Llvmlite: A Python gym for LLVM

Yashwant Singh | NVIDIA | EuroLLVM 2025 | Berlin, Germany

Who am I?

What are llvmlite and Numba?

What are llvmlite and Numba?

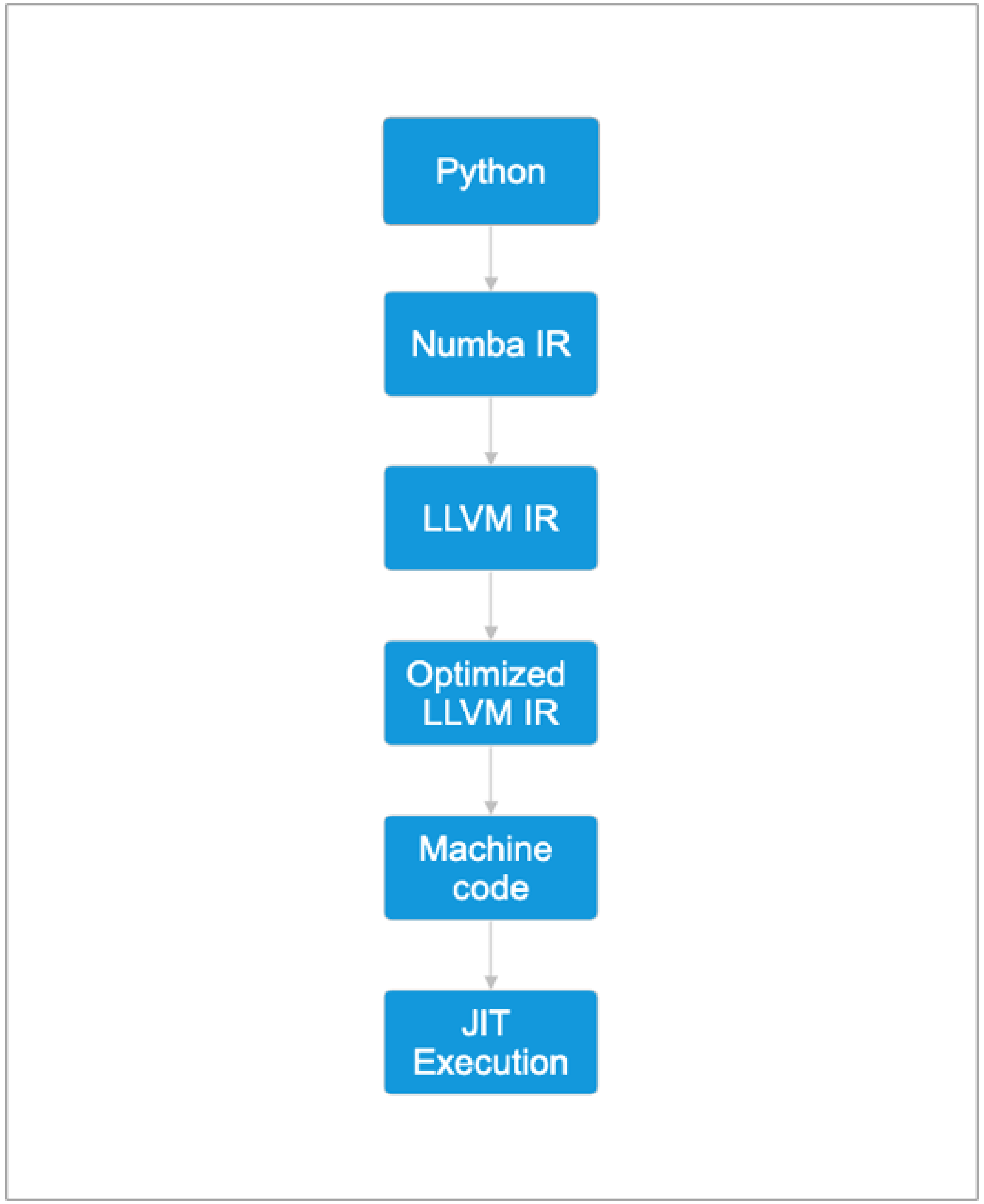
- **Numba**: An LLVM-based JIT compiler for Python

What are llvmlite and Numba?

- **Numba**: An LLVM-based JIT compiler for Python
- **Llvmlite**: Python wrapper around LLVM's C/C++ APIs

What are llvmlite and Numba?

- **Numba**: An LLVM-based JIT compiler for Python
- **Llvmlite**: Python wrapper around LLVM's C/C++ APIs



What are llvmlite and Numba?

- **Numba**: An LLVM-based JIT compiler for Python
- **Llvmlite**: Python wrapper around LLVM's C/C++ APIs

```python
import numpy as np
from numba import jit

arr = np.random.randn(100000)

@jit
def get_sum_jit(arr):
    s = 0.0
    for x in arr:
        s += x
    return s

def get_sum_no_jit(arr):
    s = 0.0
    for x in arr:
        s += x
    return s
```

What are llvmlite and Numba?

- **Numba**: An LLVM-based JIT compiler for Python
- **Llvmlite**: Python wrapper around LLVM's C/C++ APIs

```python
import numpy as np
from numba import jit

arr = np.random.randn(100000)

@jit
def get_sum_jit(arr):
    s = 0.0
    for x in arr:
        s += x
    return s

def get_sum_no_jit(arr):
    s = 0.0
    for x in arr:
        s += x
    return s
```

```
%timeit get_sum_jit(arr)
%timeit get_sum_no_jit(arr)
```

```
60.1 µs ± 60.7 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
5.14 ms ± 37.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Agenda

- IR Builder
- LLVM Playground
- Executing your IR

Disclaimer

Some of the code examples might not work with upstream/release version of llvmlite depending on the status of the below merge request:

- Shift llvmlite to LLVM 19: https://github.com/numba/llvmlite/pull/1182
- Code examples from the presentation: https://github.com/numba/llvmlite/pull/1192

IR Builder

How to use llvmlite's IRBuilder APIs to build your own LLVM based compiler?

Building an LLVM function to add 2 integers

```python
from llvmlite import ir
```

Building an LLVM function to add 2 integers

```python
from llvmlite import ir

# An empty module
mod = ir.Module(name='my-module')
```

Building an LLVM function to add 2 integers

```python
from llvmlite import ir

# An empty module
mod = ir.Module(name='my-module')

# Return type is int32 and 2 parameters of int32 type
foo_type = ir.FunctionType(ir.IntType(32), [ir.IntType(32), ir.IntType(32)])
foo = ir.Function(mod, foo_type, "add2")
```

Building an LLVM function to add 2 integers

```python
from llvmlite import ir

# An empty module
mod = ir.Module(name='my-module')

# Return type is int32 and 2 parameters of int32 type
foo_type = ir.FunctionType(ir.IntType(32), [ir.IntType(32), ir.IntType(32)])
foo = ir.Function(mod, foo_type, "add2")

# Add the entry basic block
foo.append_basic_block(name="entry")
```

Building an LLVM function to add 2 integers

Building an LLVM function to add 2 integers

```python
from llvmlite import ir

# An empty module
mod = ir.Module(name='my-module')

# Return type is int32 and 2 parameters of int32 type
foo_type = ir.FunctionType(ir.IntType(32), [ir.IntType(32), ir.IntType(32)])
foo = ir.Function(mod, foo_type, "add2")

# Add the entry basic block
foo.append_basic_block(name="entry")

builder = ir.IRBuilder(foo.entry_basic_block)
"""
builder acts as pointer and you can use it to modify the IR at 3 levels of abstraction
1) builder.block -> For adding instructions at the basic block level
2) builder.function -> For adding function level things like, arguments
3) builder.module -> For module wide changes, eg module name, target triple, etc
"""
```

Building an LLVM function to add 2 integers

```python
from llvmlite import ir

# An empty module
mod = ir.Module(name='my-module')

# Return type is int32 and 2 parameters of int32 type
foo_type = ir.FunctionType(ir.IntType(32), [ir.IntType(32), ir.IntType(32)])
foo = ir.Function(mod, foo_type, "add2")

# Add the entry basic block
foo.append_basic_block(name="entry")


builder = ir.IRBuilder(foo.entry_basic_block)
"""
builder acts as pointer and you can use it to modify the IR at 3 levels of abstraction
1) builder.block -> For adding instructions at the basic block level
2) builder.function -> For adding function level things like, arguments
3) builder.module -> For module wide changes, eg module name, target triple, etc
"""

# Let's capture the function args in a, b
a, b = builder.function.args
```

Building an LLVM function to add 2 integers

```python
from llvmlite import ir

# An empty module
mod = ir.Module(name='my-module')

# Return type is int32 and 2 parameters of int32 type
foo_type = ir.FunctionType(ir.IntType(32), [ir.IntType(32), ir.IntType(32)])
foo = ir.Function(mod, foo_type, "add2")

# Add the entry basic block
foo.append_basic_block(name="entry")

builder = ir.IRBuilder(foo.entry_basic_block)
"""
builder acts as pointer and you can use it to modify the IR at 3 levels of abstraction
1) builder.block -> For adding instructions at the basic block level
2) builder.function -> For adding function level things like, arguments
3) builder.module -> For module wide changes, eg module name, target triple, etc
"""

# Let's capture the function args in a, b
a, b = builder.function.args

# Add an 'add' instruction, that adds 'a' and 'b' and stores in 'c'
c = builder.add(a, b, 'c')
```

Building an LLVM function to add 2 integers

```python
from llvmlite import ir

# An empty module
mod = ir.Module(name='my-module')

# Return type is int32 and 2 parameters of int32 type
foo_type = ir.FunctionType(ir.IntType(32), [ir.IntType(32), ir.IntType(32)])
foo = ir.Function(mod, foo_type, "add2")

# Add the entry basic block
foo.append_basic_block(name="entry")

builder = ir.IRBuilder(foo.entry_basic_block)
"""
builder acts as pointer and you can use it to modify the IR at 3 levels of abstraction
1) builder.block -> For adding instructions at the basic block level
2) builder.function -> For adding function level things like, arguments
3) builder.module -> For module wide changes, eg module name, target triple, etc
"""

# Let's capture the function args in a, b
a, b = builder.function.args

# Add an 'add' instruction, that adds 'a' and 'b' and stores in 'c'
c = builder.add(a, b, 'c')

# Add the 'ret' instruction to return value 'c'
builder.ret(c)
```

Final code

```python
from llvmlite import ir

# An empty module
mod = ir.Module(name='my-module')

# Return type is int32 and 2 parameters of int32 type
foo_type = ir.FunctionType(ir.IntType(32), [ir.IntType(32), ir.IntType(32)])
foo = ir.Function(mod, foo_type, "add2")

# Add the entry basic block
foo.append_basic_block(name="entry")

builder = ir.IRBuilder(foo.entry_basic_block)
"""
builder acts as pointer and you can use it to modify the IR at 3 levels of abstraction
1) builder.block -> For adding instructions at the basic block level
2) builder.function -> For adding function level things like, arguments
3) builder.module -> For module wide changes, eg module name, target triple, etc
"""

# Let's capture the function args in a, b
a, b = builder.function.args

# Add an 'add' instruction, that adds 'a' and 'b' and stores in 'c'
c = builder.add(a, b, 'c')

# Add the 'ret' instruction to return value 'c'
builder.ret(c)

print(mod)
```

Final code

```python
from llvmlite import ir

# An empty module
mod = ir.Module(name='my-module')

# Return type is int32 and 2 parameters of int32 type
foo_type = ir.FunctionType(ir.IntType(32), [ir.IntType(32), ir.IntType(32)])
foo = ir.Function(mod, foo_type, "add2")

# Add the entry basic block
foo.append_basic_block(name="entry")

builder = ir.IRBuilder(foo.entry_basic_block)
"""
builder acts as pointer and you can use it to modify the IR at 3 levels of abstraction
1) builder.block -> For adding instructions at the basic block level
2) builder.function -> For adding function level things like, arguments
3) builder.module -> For module wide changes, eg module name, target triple, etc
"""

# Let's capture the function args in a, b
a, b = builder.function.args

# Add an 'add' instruction, that adds 'a' and 'b' and stores in 'c'
c = builder.add(a, b, 'c')

# Add the 'ret' instruction to return value 'c'
builder.ret(c)

print(mod)
```

```
; ModuleID = "my-module"
target triple = "unknown-unknown-unknown"
target datalayout = ""

define i32 @"add2"(i32 %".1", i32 %".2")
{
entry:
  %"c" = add i32 %".1", %".2"
  ret i32 %"c"
}
```

Adding more details to the IR

```python
builder.module.name = "test_module"
builder.module.triple = "aarch64-unknown-linux"
```

Adding more details to the IR

```python
builder.module.name = "test_module"
builder.module.triple = "aarch64-unknown-linux"

builder.function.attributes.add("noinline")

print(mod)
```

Adding more details to the IR

Adding more details to the IR

```
builder.module.name = "test_module"
builder.module.triple = "aarch64-unknown-linux"

builder.function.attributes.add("noinline")

print(mod)
```

```
; ModuleID = "test_module"
target triple = "aarch64-unknown-linux"
target datalayout = ""

define i32 @"add2"(i32 %".1", i32 %".2") noinline
{
entry:
  %"c" = add i32 %".1", %".2"
  ret i32 %"c"
}
```

Let's add another function to the module

Add a function that adds 3 integers

Let's add another function to the module

Add a function that adds 3 integers

```
int32 = ir.IntType(32)
fnty = ir.FunctionType(int32, (int32, int32, int32))

# Adding the new function to module 'mod'
bar = ir.Function(mod, fnty, "add3")

bar.append_basic_block(name="entry")

builder_bar = ir.IRBuilder(bar.entry_basic_block)

a, b, c = builder_bar.function.args[:3]
print("Function args are:", a, b, c)

# Add the 'add' instructions
sum1 = builder_bar.add(a, b, 'sum1')
sum2 = builder_bar.add(c, sum1, 'sum2')

# Add the 'ret' instruction
builder_bar.ret(sum2)

print(mod)
```

Let's add another function to the module

Add a function that adds 3 integers

```python
int32 = ir.IntType(32)
fnty = ir.FunctionType(int32, (int32, int32, int32))

# Adding the new function to module 'mod'
bar = ir.Function(mod, fnty, "add3")

bar.append_basic_block(name="entry")

builder_bar = ir.IRBuilder(bar.entry_basic_block)

a, b, c = builder_bar.function.args[:3]
print("Function args are:", a, b, c)

# Add the 'add' instructions
sum1 = builder_bar.add(a, b, 'sum1')
sum2 = builder_bar.add(c, sum1, 'sum2')

# Add the 'ret' instruction
builder_bar.ret(sum2)

print(mod)
```

```
Function args are: i32 %".1" i32 %".2" i32 %".3"
; ModuleID = "add_module"
target triple = "aarch64-unknown-linux"
target datalayout = ""

define i32 @"add2"(i32 %".1", i32 %".2") noinline
{
entry:
  %"c" = add i32 %".1", %".2"
  ret i32 %"c"
}

define i32 @"add3"(i32 %".1", i32 %".2", i32 %".3")
{
entry:
  %"sum1" = add i32 %".1", %".2"
  %"sum2" = add i32 %".3", %"sum1"
  ret i32 %"sum2"
}
```

LLVM Playground

How to leverage llvmlite to hack around with LLVM-IR using python

Code examples around:

1. Experimenting with individual LLVM passes
2. Experimenting with pass pipelines
3. Accessing LLVM visualization passes
4. Building custom optimization pipelines
5. Codegen and assembly output

1) Experimenting with LLVM's optimization passes

Let's parse the LLVM IR we want to experiment with

```
# Import the binding layer
import llvmlite.binding as llvm

# Below function takes a pointer to an array and return the number of 0s in first 10 elements
ir = r"""
define i32 @count_zeroes(i32* noalias nocapture readonly %src) {
entry:
  br label %loop.header

loop.header:
  %iv = phi i64 [ 0, %entry ], [ %inc, %loop.latch ]
  %r1  = phi i32 [ 0, %entry ], [ %r3, %loop.latch ]
  %arrayidx = getelementptr inbounds i32, i32* %src, i64 %iv
  %src_element = load i32, i32* %arrayidx, align 4
  %cmp = icmp eq i32 0, %src_element
  br i1 %cmp, label %loop.if, label %loop.latch

loop.if:
  %r2 = add i32 %r1, 1
  br label %loop.latch
loop.latch:
  %r3 = phi i32 [%r1, %loop.header], [%r2, %loop.if]
  %inc = add nuw nsw i64 %iv, 1
  %exitcond = icmp eq i64 %inc, 9
  br i1 %exitcond, label %loop.end, label %loop.header
loop.end:
  %r.lcssa = phi i32 [ %r3, %loop.latch ]
  ret i32 %r.lcssa
}
"""

# Parse the IR as a string to module object
count_zeroes_mod = llvm.parse_assembly(ir)
```

```
llvm.initialize_native_target()
llvm.initialize_native_asmprinter()
```

Initializations and helper functions

```python
llvm.initialize_native_target()
llvm.initialize_native_asmprinter()


# Helper function to create a TargetMachine object
def target_machine(jit):
    target = llvm.Target.from_default_triple()
    return target.create_target_machine(jit=jit)
```

Initializations and helper functions

Initializations and helper functions

```
llvm.initialize_native_target()
llvm.initialize_native_asmprinter()


# Helper function to create a TargetMachine object
def target_machine(jit):
    target = llvm.Target.from_default_triple()
    return target.create_target_machine(jit=jit)


# Helper function to create a PassBuilder object
def pass_builder(speed_level=0, size_level=0):
    tm = target_machine(jit=False)
    pto = llvm.\
    create_pipeline_tuning_options(speed_level, size_level)
    pb = llvm.create_pass_builder(tm, pto)
    return pb
```

Initializations and helper functions

```python
llvm.initialize_native_target()
llvm.initialize_native_asmprinter()


# Helper function to create a TargetMachine object
def target_machine(jit):
    target = llvm.Target.from_default_triple()
    return target.create_target_machine(jit=jit)


# Helper function to create a PassBuilder object
def pass_builder(speed_level=0, size_level=0):
    tm = target_machine(jit=False)
    pto = llvm.\
    create_pipeline_tuning_options(speed_level, size_level)
    pb = llvm.create_pass_builder(tm, pto)
    return pb


# Helper function to create PassManager object
def mpm():
    return llvm.create_new_module_pass_manager()
```

Running the "simplifycfg" pass on our IR

```
print(count_zeroes_mod)
```

```
; ModuleID = '<string>'
source_filename = "<string>"

define i32 @count_zeroes(ptr noalias nocapture readonly %src) {
entry:
  br label %loop.header

loop.header:                                      ; preds = %loop.latch, %entry
  %iv = phi i64 [ 0, %entry ], [ %inc, %loop.latch ]
  %r1 = phi i32 [ 0, %entry ], [ %r3, %loop.latch ]
  %arrayidx = getelementptr inbounds i32, ptr %src, i64 %iv
  %src_element = load i32, ptr %arrayidx, align 4
  %cmp = icmp eq i32 0, %src_element
  br i1 %cmp, label %loop.if, label %loop.latch

loop.if:                                          ; preds = %loop.header
  %r2 = add i32 %r1, 1
  br label %loop.latch

loop.latch:                                       ; preds = %loop.if, %loop.header
  %r3 = phi i32 [ %r1, %loop.header ], [ %r2, %loop.if ]
  %inc = add nuw nsw i64 %iv, 1
  %exitcond = icmp eq i64 %inc, 9
  br i1 %exitcond, label %loop.end, label %loop.header

loop.end:                                         ; preds = %loop.latch
  %r.lcssa = phi i32 [ %r3, %loop.latch ]
  ret i32 %r.lcssa
}
```

Running the "simplifycfg" pass on our IR

```
# Let's run simplify-cfg pass on this module
pm = mpm()
pb = pass_builder()
pm.add_simplify_cfg_pass()
pm.run(count_zeroes_mod, pb)
```

```
print(count_zeroes_mod)
```

```
; ModuleID = '<string>'
source_filename = "<string>"

define i32 @count_zeroes(ptr noalias nocapture readonly %src) {
entry:
  br label %loop.header

loop.header:                                      ; preds = %loop.latch, %entry
  %iv = phi i64 [ 0, %entry ], [ %inc, %loop.latch ]
  %r1 = phi i32 [ 0, %entry ], [ %r3, %loop.latch ]
  %arrayidx = getelementptr inbounds i32, ptr %src, i64 %iv
  %src_element = load i32, ptr %arrayidx, align 4
  %cmp = icmp eq i32 0, %src_element
  br i1 %cmp, label %loop.if, label %loop.latch

loop.if:                                          ; preds = %loop.header
  %r2 = add i32 %r1, 1
  br label %loop.latch

loop.latch:                                       ; preds = %loop.if, %loop.header
  %r3 = phi i32 [ %r1, %loop.header ], [ %r2, %loop.if ]
  %inc = add nuw nsw i64 %iv, 1
  %exitcond = icmp eq i64 %inc, 9
  br i1 %exitcond, label %loop.end, label %loop.header

loop.end:                                         ; preds = %loop.latch
  %r.lcssa = phi i32 [ %r3, %loop.latch ]
  ret i32 %r.lcssa
}
```

Running the "simplifycfg" pass on our IR

```
# Let's run simplify-cfg pass on this module
pm = mpm()
pb = pass_builder()
pm.add_simplify_cfg_pass()
pm.run(count_zeroes_mod, pb)
```

```
print(count_zeroes_mod)
```

```
; ModuleID = '<string>'
source_filename = "<string>"

define i32 @count_zeroes(ptr noalias nocapture readonly %src) {
entry:
  br label %loop.header

loop.header:                                      ; preds = %loop.latch, %entry
  %iv = phi i64 [ 0, %entry ], [ %inc, %loop.latch ]
  %r1 = phi i32 [ 0, %entry ], [ %r3, %loop.latch ]
  %arrayidx = getelementptr inbounds i32, ptr %src, i64 %iv
  %src_element = load i32, ptr %arrayidx, align 4
  %cmp = icmp eq i32 0, %src_element
  br i1 %cmp, label %loop.if, label %loop.latch

loop.if:                                          ; preds = %loop.header
  %r2 = add i32 %r1, 1
  br label %loop.latch

loop.latch:                                       ; preds = %loop.if, %loop.header
  %r3 = phi i32 [ %r1, %loop.header ], [ %r2, %loop.if ]
  %inc = add nuw nsw i64 %iv, 1
  %exitcond = icmp eq i64 %inc, 9
  br i1 %exitcond, label %loop.end, label %loop.header

loop.end:                                         ; preds = %loop.latch
  %r.lcssa = phi i32 [ %r3, %loop.latch ]
  ret i32 %r.lcssa
}
```

```
print(count_zeroes_mod)
```

```
; ModuleID = '<string>'
source_filename = "<string>"

define i32 @count_zeroes(ptr noalias nocapture readonly %src) {
entry:
  br label %loop.header

loop.header:                                      ; preds = %loop.header, %entry
  %iv = phi i64 [ 0, %entry ], [ %inc, %loop.header ]
  %r1 = phi i32 [ 0, %entry ], [ %spec.select, %loop.header ]
  %arrayidx = getelementptr inbounds i32, ptr %src, i64 %iv
  %src_element = load i32, ptr %arrayidx, align 4
  %cmp = icmp eq i32 0, %src_element
  %r2 = add i32 %r1, 1
  %spec.select = select i1 %cmp, i32 %r2, i32 %r1
  %inc = add nuw nsw i64 %iv, 1
  %exitcond = icmp eq i64 %inc, 9
  br i1 %exitcond, label %loop.end, label %loop.header

loop.end:                                         ; preds = %loop.header
  %r.lcssa = phi i32 [ %spec.select, %loop.header ]
  ret i32 %r.lcssa
}
```

2) How about default optimization pipelines (O0, O3, etc)?

```python
# Given function takes 2 values as args and return their sum
addFunc = r"""
define noundef i32 @add2nums(i32 noundef %0, i32 noundef %1) {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
"""

# Initialize and parse the llvm module from a python string
mod = llvm.parse_assembly(addFunc)
```

2) How about default optimization pipelines (O0, O3, etc)?

2) How about default optimization pipelines (O0, O3, etc)?

```
# Given function takes 2 values as args and return their sum
addFunc = r"""
define noundef i32 @add2nums(i32 noundef %0, i32 noundef %1) {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
"""

# Initialize and parse the llvm module from a python string
mod = llvm.parse_assembly(addFunc)
```

```
# Initialize pass builder with speed_level=3, i.e -O3
pb = pass_builder(speed_level=3)
```

2) How about default optimization pipelines (O0, O3, etc)?

```
# Given function takes 2 values as args and return their sum
addFunc = r"""
define noundef i32 @add2nums(i32 noundef %0, i32 noundef %1) {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
"""

# Initialize and parse the llvm module from a python string
mod = llvm.parse_assembly(addFunc)
```

```
# Initialize pass builder with speed_level=3, i.e -O3
pb = pass_builder(speed_level=3)
```

```
# Get appropriate pass manager for this speed level and optimise
pm = pb.getModulePassManager()
pm.run(mod, pb)
print(mod)
```

2) How about default optimization pipelines (O0, O3, etc)?

```
# Given function takes 2 values as args and return their sum
addFunc = r"""
define noundef i32 @add2nums(i32 noundef %0, i32 noundef %1) {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
"""

# Initialize and parse the llvm module from a python string
mod = llvm.parse_assembly(addFunc)
```

```
# Initialize pass builder with speed_level=3, i.e -O3
pb = pass_builder(speed_level=3)
```

```
# Get appropriate pass manager for this speed level and optimise
pm = pb.getModulePassManager()
pm.run(mod, pb)
print(mod)
```

```
; ModuleID = '<string>'
source_filename = "<string>"

; Function Attrs: mustprogress nofree norecurse nosync nounwind willreturn memory(none)
define noundef i32 @add2nums(i32 noundef %0, i32 noundef %1) local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0
  ret i32 %3
}

attributes #0 = { mustprogress nofree norecurse nosync nounwind willreturn memory(none) }
```

3) Experimenting with LLVM'S visualization passes

```
print(count_zeroes_mod)
```

```
; ModuleID = '<string>'
source_filename = "<string>"

define i32 @count_zeroes(ptr noalias nocapture readonly %src) {
entry:
  br label %loop.header

loop.header:                                      ; preds = %loop.header, %entry
  %iv = phi i64 [ 0, %entry ], [ %inc, %loop.header ]
  %r1 = phi i32 [ 0, %entry ], [ %spec.select, %loop.header ]
  %arrayidx = getelementptr inbounds i32, ptr %src, i64 %iv
  %src_element = load i32, ptr %arrayidx, align 4
  %cmp = icmp eq i32 0, %src_element
  %r2 = add i32 %r1, 1
  %spec.select = select i1 %cmp, i32 %r2, i32 %r1
  %inc = add nuw nsw i64 %iv, 1
  %exitcond = icmp eq i64 %inc, 9
  br i1 %exitcond, label %loop.end, label %loop.header

loop.end:                                         ; preds = %loop.header
  %r.lcssa = phi i32 [ %spec.select, %loop.header ]
  ret i32 %r.lcssa
}
```

3) Experimenting with LLVM'S visualization passes

```python
def renderModuleAsDotGraph(mod, func_name):
  pm = mpm()
  pm.add_cfg_printer_pass()
  pm.run(mod, pass_builder())   # dot graph written to ".func_name.dot"
  !dot -Tpng .{func_name}.dot > {func_name}.png
```

```python
print(count_zeroes_mod)
```

```
; ModuleID = '<string>'
source_filename = "<string>"

define i32 @count_zeroes(ptr noalias nocapture readonly %src) {
entry:
  br label %loop.header

loop.header:                                      ; preds = %loop.header, %entry
  %iv = phi i64 [ 0, %entry ], [ %inc, %loop.header ]
  %r1 = phi i32 [ 0, %entry ], [ %spec.select, %loop.header ]
  %arrayidx = getelementptr inbounds i32, ptr %src, i64 %iv
  %src_element = load i32, ptr %arrayidx, align 4
  %cmp = icmp eq i32 0, %src_element
  %r2 = add i32 %r1, 1
  %spec.select = select i1 %cmp, i32 %r2, i32 %r1
  %inc = add nuw nsw i64 %iv, 1
  %exitcond = icmp eq i64 %inc, 9
  br i1 %exitcond, label %loop.end, label %loop.header

loop.end:                                         ; preds = %loop.header
  %r.lcssa = phi i32 [ %spec.select, %loop.header ]
  ret i32 %r.lcssa
}
```

3) Experimenting with LLVM'S visualization passes

```python
def renderModuleAsDotGraph(mod, func_name):
  pm = mpm()
  pm.add_cfg_printer_pass()
  pm.run(mod, pass_builder())   # dot graph written to ".func_name.dot"
  !dot -Tpng .{func_name}.dot > {func_name}.png
```

```python
from IPython.display import Image
renderModuleAsDotGraph(count_zeroes_mod, "count_zeroes")
Image('count_zeroes.png')
```

```
Writing '.count_zeroes.dot'...
```

```python
print(count_zeroes_mod)
```

```
; ModuleID = '<string>'
source_filename = "<string>"

define i32 @count_zeroes(ptr noalias nocapture readonly %src) {
entry:
  br label %loop.header

loop.header:                                      ; preds = %loop.header, %entry
  %iv = phi i64 [ 0, %entry ], [ %inc, %loop.header ]
  %r1 = phi i32 [ 0, %entry ], [ %spec.select, %loop.header ]
  %arrayidx = getelementptr inbounds i32, ptr %src, i64 %iv
  %src_element = load i32, ptr %arrayidx, align 4
  %cmp = icmp eq i32 0, %src_element
  %r2 = add i32 %r1, 1
  %spec.select = select i1 %cmp, i32 %r2, i32 %r1
  %inc = add nuw nsw i64 %iv, 1
  %exitcond = icmp eq i64 %inc, 9
  br i1 %exitcond, label %loop.end, label %loop.header

loop.end:                                         ; preds = %loop.header
  %r.lcssa = phi i32 [ %spec.select, %loop.header ]
  ret i32 %r.lcssa
}
```

3) Experimenting with LLVM'S visualization passes

```python
def renderModuleAsDotGraph(mod, func_name):
  pm = mpm()
  pm.add_cfg_printer_pass()
  pm.run(mod, pass_builder())   # dot graph written to ".func_name.dot"
  !dot -Tpng .{func_name}.dot > {func_name}.png
```

```python
from IPython.display import Image
renderModuleAsDotGraph(count_zeroes_mod, "count_zeroes")
Image('count_zeroes.png')
```

```
Writing '.count_zeroes.dot'...
```

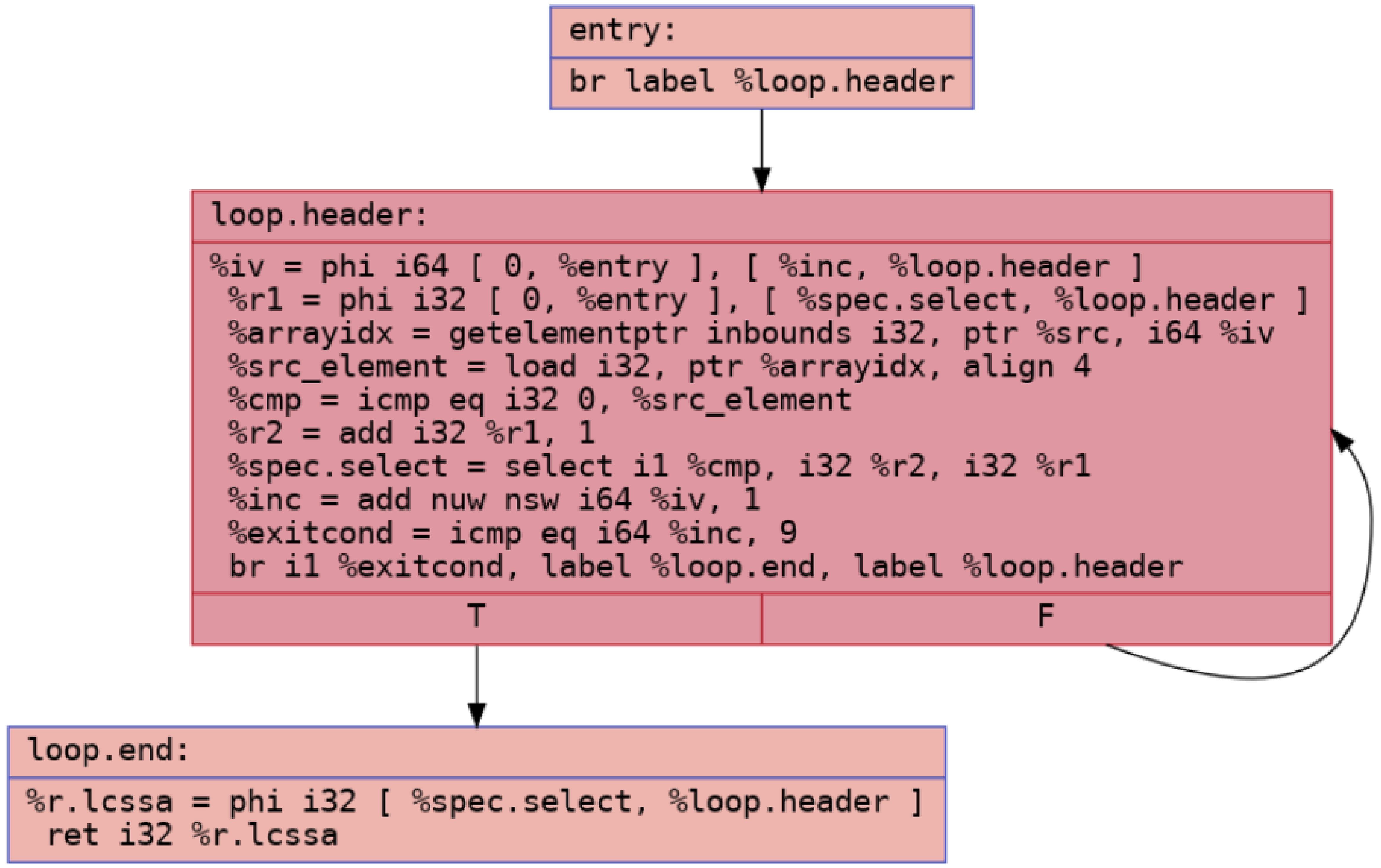
```python
print(count_zeroes_mod)
```

```
; ModuleID = '<string>'
source_filename = "<string>"

define i32 @count_zeroes(ptr noalias nocapture readonly %src) {
entry:
  br label %loop.header

loop.header:                                      ; preds = %loop.header, %entry
  %iv = phi i64 [ 0, %entry ], [ %inc, %loop.header ]
  %r1 = phi i32 [ 0, %entry ], [ %spec.select, %loop.header ]
  %arrayidx = getelementptr inbounds i32, ptr %src, i64 %iv
  %src_element = load i32, ptr %arrayidx, align 4
  %cmp = icmp eq i32 0, %src_element
  %r2 = add i32 %r1, 1
  %spec.select = select i1 %cmp, i32 %r2, i32 %r1
  %inc = add nuw nsw i64 %iv, 1
  %exitcond = icmp eq i64 %inc, 9
  br i1 %exitcond, label %loop.end, label %loop.header

loop.end:                                         ; preds = %loop.header
  %r.lcssa = phi i32 [ %spec.select, %loop.header ]
  ret i32 %r.lcssa
}
```



CFG for 'count_zeroes' function

More visualizations?

More visualizations?

```
# Function to generate dominator tree of a LLVM function

def generateDom(mod, func_name):
  pm = mpm()
  pm.add_dom_printer_pass()
  pm.run(mod, pass_builder())   # dot graph written to ".func_name.dot"
  !dot -Tpng dom.{func_name}.dot > {func_name}.png
```

```python
# Function to generate post dominator tree of a LLVM function

def generatePostDom(mod, func_name):
  pm = mpm()
  pm.add_post_dom_printer_pass()
  pm.run(mod, pass_builder())   # dot graph written to ".func_name.dot"
  !dot -Tpng postdom.{func_name}.dot > {func_name}.png
```

More visualizations?

```python
# Function to generate dominator tree of a LLVM function

def generateDom(mod, func_name):
  pm = mpm()
  pm.add_dom_printer_pass()
  pm.run(mod, pass_builder())   # dot graph written to ".func_name.dot"
  !dot -Tpng dom.{func_name}.dot > {func_name}.png
```

4) Building custom optimization pipelines

```
asm_inlineasm2 = r"""
    define i32 @caller(i32 %.1, i32 %.2) {
    entry:
      %stack = alloca i32
      store i32 %.1, i32* %stack
      br label %main
    main:
      %loaded = load i32, i32* %stack
      %.3 = add i32 %loaded, %.2
      %.4 = add i32 0, %.3
      ret i32 %.4
    }
"""

mod = llvm.parse_assembly(asm_inlineasm2)
```

```python
asm_inlineasm2 = r"""
    define i32 @caller(i32 %.1, i32 %.2) {
    entry:
      %stack = alloca i32
      store i32 %.1, i32* %stack
      br label %main
    main:
      %loaded = load i32, i32* %stack
      %.3 = add i32 %loaded, %.2
      %.4 = add i32 0, %.3
      ret i32 %.4
    }
"""

mod = llvm.parse_assembly(asm_inlineasm2)
```

```python
pm = mpm()
pm.add_constant_merge_pass()
pm.add_dead_arg_elimination_pass()
pm.add_post_order_function_attributes_pass()
# pm.add_function_inlining_pass(225)
pm.add_global_dead_code_eliminate_pass()
pm.add_global_opt_pass()
pm.add_ipsccp_pass()
pm.add_dead_code_elimination_pass()
pm.add_simplify_cfg_pass()
pm.add_new_gvn_pass()
pm.add_instruction_combine_pass()
# pm.add_licm_pass()
pm.add_sccp_pass()
# pm.add_sroa_pass()
# pm.add_type_based_alias_analysis_pass()
# pm.add_basic_alias_analysis_pass()
# pm.add_loop_rotate_pass()
# pm.add_region_info_pass()
# pm.add_scalar_evolution_aa_pass()
# pm.add_aggressive_dead_code_elimination_pass()
# pm.add_aa_eval_pass()
# pm.add_always_inliner_pass()
# pm.add_break_critical_edges_pass()
# pm.add_dead_store_elimination_pass()
# pm.add_reverse_post_order_function_attrs_pass()
pm.run(mod, pass_builder())
print(mod)
```

```
; ModuleID = '<string>'
source_filename = "<string>"

; Function Attrs: mustprogress nofree norecurse nosync nounwind willreturn memory(none)
define i32 @caller(i32 %.1, i32 %.2) local_unnamed_addr #0 {
entry:
  %.3 = add i32 %.1, %.2
  ret i32 %.3
}

attributes #0 = { mustprogress nofree norecurse nosync nounwind willreturn memory(none) }
```

5) CodeGen?

```python
llvm.initialize_all_targets()
#llvm.initialize_native_target()

llvm.initialize_all_asmprinters()
# llvm.initialize_native_asmprinter()
```

```python
llvm.initialize_all_targets()
#llvm.initialize_native_target()

llvm.initialize_all_asmprinters()
# llvm.initialize_native_asmprinter()


target_riscv = llvm.Target.from_triple("riscv32-unknown-linux")
riscv_tm = target_riscv.create_target_machine()
```

```python
llvm.initialize_all_targets()
#llvm.initialize_native_target()

llvm.initialize_all_asmprinters()
# llvm.initialize_native_asmprinter()


target_riscv = llvm.Target.from_triple("riscv32-unknown-linux")
riscv_tm = target_riscv.create_target_machine()


target_x86 = llvm.Target.from_triple("x86_64-pc-windows-msvc")
x86_tm = target_x86.create_target_machine()
```

```python
llvm.initialize_all_targets()
#llvm.initialize_native_target()

llvm.initialize_all_asmprinters()
# llvm.initialize_native_asmprinter()


target_riscv = llvm.Target.from_triple("riscv32-unknown-linux")
riscv_tm = target_riscv.create_target_machine()


target_x86 = llvm.Target.from_triple("x86_64-pc-windows-msvc")
x86_tm = target_x86.create_target_machine()


native_tm = llvm.Target.from_default_triple().create_target_machine()
```

```python
llvm.initialize_all_targets()
#llvm.initialize_native_target()

llvm.initialize_all_asmprinters()
# llvm.initialize_native_asmprinter()


target_riscv = llvm.Target.from_triple("riscv32-unknown-linux")
riscv_tm = target_riscv.create_target_machine()


target_x86 = llvm.Target.from_triple("x86_64-pc-windows-msvc")
x86_tm = target_x86.create_target_machine()


native_tm = llvm.Target.from_default_triple().create_target_machine()


ir = """
define dso_local noundef i32 @add(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
"""
mod = llvm.parse_assembly(ir)
```

```python
print("*" * 40, "X86 asm")
print(x86_tm.emit_assembly(mod))
print("*" * 40, "RISCV asm")
print(riscv_tm.emit_assembly(mod))
print("*" * 40, "Native asm")
print(native_tm.emit_assembly(mod))
```

```python
print("*" * 40, "X86 asm")
print(x86_tm.emit_assembly(mod))
print("*" * 40, "RISCV asm")
print(riscv_tm.emit_assembly(mod))
print("*" * 40, "Native asm")
print(native_tm.emit_assembly(mod))
```

```
**************************************** X86 asm
	.def	@feat.00;
	.scl	3;
	.type	0;
	.endef
	.globl	@feat.00
.set @feat.00, 0
	.file	"<string>"
	.def	add;
	.scl	2;
	.type	32;
	.endef
	.text
	.globl	add
	.p2align	4
add:
.seh_proc add
	pushq	%rax
	.seh_stackalloc 8
	.seh_endprologue
	movl	%ecx, 4(%rsp)
	movl	%edx, (%rsp)
	leal	(%rcx,%rdx), %eax
	popq	%rcx
	retq
	.seh_endproc


**************************************** RISCV asm
	.attribute	4, 16
	.attribute	5, "rv32i2p1"
	.file	"<string>"
	.text
	.globl	add
	.p2align	2
	.type	add,@function
add:
	.cfi_startproc
	addi	sp, sp, -16
	.cfi_def_cfa_offset 16
	sw	a0, 12(sp)
	add	a0, a0, a1
	sw	a1, 8(sp)
	addi	sp, sp, 16
	.cfi_def_cfa_offset 0
	ret
.Lfunc_end0:
	.size	add, .Lfunc_end0-add
	.cfi_endproc

	.section	".note.GNU-stack","",@progbits

**************************************** Native asm
	.file	"<string>"
	.text
	.globl	add
	.p2align	2
	.type	add,@function
add:
	.cfi_startproc
	sub	sp, sp, #16
	.cfi_def_cfa_offset 16
	stp	w1, w0, [sp, #8]
	add	w0, w0, w1
	add	sp, sp, #16
	ret
.Lfunc_end0:
	.size	add, .Lfunc_end0-add
	.cfi_endproc

	.section	".note.GNU-stack","",@progbits
```

Executing your IR

Using the LLVM's JIT execution engine to execute the LLVM IR

Let's execute the "add2" function that we created earlier

Let's execute the "add2" function that we created earlier

```
from ctypes import CFUNCTYPE, c_int, POINTER

ir = """
; ModuleID = "add_module"
target triple = "aarch64-unknown-linux"
target datalayout = ""

define i32 @"add2"(i32 %".1", i32 %".2") noinline
{
entry:
  %"c" = add i32 %".1", %".2"
  ret i32 %"c"
}
"""
llmod = llvm.parse_assembly(ir)
```

Let's execute the "add2" function that we created earlier

```python
from ctypes import CFUNCTYPE, c_int, POINTER

ir = """
; ModuleID = "add_module"
target triple = "aarch64-unknown-linux"
target datalayout = ""

define i32 @"add2"(i32 %".1", i32 %".2") noinline
{
entry:
  %"c" = add i32 %".1", %".2"
  ret i32 %"c"
}
"""
llmod = llvm.parse_assembly(ir)

tm = target_machine(False)
compiler = llvm.create_mcjit_compiler(llmod, tm)
compiler.finalize_object()
```

Let's execute the "add2" function that we created earlier

```
from ctypes import CFUNCTYPE, c_int, POINTER

ir = """
; ModuleID = "add_module"
target triple = "aarch64-unknown-linux"
target datalayout = ""

define i32 @"add2"(i32 %".1", i32 %".2") noinline
{
entry:
  %"c" = add i32 %".1", %".2"
  ret i32 %"c"
}
"""
llmod = llvm.parse_assembly(ir)

tm = target_machine(False)
compiler = llvm.create_mcjit_compiler(llmod, tm)
compiler.finalize_object()
```

```
cfptr_add2 = compiler.get_function_address("add2")
cfunc_add2 = CFUNCTYPE(c_int, c_int)(cfptr_add2)
```

```
print(cfunc_add2(-1, 2))
print(cfunc_add2(1, 2))
```

```
1
3
```

NVIDIA

Thank you!!

Questions?

Bonus slides

Custom target machines

Customized target machines for specific CPUs/features

```
target = llvm.Target.from_triple("aarch64-unknown-linux")


tm_default_aarch64 = target.create_target_machine(cpu='', features='',
                                opt=2, reloc='default', codemodel='jitdefault',
                                printmc=False, jit=False, abiname='')


tm_neoverse_v2 = tm = target.create_target_machine(cpu='neoverse-v2', features='',
                                opt=2, reloc='default', codemodel='jitdefault',
                                printmc=False, jit=False, abiname='')


tm_features = tm = target.create_target_machine(cpu='', features='+crc,+crypto,\
                                 +fp-armv8,+lse,+neon,+sve,+sve2',
                                opt=2, reloc='default', codemodel='jitdefault',
                                printmc=False, jit=False, abiname='')
```

```
code_object_native = tm.emit_object(mod)
code_object_aarch64 = aarch64_tm.emit_object(mod)
print(code_object_native)
```

Object code?

```python
code_object_native = tm.emit_object(mod)
code_object_aarch64 = aarch64_tm.emit_object(mod)
print(code_object_native)
```

Object code?


```
b'\x7fELF\x02\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\xb7\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x80\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00@\x00\x07\x00\x01\x00\xffC\x00\xd1\xe1\x03\x01)\x00\x00\x0
1\x0b\xffC\x00\x91\xc0\x03_\xd6\x00\x00\x00\x00\x10\x00\x00\x00\x00\x00\x00\x00\x01zR\x00\x01|\x1e\x01\x1c\x0c\x1f\x00\x18\x00\x00\x00\x18\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x14\x00\x00\x00\x00\x00\x00\x00\x00D\x0e\x10\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x04\x00\xf1\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x03\x00\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x02\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00-\x00\x00\x00\x00\x00\x04\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00)\x00\x00\x00\x12\x00\x02\x00\x00\x00\x00\x00\x00\x00\x00\x14\x00\x00\x00\x00\x00\x00\x00\x1c\x00\x00\x00\x00\x00\x00\x00\x04\x01\x00
\x00\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00$x\x00.text\x00.note.GNU-stack\x00.rela.eh_frame\x00add\x00$d\x00.strtab\x00.symtab
\x00<string>\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x000\x00\x00\x00\x03\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x000\x01\x00\x00\x00\x00\x00\x
00I\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00\x00
\x00\x01\x00\x00\x00\x06\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x14\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\n\x00\x00\x00\x01\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00T\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x1f\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00X\x00\x00\x00\x00\x00\x000\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x1a\x00\x00\x00\x04\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x1
8\x01\x00\x00\x00\x00\x00\x00\x18\x00\x00\x00\x00\x00\x00\x00\x06\x00\x00\x00\x04\x00\x00\x00\x08\x00\x00\x00\x00\x00\x00\x00\x18\x00\x00\x0
0\x00\x00\x00\x008\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x88\x00\x00\x00\x00\x00\x00\x
00\x90\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x05\x00\x00\x00\x08\x00\x00\x00\x00\x00\x00\x00\x18\x00\x00\x00\x00\x00\x00\x00'
```

Optimizing and executing a more complex example:

Sum of all elements in an array

Optimizing and executing a more complex example:

Sum of all elements in an array

```
ir = """
; ModuleID = '<string>'
source_filename = "<string>"
target triple = "unknown-unknown-unknown"

define i32 @sum(i32* %.1, i32 %.2) {
.4:
  br label %.5

.5:                                               ; preds = %.5, %.4
  %.8 = phi i32 [ 0, %.4 ], [ %.13, %.5 ]
  %.9 = phi i32 [ 0, %.4 ], [ %.12, %.5 ]
  %.10 = getelementptr i32, i32* %.1, i32 %.8
  %.11 = load i32, i32* %.10, align 4
  %.12 = add i32 %.9, %.11
  %.13 = add i32 %.8, 1
  %.14 = icmp ult i32 %.13, %.2
  br i1 %.14, label %.5, label %.6

.6:                                               ; preds = %.5
  ret i32 %.12
}
"""

mod = llvm.parse_assembly(ir)
pb = pass_builder(speed_level=3)

# Get appropriate pass manager for this speed level and optimise the module
pm = pb.getModulePassManager()
pm.run(mod, pb)
print(mod)
```

Optimizing and executing a more complex example:

Sum of all elements in an array

```
; ModuleID = '<string>'
source_filename = "<string>"
target triple = "unknown-unknown-unknown"

; Function Attrs: nofree norecurse nosync nounwind memory(argmem: read)
define i32 @sum(ptr nocapture readonly %.1, i32 %.2) local_unnamed_addr #0 {
.4:
  %0 = add i32 %.2, 2147483647
  %or.cond = icmp ult i32 %0, -2147483641
  br i1 %or.cond, label %.5.preheader, label %vector.ph

vector.ph:                                        ; preds = %.4
  %n.vec = and i32 %.2, -8
  br label %vector.body

vector.body:                                      ; preds = %vector.body, %vector.ph
  %index = phi i32 [ 0, %vector.ph ], [ %index.next, %vector.body ]
  %vec.phi = phi <4 x i32> [ zeroinitializer, %vector.ph ], [ %4, %vector.body ]
  %vec.phi2 = phi <4 x i32> [ zeroinitializer, %vector.ph ], [ %5, %vector.body ]
  %1 = sext i32 %index to i64
  %2 = getelementptr i32, ptr %.1, i64 %1
  %3 = getelementptr i8, ptr %2, i64 16
  %wide.load = load <4 x i32>, ptr %2, align 4
  %wide.load3 = load <4 x i32>, ptr %3, align 4
  %4 = add <4 x i32> %wide.load, %vec.phi
  %5 = add <4 x i32> %wide.load3, %vec.phi2
  %index.next = add nuw i32 %index, 8
  %6 = icmp eq i32 %index.next, %n.vec
  br i1 %6, label %middle.block, label %vector.body, !llvm.loop !0

middle.block:                                     ; preds = %vector.body
  %bin.rdx = add <4 x i32> %5, %4
  %7 = tail call i32 @llvm.vector.reduce.add.v4i32(<4 x i32> %bin.rdx)
  %cmp.n = icmp eq i32 %.2, %n.vec
  br i1 %cmp.n, label %.6, label %.5.preheader

.5.preheader:                                     ; preds = %.4, %middle.block
  %.8.ph = phi i32 [ 0, %.4 ], [ %n.vec, %middle.block ]
  %.9.ph = phi i32 [ 0, %.4 ], [ %7, %middle.block ]
  br label %.5

.5:                                               ; preds = %.5.preheader, %.5
  %.8 = phi i32 [ %.13, %.5 ], [ %.8.ph, %.5.preheader ]
  %.9 = phi i32 [ %.12, %.5 ], [ %.9.ph, %.5.preheader ]
  %8 = sext i32 %.8 to i64
  %.10 = getelementptr i32, ptr %.1, i64 %8
  %.11 = load i32, ptr %.10, align 4
  %.12 = add i32 %.11, %.9
  %.13 = add nuw i32 %.8, 1
  %.14 = icmp ult i32 %.13, %.2
  br i1 %.14, label %.5, label %.6, !llvm.loop !3

.6:                                               ; preds = %.5, %middle.block
  %.12.lcssa = phi i32 [ %7, %middle.block ], [ %.12, %.5 ]
  ret i32 %.12.lcssa
}

; Function Attrs: nocallback nofree nosync nounwind speculatable willreturn memory(none)
declare i32 @llvm.vector.reduce.add.v4i32(<4 x i32>) #1

attributes #0 = { nofree norecurse nosync nounwind memory(argmem: read) }
attributes #1 = { nocallback nofree nosync nounwind speculatable willreturn memory(none) }

!0 = distinct !{!0, !1, !2}
!1 = !{!"llvm.loop.isvectorized", i32 1}
!2 = !{!"llvm.loop.unroll.runtime.disable"}
!3 = distinct !{!3, !1}
```

Optimizing and executing a more complex example:

Sum of all elements in an array

```
from ctypes import CFUNCTYPE, c_int, POINTER

tm = target_machine(True)
compiler = llvm.create_mcjit_compiler(mod, tm)

compiler.finalize_object()
cfptr = compiler.get_function_address("sum")

cfunc = CFUNCTYPE(c_int, POINTER(c_int), c_int)(cfptr)

A = np.arange(10, dtype=np.int32)
res = cfunc(A.ctypes.data_as(POINTER(c_int)), A.size)

B = [1, 2, 3]
arr = (c_int * len(B))(*B)
res = cfunc(arr, len(B))


print(A)
print(res, A.sum())

print(B)
print(res, sum(B))
```

```
[0 1 2 3 4 5 6 7 8 9]
45 45
[1, 2, 3]
6 6
```