llvmlite Documentation

Release 0.44.0

Continuum Analytics

Aug 19, 2026

CONTENTS

1 Philosophy 3

2 LLVM compatibility 5

3 API stability 7

Python Module Index 83

Index 85

A lightweight LLVM-Python binding for writing JIT compilers

llvmlite provides a Python binding to LLVM for use in Numba. Numba previously relied on llvmpy.

Llvmpy became hard to maintain because:

- It has a cumbersome architecture.
- The C++11 requirement of recent LLVM versions does not go well with the compiler and runtime ABI requirements of some Python versions, especially under Windows.

Llvmpy also proved to be responsible for a sizable part of Numba's compilation times, because of its inefficient layering and object encapsulation. Fixing this issue inside the llvmpy codebase seemed a time-consuming and uncertain task.

The Numba developers decided to start a new binding from scratch, with an entirely different architecture, centered around the specific requirements of a JIT compiler.

CHAPTER

ONE

PHILOSOPHY

While llvmpy exposed large parts of the LLVM C++ API for direct calls into the LLVM library, llvmlite takes an entirely different approach. Llvmlite starts from the needs of a JIT compiler and splits them into two decoupled tasks:

- Construction of a *Module*, function by function, *Instruction* by instruction.
- Compilation and optimization of the module into machine code.

The construction of an LLVM module does not call the LLVM C++ API. Rather, it constructs the LLVM *intermediate representation* (IR) in pure Python. This is the role of the *IR layer*.

The compilation of an LLVM module takes the IR in textual form and feeds it into LLVM's parsing API. It then returns a thin wrapper around LLVM's C++ module object. This is the role of the *binding layer*.

Once parsed, the module's source code cannot be modified, which loses the flexibility of the direct mapping of C++ APIs into Python that was provided by llvmpy but saves a great deal of maintenance.

CHAPTER

TWO

LLVM COMPATIBILITY

Despite minimizing the API surface with LLVM, llvmlite is impacted by changes to LLVM's C++ API, which can occur at every feature release. Therefore, each llvmlite version is targeted to a specific LLVM feature version and works across all given bugfix releases of that version.

EXAMPLE: Llvmlite 0.12.0 works with LLVM 3.8.0 and 3.8.1, but it does not work with LLVM 3.7.0 or 3.9.0.

Numba's requirements determine which LLVM version is supported.

CHAPTER

THREE

API STABILITY

At this time, we reserve the possibility of slightly breaking the llvmlite API at each release, for the following reasons:

- Changes in LLVM behaviour, such as differences in the IR across versions.
- As a young library, llvmlite has room for improvement or fixes to the existing APIs.

3.1 Installation

The Numba/llvmlite stack consists of the following major components:

- *Numba* is the compiler package, this depends on llvmlite.
- *llvmlite* is a lightweight binding package to the LLVM APIs, it depends on LLVM.
- *LLVM* is the JIT compiler framework for producing executable code from various inputs.

All components must be compiled in order to be used. And, since each component on the stack depends on the previous one, you need to compile LLVM in order to compile llvmlite in order to compile Numba. The LLVM package is a significant size and may take significant time (magnitude, roughly an hour) and skill to compile depending on the platform.

3.1.1 Pre-built binaries

As mentioned above, building LLVM for llvmlite is challenging. Installing a binary package that has been built and tested is *strongly* recommend.

Official Conda packages are available in the Anaconda distribution:

```
conda install llvmlite
```

Development releases are built from the Git main branch and uploaded to the Numba development channel on Anaconda Cloud:

```
conda install -c numba/label/dev llvmlite
```

Binary wheels are also available for installation from PyPI:

```
pip install llvmlite
```

Development releases of binary wheels are not made available.

Contrary to what might be expected, the llvmlite packages built by the Numba maintainers do *not* use any LLVM shared libraries that may be present on the system, and/or in the Conda environment. The parts of LLVM required by llvmlite are statically linked at build time. As a result, installing llvmlite from a binary package from the Numba channel does not also require the end user to install LLVM. (For more details on the reasoning behind this, see: *Why Static Linking*

to LLVM?). Note however also that llvmlite packages compiled by other parties, e.g. conda-forge may split this into and `llvmlite` and `llvm` package and link dynamically.

Conda packages:

The Numba maintainers ship to the Numba channel:

- Numba packages
- llvmlite packages
- llvmdev packages (this contains a build of LLVM)

The llvmdev packages are not needed at runtime by llvmlite packages as llvmlite's dynamic libraries are statically linked (see above) at compile time against LLVM through the dependency on the `llvmdev` package.

The Anaconda distribution and conda-forge channels ship:

- Numba packages
- llvmlite packages
- LLVM split into runtime libraries (package called `llvm`) and compile time libraries/headers etc this contains a build of LLVM (package called `llvmdev`)

At compile time the `llvmdev` and `llvm` packages are used to build llvmlite and llvmlite's dynamic libraries are dynamically linked against the libraries in the `llvm` meta-package. This means at runtime `llvmlite` depends on the `llvm` package which has the LLVM shared libraries in it (it's actually a package called `libllvm` that contains the DSOs, but the `llvm` package is referred to so as to get the `run_exports`).

Using `pip`

The Numba maintainers ship binary wheels:

- Numba wheels (`x86*` architectures)
- llvmlite wheels (`x86*` architectures)

Note that the llvmlite wheels are statically linked against LLVM, as per the conda packages on the Numba channel. This mitigates the need for a LLVM based binary wheel. Note also that this, as of version 0.36, does not include the `aarch64` architectures, for example installation on a Raspberry Pi is not supported.

The Numba maintainers ship an `sdist` for:

- Numba
- llvmlite

Note that there is no `sdist` provided for LLVM. If you try and build `llvmlite` from `sdist` you will need to bootstrap the package with your own appropriate LLVM.

How this ends up being a problem.

1. If you are on an unsupported architecture (i.e. not `x86*`) or unsupported Python version for binary wheels (e.g. Python alphas) then `pip` will try and build Numba from `sdist` which in turn will try and build `llvmlite` from `sdist`. This will inevitably fail as the `llvmlite` source distribution needs an appropriate LLVM installation to build.
2. If you are using `pip < 19.0` then `manylinux2010` wheels will not install and you end up in the situation in 1. i.e. something unsupported so building from `sdist`.

Historically, this issues has manifested itself as the following error message, which included here verbatim for future reference:

```
FileNotFoundError: [Errno 2] No such file or directory: 'llvm-config'
```

Things to "fix" it...

1. If you are using `pip < 19.0` and on `x86*`, then update it if you can, this will let you use the `manylinux2010` binary wheels.
2. If you are on an unsupported architecture, for example Raspberry Pi, please use `conda` if you have that available.
3. Otherwise: you will probably need to build from source, this means providing an LLVM. If you have conda available you could use this to bootstrap the installation with a working `llvm/llvmdev` package. Learn more about compiling from source in the section on *Building manually* below. and in particular note the use of the `LLVM_CONFIG` environment variable for specifying where your LLVM install is.

What to be aware of when using a system provided LLVM package.

When using a system provided LLVM package, there are a number of things that could go wrong:

1. The LLVM package may not work with Numba/llvmlite at all.
2. If it does work to some degree it is unlikely the carry the correct patches for Numba/llvmlite to work entirely correctly.
3. Since the Numba/llvmlite maintainers may not know how the package was compiled it may be more difficult to get help when things do go wrong.

3.1.2 Building manually

Building llvmlite requires first building LLVM. Do not use prebuilt LLVM binaries from your OS distribution or the LLVM website! There will likely be a mismatch in version or build options, and LLVM will be missing certain patches that are critical for llvmlite operation.

Prerequisites

Before building, you must have the following:

- On Windows:
  - Visual Studio 2015 (Update 3) or later, to compile LLVM and llvmlite. The free Express edition is acceptable.
  - CMake installed.
- On Linux:
  - g++ (>= 4.8) and CMake
  - If building LLVM on Ubuntu, the linker may report an error if the development version of `libedit` is not installed. If you run into this problem, install `libedit-dev`.
- On Mac:
  - Xcode for the compiler tools, and CMake

Compiling LLVM

If you can build llvmlite inside a conda environment, you can install a prebuilt LLVM binary package and skip this step:

```
conda install -c numba llvmdev
```

The LLVM build process is fully scripted by conda-build, and the llvmdev recipe is the canonical reference for building LLVM for llvmlite. Please use it if at all possible!

The manual instructions below describe the main steps, but refer to the recipe for details:

1. Download the LLVM source code. You can download the complete "project" package, or llvm, lldd, and libunwind.
2. Download or git checkout the llvmlite source code.
3. Decompress the LLVM tar files and apply the appropriate patches from the `llvmlite/conda-recipes/` directory. You can apply each patch using the Linux `patch -p1 -i {patch-file}` command. Patches are prefixed with the LLVM version they apply cleanly to.
4. For Linux/macOS:
    1. `export PREFIX=desired_install_location CPU_COUNT=N` (`N` is number of parallel compile tasks)
    2. Run the build.sh script in the llvmdev conda recipe from the LLVM source directory.
5. For Windows:
    1. `set PREFIX=desired_install_location`
    2. Run the bld.bat script in the llvmdev conda recipe from the LLVM source directory.

Compiling llvmlite

1. To build the llvmlite C wrapper, which embeds a statically linked copy of the required subset of LLVM, run the following from the llvmlite source directory:

    ```
    python setup.py build
    ```

2. If your LLVM is installed in a nonstandard location, set the `LLVM_CONFIG` environment variable to the location of the corresponding `llvm-config` or `llvm-config.exe` executable. This variable must persist into the installation of llvmlite—for example, into a Python environment.

    EXAMPLE: If LLVM is installed in `/opt/llvm/` with the `llvm-config` binary located at `/opt/llvm/bin/llvm-config`, set `LLVM_CONFIG=/opt/llvm/bin/llvm-config`.

3. If you wish to build against an unsupported LLVM version, set the environment variable `LLVMLITE_SKIP_LLVM_VERSION_CHECK` to non-zero. Note that this is useful for e.g. testing new versions of llvmlite, but support for llvmlite built in this manner is limited/it's entirely possible that llvmlite will not work as expected. See also: *why llvmlite doesn't always support the latest release(s) of LLVM*.

Installing

1. To validate your build, run the test suite by running:

    ```
    python runtests.py
    ```

    or:

    ```
    python -m llvmlite.tests
    ```

2. If the validation is successful, install by running:

    ```
    python setup.py install
    ```

Installing from sdist

If you don't want to do any modifications to llvmlite itself, it's also possible to use `pip` to compile and install llvmlite from the latest released sdist package. You'll still need to point to your `llvm-config` if it's not in the `PATH`:

```
LLVM_CONFIG=/path/to/llvm-config pip3 install llvmlite
```

This should work on any platform that runs Python and llvm. It has been observed to work on `arm`, `ppc64le`, and also `pypy3` on `arm`.

x86 users will need to pass an extra flag (see issue #522):

```
LLVM_CONFIG=/path/to/llvm-config CXXFLAGS=-fPIC pip3 install llvmlite
```

This is known to work with `pypy3` on `Linux x64`.

It's also possible to force `pip` to rebuild `llvmlite` locally with a custom version of `llvm` :

```
LLVM_CONFIG=/path/to/custom/llvm-config CXXFLAGS=-fPIC pip3 install --no-binary :all:
llvmlite
```

3.2 User guide

3.2.1 IR layer—llvmlite.ir

The *llvmlite.ir* module contains classes and utilities to build the LLVM *intermediate representation* (IR) of native functions.

The provided APIs may sometimes look like LLVM's C++ APIs, but they never call into LLVM, unless otherwise noted. Instead, they construct a pure Python representation of the IR.

To use this module, you should be familiar with the concepts in the LLVM Language Reference.

Types

- *Atomic types*
- *Pointer Types*
- *Aggregate types*
- *Other types*

All *values* used in an LLVM module are explicitly typed. All types derive from a common base class *Type*. You can instantiate most of them directly. Once instantiated, a type should be considered immutable.

class llvmlite.ir.**Type**

The base class for all types. Never instantiate it directly. Types have the following methods in common:

- **as_pointer**(*addrspace=0*)

  Return a *PointerType* pointing to this type. The optional *addrspace* integer allows you to choose a non-default address space—the meaning is platform dependent.

- **get_abi_size**(*target_data*)

  Get the ABI size of this type, in bytes, according to the *target_data*—an *llvmlite.binding.TargetData* instance.

- **get_abi_alignment**(*target_data*)

  Get the ABI alignment of this type, in bytes, according to the *target_data*—an *llvmlite.binding.TargetData* instance.

- **get_element_offset**(*target_data*, *position*)

  Get the byte offset for the struct element at *position*, according to the *target_data*—an *llvmlite.binding.TargetData* instance.

  NOTE: *get_abi_size()*, *get_abi_alignment()*, and *get_element_offset()* call into the LLVM C++ API to get the requested information.

- **__call__**(*value*)

  A convenience method to create a *Constant* of this type with the given *value*:

```
>>> int32 = ir.IntType(32)
>>> c = int32(42)
>>> c
<ir.Constant type='i32' value=42>
>>> print(c)
i32 42
```

Atomic types

class llvmlite.ir.**IntType**(*bits*)

The type of integers. The Python integer *bits* specifies the bitwidth of the integers having this type.

width

The width in bits.

class llvmlite.ir.**HalfType**

The type of half-precision, floating-point, real numbers.

class llvmlite.ir.**FloatType**

The type of single-precision, floating-point, real numbers.

class llvmlite.ir.**DoubleType**

The type of double-precision, floating-point, real numbers.

class llvmlite.ir.**VoidType**

The class for void types. Used only as the return type of a function without a return value.

Pointer Types

llvmlite presently supports both *Typed Pointers* and *Opaque Pointers*. Typed Pointers are currently the default; Opaque Pointers will become the default in future, and support for Typed Pointers will be eventually be removed.

Note

Further details of the migration to Opaque Pointers are outlined in the section on *Deprecation of Typed Pointers*.

When Typed Pointers are enabled, the pointer type is represented using:

class llvmlite.ir.**PointerType**(*pointee*, *addrspace=0*)

The type of pointers to another type.

Pointer types expose the following attributes:

- **addrspace**

  The pointer's address space number. This optional integer allows you to choose a non-default address space—the meaning is platform dependent.

- **pointee**

  The type pointed to.

Opaque pointers can be enabled by setting the environment variable:

```
LLVMLITE_ENABLE_OPAQUE_POINTERS=1
```

or by setting the `opaque_pointers_enabled` attribute after importing llvmlite, but prior to using any of its functionality. For example:

```
import llvmlite
llvmlite.opaque_pointers_enabled = True

# ... continue using llvmlite ...
```

When Opaque Pointers are enabled, the pointer type is represented using:

class llvmlite.ir.**PointerType**(*addrspace=0*)

The type of pointers.

Pointer types expose the following attribute:

- **addrspace**

  The pointer's address space number. This optional integer allows you to choose a non-default address space—the meaning is platform dependent.

Aggregate types

class llvmlite.ir.**Aggregate**

The base class for aggregate types. Never instantiate it directly. Aggregate types have the elements attribute in common.

elements

A tuple-like immutable sequence of element types for this aggregate type.

class llvmlite.ir.**ArrayType**(*element*, *count*)

The class for array types.

- *element* is the type of every element.
- *count* is a Python integer representing the number of elements.

class llvmlite.ir.**VectorType**(*element*, *count*)

The class for vector types.

- *element* is the type of every element.
- *count* is a Python integer representing the number of elements.

class llvmlite.ir.**LiteralStructType**(*elements*[, *packed=False*])

The class for literal struct types.

- *elements* is a sequence of element types for each member of the structure.
- *packed* controls whether to use packed layout.

class llvmlite.ir.**IdentifiedStructType**

The class for identified struct types. Identified structs are compared by name. It can be used to make opaque types.

Users should not create new instance directly. Use the `Context.get_identified_type` method instead.

An identified struct is created without a body (thus opaque). To define the struct body, use the `.set_body` method.

set_body(**elems*)

Define the structure body with a sequence of element types.

Other types

class llvmlite.ir.**FunctionType**(*return_type*, *args*, *var_arg=False*)

The type of a function.

- *return_type* is the return type of the function.
- *args* is a sequence describing the types of argument to the function.
- If *var_arg* is `True`, the function takes a variable number of additional arguments of unknown types after the explicit args.

  EXAMPLE:

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

  An equivalent C declaration would be:

```
typedef int32_t (*fnty)(double, int32_t *);
```

class llvmlite.ir.**LabelType**

The type for *labels*. You do not need to instantiate this type.

class llvmlite.ir.**MetaDataType**

The type for *Metadata*. You do not need to instantiate this type.

NOTE: This class was previously called "MetaData," but it was renamed for clarity.

Values

- *Metadata*
- *Global values*
- *Instructions*
- *Landing pad clauses*

A *Module* consists mostly of values.

llvmlite.ir.**Undefined**

An undefined value, mapping to LLVM's `undef`.

class llvmlite.ir.**Value**

The base class for all IR values.

class llvmlite.ir.**_ConstOpMixin**

This is the base class for *Constant* and *GlobalValue*; do not instantiate it directly.

Integer arithmetic operations:

- **add**(*other*)
  Integer add *self* and *other*.
- **sub**(*other*)
  Integer subtract *other* from *self*.
- **mul**(*other*)
  Integer multiply *self* with *other*.
- **udiv**(*other*)
  Unsigned integer divide *self* by *other*.
- **sdiv(other)a**
  Signed integer divide *self* by *other*.
- **urem**(*other*)
  Unsigned integer remainder of *self* divided by *other*.
- **srem**(*other*)
  Signed integer remainder of *self* divided by *other*.
- **neg**()
  Negate *self*.

Integer logical operations:

- **shl**(*other*)
  Left-shift *self* by *other* bits.
- **ashr**(*other*)
  Arithmetic, signed, right-shift *self* by *other* bits.
- **lshr**(*other*)
  Logical right-shift *self* by *other* bits.
- **or_**(*other*)
  Bitwise OR *self* with *other*.
- **and_**(*other*)
  Bitwise AND *self* with *other*.
- **xor**(*other*)
  Bitwise XOR *self* with *other*.

Floating point arithmetic:

- **fadd**(*other*)
  Floating-point add *self* and *other*.
- **fsub**(*other*)
  Floating-point subtract *other* from *self*.

- **fmul**(*other*)

  Floating-point multiply *self* by *other*.

- **fdiv**(*other*)

  Floating-point divide *self* by *other*.

- **frem**(*other*)

  Floating-point remainder of *self* divided by *other*.

Comparisons:

- **icmp_signed**(*cmpop*, *other*)

  Signed integer compare *self* with *other*. The string *cmpop* can be one of <, <=, ==, !=, >= or >.

- **icmp_unsigned**(*cmpop*, *other*)

  Unsigned integer compare *self* with *other*. The string *cmpop* can be one of <, <=, ==, !=, >= or >.

- **fcmp_ordered**(*cmpop*, *other*)

  Floating-point ordered compare *self* with *other*. The string *cmpop* can be one of <, <=, ==, !=, >= or >.

- **fcmp_unordered**(*cmpop*, *other*)

  Floating-point unordered compare *self* with *other*. The string *cmpop* can be one of <, <=, ==, !=, >= or >.

Integer casts:

- **trunc**(*typ*)

  Truncate *self* to integer type *typ*.

- **zext**(*typ*)

  Zero-extend *self* to integer type *typ*.

- **sext**(*typ*)

  Sign-extend *self* to integer type *typ*.

- **bitcast**(*typ*)

  Convert this pointer constant to a constant of the given pointer type *typ*.

Floating-point casts:

- **fptrunc**(*typ*)

  Truncate (approximate) floating-point value *self* to floating-point type *typ*.

- **fpext**(*typ*)

  Extend floating-point value *self* to floating-point type *typ*.

Integer / floating-point conversion:

- **fptoui**(*typ*)

  Convert floating-point value *self* to unsigned integer type *typ*.

- **uitofp**(*typ*)

  Convert unsigned integer value *self* to floating-point type *typ*.

- **fptosi**(*typ*)

  Convert floating-point value *self* to signed integer type *typ*.

- **sitofp**(*typ*)

  Convert signed integer value *self* to floating-point type *typ*.

Integer / pointer conversions:

- **inttoptr**(*typ*)

  Convert this integer constant *self* to a constant of the given pointer type *typ*.

- **ptrtoint**(*typ*)

  Convert this pointer constant *self* to a constant of the given integer type *typ*.

class llvmlite.ir.**Constant**(*typ*, *constant*)

A literal value.

- *typ* is the type of the represented value—a *Type* instance.
- *constant* is the Python value to be represented.

Which Python types are allowed for *constant* depends on *typ*:

- All types accept *Undefined* and convert it to LLVM's `undef`.
- All types accept `None` and convert it to LLVM's `zeroinitializer`.
- *IntType* accepts any Python integer or boolean.
- *FloatType* and *DoubleType* accept any Python real number.
- Aggregate types—array and structure types—accept a sequence of Python values corresponding to the type's element types.
- *ArrayType* accepts a single `bytearray` instance to initialize the array from a string of bytes. This is useful for character constants.

See also *_ConstOpMixin*.

classmethod literal_array(*elements*)

An alternate constructor for constant arrays.

- *elements* is a sequence of values, *Constant* or otherwise.
- All *elements* must have the same type.
- Returns a constant array containing the *elements*, in order.

classmethod literal_struct(*elements*, *packed=False*)

An alternate constructor for constant structs.

- *elements* is a sequence of values, *Constant* or otherwise.
- *packed* controls whether to use packed layout.
- Returns a constant struct containing the *elements* in order.

gep(*indices*)

Compute the address of the inner element given by the sequence of *indices*. The constant must have a pointer type.

NOTE: You cannot define constant functions. Use a *Function declaration* instead.

class llvmlite.ir.**Argument**

One of a function's arguments. Arguments have the *add_attribute()* method.

add_attribute(*attr*)

Add an argument attribute to this argument. *attr* is a Python string.

class llvmlite.ir.**Block**

A *Basic block*. Do not instantiate or mutate this type directly. Instead, call the helper methods on *Function* and *IRBuilder*.

Basic blocks have the following methods and attributes:

- **replace**(*old*, *new*)

  Replace the instruction *old* with the other instruction *new* in this block's list of instructions. All uses of *old* in the whole function are also patched. *old* and *new* are *Instruction* objects.

- **function**

  The function this block is defined in.

- **is_terminated**

  Whether this block ends with a *terminator instruction*.

- **terminator**

  The block's *terminator instruction*, if any. Otherwise None.

class llvmlite.ir.**BlockAddress**

A constant representing an address of a basic block.

Block address constants have the following attributes:

- **function**

  The function in which the basic block is defined.

- **basic_block**

  The basic block. Must be a part of *function*.

Metadata

There are several kinds of *Metadata* values.

class llvmlite.ir.**MetaDataString**(*module*, *value*)

A string literal for use in metadata.

- *module* is the module that the metadata belongs to.
- *value* is a Python string.

class llvmlite.ir.**MDValue**

A metadata node. To create an instance, call *Module.add_metadata()*.

class llvmlite.ir.**DIValue**

A debug information descriptor, containing key-value pairs. To create an instance, call *Module.add_debug_info()*.

class llvmlite.ir.**DIToken**(*value*)

A debug information "token," representing a well-known enumeration value. *value* is the enumeration name.

EXAMPLE: 'DW_LANG_Python'

class llvmlite.ir.**NamedMetaData**

A named metadata node. To create an instance, call *Module.add_named_metadata()*. *NamedMetaData* has the *add()* method:

add(*md*)

Append the given piece of metadata to the collection of operands referred to by the *NamedMetaData*. *md* can be either a *MetaDataString* or a *MDValue*.

Global values

Global values are values accessible using a module-wide name.

class llvmlite.ir.**GlobalValue**

The base class for global values. Global values have the following writable attributes:

- **linkage**

  A Python string describing the linkage behavior of the global value—for example, whether it is visible from other modules. The default is the empty string, meaning "external."

- **storage_class**

  A Python string describing the storage class of the global value.
  - The default is the empty string, meaning "default."
  - Other possible values include `dllimport` and `dllexport`.

- **section**

  A Python string describing the section a global value should be in after translation. The default is the empty string, meaning no specific section.

See also *_ConstOpMixin*.

class llvmlite.ir.**GlobalVariable**(*module*, *typ*, *name*, *addrspace=0*)

A global variable.

- *module* is where the variable is defined.
- *typ* is the variable's type. It cannot be a function type. To declare a global function, use *Function*.

  The type of the returned Value is a pointer to *typ*. To read the contents of the variable, you need to *load()* from the returned Value. To write to the variable, you need to *store()* to the returned Value.

- *name* is the variable's name—a Python string.
- *addrspace* is an optional address space to store the variable in.

Global variables have the following writable attributes:

- **set_metadata**(*name*, *node*)

  Add metadata with the given *name*, pointing to the given metadata *node*—an instance of *MDValue*.

- **global_constant**
  - If `True`, the variable is declared a constant, meaning that its contents cannot be modified.
  - The default is `False`.

- **unnamed_addr**
  - If `True`, the address of the variable is deemed insignificant, meaning that it is merged with other variables that have the same initializer.
  - The default is `False`.

- **initializer**

  The variable's initialization value—probably a *Constant* of type *typ*. The default is `None`, meaning that the variable is uninitialized.

- **align**

  An explicit alignment in bytes. The default is None, meaning that the default alignment for the variable's type is used.

class llvmlite.ir.**Function**(*module*, *typ*, *name*)

A global function.

- *module* is where the function is defined.
- *typ* is the function's type—a *FunctionType* instance.
- *name* is the function's name—a Python string.

If a global function has any basic blocks, it is a *Function definition*. Otherwise, it is a *Function declaration*.

Functions have the following methods and attributes:

- **append_basic_block**(*name=''*)

  Append a *Basic block* to this function's body.
  - If *name* is not empty, it names the block's entry *Label*.
  - Returns a new *Block*.

- **insert_basic_block**(*before*, *name=''*)

  Similar to *append_basic_block()*, but inserts it before the basic block *before* in the function's list of basic blocks.

- **set_metadata**(*name*, *node*)

  Add a function-specific metadata named *name* pointing to the given metadata *node*—an *MDValue*.

- **args**

  The function's arguments as a tuple of *Argument* instances.

- **attributes**

  A set of function attributes. Each optional attribute is a Python string. By default this is empty. Use the *.add()* method to add an attribute:

  ```
  fnty = ir.FunctionType(ir.DoubleType(), (ir.DoubleType(),))
  fn = Function(module, fnty, "sqrt")
  fn.attributes.add("alwaysinline")
  ```

- **calling_convention**

  The function's calling convention—a Python string. The default is the empty string.

- **is_declaration**

  Indicates whether the global function is a declaration or a definition.
  - If True, it is a declaration.
  - If False, it is a definition.

Instructions

Every *Instruction* is also a value:

- It has a name—the recipient's name.
- It has a well-defined type.
- It can be used as an operand in other instructions or in literals.

Usually, you should not instantiate instruction types directly. Use the helper methods on the *IRBuilder* class.

class llvmlite.ir.**Instruction**

The base class for all instructions. Instructions have the following method and attributes:

- **set_metadata**(*name*, *node*)

  Add an instruction-specific metadata *name* pointing to the given metadata *node*—an *MDValue*.

- **replace_usage**(*old*, *new*)

  Replace the operand *old* with the other instruction *new*.

- **function**

  The function that contains this instruction.

- **module**

  The module that defines this instruction's function.

class llvmlite.ir.**PredictableInstr**

The class of instructions for which we can specify the probabilities of different outcomes—for example, a switch or a conditional branch. Predictable instructions have the *set_weights()* method.

set_weights(*weights*)

Set the weights of the instruction's possible outcomes. *weights* is a sequence of positive integers, each corresponding to a different outcome and specifying its relative probability compared to other outcomes. The greater the number, the likelier the outcome.

class llvmlite.ir.**SwitchInstr**

A switch instruction. Switch instructions have the *add_case()* method.

add_case(*val*, *block*)

Add a case to the switch instruction.

- *val* should be a *Constant* or a Python value compatible with the switch instruction's operand type.
- *block* is a *Block* to jump to if val and the switch operand compare equal.

class llvmlite.ir.**IndirectBranch**

An indirect branch instruction. Indirect branch instructions have the *add_destination()* method.

add_destination(*value*, *block*)

Add an outgoing edge. The indirect branch instruction must refer to every basic block it can transfer control to.

class llvmlite.ir.**PhiInstr**

A phi instruction. Phi instructions have the *add_incoming()* method.

add_incoming(*value*, *block*)

Add an incoming edge. Whenever transfer is controlled from *block*—a *Block*—the phi instruction takes the given *value*.

class llvmlite.ir.**LandingPad**

A landing pad. Landing pads have the *add_clause()* method:

add_clause(*value*, *block*)

Add a catch or filter clause. Create catch clauses using *CatchClause* and filter clauses using *FilterClause*.

Landing pad clauses

Landing pads have the following classes associated with them.

class llvmlite.ir.**CatchClause**(*value*)

A catch clause. Instructs the personality function to compare the in-flight exception typeinfo with *value*, which should have type *IntType(8).as_pointer().as_pointer()*.

class llvmlite.ir.**FilterClause**(*value*)

A filter clause. Instructs the personality function to check inclusion of the the in-flight exception typeinfo in *value*, which should have type *ArrayType(IntType(8).as_pointer().as_pointer(), ...)*.

Modules

A module is a compilation unit. It defines a set of related functions, global variables and metadata. In the IR layer, a module is represented by the *Module* class.

class llvmlite.ir.**Module**(*name=''*)

Create a module. For informational purposes, you can specify the optional *name*, a Python string.

Modules have the following methods and attributes:

- **add_debug_info**(*kind*, *operands*, *is_distinct=False*)

  Add debug information metadata to the module with the given *operands*—a mapping of string keys to values—or return a previous equivalent metadata. *kind* is the name of the debug information kind.

  EXAMPLE: `'DICompileUnit'`

  A *DIValue* instance is returned. You can then associate it to, for example, an instruction.

  EXAMPLE:

```
di_file = module.add_debug_info("DIFile", {
     "filename": "factorial.py",
     "directory": "bar",
})
di_compile_unit = module.add_debug_info("DICompileUnit", {
     "language": ir.DIToken("DW_LANG_Python"),
     "file": di_file,
     "producer": "llvmlite x.y",
     "runtimeVersion": 2,
     "isOptimized": False,
}, is_distinct=True)
```

- **add_global**(*globalvalue*)

  Add the given *globalvalue*—a *GlobalValue*—to this module. It should have a unique name in the whole module.

- **add_metadata**(*operands*)

  Add an unnamed *Metadata* node to the module with the given *operands*—a list of metadata-compatible values. If another metadata node with equal operands already exists in the module, it is reused instead. Returns an *MDValue*.

- **add_named_metadata**(*name*, *element=None*)

  Return the metadata node with the given *name*. If it does not already exist, the named metadata node is created first. If *element* is not `None`, it can be a metadata value or a sequence of values to append to the metadata node's elements. Returns a *NamedMetaData*.

  EXAMPLE:

```
module.add_named_metadata("llvm.ident", ["llvmlite/1.0"])
```

- **get_global**(*name*)

  Get the *Global value*—a *GlobalValue*—with the given name. KeyError is raised if the name does not exist.

- **get_named_metadata**(*name*)

  Return the metadata node with the given *name*. KeyError is raised if the name does not exist.

- **get_unique_name**(*name*)

  Return a unique name across the whole module. *name* is the desired name, but a variation can be returned if it is already in use.

- **data_layout**

  A string representing the data layout in LLVM format.

- **functions**

  The list of functions, as *Function* instances, declared or defined in the module.

- **global_values**

  An iterable of global values in this module.

- **triple**

  A string representing the target architecture in LLVM "triple" form.

IR builders

- *Instantiation*
- *Attributes*
- *Utilities*
- *Positioning*
- *Flow control helpers*
- *Instruction building*
  - *Arithmetic*
  - *Conversions*
  - *Comparisons*
  - *Conditional move*
  - *Phi*
  - *Aggregate operations*
  - *Vector operations*
  - *Memory*
  - *Function call*
  - *Branches*
  - *Exception handling*

- *Inline assembler*
- *Miscellaneous*

IRBuilder is the workhorse of LLVM *Intermediate representation (IR)* generation. It allows you to fill the *basic blocks* of your functions with LLVM instructions.

An *IRBuilder* internally maintains a current basic block and a pointer inside the block's list of instructions. When a new instruction is added, it is inserted at that point, and then the pointer is advanced after the new instruction.

A *IRBuilder* also maintains a reference to metadata describing the current source location, which is attached to all inserted instructions.

Instantiation

class llvmlite.ir.**IRBuilder**(*block=None*)

Create a new IR builder. If *block*—a *Block*—is given, the builder starts at the end of this basic block.

Attributes

IRBuilder has the following attributes:

- IRBuilder.**block**

  The basic block that the builder is operating on.

- IRBuilder.**function**

  The function that the builder is operating on.

- IRBuilder.**module**

  The module that the builder's function is defined in.

- IRBuilder.**debug_metadata**

  If not None, the metadata that is attached to any inserted instructions as !dbg, unless the instruction already has !dbg set.

Utilities

IRBuilder.**append_basic_block**(*name=''*)

Append a basic block, with the given optional *name*, to the current function. The current block is not changed. A *Block* is returned.

Positioning

The following *IRBuilder* methods help you move the current instruction pointer:

- IRBuilder.**position_before**(*instruction*)

  Position immediately before the given *instruction*. The current block is also changed to the instruction's basic block.

- IRBuilder.**position_after**(*instruction*)

  Position immediately after the given *instruction*. The current block is also changed to the instruction's basic block.

- IRBuilder.**position_at_start**(*block*)

  Position at the start of the basic *block*.

- IRBuilder.**position_at_end**(*block*)

  Position at the end of the basic *block*.

The following context managers allow you to temporarily switch to another basic block and then go back to where you were.

- IRBuilder.**goto_block**(*block*)

  Position the builder either at the end of the basic *block*, if it is not terminated, or just before the *block*'s terminator:

  ```
  new_block = builder.append_basic_block('foo')
  with builder.goto_block(new_block):
     # Now the builder is at the end of *new_block*
     # ... add instructions

  # Now the builder has returned to its previous position
  ```

- IRBuilder.**goto_entry_block**()

  The same as *goto_block()*, but with the current function's entry block.

Flow control helpers

The following context managers make it easier to create conditional code.

- IRBuilder.**if_then**(*pred*, *likely=None*)

  Create a basic block whose execution is conditioned on predicate *pred*, a value of type `IntType(1)`. Another basic block is created for instructions after the conditional block. The current basic block is terminated with a conditional branch based on *pred*.

  When the context manager is entered, the builder positions at the end of the conditional block. When the context manager is exited, the builder positions at the start of the continuation block.

  If likely is not `None`, it indicates whether *pred* is likely to be `True`, and metadata is emitted to specify branch weights accordingly.

- IRBuilder.**if_else**(*pred*, *likely=None*)

  Set up 2 basic blocks whose execution is conditioned on predicate *pred*, a value of type `IntType(1)`. *likely* has the same meaning as in `if_then()`.

  A pair of context managers is yielded. Each of them acts as an *if_then()* context manager—the first for the block to be executed if pred is `True` and the second for the block to be executed if pred is `False`.

  When the context manager is exited, the builder is positioned on a new continuation block that both conditional blocks jump into.

  Typical use:

  ```
  with builder.if_else(pred) as (then, otherwise):
      with then:
          # emit instructions for when the predicate is true
      with otherwise:
          # emit instructions for when the predicate is false
  # emit instructions following the if-else block
  ```

Instruction building

The following methods insert a new instruction—an *Instruction* instance—at the current index in the current block. The new instruction is returned.

An instruction's operands are almost always *values*.

Many of these methods also take an optional *name* argument, specifying the local *name* of the result value. If not given, a unique name is automatically generated.

Arithmetic

In the methods below, the *flags* argument is an optional sequence of strings that modify the instruction's semantics. Examples include the fast-math flags for floating-point operations, and whether wraparound on overflow can be ignored on integer operations.

Integer

- `IRBuilder.`**`shl`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Left-shift *lhs* by *rhs* bits.
- `IRBuilder.`**`lshr`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Logical right-shift *lhs* by *rhs* bits.
- `IRBuilder.`**`ashr`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Arithmetic, signed, right-shift *lhs* by *rhs* bits.
- `IRBuilder.`**`cttz`**(*value*, *flag*)
  Counts trailing zero bits in *value*. Boolean *flag* indicates whether the result is defined for `0`.
- `IRBuilder.`**`ctlz`**(*value*, *flag*)
  Counts leading zero bits in *value*. Boolean *flag* indicates whether the result is defined for `0`.
- `IRBuilder.`**`add`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Integer add *lhs* and *rhs*.
- `IRBuilder.`**`sadd_with_overflow`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Integer add *lhs* and *rhs*. A `{ result, overflow bit }` structure is returned.
- `IRBuilder.`**`sub`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Integer subtract *rhs* from *lhs*.
- `IRBuilder.`**`ssub_with_overflow`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Integer subtract *rhs* from *lhs*. A `{ result, overflow bit }` structure is returned.
- `IRBuilder.`**`mul`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Integer multiply *lhs* with *rhs*.
- `IRBuilder.`**`smul_with_overflow`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Integer multiply *lhs* with *rhs*. A `{ result, overflow bit }` structure is returned.
- `IRBuilder.`**`sdiv`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Signed integer divide *lhs* by *rhs*.
- `IRBuilder.`**`udiv`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Unsigned integer divide *lhs* by *rhs*.

- IRBuilder.**srem**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Signed integer remainder of *lhs* divided by *rhs*.
- IRBuilder.**urem**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Unsigned integer remainder of *lhs* divided by *rhs*.
- IRBuilder.**and_**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Bitwise AND *lhs* with *rhs*.
- IRBuilder.**or_**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Bitwise OR *lhs* with *rhs*.
- IRBuilder.**xor**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Bitwise XOR *lhs* with *rhs*.
- IRBuilder.**not_**(*value*, *name=''*)
  Bitwise complement *value*.
- IRBuilder.**neg**(*value*, *name=''*)
  Negate *value*.

Floating-point

- IRBuilder.**fadd**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Floating-point add *lhs* and *rhs*.
- IRBuilder.**fsub**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Floating-point subtract *rhs* from *lhs*.
- IRBuilder.**fmul**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Floating-point multiply *lhs* by *rhs*.
- IRBuilder.**fdiv**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Floating-point divide *lhs* by *rhs*.
- IRBuilder.**frem**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Floating-point remainder of *lhs* divided by *rhs*.
- IRBuilder.**fneg**(*arg*, *name=''*, *flags=()*)
  Floating-point negation of *arg*.

Conversions

- IRBuilder.**trunc**(*value*, *typ*, *name=''*)
  Truncate integer *value* to integer type *typ*.
- IRBuilder.**zext**(*value*, *typ*, *name=''*)
  Zero-extend integer *value* to integer type *typ*.
- IRBuilder.**sext**(*value*, *typ*, *name=''*)
  Sign-extend integer *value* to integer type *typ*.
- IRBuilder.**fptrunc**(*value*, *typ*, *name=''*)
  Truncate—approximate—floating-point *value* to floating-point type *typ*.
- IRBuilder.**fpext**(*value*, *typ*, *name=''*)
  Extend floating-point *value* to floating-point type *typ*.

- IRBuilder.**fptosi**(*value*, *typ*, *name=''*)
  Convert floating-point *value* to signed integer type *typ*.
- IRBuilder.**fptoui**(*value*, *typ*, *name=''*)
  Convert floating-point *value* to unsigned integer type *typ*.
- IRBuilder.**sitofp**(*value*, *typ*, *name=''*)
  Convert signed integer *value* to floating-point type *typ*.
- IRBuilder.**uitofp**(*value*, *typ*, *name=''*)
  Convert unsigned integer *value* to floating-point type *typ*.
- IRBuilder.**ptrtoint**(*value*, *typ*, *name=''*)
  Convert pointer *value* to integer type *typ*.
- IRBuilder.**inttoptr**(*value*, *typ*, *name=''*)
  Convert integer *value* to pointer type *typ*.
- IRBuilder.**bitcast**(*value*, *typ*, *name=''*)
  Convert pointer *value* to pointer type *typ*.
- IRBuilder.**addrspacecast**(*value*, *typ*, *name=''*)
  Convert pointer *value* to pointer type *typ* of different address space.

Comparisons

- IRBuilder.**icmp_signed**(*cmpop*, *lhs*, *rhs*, *name=''*)
  Signed integer compare *lhs* with *rhs*. The string *cmpop* can be one of <, <=, ==, !=, >= or >.
- IRBuilder.**icmp_unsigned**(*cmpop*, *lhs*, *rhs*, *name=''*)
  Unsigned integer compare *lhs* with *rhs*. The string *cmpop* can be one of <, <=, ==, !=, >= or >.
- IRBuilder.**fcmp_ordered**(*cmpop*, *lhs*, *rhs*, *name=''*, *flags=[]*)
  Floating-point ordered compare *lhs* with *rhs*.
  - The string *cmpop* can be one of <, <=, ==, !=, >=, >, `ord` or `uno`.
  - The *flags* list can include any of `nnan`, `ninf`, `nsz`, `arcp` and `fast`, which implies all previous flags.
- IRBuilder.**fcmp_unordered**(*cmpop*, *lhs*, *rhs*, *name=''*, *flags=[]*)
  Floating-point unordered compare *lhs* with *rhs*.
  - The string *cmpop*, can be one of <, <=, ==, !=, >=, >, `ord` or `uno`.
  - The *flags* list can include any of `nnan`, `ninf`, `nsz`, `arcp` and `fast`, which implies all previous flags.

Conditional move

IRBuilder.**select**(*cond*, *lhs*, *rhs*, *name=''*)
A 2-way select—*lhs* if *cond*, else *rhs*.

Phi

IRBuilder.**phi**(*typ*, *name=''*)
Create a phi node. To add incoming edges and their values, use the *add_incoming()* method on the return value.

Aggregate operations

- IRBuilder.**extract_value**(*agg*, *index*, *name=''*)

  Extract the element at *index* of the *aggregate value* *agg*.

  - *index* may be an integer or a sequence of integers.
  - Indices must be constant.

- IRBuilder.**insert_value**(*agg*, *value*, *index*, *name=''*)

  Build a copy of *aggregate value* *agg* by setting the new *value* at *index*. The value for *index* can be of the same types as in `extract_value()`.

Vector operations

- IRBuilder.**extract_element**(*vector*, *idx*, *name=''*)

  Returns the *value* at position *idx*.

- IRBuilder.**insert_element**(*vector*, *value*, *idx*, *name=''*)

  Returns vector with `vector[idx]` replaced by `value`. The result is undefined if the idx is larger or equal the vector length.

- IRBuilder.**shuffle_vector**(*vector1*, *vector2*, *mask*, *name=''*)

  Constructs a permutation of elements from *vector1* and *vector2*. Returns a new vector in the same length of *mask*.

  - *vector1* and *vector2* must have the same element type.
  - *mask* must be a constant vector of integer types.

Memory

- IRBuilder.**alloca**(*typ*, *size=None*, *name=''*)

  Statically allocate a stack slot for *size* values of type *typ*. If *size* is not given, a stack slot for 1 value is allocated.

- IRBuilder.**load**(*ptr*, *name=''*, *align=None*)

  Load value from pointer *ptr*. If *align* is passed, it should be a Python integer specifying the guaranteed pointer alignment.

- IRBuilder.**store**(*value*, *ptr*, *align=None*)

  Store *value* to pointer *ptr*. If *align* is passed, it should be a Python integer specifying the guaranteed pointer alignment.

- IRBuilder.**load_atomic**(*ptr*, *ordering*, *align*, *name=''*)

  Load value from pointer *ptr* as an atomic operation with the given *ordering*. *align* must be a Python integer specifying the guaranteed pointer alignment.

- IRBuilder.**store_atomic**(*value*, *ptr*, *ordering*, *align*)

  Store *value* to pointer *ptr* as an atomic operation with the given *ordering*. *align* must be a Python integer specifying the guaranteed pointer alignment.

- IRBuilder.**gep**(*ptr*, *indices*, *inbounds=False*, *name=''*)

  The *getelementptr* instruction. Given a pointer *ptr* to an aggregate value, compute the address of the inner element given by the sequence of *indices*.

- `llvmlite.ir.cmpxchg`(*ptr*, *cmp*, *val*, *ordering*, *failordering=None*, *name=''*)

  Atomic compare-and-swap at address *ptr*.

  - *cmp* is the value to compare the contents with.
  - *val* is the new value to be swapped into.
  - Optional *ordering* and *failordering* specify the memory model for this instruction.

- `llvmlite.ir.atomic_rmw`(*op*, *ptr*, *val*, *ordering*, *name=''*)

  Atomic in-memory operation *op* at address *ptr*, with operand *val*.

  - The string *op* specifies the operation—for example, `add` or `sub`.
  - The optional *ordering* specifies the memory model for this instruction.

Function call

`IRBuilder.call`(*fn*, *args*, *name=''*, *cconv=None*, *tail=None*, *fastmath=()*, *attrs=()*, *arg_attrs=None*)

Call function *fn* with arguments *args*, a sequence of values.

- *cconv* is the optional calling convention.
- *tail* controls tail-call optimization behavior. It may be one of:
  - `None` (the default): indicates no specific tail-call optimization behavior.
  - `"tail"`: a hint that indicates that the call should be tail-call optimized, but may be ignored.
  - `"musttail"`: indicates that the call must be tail-call optimized for program correctness.
  - `"notail"`: indicates that the call should never be tail-call optimized.

  For backwards compatibility with previous versions, the following values are also accepted:

  - `False` is equivalent to `None`, indicating no specific behavior.
  - `True` is equivalent to `"tail"`, suggesting tail-call optimization.
- *fastmath* is a string or a sequence of strings of names for fast-math flags.
- *attrs* is a string or sequence of strings of function attributes to attach to the call site.
- *arg_attrs* is a dictionary matching argument indices (as regular integers, starting at zero) to strings or sequences of strings giving the attributes to attach to the respective argument at this call site. If an index is not present in the dictionary, or *arg_attrs* is missing entirely, no attributes are emitted for the given argument.

  If some attributes, such as `sret`, are specified at the function declaration, they must also be specified at each call site for correctness. (As of LLVM 11, this does not seem to be explicitly specified in the LLVM language reference.)

Branches

The following methods are all *terminators*:

- `IRBuilder.branch`(*target*)

  Unconditional jump to the *target*, a `Block`.

- `IRBuilder.cbranch`(*cond*, *truebr*, *falsebr*)

  Conditional jump to either *truebr* or *falsebr*—both `Block` instances—depending on *cond*, a value of type `IntType(1)`. This instruction is a `PredictableInstr`.

- IRBuilder.**ret**(*value*)

  Return the *value* from the current function.

- IRBuilder.**ret_void**()

  Return from the current function without a value.

- IRBuilder.**switch**(*value*, *default*)

  Switch to different blocks based on the *value*. *default* is the block to switch to if no other block is matched.

  To add non-default targets, use the `add_case()` method on the return value.

- IRBuilder.**branch_indirect**(*address*)

  Jump to the basic block with the address *address*, a value of type *IntType(8).as_pointer()*.

  To obtain a block address, use the `BlockAddress` constant.

  To add all possible jump destinations, use the `add_destination()` method on the return value.

Exception handling

- IRBuilder.**invoke**(*fn*, *args*, *normal_to*, *unwind_to*, *name=''*, *cconv=None*, *fastmath=()*, *attrs=()*, *arg_attrs=None*)

  Call function *fn* with arguments *args*, a sequence of values.

  If the function *fn* returns normally, control is transferred to *normal_to*. Otherwise, it is transferred to *unwind_to*, whose first non-phi instruction must be `LandingPad`.

  The remaining arguments give additional attributes to specify at the call site; see `call()` for a description.

- IRBuilder.**landingpad**(*typ*, *personality*, *name=''*, *cleanup=False*)

  Describe which exceptions this basic block can handle.

  - *typ* specifies the return type of the landing pad. It is a structure with 2 pointer-sized fields.
  - *personality* specifies an exception personality function.
  - *cleanup* specifies whether control should always be transferred to this landing pad, even when no matching exception is caught.

  To add landing pad clauses, use the `add_clause()` method on the return value.

  There are 2 kinds of landing pad clauses:

  - A `CatchClause`, which specifies a typeinfo for a single exception to be caught. The typeinfo is a value of type *IntType(8).as_pointer().as_pointer()*;
  - A `FilterClause`, which specifies an array of typeinfos.

  Every landing pad must either contain at least 1 clause or be marked for cleanup.

  The semantics of a landing pad are entirely determined by the personality function. For details on the way LLVM handles landing pads in the optimizer, see Exception handling in LLVM. For details on the implementation of personality functions, see Itanium exception handling ABI.

- IRBuilder.**resume**(*landingpad*)

  Resume an exception caught by *landingpad*. Used to indicate that the landing pad did not catch the exception after all, perhaps because it only performed cleanup.

Inline assembler

- IRBuilder.**asm**(*ftype*, *asm*, *constraint*, *args*, *side_effect*, *name=''*)

  Add an inline assembler call instruction. For example, this is used in *load_reg()* and *store_reg()*.

  Arguments:

  - *ftype* is a function type specifying the inputs and output of the inline assembler call.
  - *asm* is the inline assembler snippet—for example, `"mov $2, $0\nadd $1, $0"`. x86 inline ASM uses the AT&T syntax.
  - *constraint* defines the input/output constraints—for example `=r,r,r`.
  - *args* is the list of inputs, as IR values.
  - *side_effect* is a boolean that specifies whether or not this instruction has side effects not visible in the constraint list.
  - *name* is the optional name of the returned LLVM value.

  For more information about these parameters, see the official LLVM documentation.

  EXAMPLE: Adding 2 64-bit values on x86:

```
fty = FunctionType(IntType(64), [IntType(64),IntType(64)])
add = builder.asm(fty, "mov $2, $0\nadd $1, $0", "=r,r,r",
                  (arg_0, arg_1), True, name="asm_add")
```

- IRBuilder.**load_reg**(*reg_type*, *reg_name*, *name=''*)

  Load a register value into an LLVM value.

  EXAMPLE: Obtaining the value of the `rax` register:

```
builder.load_reg(IntType(64), "rax")
```

- IRBuilder.**store_reg**(*value*, *reg_type*, *reg_name*, *name=''*)

  Store an LLVM value inside a register.

  EXAMPLE: Storing `0xAAAAAAAAAAAAAAAA` into the `rax` register:

```
builder.store_reg(Constant(IntType(64), 0xAAAAAAAAAAAAAAAA), IntType(64), "rax")
```

Miscellaneous

- IRBuilder.**assume**(*cond*)

  Let the LLVM optimizer assume that *cond*—a value of type `IntType(1)`—is `True`.

- IRBuilder.**unreachable**()

  Mark an unreachable point in the code.

- IRBuilder.**comment**(*text*)

  Puts a single-line comment into the generated IR. This will be ignored by LLVM, but can be useful for debugging the output of a compiler.

  Arguments:

  - *text* is a string that does not contain new line characters.

Example—defining a simple function

This example defines a function that adds 2 double-precision, floating-point numbers.

```
"""
This file demonstrates a trivial function "fpadd" returning the sum of
two floating-point numbers.
"""

from llvmlite import ir

# Create some useful types
double = ir.DoubleType()
fnty = ir.FunctionType(double, (double, double))

# Create an empty module...
module = ir.Module(name=__file__)
# and declare a function named "fpadd" inside it
func = ir.Function(module, fnty, name="fpadd")

# Now implement the function
block = func.append_basic_block(name="entry")
builder = ir.IRBuilder(block)
a, b = func.args
result = builder.fadd(a, b, name="res")
builder.ret(result)

# Print the module IR
print(module)
```

The generated LLVM *intermediate representation* is printed at the end:

```
; ModuleID = "examples/ir_fpadd.py"
target triple = "unknown-unknown-unknown"
target datalayout = ""

define double @"fpadd"(double %".1", double %".2")
{
entry:
  %"res" = fadd double %".1", %".2"
  ret double %"res"
}
```

To learn how to compile and execute this function, see *LLVM binding layer—llvmlite.binding*.

3.2.2 LLVM binding layer—llvmlite.binding

The *llvmlite.binding* module provides classes to interact with functionalities of the LLVM library. Generally, they closely mirror concepts of the C++ API. Only a small subset of the LLVM API is mirrored: those parts that have proven useful to implement Numba's JIT compiler.

Initialization and finalization

You only need to call these functions once per process invocation.

- `llvmlite.binding.`**`initialize`**`()`

  Initialize the LLVM core.

- `llvmlite.binding.`**`initialize_all_targets`**`()`

  Initialize all targets. Must be called before targets can be looked up via the *`Target`* class.

- `llvmlite.binding.`**`initialize_all_asmprinters`**`()`

  Initialize all code generators. Must be called before generating any assembly or machine code via the *`TargetMachine.emit_object()`* and *`TargetMachine.emit_assembly()`* methods.

- `llvmlite.binding.`**`initialize_native_target`**`()`

  Initialize the native—host—target. Must be called once before doing any code generation.

- `llvmlite.binding.`**`initialize_native_asmprinter`**`()`

  Initialize the native assembly printer.

- `llvmlite.binding.`**`initialize_native_asmparser`**`()`

  Initialize the native assembly parser. Must be called for inline assembly to work.

- `llvmlite.binding.`**`shutdown`**`()`

  Shut down the LLVM core.

- `llvmlite.binding.`**`llvm_version_info`**

  A 3-integer tuple representing the LLVM version number.

  EXAMPLE: `(3, 7, 1)`

  Since LLVM is statically linked into the `llvmlite` DLL, this is guaranteed to represent the true LLVM version in use.

Dynamic libraries and symbols

These functions tell LLVM how to resolve external symbols referred from compiled LLVM code.

- `llvmlite.binding.`**`add_symbol`**`(`*name*, *address*`)`

  Register the *address* of global symbol *name*, for use from LLVM-compiled functions.

- `llvmlite.binding.`**`address_of_symbol`**`(`*name*`)`

  Get the in-process address of symbol *name*. An integer is returned, or `None` if the symbol is not found.

- `llvmlite.binding.`**`load_library_permanently`**`(`*filename*`)`

  Load an external shared library. *filename* is the path to the shared library file.

Target information

Target information allows you to inspect and modify aspects of the code generation, such as which CPU is targeted or what optimization level is desired.

Minimal use of this module would be to create a *`TargetMachine`* for later use in code generation.

EXAMPLE:

```
from llvmlite import binding
target = binding.Target.from_default_triple()
target_machine = target.create_target_machine()
```

Functions

- llvmlite.binding.**get_default_triple**()

  Return a string representing the default target triple that LLVM is configured to produce code for. This represents the host's architecture and platform.

- llvmlite.binding.**get_process_triple**()

  Return a target triple suitable for generating code for the current process.

  EXAMPLE: The default triple from `get_default_triple()` is not suitable when LLVM is compiled for 32-bit, but the process is executing in 64-bit mode.

- llvmlite.binding.**get_object_format**(*triple=None*)

  Get the object format for the given *triple* string, or the default triple if `None`. Returns a string such as `"ELF"`, `"COFF"` or `"MachO"`.

- llvmlite.binding.**get_host_cpu_name**()

  Get the name of the host's CPU as a string. You can use the return value with *Target.create_target_machine()*.

- llvmlite.binding.**get_host_cpu_features**()

  Return a dictionary-like object indicating the CPU features for the current architecture and whether they are enabled for this CPU.

  The key-value pairs contain the feature name as a string and a boolean indicating whether the feature is available.

  The returned value is an instance of the `FeatureMap` class, which adds a new method `.flatten()` for returning a string suitable for use as the `features` argument to *Target.create_target_machine()*.

  If LLVM has not implemented this feature or it fails to get the information, a `RuntimeError` exception is raised.

- llvmlite.binding.**create_target_data**(*data_layout*)

  Create a *TargetData* representing the given *data_layout* string.

Classes

class llvmlite.binding.**TargetData**

Provides functionality around a given data layout. It specifies how the different types are to be represented in memory. Use *create_target_data()* to instantiate.

- **get_abi_size**(*type*)

  Get the ABI-mandated size of a *TypeRef* object. Returns an integer.

- **get_abi_alignment**(*type*)

  Similar to *get_abi_size()*, but returns the ABI-mandated alignment rather that the ABI size.

- **get_pointee_abi_size**(*type*)

  Similar to *get_abi_size()*, but assumes that *type* is an LLVM pointer type and returns the ABI-mandated size of the type pointed to. This is useful for a global variable, whose type is really a pointer to the declared type.

- **get_pointee_abi_alignment**(*type*)

  Similar to *get_pointee_abi_size()*, but returns the ABI-mandated alignment rather than the ABI size.

- **get_element_offset**(*type*, *position*)

  Computes the byte offset of the struct element at position.

class llvmlite.binding.**Target**

Represents a compilation target. The following factories are provided:

- **classmethod from_triple**(*triple*)

  Create a new *Target* instance for the given *triple* string denoting the target platform.

- **classmethod from_default_triple**()

  Create a new *Target* instance for the default platform that LLVM is configured to produce code for. This is equivalent to calling `Target.from_triple(get_default_triple())`.

The following attributes and methods are available:

- **description**

  A description of the target.

- **name**

  The name of the target.

- **triple**

  A string that uniquely identifies the target.

  EXAMPLE: `"x86_64-pc-linux-gnu"`

- **create_target_machine**(*cpu=''*, *features=''*, *opt=2*, *reloc='default'*, *codemodel='jitdefault'*, *abiname=''*)

  Create a new *TargetMachine* instance for this target and with the given options:
  - *cpu* is an optional CPU name to specialize for.
  - *features* is a comma-separated list of target-specific features to enable or disable.
  - *opt* is the optimization level, from 0 to 3.
  - *reloc* is the relocation model.
  - *codemodel* is the code model.
  - *abiname* is the name of the ABI.

  The defaults for reloc and codemodel are appropriate for JIT compilation.

  TIP: To list the available CPUs and features for a target, run the command `llc -mcpu=help`.

class llvmlite.binding.**TargetMachine**

Holds all the settings necessary for proper code generation, including target information and compiler options. Instantiate using *Target.create_target_machine()*.

- **add_analysis_passes**(*pm*)

  Register analysis passes for this target machine with the *PassManager* instance *pm*.

- **emit_object**(*module*)

  Represent the compiled *module*—a *ModuleRef* instance—as a code object that is suitable for use with the platform's linker. Returns a bytestring.

- **set_asm_verbosity**(*is_verbose*)

  Set whether this target machine emits assembly with human-readable comments, such as those describing control flow or debug information.

- **emit_assembly**(*module*)

  Return a string representing the compiled *module*'s native assembler. You must first call *initialize_native_asmprinter()*.

- **target_data**

  The *TargetData* associated with this target machine.

class llvmlite.binding.**FeatureMap**

Stores processor feature information in a dictionary-like object. This class extends `dict` and adds only the `.flatten()` method.

flatten(*sort=True*)

Returns a string representation of the stored information that is suitable for use in the `features` argument of *Target.create_target_machine()*.

If the `sort` keyword argument is `True`—the default—the features are sorted by name to give a stable ordering between Python sessions.

Context

LLVMContext is an opaque context reference used to group modules into logical groups. For example, the type names are unique within a context, the name collisions are resolved by LLVM automatically.

LLVMContextRef

A wrapper around LLVMContext. Should not be instantiated directly, use the following methods:

class LLVMContextRef

- **create_context():**

  Create a new LLVMContext instance.

- **get_global_context():**

  Get the reference to the global context.

Modules

Although they conceptually represent the same thing, modules in the *IR layer* and modules in the *binding layer* do not have the same roles and do not expose the same API.

While modules in the IR layer allow you to build and group functions together, modules in the binding layer give access to compilation, linking and execution of code. To distinguish between them, the module class in the binding layer is called *ModuleRef* as opposed to *llvmlite.ir.Module*.

To go from the IR layer to the binding layer, use the *parse_assembly()* function.

Factory functions

You can create a module from the following factory functions:

- llvmlite.binding.**parse_assembly**(*llvmir*, *context=None*)

  Parse the given *llvmir*, a string containing some LLVM IR code. If parsing is successful, a new *ModuleRef* instance is returned.

  – context: an instance of *LLVMContextRef*.

  Defaults to the global context.

  EXAMPLE: You can obtain *llvmir* by calling `str()` on an *llvmlite.ir.Module* object.

- llvmlite.binding.**parse_bitcode**(*bitcode*, *context=None*)

  Parse the given *bitcode*, a bytestring containing the LLVM bitcode of a module. If parsing is successful, a new *ModuleRef* instance is returned.

- context: an instance of *LLVMContextRef*.

  Defaults to the global context.

EXAMPLE: You can obtain the *bitcode* by calling *ModuleRef.as_bitcode()*.

The ModuleRef class

class llvmlite.binding.**ModuleRef**

A wrapper around an LLVM module object. The following methods and properties are available:

- **as_bitcode**()

  Return the bitcode of this module as a bytes object.

- **get_function**(*name*)

  Get the function with the given *name* in this module.

  If found, a *ValueRef* is returned. Otherwise, `NameError` is raised.

- **get_global_variable**(*name*)

  Get the global variable with the given *name* in this module.

  If found, a *ValueRef* is returned. Otherwise, `NameError` is raised.

- **get_struct_type**(*name*)

  Get the struct type with the given *name* in this module.

  If found, a *TypeRef* is returned. Otherwise, `NameError` is raised.

- **link_in**(*other*, *preserve=False*)

  Link the *other* module into this module, resolving references wherever possible.
  - If *preserve* is `True`, the other module is first copied in order to preserve its contents.
  - If *preserve* is `False`, the other module is not usable after this call.

- **verify**()

  Verify the module's correctness. On error, raise `RuntimeError`.

- **data_layout**

  The data layout string for this module. This attribute can be set.

- **functions**

  An iterator over the functions defined in this module. Each function is a *ValueRef* instance.

- **global_variables**

  An iterator over the global variables defined in this module. Each global variable is a *ValueRef* instance.

- **struct_types**

  An iterator over the struct types defined in this module. Each type is a *TypeRef* instance.

- **name**

  The module's identifier, as a string. This attribute can be set.

- **source_file**

  The module's reported source file, as a string. This attribute can not be set.

- **triple**

  The platform "triple" string for this module. This attribute can be set.

Value references

A value reference is a wrapper around an LLVM value for you to inspect. You cannot create a value reference yourself. You get them from methods of the *ModuleRef* and *ValueRef* classes.

Enumerations

class llvmlite.binding.**Linkage**

The linkage types allowed for global values are:

- **external**
- **available_externally**
- **linkonce_any**
- **linkonce_odr**
- **linkonce_odr_autohide**
- **weak_any**
- **weak_odr**
- **appending**
- **internal**
- **private**
- **dllimport**
- **dllexport**
- **external_weak**
- **ghost**
- **common**
- **linker_private**
- **linker_private_weak**

class llvmlite.binding.**Visibility**

The visibility styles allowed for global values are:

- **default**
- **hidden**
- **protected**

class llvmlite.binding.**StorageClass**

The storage classes allowed for global values are:

- **default**
- **dllimport**

- dllexport

class llvmlite.binding.**ValueKind**

The value kinds allowed are:

- argument
- basic_block
- memory_use
- memory_def
- memory_phi
- function
- global_alias
- global_ifunc
- global_variable
- block_address
- constant_expr
- constant_array
- constant_struct
- constant_vector
- undef_value
- constant_aggregate_zero
- constant_data_array
- constant_data_vector
- constant_int
- constant_fp
- constant_pointer_null
- constant_token_none
- metadata_as_value
- inline_asm
- instruction
- poison_value

The ValueRef class

class llvmlite.binding.**ValueRef**

A wrapper around an LLVM value. The attributes available are:

- **is_declaration**
  - `True`—The global value is a mere declaration.
  - `False`—The global value is defined in the given module.
- **linkage**

  The linkage type—a *Linkage* instance—for this value. This attribute can be set.
- **module**

  The module—a *ModuleRef* instance—that this value is defined in.
- **function**

  The function—a *ValueRef* instance—that this value is defined in.
- **block**

  The basic block—a *ValueRef* instance—that this value is defined in.
- **instruction**

  The instruction—a *ValueRef* instance—that this value is defined in.
- **name**

  This value's name, as a string. This attribute can be set.
- **type**

  This value's LLVM type as *TypeRef* object.
- **storage_class**

  The storage class—a *StorageClass* instance—for this value. This attribute can be set.
- **visibility**

  The visibility style—a *Visibility* instance—for this value. This attribute can be set.
- **value_kind**

  The LLVM value kind—a *ValueKind* instance—for this value.
- **blocks**

  An iterator over the basic blocks in this function. Each block is a *ValueRef* instance.
- **arguments**

  An iterator over the arguments of this function. Each argument is a *ValueRef* instance.
- **instructions**

  An iterator over the instructions in this basic block. Each instruction is a *ValueRef* instance.
- **incoming_blocks**

  An iterator over the incoming blocks of a phi instruction. Each block is a *ValueRef* instance.
- **operands**

  An iterator over the operands in this instruction. Each operand is a *ValueRef* instance.
- **opcode**

  The instruction's opcode, as a string.

- **attributes**

  An iterator over the attributes in this value. Each attribute is a `bytes` instance. Values that have attributes are: function, argument (and others for which attributes support has not been implemented)

- **is_global**

  The value is a global variable.

- **is_function**

  The value is a function.

- **is_argument**

  The value is a function's argument.

- **is_block**

  The value is a function's basic block.

- **is_instruction**

  The value is a basic block's instruction.

- **is_operand**

  The value is a instruction's operand.

- **is_constant**

  The value is a constant.

- **get_constant_value**(*self*, *signed_int=False*, *round_fp=False*)

  Return the constant value, either as a literal (for example, int or float) when supported, or as a string otherwise. Keyword arguments specify the preferences during conversion:
  - If `signed_int` is True and the constant is an integer, returns a signed integer.
  - If `round_fp` True and the constant is a floating point value, rounds the result upon accuracy loss (e.g., when querying an fp128 value). By default, raises an exception on accuracy loss.

Type references

A type reference wraps an LLVM type. It allows accessing type's name and IR representation. It is also accepted by methods like *TargetData.get_abi_size()*.

The TypeRef class

class llvmlite.binding.**TypeRef**

A wrapper around an LLVM type. The attributes available are:

- **name**

  This type's name, as a string.

- **is_struct**
  - `True`—The type is a struct type
  - `False`—The type is not a struct type

- **is_pointer**
  - `True`—The type is a pointer type
  - `False`—The type is not a pointer type

- **is_array**
  - `True`—The type is an array type
  - `False`—The type is not an array type

- **is_vector**
  - `True`—The type is a vector type
  - `False`—The type is not a vector type
- **is_function_vararg**
  - `True`— The function type accepts a variable number of arguments
  - `False`—The function type accepts a fixed number of arguments
- **elements**

  Returns an iterator over enclosing types. For example, the elements of an array or members of a struct.
- **element_type**

  If the type is a pointer, return the pointed-to type. Raises a ValueError if the type is not a pointer type.
- **element_count**

  Returns the number of elements in an array or a vector. For scalable vectors, returns minimum number of elements. When the type is neither an array nor a vector, raises exception.
- **type_width**

  Return the basic size of this type if it is a primitive type. These are fixed by LLVM and are not target-dependent. This will return zero if the type does not have a size or is not a primitive type.

  If this is a scalable vector type, the scalable property will be set and the runtime size will be a positive integer multiple of the base size.

  Note that this may not reflect the size of memory allocated for an instance of the type or the number of bytes that are written when an instance of the type is stored to memory.
- **type_kind**

  Returns the `LLVMTypeKind` enumeration of this type.
- **as_ir**(*self*, *ir_ctx*)

  Convert into an `llvmlite.ir.Type`.
- **__str__**(*self*)

  Get the string IR representation of the type.

Execution engine

The execution engine is where actual code generation and execution happen. At present a single execution engine, `MCJIT`, is exposed.

Functions

- `llvmlite.binding.`**create_mcjit_compiler**(*module*, *target_machine*, *use_lmm=None*)

  Create a MCJIT-powered engine from the given *module* and *target_machine*.

  lmm controls whether the llvmlite memory manager is used. If not supplied, the default choice for the platform will be used (`True` on 64-bit ARM systems, `False` otherwise).

  - *module* does not need to contain any code.
  - Returns a *ExecutionEngine* instance.
- `llvmlite.binding.`**check_jit_execution**()

  Ensure that the system allows creation of executable memory ranges for JIT-compiled code. If some security mechanism such as SELinux prevents it, an exception is raised. Otherwise the function returns silently.

  Calling this function early can help diagnose system configuration issues, instead of letting JIT-compiled functions crash mysteriously.

The ExecutionEngine class

class llvmlite.binding.**ExecutionEngine**

A wrapper around an LLVM execution engine. The following methods and properties are available:

- **add_module**(*module*)

  Add the *module*—a *ModuleRef* instance—for code generation. When this method is called, ownership of the module is transferred to the execution engine.

- **finalize_object**()

  Make sure all modules owned by the execution engine are fully processed and usable for execution.

- **get_function_address**(*name*)

  Return the address of the function *name* as an integer. It's a fatal error in LLVM if the symbol of *name* doesn't exist.

- **get_global_value_address**(*name*)

  Return the address of the global value *name* as an integer. It's a fatal error in LLVM if the symbol of *name* doesn't exist.

- **remove_module**(*module*)

  Remove the *module*—a *ModuleRef* instance—from the modules owned by the execution engine. This allows releasing the resources owned by the module without destroying the execution engine.

- **add_object_file**(*object_file*)

  Add the symbols from the specified object file to the execution engine.
  - ***object_file*** **str or *ObjectFileRef*: a path to the object file**
    or a object file instance. Object file instance is not usable after this call.

- **set_object_cache**(*notify_func=None*, *getbuffer_func=None*)

  Set the object cache callbacks for this engine.
  - *notify_func*, if given, is called whenever the engine has finished compiling a module. It is passed the `(module, buffer)` arguments:
    * *module* is a *ModuleRef* instance.
    * *buffer* is a bytes object of the code generated for the module.

    The return value is ignored.
  - *getbuffer_func*, if given, is called before the engine starts compiling a module. It is passed an argument, *module*, a *ModuleRef* instance of the module being compiled.
    * It can return `None`, in which case the module is compiled normally.
    * It can return a bytes object of native code for the module, which bypasses compilation entirely.

- **target_data**

  The *TargetData* used by the execution engine.

Object file

The object file is an abstraction of LLVM representation of the static object code files. This class provides methods to examine the contents of the object files. It also can be passed as parameter to *ExecutionEngine.add_object_file()* to make the symbols available to the JIT.

The ObjectFileRef class

class llvmlite.binding.**ObjectFileRef**

A wrapper around LLVM object file. The following methods and properties are available:

- **from_data(data):**

  Create an instance of ObjectFileRef from the provided binary data.

- **from_path(path):**

  Create an instance of ObjectFileRef from the supplied filesystem path. Raises IOError if the path does not exist.

- **sections:**

  Return an iterator to the sections objects consisting of the instance of *SectionIteratorRef*

The SectionIteratorRef class

class llvmlite.binding.**SectionIteratorRef**
A wrapper around the section class which provides information like section name, type and size, etc.

- **name():**

  Get section name.

- **is_text():**

  Returns true when section is of type text.

- **size():**

  Get section size.

- **address():**

  Get section address.

- **data():**

  Get section contents.

- **is_end(object_file):**

  Return true if the section iterator is the last element of the object_file.
  - object_file: an instance of *ObjectFileRef*

- **next():**

  Get the next section instance.

Optimization passes

LLVM gives you the opportunity to fine-tune optimization passes. Optimization passes are managed by a pass manager. There are two kinds of pass managers:

- *FunctionPassManager*, for optimizations that work on single functions.
- *ModulePassManager*, for optimizations that work on whole modules.

llvmlite provides bindings for LLVM's *New* and *Legacy* pass managers, which have slightly different APIs and behaviour. The differences between them and the motivations for the New Pass Manager are outlined in the LLVM Blog post on the New Pass Manager.

In a future version of llvmlite, likely coinciding with a minimum LLVM version requirement of 17, support for the Legacy Pass Manager will be removed. It is recommended that new code using llvmlite uses the New Pass Manager, and existing code using the Legacy Pass Manager be updated to use the New Pass Manager.

New Pass Manager APIs

To manage the optimization attributes we first need to instantiate a *PipelineTuningOptions* instance:

class llvmlite.binding.**PipelineTuningOptions**(*speed_level=2*, *size_level=0*)

Creates a new PipelineTuningOptions object.

The following writable attributes are available, whose default values depend on the initial setting of the speed and size optimization levels:

- **loop_interleaving**

  Enable loop interleaving.

- **loop_vectorization**

  Enable loop vectorization.

- **slp_vectorization**

  Enable SLP vectorization, which uses a different algorithm to loop vectorization. Both may be enabled at the same time.

- **loop_unrolling**

  Enable loop unrolling.

- **speed_level**

  The level of optimization for speed, as an integer between 0 and 3.

- **size_level**

  The level of optimization for size, as an integer between 0 and 2.

We also need a *PassBuilder* object to manage the respective function and module pass managers:

class llvmlite.binding.**PassBuilder**(*target_machine*, *pipeline_tuning_options*)

A pass builder that uses the given *TargetMachine* and *PipelineTuningOptions* instances.

getModulePassManager()

Return a populated *ModulePassManager* object based on PTO settings.

getFunctionPassManager()

Return a populated *FunctionPassManager* object based on PTO settings.

The *ModulePassManager* and *FunctionPassManager* classes implement the module and function pass managers:

class llvmlite.binding.**ModulePassManager**

A pass manager for running optimization passes on an LLVM module.

add_verifier()

Add the Module Verifier pass.

run(*module*, *passbuilder*)

Run optimization passes on *module*, a *ModuleRef* instance.

class llvmlite.binding.**FunctionPassManager**

A pass manager for running optimization passes on an LLVM function.

run(*function*, *passbuilder*)

Run optimization passes on *function*, a *ValueRef* instance.

These can be created with passes populated by using the *PassBuilder.getModulePassManager()* and *PassBuilder.getFunctionPassManager()* methods, or they can be instantiated unpopulated, then passes can be added using the add_* methods.

To instantiate the unpopulated instances, use:

llvmlite.binding.**create_new_module_pass_manager**()

Create an unpopulated *ModulePassManager* instance.

and

llvmlite.binding.**create_new_function_pass_manager**()

Create an unpopulated *FunctionPassManager* instance.

The add_* methods supported by both pass manager classes are:

add_aa_eval_pass()

Add the Exhaustive Alias Analysis Precision Evaluator pass.

add_loop_unroll_pass()

Add the Loop Unroll pass.

add_loop_rotate_pass()

Add the Loop Rotate pass.

add_instruction_combine_pass()

Add the Combine Redundant Instructions pass.

add_jump_threading_pass()

Add the Jump Threading pass.

add_simplify_cfg_pass()

Add the Simplify CFG pass.

add_refprune_pass()

Add the Reference pruning pass.

Legacy Pass Manager APIs

To instantiate either of these pass managers, you first need to create and configure a *PassManagerBuilder*.

class llvmlite.binding.**PassManagerBuilder**

Create a new pass manager builder. This object centralizes optimization settings.

The populate method is available:

populate(*pm*)

Populate the pass manager *pm* with the optimization passes configured in this pass manager builder.

The following writable attributes are available:

- **disable_unroll_loops**

  If True, disable loop unrolling.

- **inlining_threshold**

  The integer threshold for inlining one function into another. The higher the number, the more likely that inlining will occur. This attribute is write-only.

- **loop_vectorize**

  If True, allow vectorizing loops.

- **opt_level**

  The general optimization level, as an integer between 0 and 3.

- **size_level**

  Whether and how much to optimize for size, as an integer between 0 and 2.

- **slp_vectorize**

  If `True`, enable the SLP vectorizer, which uses a different algorithm than the loop vectorizer. Both may be enabled at the same time.

class `llvmlite.binding.`**PassManager**

The base class for pass managers. Use individual `add_*` methods or *PassManagerBuilder.populate()* to add optimization passes.

- **add_constant_merge_pass**()

  See constmerge pass documentation.

- **add_dead_arg_elimination_pass**()

  See deadargelim pass documentation.

- **add_function_attrs_pass**()

  See functionattrs pass documentation.

- **add_function_inlining_pass**(*self*)

  See inline pass documentation.

- **add_global_dce_pass**()

  See globaldce pass documentation.

- **add_global_optimizer_pass**()

  See globalopt pass documentation.

- **add_ipsccp_pass**()

  See ipsccp pass documentation.

- **add_dead_code_elimination_pass**()

  See dce pass documentation.

- **add_cfg_simplification_pass**()

  See simplifycfg pass documentation.

- **add_gvn_pass**()

  See gvn pass documentation.

- **add_instruction_combining_pass**()

  See instcombine pass documentation.

- **add_licm_pass**()

  See licm pass documentation.

- **add_sccp_pass**()

  See sccp pass documentation.

- **add_sroa_pass**()

  See scalarrepl pass documentation.

  While the scalarrepl pass documentation describes the transformation performed by the pass added by this function, the pass corresponds to the `opt -sroa` command-line option and not to `opt -scalarrepl`.

- **add_type_based_alias_analysis_pass**()

  See tbaa metadata documentation.

- **add_basic_alias_analysis_pass**()

  See basicaa pass documentation.

- **add_instruction_namer_pass**()

  See instnamer pass documentation.

- **add_refprune_pass**()

  Add the Reference pruning pass.

class llvmlite.binding.**ModulePassManager**

Create a new pass manager to run optimization passes on a module.

The `run` method is available:

run(*module*)

Run optimization passes on the *module*, a *ModuleRef* instance.

Returns `True` if the optimizations made any modification to the module. Otherwise returns `False`.

class llvmlite.binding.**FunctionPassManager**(*module*)

Create a new pass manager to run optimization passes on a function of the given *module*, a *ModuleRef* instance.

The following methods are available:

- **finalize**()

  Run all the finalizers of the optimization passes.

- **initialize**()

  Run all the initializers of the optimization passes.

- **run**(*function*)

  Run optimization passes on *function*, a *ValueRef* instance.

  Returns `True` if the optimizations made any modification to the module. Otherwise returns `False`.

Analysis utilities

llvmlite.binding.**get_function_cfg**(*func*, *show_inst=True*)

Return a string of the control-flow graph of the function, in DOT format.

- If *func* is not a materialized function, the module containing the function is parsed to create an actual LLVM module.
- The *show_inst* flag controls whether the instructions of each block are `printed.functions`.

llvmlite.binding.**view_dot_graph**(*graph*, *filename=None*, *view=False*)

View the given DOT source. This function requires the graphviz package.

- If view is `True`, the image is rendered and displayed in the default application in the system. The file path of the output is returned.
- If view is `False`, a `graphviz.Source` object is returned.
- If view is `False` and the environment is in an IPython session, an IPython image object is returned and can be displayed inline in the notebook.

Pass Timings

llvmlite.binding.**set_time_passes**(*enable*)

Enable or disable the pass timers.

llvmlite.binding.**report_and_reset_timings**()

Returns the pass timings report and resets the LLVM internal timers.

Pass timers are enabled by `set_time_passes()`. If the timers are not enabled, this function will return an empty string.

Misc

llvmlite.binding.ffi.**register_lock_callback**(*acq_fn*, *rel_fn*)

Register callback functions for lock acquire and release. `acq_fn` and `exit_fn` are callables that take no arguments.

llvmlite.binding.ffi.**unregister_lock_callback**(*acq_fn*, *rel_fn*)

Remove the registered callback functions for lock acquire and release. The arguments are the same as used in `register_lock_callback()`.

Example—compiling a simple function

Compile and execute the function defined in `ir-fpadd.py`. For more information on `ir-fpadd.py`, see *Example—Defining a simple function*. The function is compiled with no specific optimizations.

```
from __future__ import print_function

from ctypes import CFUNCTYPE, c_double

import llvmlite.binding as llvm


# All these initializations are required for code generation!
llvm.initialize()
llvm.initialize_native_target()
llvm.initialize_native_asmprinter()  # yes, even this one

llvm_ir = """
   ; ModuleID = "examples/ir_fpadd.py"
   target triple = "unknown-unknown-unknown"
   target datalayout = ""

   define double @"fpadd"(double %".1", double %".2")
   {
   entry:
     %"res" = fadd double %".1", %".2"
     ret double %"res"
   }
   """

def create_execution_engine():
    """
    Create an ExecutionEngine suitable for JIT code generation on
    the host CPU.  The engine is reusable for an arbitrary number of
```

(continues on next page)

(continued from previous page)

```
    modules.
    """
    # Create a target machine representing the host
    target = llvm.Target.from_default_triple()
    target_machine = target.create_target_machine()
    # And an execution engine with an empty backing module
    backing_mod = llvm.parse_assembly("")
    engine = llvm.create_mcjit_compiler(backing_mod, target_machine)
    return engine


def compile_ir(engine, llvm_ir):
    """
    Compile the LLVM IR string with the given engine.
    The compiled module object is returned.
    """
    # Create a LLVM module object from the IR
    mod = llvm.parse_assembly(llvm_ir)
    mod.verify()
    # Now add the module and make sure it is ready for execution
    engine.add_module(mod)
    engine.finalize_object()
    engine.run_static_constructors()
    return mod


engine = create_execution_engine()
mod = compile_ir(engine, llvm_ir)

# Look up the function pointer (a Python int)
func_ptr = engine.get_function_address("fpadd")

# Run the function via ctypes
cfunc = CFUNCTYPE(c_double, c_double, c_double)(func_ptr)
res = cfunc(1.0, 3.5)
print("fpadd(...) =", res)
```

3.2.3 Deprecation Notices

This section contains information about deprecation of behaviours, features and APIs that have become undesirable/obsolete. Any information about the schedule for their deprecation and reasoning behind the changes, along with examples, is provided.

Deprecation of Typed Pointers

The use of Typed Pointers is deprecated, and *Opaque Pointers* will be the default (and eventually required) in a future llvmlite version.

Reason for deprecation

llvmlite aims to move forward to newer LLVM versions, which will necessitate switching to Opaque Pointers:

- In LLVM 15, Opaque Pointers are the default.
- In LLVM 16, Typed Pointers are only supported on a best-effort basis (and therefore may have bugs that go unfixed).
- In LLVM 17, support for Typed Pointers is removed.

Although Opaque Pointers are already the default in LLVM 15, llvmlite still uses Typed Pointers by default with LLVM 15.

Examples(s) of the impact

Code that uses llvmlite to work with pointers or to parse assembly that uses pointers will break if not modified to use opaque pointers.

Schedule

- In llvmlite 0.44, Opaque Pointers will be the default pointer kind.
- In llvmlite 0.45, support for Typed Pointers will be removed.

Recommendations

Code using llvmlite should be updated as follows when switching to Opaque Pointers.

IR layer

Modify uses of `.type.pointee` of instructions to use `.allocated_type` instead. For example:

```
# Allocating an integer on the stack and storing a value to it
stackint = builder.alloca(ir.IntType(32))
builder.store(ir.Constant(stackint.type.pointee, 123), stackint)
```

becomes:

```
# Allocating an integer on the stack and storing a value to it
stackint = builder.alloca(ir.IntType(32))
builder.store(ir.Constant(stackint.allocated_type, 123), stackint)
```

Replace the use of `.as_pointer()` of types with the `PointerType` class. For example:

```
# Declaring a function of type i32(i32*, i32)
fnty = ir.FunctionType(ir.IntType(32), [ir.IntType(32).as_pointer(),
                       ir.IntType(32)])
```

becomes:

```
# Declaring a function of type i32(ptr, i32)
fnty = ir.FunctionType(ir.IntType(32), [ir.PointerType(),
                       ir.IntType(32)])
```

Modify calls to `ir.load`, `ir.load_atomic`, and `ir.gep` instructions to pass in pointer types. For example:

```
ptr = builder.gep(func.args[0], [index])
value = builder.load(ptr)
```

becomes:

```
ptr = builder.gep(func.args[0], [index], source_etype=ll.IntType(32))
value = builder.load(ptr, typ=ll.IntType(32))
```

Binding layer

When working with *TargetData* instances:

- Replace calls to *get_pointee_abi_size()* with calls to *get_abi_size()*.
- Replace calls to *get_pointee_abi_alignment()* with calls to *get_abi_alignment()*.

When working with global variables and functions (which will be *ValueRef* instances):

- Replace any use of `valueref.type` with `valueref.global_value_type` for any `valueref` that is a global variable or function.

When passing assembly to *llvmlite.binding.parse_assembly()*:

- Ensure that any IR passed to `parse_assembly()` uses Opaque Pointers.

Deprecation of *llvmlite.llvmpy* module

The *llvmlite.llvmpy* module was originally created for compatibility with *llvmpy*. As time has passed, that functionality was redesigned and put in *llvmlite.ir* with *llvmlite.llvmpy* remaining as a compatibility layer. No continued maintenance has ensured that it provides a matching API to *llvmpy* and it provides no advantage over the *llvmlite.ir* module.

Reason for deprecation

The functionality provided by *llvmlite.llvmpy* and its child modules is now present in *llvmlite.ir*, so this module will be dropped.

Example(s) of the impact

Code that imports *llvmlite.llvmpy*, *llvmlite.llvmpy.core* or *llvmlite.llvmpy.passes* will break.

Schedule

The feature change was implemented as follows:

- v0.39 module is deprecated
- v0.40 module is removed

Recommendations

Since similar functionality already exists in *llvmlite.ir*, the transition path is relatively short:

- replace *llvmlite.llvmpy.core.Builder* with *llvmlite.ir.IRBuilder*
- replace *llvmlite.llvmpy.core.Builder.icmp* with *llvmlite.ir.IRBuilder.icmp_signed* and *icmp_unsigned*, as appropriate
- replace *llvmlite.llvmpy.core.Builder.fcmp* with *llvmlite.ir.IRBuilder.fcmp_ordered* and *fcmp_unordered*, as appropriate

- replace calls to the static methods of *llvmlite.llvmpy.core.Type* with the constructors provided in *llvmlite.ir* (_e.g._, *Type.int(8)* with *IntType(8)*)
- Replace calls to the static methods of *llvmlite.llvmpy.core.Constant* with calls to the constructor of *llvmlite.ir.Constant* or *llvmlite.ir.Constant.literal_struct*, as appropriate. Note that *stringz* and *array* have no direct equivalents.
- replace *llvmlite.llvmpy.core.Module*, *Function*, *MetaDataString*, *InlineAsm* with the classes of the same name in *llvmlite.ir*.
- replace *llvmlite.llvmpy.core.MetaData.get* with *llvmlite.ir.Module.add_metadata*
- replace *llvmlite.llvmpy.core.Function.intrinsic* with *llvmlite.ir.Module.declare_intrinsic*
- for *llvmlite.llvmpy.passes*, create the pass manager directly using *llvmlite.binding*

Deprecation of use of memset/memcpy llvm intrinsic with specified alignment

From LLVM 7 onward the memset and memcpy intrinsics dropped the use of an alignment, specified as the third argument, and instead use the alignment of the first argument for this purpose. Specifying the alignment in third argument continued to work as LLVM auto-updates this use case.

Reason for deprecation

LLVM has changed the behaviour of the previously mentioned intrinsics, and so as to increase compatibility with future releases of LLVM, llvmlite is adapting to match.

Example(s) of the impact

As of 0.30 the following worked:

```
from llvmlite import ir

bit = ir.IntType(1)
int8 = ir.IntType(8)
int32 = ir.IntType(32)
int64 = ir.IntType(64)
int8ptr = int8.as_pointer()

mod = ir.Module()
fnty = ir.FunctionType(int32, ())
func = ir.Function(mod, fnty, "some_function")
block = func.append_basic_block('some_block')
builder = ir.IRBuilder(block)

some_address = int64(0xdeaddead)
dest = builder.bitcast(some_address, int8ptr)
value = int8(0xa5)
memset = mod.declare_intrinsic('llvm.memset', [int8ptr, int32])
memcpy = mod.declare_intrinsic('llvm.memcpy', [int8ptr, int8ptr, int32])

# NOTE: 5 argument call site (dest, value, length, align, isvolatile)
builder.call(memset, [dest, value, int32(10), int32(0), bit(0)])

some_other_address = int64(0xcafecafe)
src = builder.bitcast(some_other_address, int8ptr)
```

(continues on next page)

(continued from previous page)

```
# NOTE: 5 argument call site (dest, src, length, align, isvolatile)
builder.call(memcpy, [dest, src, int32(10), int32(0), bit(0)])

builder.ret(int32(0))
print(str(mod))
```

From 0.31 onwards only the following works:

```
from llvmlite import ir

bit = ir.IntType(1)
int8 = ir.IntType(8)
int32 = ir.IntType(32)
int64 = ir.IntType(64)
int8ptr = int8.as_pointer()

mod = ir.Module()
fnty = ir.FunctionType(int32, ())
func = ir.Function(mod, fnty, "some_function")
block = func.append_basic_block('some_block')
builder = ir.IRBuilder(block)

some_address = int64(0xdeaddead)
dest = builder.bitcast(some_address, int8ptr)
value = int8(0xa5)
memset = mod.declare_intrinsic('llvm.memset', [int8ptr, int32])
memcpy = mod.declare_intrinsic('llvm.memcpy', [int8ptr, int8ptr, int32])

# NOTE: 4 argument call site (dest, value, length, isvolatile)
builder.call(memset, [dest, value, int32(10), bit(0)])

some_other_address = int64(0xcafecafe)
src = builder.bitcast(some_other_address, int8ptr)

# NOTE: 4 argument call site (dest, src, length, isvolatile)
builder.call(memcpy, [dest, src, int32(10), bit(0)])

builder.ret(int32(0))
print(str(mod))
```

Schedule

The feature change was implemented as follows:

- v0.30 was the last release to support an alignment specified as the third argument (5 argument style).
- v0.31 onwards supports the 4 argument style call only.

Recommendations

Projects that need/rely on the deprecated behaviour should pin their dependency on llvmlite to a version prior to removal of this behaviour.

3.3 Frequently Asked Questions

3.3.1 Why doesn't `llvmlite` always support the latest release(s) of LLVM?

There's a few reasons for this:

1. Most importantly, if llvmlite declares it supports a given version of LLVM, it means that the development team has verified that the particular LLVM version will work for llvmlite uses, mainly that it works fine for JIT compiler projects (like Numba). This testing is extensive and it's not uncommon to have to patch LLVM in the process/make decisions about the severity of problems/report problems upstream to LLVM. Ultimately, if the build is not considered stable across all supported architectures/OS combinations then support is not declared.
2. LLVM sometimes updates its API or internal behaviour and it takes time to work out what needs to be done to accommodate this on all of llvmlite's supported architectures/OS combinations.
3. There is a set of patches that ship along with the LLVM builds for llvmlite. These patches are necessary and often need porting to work on new LLVM versions, this takes time and requires testing. It should be noted that LLVM builds that come with e.g. linux distributions may not work well with the llvmlite intended use cases.

3.3.2 Why Static Linking to LLVM?

The llvmlite package uses LLVM via ctypes calls to a C wrapper that is statically linked to LLVM. Some people are surprised that llvmlite uses static linkage to LLVM, but there are several important reasons for this:

1. *The LLVM API has not historically been stable across releases* - Although things have improved since LLVM 4.0, there are still enough changes between LLVM releases to cause compilation issues if the right version is not matched with llvmlite.
2. *The LLVM shipped by most Linux distributions is not the version llvmlite needs* - The release cycles of Linux distributions will never line up with LLVM or llvmlite releases.
3. *We need to patch LLVM* - The binary packages of llvmlite are built against LLVM with a handful of patches to either fix bugs or to add features that have not yet been merged upstream. In some cases, we've had to carry patches for several releases before they make it into LLVM.
4. *We don't need most of LLVM* - We are sensitive to the install size of llvmlite, and a full build of LLVM is quite large. We can dramatically reduce the total disk needed by an llvmlite user (who typically doesn't need the rest of LLVM, ignoring the version matching issue) by statically linking to the library and pruning the symbols we do not need.
5. *Numba can use multiple LLVM builds at once* - Some Numba targets (AMD GPU, for example) may require different LLVM versions or non-mainline forks of LLVM to work. These other LLVMs can be wrapped in a similar fashion as llvmlite, and will stay isolated.

Static linkage of LLVM was definitely not our goal early in Numba development, but seems to have become the only workable solution given our constraints.

3.4 Contributing to llvmlite

llvmlite originated to fulfill the needs of the Numba project. It is maintained mostly by the Numba team. We tend to prioritize the needs and constraints of Numba over other conflicting desires.

We do welcome any contributions in the form of *bug reports* or *pull requests*.

- *Communication methods*
- *Development rules*
- *Documentation*

3.4.1 Communication methods

Forum

llvmlite uses the Numba Discourse as a forum for longer running threads such as design discussions and roadmap planning. There are various categories available and it can be reached at: numba.discourse.group. It also has a llvmlite topic.

Bug reports

We use the Github issue tracker to track both bug reports and feature requests. If you report an issue, please include:

- What you are trying to do.
- Your operating system.
- What version of llvmlite you are running.
- A description of the problem—for example, the full error traceback or the unexpected results you are getting.
- As far as possible, a code snippet that allows full reproduction of your problem.

Pull requests

To contribute code:

1. Fork our Github repository.
2. Create a branch representing your work.
3. When your work is ready, submit it as a pull request from the Github interface.

3.4.2 Development rules

Coding conventions

- All Python code should follow PEP 8.
- All C++ code is formatted using `clang-format-13` from the `clang-format-13` package available in the `conda-forge` conda channel.
- Code and documentation should generally fit within 80 columns, for maximum readability with all existing tools, such as code review user interfaces.

Optionally, you may wish to setup pre-commit hooks to automatically run `clang-format` when you make a git commit. This can be done by installing `pre-commit`:

```
pip install pre-commit
```

and then running:

```
pre-commit install
```

from the root of the Numba repository. Now `clang-format` will be run each time you commit changes. You can skip this check with `git commit --no-verify`.

Platform support

Llvmlite will be kept compatible with Python 3.10 and later under at least Windows, macOS and Linux.

We do not expect contributors to test their code on all platforms. Pull requests are automatically built and tested using Azure Pipelines for Winows, OSX and Linux.

3.4.3 Documentation

This llvmlite documentation is built using Sphinx and maintained in the `docs` directory inside the llvmlite repository.

1. Edit the source files under `docs/source/`.
2. Build the documentation:

```
make html
```

3. Check the documentation:

```
open _build/html/index.html
```

3.5 Release Notes

3.5.1 v0.44.0 (16 January, 2025)

Highlights of this release are:

- Official support for Python 3.13.
- Dropped official support for Python 3.9, the minimum supported Python version is 3.10.
- LLVM 15 is now the default LLVM.
- Added support for LLVM's new PassManager.
- Support for LLVM based target triple partitioning.
- `llvmlite.binding.TypeRef` now roundtrips back into `llvmlite.ir.Type`.
- API updates to accommodate packed Literal Structs.
- Added NetBSD support.
- Support for opaque pointers.

Pull-Requests:

- PR #1036: LLVM 15 conda recipe (sklam gmarkall)
- PR #1046: Add basic infra required to move Numba to NewPassManager (yashssh gmarkall)
- PR #1050: Add conda-forge pre-tag testing to release checklist (gmarkall)
- PR #1051: Roundtrip llvmlite.binding.TypeRef back into llvmlite.ir.Type (sklam)
- PR #1053: Use RTD for all docs CI / testing (was: Try and get CI to use a newer Sphinx) (gmarkall)
- PR #1057: Port refprune pass to NewPassManager (yashssh)
- PR #1060: port changelog for 0.43.0 to main (esc)

- PR #1063: Added llvm based process triple partitioning (kc611)
- PR #1064: Add support for opaque pointers (gmarkall rj-jesus)
- PR #1066: Move Azure to use macos-12 (gmarkall)
- PR #1067: Use LLVM 15 by default, add experimental LLVM 16 support (gmarkall)
- PR #1068: Early exit from fpm.run() if called on function declarations (yashssh)
- PR #1069: Add instrumentation callback hook for new pass managers (yashssh)
- PR #1073: bump compiler for linux-aarch64 to 11 (esc)
- PR #1077: Add NetBSD support. (0-wiz-0)
- PR #1080: Added Python 3.13 to CI and Descriptions (kc611)
- PR #1081: Add convenience func and args (thomaspinckney3)
- PR #1085: Fix RTD build, no need for get_html_theme_path(). (stuartarchibald)
- PR #1087: Add missing argtypes description for refprune pass related APIs (yashssh)
- PR #1090: Add MSVC *-GL* workaround (sklam)
- PR #1094: Fix CI by not removing nonexistent environment (gmarkall)
- PR #1097: update azure-pipelines to macos-13 (swap357)
- PR #1098: Fix store to opaque ptr (alexander-shaposhnikov)
- PR #1104: Target 0.44.0: Fix osx wheel build (sklam)
- PR #1108: Fix Windows llvmdev build due to zlib linkage (sklam)

Authors:

- 0-wiz-0
- esc
- gmarkall
- kc611
- rj-jesus
- sklam
- stuartarchibald
- thomaspinckney3
- yashssh
- alexander-shaposhnikov
- swap357

3.5.2 v0.43.0 (June 13, 2024)

Highlights of this release are:

- Support for building against LLVM 15.
- A fix for *refpruning* algorithm in specific *fanout_raise* cases.

Pull-Requests:

- PR #1025: skip *raise* basic blocks in *verifyFanoutBackward* (dlee992)

- PR #1029: Update CHANGE_LOG for 0.42.0 final. (stuartarchibald)
- PR #1032: v0.42 Post release (sklam)
- PR #1035: Support building against llvm15 (gmarkall yashssh)
- PR #1059: update CHANGE_LOG and release date for 0.43.0 final (esc)

Authors:

- dlee992
- esc
- gmarkall
- sklam
- stuartarchibald
- yashssh

3.5.3 v0.42.0 (January 31, 2024)

Highlights of this release include:

- Support for Python 3.12.
- A fix for relocation overflows on AArch64 systems.
- Binding layer: new queries for incoming blocks of phi instructions, type kinds, and elements. Addition of the Instruction Namer pass.
- IR layer: Support *convergent* as an attribute of function calls and call instructions.

Pull-Requests:

- PR #973: Bindings: Query incoming blocks of a phi instruction (tbennun)
- PR #978: Bindings: Query type kinds, derived types, and elements (tbennun sklam)
- PR #981: Add Instruction Namer pass to PassManager (tbennun)
- PR #993: Update changelog on main for 0.41.0 (esc)
- PR #1005: Remove suggestion that add_global_mapping() is unused (gmarkall)
- PR #1006: Release Notes 0.41.1 for main (esc)
- PR #1007: update release checklists post 0.41.1 (esc)
- PR #1009: Fix relocation overflows by implementing preallocation in the memory manager (gmarkall)
- PR #1010: Python 3.12 (esc)
- PR #1012: conda-recipe cleanups (esc)
- PR #1014: Fix conda-recipe syntax errors from #1012 (esc)
- PR #1017: add 3.12 to azure (esc)
- PR #1018: Bump minimum supported Python version to 3.9 (kc611)
- PR #1019: Add convergent as a supported FunctionAttribute and CallInstrAttribute. (diptorupd)

Authors:

- diptorupd
- esc

- gmarkall
- kc611
- sklam
- tbennun

3.5.4 v0.41.1 (October 17, 2023)

This is a maintenance release that includes a workaround in the test suite for ORCJit issues on the `aarch64` platform. Also, this is the last release to support the Windows 32-bit platform (`win32`).

Pull-Requests:

- PR #996: fix typos found by codespell (esc)
- PR #997: Fix issue #880 by ensuring all sources are compiled under FreeBSD. (ke6jjj)
- PR #998: adding sphinx_rtd_theme to RTD build to fix build (esc)
- PR #1001: Fix / workaround for OrcJIT blocking issues (gmarkall)

Authors:

- esc
- ke6jjj
- gmarkall

3.5.5 v0.41.0 (September 20, 2023)

Pull-Requests:

- PR #871: Refactor native library loading (folded sklam)
- PR #896: drop upper limit on Python for conda recipe (esc)
- PR #904: Create GitHub Action for llvmlite release (apmasell)
- PR #934: Expose TargetLibraryInfo pass (sklam)
- PR #935: Disable zlib for LLVM on Windows (apmasell)
- PR #936: Enable querying constants and value kinds (tbennun)
- PR #939: Bump llvmdev build number to include the nozlib change for windows (sklam)
- PR #940: Update CHANGE_LOG for 0.40.0 final. (stuartarchibald)
- PR #942: Add ORCJITv2 support (apmasell)
- PR #951: Add a type hint for *IntType.width* (apmasell)
- PR #952: Fix CI failing due to unsupported target triple on non-x86 platforms. (sklam)
- PR #958: fixup LLVM versions in version compat table (esc)
- PR #959: Remove support for LLVM < 14 (apmasell)
- PR #960: add various bullets to release checklists and sync (esc)
- PR #963: Allow adding comments to generated IR (apmasell)
- PR #966: build: support building on GNU/Hurd (pinotree)
- PR #967: Expose library name in OrcJIT tracker (apmasell)

- PR #968: Update LLVM manual build instructions (apmasell)
- PR #969: update changelog on main for v0.40.1 (esc)
- PR #983: adding RTD conf file V2 as per request (esc)
- PR #985: Update release checklist post 0.41.0rc1 (esc)
- PR #988: Fix FreeBsd build (sklam)

Authors:

- apmasell
- esc
- folded
- pinotree
- sklam
- stuartarchibald
- tbennun

3.5.6 v0.40.1 (June 21, 2023)

Pull-Requests:

- PR #945: Fix #944. Add *.argtypes* to prevent errors in pypy. (Siu Kwan Lam)
- PR #947: Update SVML patch for LLVM 14 (Andre Masella)
- PR #949: Handle PowerPC synonyms (Andre Masella)
- PR #950: Fix incorrect *byval* and other attributes on LLVM 14 (Andre Masella)

Authors:

- Andre Masella
- Siu Kwan Lam

3.5.7 v0.40.0 (May 1, 2023)

This release predominantly upgrades to LLVM 14 and Python 3.11. Bindings to a large number of passes are added. The minimum supported Python version is now Python 3.8.

Note: A bug was discovered in LLVM's RuntimeDyldELF on the Aarch64 platform that can cause segfaults when cross module symbols are linked. It is necessary for JIT users to build LLVM with the patch added in PR#926.

Pull-Requests:

- PR #827: Add more LLVM pass bindings (apmasell)
- PR #830: Add LLVM 14 support (apmasell)
- PR #860: the git tag for the RC needs an rc1 suffix (esc)
- PR #869: bump max Python version to 3.11 (esc sklam)
- PR #876: Remove `llvmlite.llvmpy` after deprecation (apmasell)
- PR #883: Adds support for calling functions with 'tail', 'notail', or 'musttail' markers. (bslatkin)
- PR #886: Simplify setup.py Python version guard (mbargull)
- PR #892: Bump minimum supported Python version to 3.8 (jamesobutler)

- PR #893: Upgrade to ubuntu-20.04 for azure pipeline CI (jamesobutler)
- PR #899: Run Minconda install with bash (gmarkall)
- PR #903: Fix flake8 config and style for flake8 6 (gmarkall)
- PR #905: Add YouCompleteMe configuration file and ignore vim swap files (gmarkall)
- PR #906: Replace importlib-resources legacy API use (sklam)
- PR #910: Aarch64 split build for LLVM14 (sklam)
- PR #921: Setup AzureCI to use py311 and llvm14 (sklam)
- PR #922: Fix AzureCI not using llvm14 on windows (sklam)
- PR #926: llvmdev recipe: Add patch that clears GOTOffsetMap (apmasell gmarkall sklam)
- PR #930: Update changelog for 0.40.0rc1 (sklam stuartarchibald)
- PR #931: Remove maximum Python version limit (sklam apmasell)
- PR #932: Fix wheel builds (sklam)
- PR #935: Disable zlib for LLVM on Windows (apmasell)
- PR #939: Bump llvmdev build number to include the nozlib change for windows (sklam)
- PR #940: Update CHANGE_LOG for 0.40.0 final. (stuartarchibald)

Authors:

- apmasell
- bslatkin
- esc
- gmarkall
- jamesobutler
- mbargull
- sklam
- stuartarchibald

3.5.8 v0.39.1 (September 1, 2022)

This is a maintenance release to fix build issues on MacOS.

Pull-Requests:

- PR #752: Skip test if libm is not found (Siu Kwan Lam)
- PR #865: Move Azure to use macos-11 (stuartarchibald)
- PR #874: Add zlib as a dependency for aarch64 (esc)
- PR #878: Update changelog (Andre Masella)

3.5.9 v0.39.0 (July 25, 2022)

This release predominantly adds new features and improves functionality.

- It's now possible to directly set LLVM `metadata` on global variables.
- Functions and global variables now support the specification of a `section` in which they should be placed.

- The attribute `source_file` had been added to the `ModuleRef` class, it returns the module's original file name.
- The FFI library binding to LLVM is now loaded with `importlib` to increase compatibility with other projects and improve start-up times.
- Linux builds now use the parallel option to `make` to speed up building the FFI library.
- Preliminary work to expose LLVM's optimization-remarks interface has been undertaken. The bindings are exposed and tested, but not yet documented for general use (additional work is needed).

Deprecations:

- The `llvmlite.llvmpy` module has been deprecated as the functionality it provides is available through the `llvmlite.ir` module. See the deprecation guide in the user documentation for details and recommendations regarding replacement.

Pull-Requests:

- PR #328: Build C files separately on Linux and support parallel make (Michał Górny)
- PR #754: manylinux2014 aarch64 wheels with system compilers (esc)
- PR #760: add support for attaching metadata to global variables (Graham Markall John Törnblom)
- PR #786: Update Windows and OSX CI images. (stuartarchibald)
- PR #801: Update ffi.py (franzhaas)
- PR #803: llvm::Module::GetSourceFileName (J. Aaron Pendergrass)
- PR #806: Bump to v0.39.0dev (esc)
- PR #807: Exclude ExecutionEngine tests on linux 32 (esc)
- PR #809: Update CHANGE_LOG for 0.38.0 (stuartarchibald)
- PR #813: Add m1 support to conda build scripts (esc Stan Seibert)
- PR #815: update local references (esc)
- PR #816: remove configuration landscape service as it is no longer used (esc)
- PR #817: remove uppper limit on Python requires (esc)
- PR #819: adding rc and final release checklist templates (esc)
- PR #823: Add section to globals (Andreas Wrisley)
- PR #824: add GitHub URL for PyPi (Andrii Oriekhov)
- PR #825: Add flag handling to more instructions. (stuartarchibald Andre Masella)
- PR #826: Deprecated *llvmlite.llvmpy* (Andre Masella)
- PR #831: Format C++ code (Andre Masella)
- PR #832: DOC: Fix the syntax for the llvmlite discourse topic link. (stuartarchibald)
- PR #835: Add pre-commit hooks for clang-format (Andre Masella)
- PR #837: Add support for optimization remarks in pass managers (Siu Kwan Lam Andre Masella)
- PR #846: Cherry-Pick: #842 –> *main* :Changelog for 0.38.1 (esc)
- PR #851: adding the llvm_11_consecutive_registers.patch (esc)
- PR #857: Delegate passmanager remarks methods (Andre Masella)
- PR #858: Update CHANGE_LOG for 0.39.0 (esc Graham Markall stuartarchibald)

- PR #863: Update changelog for 0.39.0 release (Siu Kwan Lam)
- PR #864: Update release date for 0.39.0 release. (stuartarchibald)
- PR #867: Update CHANGE_LOG 0.39.0 final. (stuartarchibald)

Authors:

- Andrii Oriekhov
- Andreas Wrisley
- Andre Masella
- esc
- franzhaas
- Graham Markall
- J. Aaron Pendergrass
- John Törnblom
- Michał Górny
- Stan Seibert
- Siu Kwan Lam
- stuartarchibald

3.5.10 v0.38.1 (May 19, 2022)

This is a maintenance release to support the Apple M1 architecture.

Pull-Requests:

- PR #841: Merge pull request #813 from seibert/m1_support (esc, Stan Seibert)
- PR #845: Backport #786 for 0.38.1 (stuartarchibald)

Authors:

- esc
- Stan Seibert
- stuartarchibald

3.5.11 v0.38.0 (January 13, 2022)

This release makes llvmlite compatible with Python 3.10. It also adds an `abiname` option to the target machine creation interface that mimics the same in LLVM. Further, a large number of functions are added to the IR API to support common uses of constant expressions. Finally, a number of bugs were fixed!

Pull-Requests:

- PR #702: Implement minimal support for call-site argument attributes (David Nadlinger)
- PR #739: Bump to 0.38.0 development series (Siu Kwan Lam)
- PR #751: Fix encoding of unicode string in metadata. (Siu Kwan Lam)
- PR #757: Add issue templates based on those in Numba (Graham Markall)
- PR #763: Update the Azure default linux image to Ubuntu 18.04 (stuartarchibald)
- PR #769: Python 3.10 (esc)

- PR #771: cleanup llvmlite metadata (esc)
- PR #774: Fix OSX breakage on master (esc)
- PR #775: targets.py: Add ABIName parameter for RISC-V hard float targets (Graham Markall occheung)
- PR #778: fix: test code typo (Itmom)
- PR #779: pin compiler toolchain to version 10 on osx (esc)
- PR #784: Adds _ConstOpMixin.ptrtoint to complement existing bitcast and inttoptr methods. (Brett Slatkin)
- PR #788: Add RISC-V + all default targets to LLVM build (Stan Seibert esc)
- PR #789: don't build docs on every platform, only once (esc)
- PR #791: update to use LLVM 11.1.0 build 4 (esc)
- PR #795: Continue PR #735 (Add support for more constant expressions) (Graham Markall John Törnblom)
- PR #797: Make tests that require cross compilation to RISCV optional. (stuartarchibald)

Authors:

- Itmom
- Brett Slatkin
- David Nadlinger
- esc
- Graham Markall
- John Törnblom
- occheung
- Stan Seibert
- Siu Kwan Lam
- stuartarchibald

3.5.12 v0.37.0 (August 19, 2021)

The biggest new feature in this release is LLVM 11 support for all platforms! This is important, because the `aarch64` platform had been 'stuck' on LLVM 9 for several releases. A special thanks to David Spickett from Linaro for supplying the relevant patch.

Pull-Requests:

- PR #711: Use conda for readthedocs with Python 3.9 (Antoine Pitrou esc)
- PR #715: LLVM 11.1.0 (Ivan Butygin esc)
- PR #726: Disallow alwaysinline and noinline on functions (Graham Markall)
- PR #730: Don't export LLVM symbols when linking statically (Chris Burr esc)
- PR #732: Docs/remove travis (esc)
- PR #734: bump llvmdev to 11 (esc)
- PR #736: manylinux llvmdev upgrade to LLVM 11 / manylinux2010 to manylinux2014 upgrade (esc)
- PR #747: Remove 3.6 Support (esc)
- PR #748: Fixup readme (esc)

- PR #749: adding PRs specific to the RC2 release (esc)
- PR #762: PR #758 continued. Ensure GCC7 is used on non-ARM platforms. (esc stuartarchibald)
- PR #764: Update CHANGE_LOG for 0.37.0 (stuartarchibald)

Authors:

- Ivan Butygin
- Chris Burr
- esc
- Graham Markall
- Antoine Pitrou
- Siu Kwan Lam
- stuartarchibald

3.5.13 v0.36.0 (March 11, 2021)

The main feature in this release is Python 3.9 support. Also, some changes to the build scripts were included: the llvmdev recipe now supports compilation on Windows 10 and llvmlite may now be compiled with cmake on POSIX systems. Additionally, both the timings from the LLVM passes themselves and the time spent within the LLVM thread-safe access lock are now available for diagnostic purposes. Lastly, several minor cosmetic changes to the documentation and project files are included.

Pull requests:

- PR #624: Expose pass timings
- PR #655: Switch encoding to UTF-8 from latin1
- PR #662: Delete `requirements.txt`
- PR #666: fix rst syntax in install docs
- PR #668: Modify cmake options to work with VS2019
- PR #670: Llvmdev windows 10
- PR #671: Python 3.9 support
- PR #673: use build 2 on windows
- PR #677: Support building with CMake on posix systems
- PR #682: slight rearrangement of intro
- PR #685: Added fneg instruction
- PR #687: Cleanup public CI configuration and badge
- PR #689: fixup azure badge to point at master branch
- PR #690: Callback to track when the llvm lock is acquired and released
- PR #699: adapt Python version clamp from Numba
- PR #701: Improve llvm not found error

Authors:

- Graham Markall
- John Kirkham

- Peter-Jan Gootzen
- Siu Kwan Lam (core dev)
- Stan Seibert (core dev)
- Stuart Archibald (core dev)
- Uwe L. Korn
- Valentin Haenel (core dev)
- `@ARF1`
- `@abebeos`
- `@0x5h4un`

3.5.14 v0.35.0 (November 30, 2020)

The main feature in this release is a Numba specific LLVM pass for pruning reference-count operations. We plan to generalize this custom LLVM pass once it is proven stable so that it can be configured for other uses. In addition, this release contains an updated SVML patch that fixes an issue for AVX512, and a patch that fixes build issues on Linux and BSD.

Pull requests:

- PR #617: Fix wheel building.
- PR #618: Fix changelog on master
- PR #626: Update SVML patch
- PR #627: Always build with -fPIC on Linux and BSD
- PR #633: Fix Issue #632 - obj file section contents truncated
- PR #634: Implement reference count pruner LLVM pass
- PR #635: Fixes #629: Default to 0 if None is passed as a value for Int32
- PR #637: Fix appveyor
- PR #640: Backport llvm refop pass for LLVM 9.
- PR #641: fix llvmdev build number
- PR #642: pin to correct version of llvmdev
- PR #644: Fix refprune fanout_raise case getting stuck on large graphs
- PR #647: Add support for general attrs on call and loop-rotate pass

Authors:

- Graham Markall
- Ivan Butygin
- NavyaZaveri
- Siu Kwan Lam (core dev)
- Stuart Archibald (core dev)
- Valentin Haenel (core dev)

3.5.15 v0.34.0 (August 12, 2020)

This release upgrades to LLVM 10 (10.0.1) for all platforms except *aarch64* which will remain at LLVM 9 (9.0.1).

Pull requests:

- PR #596: Use std::make_unique on LLVM 10 (Cherry-Pick via #599)
- PR #606: Revert "Fix CUDA with LLVM9" (Cherry-Pick via #599)
- PR #599: LLVM 10
- PR #598: Fix flake in setup.py
- PR #602: add missing targets to wheels
- PR #614: Fix wheel building
- PR #616: fix release date in changelog

Authors:

- Graham Markall
- Michał Górny
- Siu Kwan Lam (core dev)
- Stuart Archibald (core dev)
- Valentin Haenel (core dev)

3.5.16 v0.33.0 (June 10, 2020)

This release upgrades to LLVM 9 and drops support for older LLVM versions.

Pull requests:

- PR #593: Fix CUDA with LLVM9
- PR #592: Fix meta.yaml
- PR #591: buildscripts: Unpin wheel
- PR #590: add python_requires to setup.py
- PR #582: Adds override for LLVM version check, re-formats docs.
- PR #581: Add FAQ entry on LLVM version support.
- PR #580: Trove classifiers may be out of date.
- PR #577: llvmlite wheel building fixes
- PR #575: Update the release date
- PR #548: Upgrade to LLVM9
- PR #521: Allow instructions to be removed from blocks

Authors:

- Graham Markall
- Jan Vesely
- Siu Kwan Lam (core dev)
- Stuart Archibald (core dev)

- Tim Babb
- Valentin Haenel (core dev)

3.5.17 v0.32.1 (May 7, 2020)

This is a small patch release that addresses some packaging issues:

Pull requests:

- PR 580: Trove classifiers may be out of date.
- PR 581: Add FAQ entry on LLVM version support.
- PR 582: Adds override for LLVM version check, re-formats docs.

Authors:

- Stuart Archibald (core dev)
- Valentin Haenel (core dev)

3.5.18 v0.32.0 (Apr 16, 2020)

The main changes in this release are the removal of specific code for Python 2 and Python <3.6, and making the code base PEP8 compliant.

Pull requests:

- PR #577: llvmlite wheel building fixes
- PR #560: ENH: Better error message
- PR #558: update install docs
- PR #556: binding: Allow empty features list
- PR #555: travis: Cleanup
- PR #554: azure-pipelines: Bump VM images.
- PR #552: Add paragraph on installing from sdist and on non-traditional platforms.
- PR #551: Remove python 2, python < 3.6, fix up, add flake8
- PR #549: Miscalled method and missing parameter in the documentation
- PR #547: Permit building on Visual Studio 2017
- PR #543: Update error message in LLVM version check.
- PR #540: update to final release date for 0.31.0

Authors:

- Arik Funke
- Eric Larson
- Jan Vesely
- Shan Sikdar
- Siu Kwan Lam (core dev)
- Stan Seibert (core dev)
- Stuart Archibald (core dev)

- Vladislav Hrčka

3.5.19 v0.31.0 (Jan 2, 2020)

This release switches memset/memcpy to use the 4 argument style as per LLVM 7+ and updates some documentation.

Commits:

- PR #485: Revert "Revert "LLVM 7 changed memset intrinsic signature, adjust it""
- PR #520: Begin development of 0.31.0
- PR #528: Add cttz and ctlz to irbuilder docs.
- PR #533: Update deprecation docs with full deprecation of 5 arg memset/memcpy
- PR #535: Update docs to not report LLVM 3.8 as latest!

Authors:

- Isaac Virshup
- Siu Kwan Lam (core dev)
- Stuart Archibald (core dev)

3.5.20 v0.30.0 (Oct 9, 2019)

This release adds support for half-precision float and schedules the deprecation of memset/memcpy accepting 5 arguments (cf. LLVM change).

- PR #518: Fix use of -fPIC flag in wheels
- PR #513: Remove restriction on sphinx version from Anaconda distro
- PR #512: fix for block labels which contain "interesting" characters
- PR #511: Deprecate the use of memset/memcpy with alias
- PR #510: Add fp16 Intrinsics
- PR #509: Add Half-Precision Type
- PR #502: Add -fPIC flag for manylinux1 wheel building
- PR #491: Fix incorrect hierarchy in the documentation for ir.Constant.
- PR #474: Update docs
- PR #470: Fix leak on string returning APIs.

3.5.21 v0.29.0 (May 29, 2019)

This release upgrades to LLVM 8.0 for all supported platforms except PPC64LE. Due to numerous problems with LLVM 8.0 running on PPC64LE, we have decided to use LLVM 7.1, which is more stable on PPC64LE. In addition, non-host LLVM targets, AMDGPU, NVPTX, and WebAssembly, are enabled and they are available in our *llvmlite* builds.

- PR #484: Revert "LLVM 7 changed memset intrinsic signature, adjust it"
- PR #483: Depend on enum34 using PEP 508 environment markers
- PR #478: Upgrade to llvm8
- PR #469: Support loading from current directory and egg files
- PR #467: Add missing fastmath flags from LLVM 7

- PR #460: LLVM 7 changed memset intrinsic signature, adjust it

3.5.22 v0.28.0 (Mar 12, 2019)

This release adds a number of community contributed features, including support for vector types, as well as atomic loads and stores.

- PR #322: Adding Vector Type
- PR #389: Add symbols from static object files
- PR #417: Add support for atomic loads/stores
- PR #422: Normalize replace_* behaviour and add docs
- PR #426: Fix pickling of IR functions and add tests
- PR #444: Setup manylinux1 buildscripts and CI
- PR #446: Document need for -p1 argument to patch command
- PR #448: Fix "SyntaxWarning: invalid escape sequence d"
- PR #449: Consolidate the two vector type PRs
- PR #452: Some adjustments to pr426
- PR #454: Truncate long label names when adding label suffix.
- PR #458: Add changelog info about v0.27.1

3.5.23 v0.27.1 (Feb 1, 2019)

Bugfix release for invalid wheel hash for OSX packages. No change to source code.

3.5.24 v0.27.0 (Dec 28, 2018)

This release updates llvmlite to LLVM 7. Note that LLVM 7.0.0 contains a critical bug that is resolved with a patch included in the llvmdev conda package recipe. The final release of LLVM 7.0.1 may also resolve the issue.

- PR #434: Add another thread for RPi builds.
- PR #430: llvm lld integration, merge #428
- PR #428: Build LLD as part of the llvmdev package
- PR #413: Set up CI with Azure Pipelines
- PR #412: LLVM 7 support

3.5.25 v0.26.0 (Nov 27, 2018)

The primary new features in this release is support for generation of Intel JIT events, which makes profiling of JIT compiled code in Intel VTune possible. This release also contains some minor build improvements for ARMv7, and some small fixes.

LLVM 7 support was originally slated for this release, but had to be delayed after some issues arose in testing. LLVM 6 is still required for llvmlite.

- PR #409: Use native cmake on armv7l
- PR #407: Throttle thread count for llvm build on armv7l.
- PR #403: Add shutdown detection to ObjectRef __del__ method.

- PR #400: conda recipe: add make as build dep
- PR #399: Add get_element_offset to TargetData
- PR #398: Fix gep method call on Constant objects
- PR #395: Fix typo in irbuilder documentation
- PR #394: Enable IntelJIT events for LLVM for VTune support

3.5.26 v0.25.0 (Sep 18, 2018)

This release adds support for the FMA instruction, and has some documentation and build improvements. Starting with this release, we are including ARMv8 (AArch64) testig in our CI process.

- PR #391: Fix broken win32 py2.7 build.
- PR #387: protect against empty features in list
- PR #384: Read CMAKE_GENERATOR which conda-build sets
- PR #382: rewrite of install instructions, calling out LLVM build challenges
- PR #380: Add FMA intrinsic support
- PR #379: ARM aarch64 test on jetson tx2
- PR #378: add slack, drop flowdock

3.5.27 v0.24.0 (Jul 6, 2018)

This release adds support for Python 3.7 and fixes some build issues. It also contains an updated SVML patch for the llvmdev package that works around some vectorization issues. It also adds a selective LLVM 6.0.1 llvmdev build for the *ppc64le* architecture.

- PR #374: Fix up broken patch selector
- PR #373: Add long description from readme
- PR #371: LLVM 6.0.1 build based on RC and fixes for PPC64LE
- PR #369: Recipe fixes for Conda Build 3
- PR #363: Workaround for incorrect vectorization factor computed for SVML functions
- PR #356: fix build on OpenBSD.
- PR #351: Python 3.7 compat: Properly escape repl in re.sub

3.5.28 v0.23.2 (Jun 1, 2018)

This is a bug fix release to assist in addressing a critical Numba issue that can affect users who download llvmlite packages from sources other than PyPI (pip), Anaconda, or Intel Python: https://github.com/numba/numba/issues/3006

Support for SVML is now detected at compile time and baked into a function that is exposed by llvmlite. This function can be queried at runtime to find out if SVML is supported by the LLVM that llvmlite was compiled against, code generation paths can then be adjusted accordingly.

The following PRs are closed in this release:

- PR #361: Add SVML detection and a function to declare support.

3.5.29 v0.23.1 (May 17, 2018)

This is a minor patch release that includes no code changes. It is issued to fix a couple of problems with the build recipes for llvmdev (on which llvmlite relies).

The following PRs are closed in this release:

- PR #353: PR Fix llvmdev build recipe.
- PR #348: llvmdev: enhancements to conda recipe

3.5.30 v0.23.0 (Apr 24, 2018)

In this release, we upgrade to LLVM 6. Two LLVM patches are added:

1. A patch to fix LLVM bug (https://bugs.llvm.org/show_bug.cgi?id=37019) that causes undefined behavior during CFG printing.
2. A patch to enable Intel SVML auto-vectorization of transcendentals.

The following PRs are closed in this release:

- PR #343: Fix undefined behavior bug due to Twine usage in LLVM
- PR #340: This moves llvmlite to use LLVM 6.0.0 as its backend.
- PR #339: Add cttz & ctlz
- PR #338: Add 3 Bit Manipulation Intrinsics
- PR #330: Add support for LLVM fence instruction
- **PR #326: Enable Intel SVML-enabled auto-vectorization for all the** transcendentals

3.5.31 v0.22.0 (Feb 15, 2018)

In this release, we have changed the locking strategy that protects LLVM from race conditions. Before, the llvmlite user (like Numba) was responsible for this locking. Now, llvmlite imposes a global thread lock on all calls into the LLVM API. This should be significantly less error prone. Future llvmlite releases may manually exempt some functions from locking once they are determined to be thread-safe.

The following PRs are closed in this release:

- PR#318: Ensuring threadsafety in concurrent usage of LLVM C-API
- PR#221: Add all function/return value attributes from LLVM 3.9
- PR#304: Expose support for static constructors/destructors

3.5.32 v0.21.0 (Dec 6, 2017)

In this release, we upgrade to LLVM 5. Our build scripts now use conda-build 3. For our prebuilt binaries, GCC 7 toolchain is used on unix-like systems and the OSX minimum deployment target is 10.9.

The following PRs are closed in this release:

- PR #315: Updates for conda build 3.
- PR #307: Fixes for LLVM5.
- PR #306: Working towards LLVM 5.0 support.

3.5.33 v0.20.0 (Sep 6, 2017)

Beginning with this minor release, we support wheels for Linux, OSX and Windows. Pull requests related to enabling wheels are #294, #295, #296 and #297. There are also fixes to the documentation (#283 and #289).

3.5.34 v0.19.0 (Jul 6, 2017)

This is a minor release with the following fixes.

- PR #281, Issue #280: Fix GEP addrspace issue
- PR #279: Fix #274 addrspace in gep
- PR #278: add Readthedocs badge
- PR #275: Add variables to pass through when doing conda-build
- PR #273: Fix the behavior of module.get_global
- PR #272: cmpop contains comparison type, not lhs
- PR #268, Fix #267: Support packed struct

The following are CI build related changes:

- PR #277: Add pass through gcc flags for llvmdev
- PR #276: Remove jenkins build scripts

3.5.35 v0.18.0 (May 4, 2017)

This is a minor release that fixes several issues (#263, #262, #258, #237) with the wheel build. In addition, we have minor fixes for running on PPC64LE platforms (#261). And, we added CI testing against PyPy (#253).

3.5.36 v0.17.1 (Apr 12, 2017)

This is a bugfix release that addresses issue #258 that our LLVM binding shared library is missing from the wheel builds.

3.5.37 v0.17.0 (Apr 9, 2017)

In this release, we are upgrading to LLVM 4.0. We are also starting to provide wheel packages for 64-bit Linux platforms (manylinux).

Fixes:

- Issue #249, PR #250: Disable static linking of libstdc++ by default.

Enhancements:

- PR #246: Add requirements.txt for pip dependency resolving
- PR #238: LLVM 4.0
- PR #222: Enable wheel builds

3.5.38 v0.16.0 (Feb 16, 2017)

API changes:

- Switched from LLVM 3.8 to 3.9
- `TargetData.add_pass` is removed in LLVM 3.9.

Enhancements:

- PR #239: Enable fastmath flags
- PR #233: Updates for llvm3.9.1
- PR #199: Update for changes in LLVM 3.9

Fixes:

- PR #236: Fix metadata with long value
- PR #231: Fix setup.py for Python2.7 so that pip auto installs dependencies
- PR #226: Fix get_host_cpu_features() so that it fails properly

3.5.39 v0.15.0 (Dec 20, 2016)

Enhancements:

- PR #213: Add partial LLVM bindings for ObjectFile.
- PR #215: Add inline assembly helpers in the builder.
- PR #216: Allow specifying alignment in alloca instructions.
- PR #219: Remove unnecessary verify in module linkage.

Fixes:

- PR #209, Issue #208: Fix overly restrictive test for library filenames.

3.5.40 v0.14.0 (Oct 17, 2016)

Enhancements:

- PR #104: Add binding to get and view function control-flow graph.
- PR #210: Improve llvmdev recipe.
- PR #212: Add initializer for the native assembly parser.

3.5.41 v0.13.0 (Aug 24, 2016)

Enhancements:

- PR #176: Switch from LLVM 3.7 to LLVM 3.8.
- PR #191: Allow setting the alignment of a global variable.
- PR #198: Add missing function attributes.
- PR #160: Escape the contents of metadata strings, to allow embedding any characters.
- PR #162: Add support for creating debug information nodes.
- PR #200: Improve the usability of metadata emission APIs.
- PR #200: Allow calling functions with metadata arguments (such as `@llvm.dbg.declare`).

Fixes:

- PR #190: Suppress optimization remarks printed out in some cases by LLVM.
- PR #200: Allow attaching metadata to a `ret` instruction.

3.5.42 v0.12.1 (Jul 12, 2016)

New release to fix packages on PyPI. Same as v0.12.0.

3.5.43 v0.12.0 (Jul 6, 2016)

Enhancements:

- PR #179: Let llvmlite build on armv7l Linux.
- PR #161: Allow adding metadata to functions.
- PR #163: Allow emitting fast-math `fcmp` instructions.
- PR #159: Allow emitting verbose assembly in TargetMachine.

Fixes:

- Issue #177: Make setup.py compatible with `pip install`.

3.5.44 v0.11.0 (May 19, 2016)

Enhancements:

- PR #175: Check LLVM version at build time
- PR #169: Default initializer for non-external global variable
- PR #168: add ir.Constant.literal_array()

3.5.45 v0.10.0 (Mar 31, 2016)

Enhancements:

- PR #146: Improve `setup.py clean` to wipe more leftovers.
- PR #135: Remove some llvmpy compatibility APIs.
- PR #151: Always copy TargetData when adding to a pass manager.
- PR #148: Make errors more explicit on loading the binding DLL.
- PR #144: Allow overriding `-flto` in Linux builds.
- PR #136: Remove Python 2.6 and 3.3 compatibility.
- Issue #131: Allow easier creation of constants by making type instances callable.
- Issue #130: The test suite now ensures the runtime DLL dependencies are within a certain expected set.
- Issue #121: Simplify build process on Unix and remove hardcoded linking with LLVMOProfileJIT.
- Issue #125: Speed up formatting of raw array constants.

Fixes:

- PR #155: Properly emit IR for metadata null.
- PR #153: Remove deprecated uses of `TargetMachine::getDataLayout()`.
- PR #156: Move personality from LandingPadInstr to FunctionAttributes. It was moved in LLVM 3.7.
- PR #149: Implement LLVM scoping correctly.
- PR #141: Ensure no CMakeCache.txt file is included in sdist.
- PR #132: Correct constant in `llvmir.py` example.

3.5.46 v0.9.0 (Feb 29, 2016)

Enhancements:

- PR #73: Add get_process_triple() and get_host_cpu_features()
- Switch from LLVM 3.6 to LLVM 3.7. The generated IR for some memory operations has changed.
- Improved performance of IR serialization.
- Issue #116: improve error message when the operands of a binop have differing types.
- PR #113: Let Type.get_abi_{size,alignment} not choke on identified types.
- PR #112: Support non-alphanumeric characters in type names.

Fixes:

- Remove the libcurses dependency on Linux.

3.5.47 v0.8.0 (Oct 23, 2015)

- Update LLVM to 3.6.2
- Add an *align* parameter to IRBuilder.load() and IRBuilder.store().
- Allow setting visibility, DLL storageclass of ValueRef
- Support profiling with OProfile

3.5.48 v0.7.0 (Aug 31, 2015)

- PR #88: Provide hooks into the MCJIT object cache
- PR #87: Add indirect branches and exception handling APIs to ir.Builder.
- PR #86: Add ir.Builder APIs for integer arithmetic with overflow
- Issue #76: Fix non-Windows builds when LLVM was built using CMake
- Deprecate .get_pointer_to_global() and add .get_function_address() and .get_global_value_address() in ExecutionEngine.

3.5.49 v0.6.0 (Jun 30, 2015)

Enhancements:

- Switch from LLVM 3.5 to LLVM 3.6. The generated IR for metadata nodes has slightly changed, and the "old JIT" engine has been remove (only MCJIT is now available).
- Add an optional flags argument to arithmetic instructions on IRBuilder.
- Support attributes on the return type of a function.

3.5.50 v0.5.1 (Jun 8, 2015)

Fixes:

- Fix implicit termination of basic block in nested if_then()

3.5.51 v0.5.0 (May 27, 2015)

New documentation hosted at http://llvmlite.pydata.org

Enhancements:

- Add code-generation helpers from numba.cgutils
- Support for memset, memcpy, memmove intrinsics

Fixes:

- Fix string encoding problem when round-triping parse_assembly()

3.5.52 v0.4.0 (Mar 30, 2015)

Enhancements:

- Add Module.get_global()
- Renamd Module.global_variables to Module.global_values
- Support loading library parmanently
- Add Type.get_abi_alignment()

Fixes:

- Expose LLVM version as a tuple

Patched LLVM 3.5.1: Updated to 3.5.1 with the same ELF relocation patched for v0.2.2.

3.5.53 v0.2.2 (Jan 29, 2015)

Enhancements:

- Support for addrspacescast
- Support for tail call, calling convention attribute
- Support for IdentifiedStructType

Fixes:

- GEP addrspace propagation
- Various installation process fixes

Patched LLVM 3.5: The binaries from the numba binstar channel use a patched LLVM3.5 for fixing a LLVM ELF relocation bug that is caused by the use of 32-bit relative offset in 64-bit binaries. The problem appears to occur more often on hardened kernels, like in CentOS. The patched source code is available at: https://github.com/numba/llvm-mirror/releases/tag/3.5p1

3.5.54 v0.2.0 (Dec 16, 2014)

This is the first official release. It contains a few feature additions and bug fixes. It meets all requirements to replace llvmpy in numba and numbapro.

3.5.55 v0.1.0 (Nov 13, 2014)

This is the first release. This is released for beta testing llvmlite and numba before the official release.

3.6 Glossary

- *Basic block*
- *Function declaration*
- *Function definition*
- *getelementptr*
- *Global value*
- *Global variable*
- *Instruction*
- *Intermediate representation (IR)*
- *Label*
- *Metadata*
- *Module*
- *Terminator, terminator instruction*

3.6.1 Basic block

A sequence of instructions inside a function. A basic block always starts with a *Label* and ends with a *terminator*. No other instruction inside the basic block can transfer control out of the block.

3.6.2 Function declaration

The specification of a function's prototype without an associated implementation. A declaration includes the argument types, return types and other information such as the calling convention. This is like an `extern` function declaration in C.

3.6.3 Function definition

A function's prototype, as in a *Function declaration*, plus a body implementing the function.

3.6.4 getelementptr

An LLVM *Instruction* that lets you get the address of a subelement of an aggregate data structure.

See the getelementptr instruction in the official LLVM documentation.

3.6.5 Global value

A named value accessible to all members of a module.

3.6.6 Global variable

A variable whose value is accessible to all members of a module. It is a constant pointer to a module-allocated slot of the given type.

All global variables are global values. However, the opposite is not true—function declarations and function definitions are not global variables, they are only *global values*.

3.6.7 Instruction

The fundamental element used in implementing an LLVM function. LLVM instructions define a procedural, assembly-like language.

3.6.8 Intermediate representation (IR)

High-level assembly-language code describing to LLVM the program to be compiled to native code.

3.6.9 Label

A branch target inside a function. A label always denotes the start of a *Basic block*.

3.6.10 Metadata

Optional information associated with LLVM instructions, functions and other code. Metadata provides information that is not critical to the compiling of an LLVM *intermediate representation*, such as the likelihood of a condition branch or the source code location corresponding to a given instruction.

3.6.11 Module

A compilation unit for LLVM *intermediate representation*. A module can contain any number of function declarations and definitions, global variables and metadata.

3.6.12 Terminator, terminator instruction

A kind of *Instruction* that explicitly transfers control to another part of the program instead of going to the next instruction after it is executed. Examples are branches and function returns.

PYTHON MODULE INDEX

l

llvmlite.binding, 33
llvmlite.ir, 11

INDEX

Symbols

_ConstOpMixin (*class in llvmlite.ir*), 15
__call__() (*llvmlite.ir.Type method*), 12
__str__() (*llvmlite.binding.TypeRef method*), 43

A

add() (*llvmlite.ir._ConstOpMixin method*), 15
add() (*llvmlite.ir.IRBuilder method*), 26
add() (*llvmlite.ir.NamedMetaData method*), 18
add_aa_eval_pass(), 47
add_analysis_passes() (*llvmlite.binding.TargetMachine method*), 36
add_attribute() (*llvmlite.ir.Argument method*), 17
add_case() (*llvmlite.ir.SwitchInstr method*), 21
add_clause() (*llvmlite.ir.LandingPad method*), 21
add_debug_info() (*llvmlite.ir.Module method*), 22
add_destination() (*llvmlite.ir.IndirectBranch method*), 21
add_global() (*llvmlite.ir.Module method*), 22
add_incoming() (*llvmlite.ir.PhiInstr method*), 21
add_instruction_combine_pass(), 47
add_jump_threading_pass(), 47
add_loop_rotate_pass(), 47
add_loop_unroll_pass(), 47
add_metadata() (*llvmlite.ir.Module method*), 22
add_module() (*llvmlite.binding.ExecutionEngine method*), 44
add_named_metadata() (*llvmlite.ir.Module method*), 22
add_object_file() (*llvmlite.binding.ExecutionEngine method*), 44
add_refprune_pass(), 47
add_simplify_cfg_pass(), 47
add_symbol() (*in module llvmlite.binding*), 34
add_verifier() (*llvmlite.binding.ModulePassManager method*), 46
address_of_symbol() (*in module llvmlite.binding*), 34
addrspace (*llvmlite.ir.PointerType attribute*), 13
addrspacecast() (*llvmlite.ir.IRBuilder method*), 28
Aggregate (*class in llvmlite.ir*), 13
align (*llvmlite.ir.GlobalVariable attribute*), 19
alloca() (*llvmlite.ir.IRBuilder method*), 29
and_() (*llvmlite.ir._ConstOpMixin method*), 15
and_() (*llvmlite.ir.IRBuilder method*), 27
append_basic_block() (*llvmlite.ir.Function method*), 20
append_basic_block() (*llvmlite.ir.IRBuilder method*), 24
args (*llvmlite.ir.Function attribute*), 20
Argument (*class in llvmlite.ir*), 17
arguments (*llvmlite.binding.ValueRef attribute*), 41
ArrayType (*class in llvmlite.ir*), 13
as_bitcode() (*llvmlite.binding.ModuleRef method*), 38
as_ir() (*llvmlite.binding.TypeRef method*), 43
as_pointer() (*llvmlite.ir.Type method*), 11
ashr() (*llvmlite.ir._ConstOpMixin method*), 15
ashr() (*llvmlite.ir.IRBuilder method*), 26
asm() (*llvmlite.ir.IRBuilder method*), 32
assume() (*llvmlite.ir.IRBuilder method*), 32
atomic_rmw() (*in module llvmlite.ir*), 30
attributes (*llvmlite.binding.ValueRef attribute*), 41
attributes (*llvmlite.ir.Function attribute*), 20

B

basic_block (*llvmlite.ir.BlockAddress attribute*), 18
bitcast() (*llvmlite.ir._ConstOpMixin method*), 16
bitcast() (*llvmlite.ir.IRBuilder method*), 28
Block (*class in llvmlite.ir*), 18
block (*llvmlite.binding.ValueRef attribute*), 41
block (*llvmlite.ir.IRBuilder attribute*), 24
BlockAddress (*class in llvmlite.ir*), 18
blocks (*llvmlite.binding.ValueRef attribute*), 41
branch() (*llvmlite.ir.IRBuilder method*), 30
branch_indirect() (*llvmlite.ir.IRBuilder method*), 31

C

call() (*llvmlite.ir.IRBuilder method*), 30
calling_convention (*llvmlite.ir.Function attribute*), 20
CatchClause (*class in llvmlite.ir*), 22
cbranch() (*llvmlite.ir.IRBuilder method*), 30
check_jit_execution() (*in module llvmlite.binding*), 43
cmpxchg() (*in module llvmlite.ir*), 29
comment() (*llvmlite.ir.IRBuilder method*), 32

Constant (*class in llvmlite.ir*), 17
create_mcjit_compiler() (*in module llvmlite.binding*), 43
create_new_function_pass_manager() (*in module llvmlite.binding*), 47
create_new_module_pass_manager() (*in module llvmlite.binding*), 47
create_target_data() (*in module llvmlite.binding*), 35
create_target_machine() (*llvmlite.binding.Target method*), 36
ctlz() (*llvmlite.ir.IRBuilder method*), 26
cttz() (*llvmlite.ir.IRBuilder method*), 26

D

data_layout (*llvmlite.binding.ModuleRef attribute*), 38
data_layout (*llvmlite.ir.Module attribute*), 23
debug_metadata (*llvmlite.ir.IRBuilder attribute*), 24
description (*llvmlite.binding.Target attribute*), 36
disable_unroll_loops (*llvmlite.binding.PassManagerBuilder attribute*), 47
DIToken (*class in llvmlite.ir*), 18
DIValue (*class in llvmlite.ir*), 18
DoubleType (*class in llvmlite.ir*), 12

E

element_count (*llvmlite.binding.TypeRef attribute*), 43
element_type (*llvmlite.binding.TypeRef attribute*), 43
elements (*llvmlite.binding.TypeRef attribute*), 43
elements (*llvmlite.ir.Aggregate attribute*), 13
emit_assembly() (*llvmlite.binding.TargetMachine method*), 36
emit_object() (*llvmlite.binding.TargetMachine method*), 36
ExecutionEngine (*class in llvmlite.binding*), 44
extract_element() (*llvmlite.ir.IRBuilder method*), 29
extract_value() (*llvmlite.ir.IRBuilder method*), 29

F

fadd() (*llvmlite.ir._ConstOpMixin method*), 15
fadd() (*llvmlite.ir.IRBuilder method*), 27
fcmp_ordered() (*llvmlite.ir._ConstOpMixin method*), 16
fcmp_ordered() (*llvmlite.ir.IRBuilder method*), 28
fcmp_unordered() (*llvmlite.ir._ConstOpMixin method*), 16
fcmp_unordered() (*llvmlite.ir.IRBuilder method*), 28
fdiv() (*llvmlite.ir._ConstOpMixin method*), 16
fdiv() (*llvmlite.ir.IRBuilder method*), 27
FeatureMap (*class in llvmlite.binding*), 37
FilterClause (*class in llvmlite.ir*), 22
finalize() (*llvmlite.binding.FunctionPassManager method*), 49
finalize_object() (*llvmlite.binding.ExecutionEngine method*), 44
flatten() (*llvmlite.binding.FeatureMap method*), 37
FloatType (*class in llvmlite.ir*), 12
fmul() (*llvmlite.ir._ConstOpMixin method*), 15
fmul() (*llvmlite.ir.IRBuilder method*), 27
fneg() (*llvmlite.ir.IRBuilder method*), 27
fpext() (*llvmlite.ir._ConstOpMixin method*), 16
fpext() (*llvmlite.ir.IRBuilder method*), 27
fptosi() (*llvmlite.ir._ConstOpMixin method*), 16
fptosi() (*llvmlite.ir.IRBuilder method*), 27
fptoui() (*llvmlite.ir._ConstOpMixin method*), 16
fptoui() (*llvmlite.ir.IRBuilder method*), 28
fptrunc() (*llvmlite.ir._ConstOpMixin method*), 16
fptrunc() (*llvmlite.ir.IRBuilder method*), 27
frem() (*llvmlite.ir._ConstOpMixin method*), 16
frem() (*llvmlite.ir.IRBuilder method*), 27
from_default_triple() (*llvmlite.binding.Target class method*), 36
from_triple() (*llvmlite.binding.Target class method*), 36
fsub() (*llvmlite.ir._ConstOpMixin method*), 15
fsub() (*llvmlite.ir.IRBuilder method*), 27
Function (*class in llvmlite.ir*), 20
function (*llvmlite.binding.ValueRef attribute*), 41
function (*llvmlite.ir.Block attribute*), 18
function (*llvmlite.ir.BlockAddress attribute*), 18
function (*llvmlite.ir.Instruction attribute*), 21
function (*llvmlite.ir.IRBuilder attribute*), 24
FunctionPassManager (*class in llvmlite.binding*), 46
functions (*llvmlite.binding.ModuleRef attribute*), 38
functions (*llvmlite.ir.Module attribute*), 23
FunctionType (*class in llvmlite.ir*), 14

G

gep() (*llvmlite.ir.Constant method*), 17
gep() (*llvmlite.ir.IRBuilder method*), 29
get_abi_alignment() (*llvmlite.binding.TargetData method*), 35
get_abi_alignment() (*llvmlite.ir.Type method*), 11
get_abi_size() (*llvmlite.binding.TargetData method*), 35
get_abi_size() (*llvmlite.ir.Type method*), 11
get_constant_value() (*llvmlite.binding.ValueRef method*), 42
get_default_triple() (*in module llvmlite.binding*), 35
get_element_offset() (*llvmlite.binding.TargetData method*), 35
get_element_offset() (*llvmlite.ir.Type method*), 12
get_function() (*llvmlite.binding.ModuleRef method*), 38
get_function_address() (*llvmlite.binding.ExecutionEngine method*), 44

get_function_cfg() (*in module llvmlite.binding*), 49
get_global() (*llvmlite.ir.Module method*), 23
get_global_value_address() (*llvmlite.binding.ExecutionEngine method*), 44
get_global_variable() (*llvmlite.binding.ModuleRef method*), 38
get_host_cpu_features() (*in module llvmlite.binding*), 35
get_host_cpu_name() (*in module llvmlite.binding*), 35
get_named_metadata() (*llvmlite.ir.Module method*), 23
get_object_format() (*in module llvmlite.binding*), 35
get_pointee_abi_alignment() (*llvmlite.binding.TargetData method*), 35
get_pointee_abi_size() (*llvmlite.binding.TargetData method*), 35
get_process_triple() (*in module llvmlite.binding*), 35
get_struct_type() (*llvmlite.binding.ModuleRef method*), 38
get_unique_name() (*llvmlite.ir.Module method*), 23
getFunctionPassManager() (*llvmlite.binding.PassBuilder method*), 46
getModulePassManager() (*llvmlite.binding.PassBuilder method*), 46
global_constant (*llvmlite.ir.GlobalVariable attribute*), 19
global_values (*llvmlite.ir.Module attribute*), 23
global_variables (*llvmlite.binding.ModuleRef attribute*), 38
GlobalValue (*class in llvmlite.ir*), 19
GlobalVariable (*class in llvmlite.ir*), 19
goto_block() (*llvmlite.ir.IRBuilder method*), 25
goto_entry_block() (*llvmlite.ir.IRBuilder method*), 25

H

HalfType (*class in llvmlite.ir*), 12

I

icmp_signed() (*llvmlite.ir._ConstOpMixin method*), 16
icmp_signed() (*llvmlite.ir.IRBuilder method*), 28
icmp_unsigned() (*llvmlite.ir._ConstOpMixin method*), 16
icmp_unsigned() (*llvmlite.ir.IRBuilder method*), 28
IdentifiedStructType (*class in llvmlite.ir*), 13
if_else() (*llvmlite.ir.IRBuilder method*), 25
if_then() (*llvmlite.ir.IRBuilder method*), 25
incoming_blocks (*llvmlite.binding.ValueRef attribute*), 41
IndirectBranch (*class in llvmlite.ir*), 21
initialize() (*in module llvmlite.binding*), 34
initialize() (*llvmlite.binding.FunctionPassManager method*), 49
initialize_all_asmprinters() (*in module llvmlite.binding*), 34
initialize_all_targets() (*in module llvmlite.binding*), 34
initialize_native_asmparser() (*in module llvmlite.binding*), 34
initialize_native_asmprinter() (*in module llvmlite.binding*), 34
initialize_native_target() (*in module llvmlite.binding*), 34
initializer (*llvmlite.ir.GlobalVariable attribute*), 19
inlining_threshold (*llvmlite.binding.PassManagerBuilder attribute*), 47
insert_basic_block() (*llvmlite.ir.Function method*), 20
insert_element() (*llvmlite.ir.IRBuilder method*), 29
insert_value() (*llvmlite.ir.IRBuilder method*), 29
Instruction (*class in llvmlite.ir*), 20
instruction (*llvmlite.binding.ValueRef attribute*), 41
instructions (*llvmlite.binding.ValueRef attribute*), 41
inttoptr() (*llvmlite.ir._ConstOpMixin method*), 17
inttoptr() (*llvmlite.ir.IRBuilder method*), 28
IntType (*class in llvmlite.ir*), 12
invoke() (*llvmlite.ir.IRBuilder method*), 31
IRBuilder (*class in llvmlite.ir*), 24
is_argument (*llvmlite.binding.ValueRef attribute*), 42
is_array (*llvmlite.binding.TypeRef attribute*), 42
is_block (*llvmlite.binding.ValueRef attribute*), 42
is_constant (*llvmlite.binding.ValueRef attribute*), 42
is_declaration (*llvmlite.binding.ValueRef attribute*), 41
is_declaration (*llvmlite.ir.Function attribute*), 20
is_function (*llvmlite.binding.ValueRef attribute*), 42
is_function_vararg (*llvmlite.binding.TypeRef attribute*), 43
is_global (*llvmlite.binding.ValueRef attribute*), 42
is_instruction (*llvmlite.binding.ValueRef attribute*), 42
is_operand (*llvmlite.binding.ValueRef attribute*), 42
is_pointer (*llvmlite.binding.TypeRef attribute*), 42
is_struct (*llvmlite.binding.TypeRef attribute*), 42
is_terminated (*llvmlite.ir.Block attribute*), 18
is_vector (*llvmlite.binding.TypeRef attribute*), 42

L

LabelType (*class in llvmlite.ir*), 14
LandingPad (*class in llvmlite.ir*), 21
landingpad() (*llvmlite.ir.IRBuilder method*), 31
link_in() (*llvmlite.binding.ModuleRef method*), 38
Linkage (*class in llvmlite.binding*), 39
linkage (*llvmlite.binding.ValueRef attribute*), 41
linkage (*llvmlite.ir.GlobalValue attribute*), 19
Linkage.appending (*in module llvmlite.binding*), 39

Linkage.available_externally (*in module llvmlite.binding*), 39
Linkage.common (*in module llvmlite.binding*), 39
Linkage.dllexport (*in module llvmlite.binding*), 39
Linkage.dllimport (*in module llvmlite.binding*), 39
Linkage.external (*in module llvmlite.binding*), 39
Linkage.external_weak (*in module llvmlite.binding*), 39
Linkage.ghost (*in module llvmlite.binding*), 39
Linkage.internal (*in module llvmlite.binding*), 39
Linkage.linker_private (*in module llvmlite.binding*), 39
Linkage.linker_private_weak (*in module llvmlite.binding*), 39
Linkage.linkonce_any (*in module llvmlite.binding*), 39
Linkage.linkonce_odr (*in module llvmlite.binding*), 39
Linkage.linkonce_odr_autohide (*in module llvmlite.binding*), 39
Linkage.private (*in module llvmlite.binding*), 39
Linkage.weak_any (*in module llvmlite.binding*), 39
Linkage.weak_odr (*in module llvmlite.binding*), 39
literal_array() (*llvmlite.ir.Constant class method*), 17
literal_struct() (*llvmlite.ir.Constant class method*), 17
LiteralStructType (*class in llvmlite.ir*), 13
llvm_version_info (*in module llvmlite.binding*), 34
LLVMContextRef (*built-in class*), 37
llvmlite.binding
    module, 33
llvmlite.ir
    module, 11
load() (*llvmlite.ir.IRBuilder method*), 29
load_atomic() (*llvmlite.ir.IRBuilder method*), 29
load_library_permanently() (*in module llvmlite.binding*), 34
load_reg() (*llvmlite.ir.IRBuilder method*), 32
loop_interleaving (*llvmlite.binding.PipelineTuningOptions attribute*), 46
loop_unrolling (*llvmlite.binding.PipelineTuningOptions attribute*), 46
loop_vectorization (*llvmlite.binding.PipelineTuningOptions attribute*), 46
loop_vectorize (*llvmlite.binding.PassManagerBuilder attribute*), 47
lshr() (*llvmlite.ir._ConstOpMixin method*), 15
lshr() (*llvmlite.ir.IRBuilder method*), 26

M

MDValue (*class in llvmlite.ir*), 18
MetaDataString (*class in llvmlite.ir*), 18
MetaDataType (*class in llvmlite.ir*), 14
module
    llvmlite.binding, 33
    llvmlite.ir, 11
Module (*class in llvmlite.ir*), 22
module (*llvmlite.binding.ValueRef attribute*), 41
module (*llvmlite.ir.Instruction attribute*), 21
module (*llvmlite.ir.IRBuilder attribute*), 24
ModulePassManager (*class in llvmlite.binding*), 46
ModuleRef (*class in llvmlite.binding*), 38
mul() (*llvmlite.ir._ConstOpMixin method*), 15
mul() (*llvmlite.ir.IRBuilder method*), 26

N

name (*llvmlite.binding.ModuleRef attribute*), 38
name (*llvmlite.binding.Target attribute*), 36
name (*llvmlite.binding.TypeRef attribute*), 42
name (*llvmlite.binding.ValueRef attribute*), 41
NamedMetaData (*class in llvmlite.ir*), 18
neg() (*llvmlite.ir._ConstOpMixin method*), 15
neg() (*llvmlite.ir.IRBuilder method*), 27
not_() (*llvmlite.ir.IRBuilder method*), 27

O

ObjectFileRef (*class in llvmlite.binding*), 44
opcode (*llvmlite.binding.ValueRef attribute*), 41
operands (*llvmlite.binding.ValueRef attribute*), 41
opt_level (*llvmlite.binding.PassManagerBuilder attribute*), 47
or_() (*llvmlite.ir._ConstOpMixin method*), 15
or_() (*llvmlite.ir.IRBuilder method*), 27

P

parse_assembly() (*in module llvmlite.binding*), 37
parse_bitcode() (*in module llvmlite.binding*), 37
PassBuilder (*class in llvmlite.binding*), 46
PassManager (*class in llvmlite.binding*), 48
PassManager.add_basic_alias_analysis_pass() (*in module llvmlite.binding*), 48
PassManager.add_cfg_simplification_pass() (*in module llvmlite.binding*), 48
PassManager.add_constant_merge_pass() (*in module llvmlite.binding*), 48
PassManager.add_dead_arg_elimination_pass() (*in module llvmlite.binding*), 48
PassManager.add_dead_code_elimination_pass() (*in module llvmlite.binding*), 48
PassManager.add_function_attrs_pass() (*in module llvmlite.binding*), 48
PassManager.add_function_inlining_pass() (*in module llvmlite.binding*), 48

PassManager.add_global_dce_pass() (*in module llvmlite.binding*), 48
PassManager.add_global_optimizer_pass() (*in module llvmlite.binding*), 48
PassManager.add_gvn_pass() (*in module llvmlite.binding*), 48
PassManager.add_instruction_combining_pass() (*in module llvmlite.binding*), 48
PassManager.add_instruction_namer_pass() (*in module llvmlite.binding*), 49
PassManager.add_ipsccp_pass() (*in module llvmlite.binding*), 48
PassManager.add_licm_pass() (*in module llvmlite.binding*), 48
PassManager.add_refprune_pass() (*in module llvmlite.binding*), 49
PassManager.add_sccp_pass() (*in module llvmlite.binding*), 48
PassManager.add_sroa_pass() (*in module llvmlite.binding*), 48
PassManager.add_type_based_alias_analysis_pass() (*in module llvmlite.binding*), 48
PassManagerBuilder (*class in llvmlite.binding*), 47
phi() (*llvmlite.ir.IRBuilder method*), 28
PhiInstr (*class in llvmlite.ir*), 21
PipelineTuningOptions (*class in llvmlite.binding*), 46
pointee (*llvmlite.ir.PointerType attribute*), 13
PointerType (*class in llvmlite.ir*), 12
populate() (*llvmlite.binding.PassManagerBuilder method*), 47
position_after() (*llvmlite.ir.IRBuilder method*), 24
position_at_end() (*llvmlite.ir.IRBuilder method*), 24
position_at_start() (*llvmlite.ir.IRBuilder method*), 24
position_before() (*llvmlite.ir.IRBuilder method*), 24
PredictableInstr (*class in llvmlite.ir*), 21
ptrtoint() (*llvmlite.ir._ConstOpMixin method*), 17
ptrtoint() (*llvmlite.ir.IRBuilder method*), 28

R

register_lock_callback() (*in module llvmlite.binding.ffi*), 50
remove_module() (*llvmlite.binding.ExecutionEngine method*), 44
replace() (*llvmlite.ir.Block method*), 18
replace_usage() (*llvmlite.ir.Instruction method*), 21
report_and_reset_timings() (*in module llvmlite.binding*), 50
resume() (*llvmlite.ir.IRBuilder method*), 31
ret() (*llvmlite.ir.IRBuilder method*), 30
ret_void() (*llvmlite.ir.IRBuilder method*), 31
run() (*llvmlite.binding.FunctionPassManager method*), 46
run() (*llvmlite.binding.ModulePassManager method*), 46

S

sadd_with_overflow() (*llvmlite.ir.IRBuilder method*), 26
sdiv() (*llvmlite.ir.IRBuilder method*), 26
section (*llvmlite.ir.GlobalValue attribute*), 19
SectionIteratorRef (*class in llvmlite.binding*), 45
select() (*llvmlite.ir.IRBuilder method*), 28
set_asm_verbosity() (*llvmlite.binding.TargetMachine method*), 36
set_body() (*llvmlite.ir.IdentifiedStructType method*), 14
set_metadata() (*llvmlite.ir.Function method*), 20
set_metadata() (*llvmlite.ir.GlobalVariable method*), 19
set_metadata() (*llvmlite.ir.Instruction method*), 21
set_object_cache() (*llvmlite.binding.ExecutionEngine method*), 44
set_time_passes() (*in module llvmlite.binding*), 50
set_weights() (*llvmlite.ir.PredictableInstr method*), 21
sext() (*llvmlite.ir._ConstOpMixin method*), 16
sext() (*llvmlite.ir.IRBuilder method*), 27
shl() (*llvmlite.ir._ConstOpMixin method*), 15
shl() (*llvmlite.ir.IRBuilder method*), 26
shuffle_vector() (*llvmlite.ir.IRBuilder method*), 29
shutdown() (*in module llvmlite.binding*), 34
sitofp() (*llvmlite.ir._ConstOpMixin method*), 16
sitofp() (*llvmlite.ir.IRBuilder method*), 28
size_level (*llvmlite.binding.PassManagerBuilder attribute*), 47
size_level (*llvmlite.binding.PipelineTuningOptions attribute*), 46
slp_vectorization (*llvmlite.binding.PipelineTuningOptions attribute*), 46
slp_vectorize (*llvmlite.binding.PassManagerBuilder attribute*), 48
smul_with_overflow() (*llvmlite.ir.IRBuilder method*), 26
source_file (*llvmlite.binding.ModuleRef attribute*), 38
speed_level (*llvmlite.binding.PipelineTuningOptions attribute*), 46
srem() (*llvmlite.ir._ConstOpMixin method*), 15
srem() (*llvmlite.ir.IRBuilder method*), 26
ssub_with_overflow() (*llvmlite.ir.IRBuilder method*), 26
storage_class (*llvmlite.binding.ValueRef attribute*), 41
storage_class (*llvmlite.ir.GlobalValue attribute*), 19
StorageClass (*class in llvmlite.binding*), 39
StorageClass.default (*in module llvmlite.binding*), 39
StorageClass.dllexport (*in module llvmlite.binding*), 39

StorageClass.dllimport (*in module llvmlite.binding*), 39
store() (*llvmlite.ir.IRBuilder method*), 29
store_atomic() (*llvmlite.ir.IRBuilder method*), 29
store_reg() (*llvmlite.ir.IRBuilder method*), 32
struct_types (*llvmlite.binding.ModuleRef attribute*), 38
sub() (*llvmlite.ir._ConstOpMixin method*), 15
sub() (*llvmlite.ir.IRBuilder method*), 26
switch() (*llvmlite.ir.IRBuilder method*), 31
SwitchInstr (*class in llvmlite.ir*), 21

T

Target (*class in llvmlite.binding*), 36
target_data (*llvmlite.binding.ExecutionEngine attribute*), 44
target_data (*llvmlite.binding.TargetMachine attribute*), 36
TargetData (*class in llvmlite.binding*), 35
TargetMachine (*class in llvmlite.binding*), 36
terminator (*llvmlite.ir.Block attribute*), 18
triple (*llvmlite.binding.ModuleRef attribute*), 38
triple (*llvmlite.binding.Target attribute*), 36
triple (*llvmlite.ir.Module attribute*), 23
trunc() (*llvmlite.ir._ConstOpMixin method*), 16
trunc() (*llvmlite.ir.IRBuilder method*), 27
Type (*class in llvmlite.ir*), 11
type (*llvmlite.binding.ValueRef attribute*), 41
type_kind (*llvmlite.binding.TypeRef attribute*), 43
type_width (*llvmlite.binding.TypeRef attribute*), 43
TypeRef (*class in llvmlite.binding*), 42

U

udiv() (*llvmlite.ir._ConstOpMixin method*), 15
udiv() (*llvmlite.ir.IRBuilder method*), 26
uitofp() (*llvmlite.ir._ConstOpMixin method*), 16
uitofp() (*llvmlite.ir.IRBuilder method*), 28
Undefined (*in module llvmlite.ir*), 14
unnamed_addr (*llvmlite.ir.GlobalVariable attribute*), 19
unreachable() (*llvmlite.ir.IRBuilder method*), 32
unregister_lock_callback() (*in module llvmlite.binding.ffi*), 50
urem() (*llvmlite.ir._ConstOpMixin method*), 15
urem() (*llvmlite.ir.IRBuilder method*), 27

V

Value (*class in llvmlite.ir*), 14
value_kind (*llvmlite.binding.ValueRef attribute*), 41
ValueKind (*class in llvmlite.binding*), 40
ValueKind.argument (*in module llvmlite.binding*), 40
ValueKind.basic_block (*in module llvmlite.binding*), 40
ValueKind.block_address (*in module llvmlite.binding*), 40
ValueKind.constant_aggregate_zero (*in module llvmlite.binding*), 40
ValueKind.constant_array (*in module llvmlite.binding*), 40
ValueKind.constant_data_array (*in module llvmlite.binding*), 40
ValueKind.constant_data_vector (*in module llvmlite.binding*), 40
ValueKind.constant_expr (*in module llvmlite.binding*), 40
ValueKind.constant_fp (*in module llvmlite.binding*), 40
ValueKind.constant_int (*in module llvmlite.binding*), 40
ValueKind.constant_pointer_null (*in module llvmlite.binding*), 40
ValueKind.constant_struct (*in module llvmlite.binding*), 40
ValueKind.constant_token_none (*in module llvmlite.binding*), 40
ValueKind.constant_vector (*in module llvmlite.binding*), 40
ValueKind.function (*in module llvmlite.binding*), 40
ValueKind.global_alias (*in module llvmlite.binding*), 40
ValueKind.global_ifunc (*in module llvmlite.binding*), 40
ValueKind.global_variable (*in module llvmlite.binding*), 40
ValueKind.inline_asm (*in module llvmlite.binding*), 40
ValueKind.instruction (*in module llvmlite.binding*), 40
ValueKind.memory_def (*in module llvmlite.binding*), 40
ValueKind.memory_phi (*in module llvmlite.binding*), 40
ValueKind.memory_use (*in module llvmlite.binding*), 40
ValueKind.metadata_as_value (*in module llvmlite.binding*), 40
ValueKind.poison_value (*in module llvmlite.binding*), 40
ValueKind.undef_value (*in module llvmlite.binding*), 40
ValueRef (*class in llvmlite.binding*), 41
VectorType (*class in llvmlite.ir*), 13
verify() (*llvmlite.binding.ModuleRef method*), 38
view_dot_graph() (*in module llvmlite.binding*), 49
Visibility (*class in llvmlite.binding*), 39
visibility (*llvmlite.binding.ValueRef attribute*), 41
Visibility.default (*in module llvmlite.binding*), 39
Visibility.hidden (*in module llvmlite.binding*), 39
Visibility.protected (*in module llvmlite.binding*),

39
VoidType (*class in llvmlite.ir*), 12

W

width (*llvmlite.ir.IntType attribute*), 12

X

xor() (*llvmlite.ir._ConstOpMixin method*), 15
xor() (*llvmlite.ir.IRBuilder method*), 27

Z

zext() (*llvmlite.ir._ConstOpMixin method*), 16
zext() (*llvmlite.ir.IRBuilder method*), 27