PyCuda Documentation

Release 0.92

Andreas Kloeckner

February 24, 2009

CONTENTS

PyCuda gives you easy, Pythonic access to Nvidia's CUDA parallel computation API. Several wrappers of the CUDA API already exist–so why the need for PyCuda?

- Object cleanup tied to lifetime of objects. This idiom, often called RAII in C++, makes it much easier to write correct, leak- and crash-free code. PyCuda knows about dependencies, too, so (for example) it won't detach from a context before all memory allocated in it is also freed.
- Convenience. Abstractions like `pycuda.driver.SourceModule` and `pycuda.gpuarray.GPUArray` make CUDA programming even more convenient than with Nvidia's C-based runtime.
- Completeness. PyCuda puts the full power of CUDA's driver API at your disposal, if you wish.
- Automatic Error Checking. All CUDA errors are automatically translated into Python exceptions.
- Speed. PyCuda's base layer is written in C++, so all the niceties above are virtually free.
- Helpful Documentation. You're looking at it. ;)

Here's an example, to given you an impression:

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

mod = drv.SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1))

print dest-a*b
```

(You can find this example as `examples/hello_gpu.py` in the PyCuda source distribution.)

On the surface, this program will print a screenful of zeros. Behind the scenes, a lot more interesting stuff is going on:

- PyCuda has compiled the CUDA source code and uploaded it to the card.

  Note: This code doesn't have to be a constant–you can easily have Python generate the code you want to compile. See *Metaprogramming with PyCuda*.

- PyCuda's numpy interaction code has automatically allocated space on the device, copied the numpy arrays *a* and *b* over, launched a 400x1x1 single-block grid, and copied *dest* back.

  Note that you can just as well keep your data on the card between kernel invocations–no need to copy data all the time.

- See how there's no cleanup code in the example? That's not because we were lazy and just skipped it. It simply isn't needed. PyCuda will automatically infer what cleanup is necessary and do it for you.

Curious? Let's get started.

CHAPTER

ONE

CONTENTS

1.1 Installation

This tutorial will walk you through the process of building PyCuda. To follow, you really only need four basic things:

- A UNIX-like machine with web access.
- Nvidia's CUDA toolkit. PyCuda was developed against version 2.0 beta. It may work with other versions, too.
- A C++ compiler, preferably a Version 4.x gcc.
- A working Python installation, Version 2.4 or newer.

1.1.1 Step 1: Install Boost

You may already have a working copy of the Boost C++ libraries. If so, make sure that it's version 1.35.0 or newer. If not, no problem, please follow this link to the simple build and install instructions that I wrote for Boost. Continue here when you're done.

1.1.2 Step 2: Download and unpack PyCuda

Download PyCuda and unpack it:

```
$ tar xfz pycuda-VERSION.tar.gz
```

1.1.3 Step 3: Install Numpy

PyCuda is designed to work in conjunction with numpy, Python's array package.

Here's an easy way to install it, if you do not have it already:

```
$ cd pycuda-VERSION
$ su -c "python ez_setup.py" # this will install setuptools
$ su -c "easy_install numpy" # this will install numpy using setuptools
```

(If you're not sure, repeating these commands will not hurt.)

1.1.4 Step 4: Build PyCuda

Next, just type:

```
$ cd pycuda-VERSION # if you're not there already
$ python configure.py \
  --boost-inc-dir=$HOME/pool/include/boost-1_35 \
  --boost-lib-dir=$HOME/pool/lib \
  --boost-python-libname=boost_python-gcc42-mt \
  --cuda-root=/where/ever/you/installed/cuda
$ su -c "make install"
```

Note that `gcc42` is a compiler tag that depends on the compiler with which you built boost. Check the contents of your boost library directory to find out what the correct tag is. Also note that you will (probably) have to change the value of *--cuda-root*.

Once that works, congratulations! You've successfully built PyCuda.

1.1.5 Step 5: Test PyCuda

If you'd like to be extra-careful, you can run PyCuda's unit tests:

```
$ cd pycuda-VERSION/test
$ python test_driver.py
```

If it says "OK" at the end, you're golden.

1.1.6 Installing on Windows

First, try running **configure.py** as above. If that fails, create a file called `siteconf.py` containing the following, adapted to match your system:

```
BOOST_INC_DIR = [r'C:\Program Files\boost\boost_1_36_0']
BOOST_LIB_DIR = [r'C:\Program Files\boost\boost_1_36_0\stage\lib']
BOOST_PYTHON_LIBNAME = ['boost_python-mgw34']
CUDA_ROOT = r'C:\CUDA'
CUDADRV_LIB_DIR = [r'C:\CUDAlib']
CUDADRV_LIBNAME = ['cuda']
CXXFLAGS = []
LDFLAGS = []
```

Subsequently, you may build and install PyCuda by typing:

```
$ python setup.py install
```

1.2 Tutorial Introduction

1.2.1 Getting started

Before you can use PyCuda, you have to initialize it and create a context:

```
import pycuda.driver as cuda
import pycuda.autoinit
```

Pretty much equivalently, you could have used the following, wordier initialization sequence:

```
import pycuda.driver as cuda
import pycuda.autoinit

cuda.init()
assert cuda.Device.count() >= 1

dev = cuda.Device(0)
ctx = dev.make_context()
```

1.2.2 Transferring Data

The next step in most programs is to transfer data onto the device. In PyCuda, you will mostly transfer data from `numpy` arrays on the host. (But indeed, everything that satisfies the Python buffer interface will work, even a `str`.) Let's make a 4x4 array of random numbers:

```
import numpy
a = numpy.random.randn(4,4)
```

But wait–*a* consists of double precision numbers, but most nVidia devices only support single precision:

```
a = a.astype(numpy.float32)
```

Finally, we need somewhere to transfer data to, so we need to allocate memory on the device:

```
a_gpu = cuda.mem_alloc(a.nbytes)
```

As a last step, we need to transfer the data to the GPU:

```
cuda.memcpy_htod(a_gpu, a)
```

1.2.3 Executing a Kernel

For this tutorial, we'll stick to something simple: We will write code to double each entry in *a_gpu*. To this end, we write the corresponding CUDA C code, and feed it into the constructor of a `pycuda.driver.SourceModule`:

```
mod = cuda.SourceModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)
```

If there aren't any errors, the code is now compiled and loaded onto the device. We find a reference to our `pycuda.driver.Function` and call it, specifying *a_gpu* as the argument, and a block size of 4x4:

```
func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))
```

Finally, we fetch the data back from the GPU and display it, together with the original *a*:

```
a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```

This will print something like this:

```
[[ 0.51360393  1.40589952  2.25009012  3.02563429]
 [-0.75841576 -1.18757617  2.72269917  3.12156057]
 [ 0.28826082 -2.92448163  1.21624792  2.86353827]
 [ 1.57651746  0.63500965  2.21570683 -0.44537592]]
[[ 0.25680196  0.70294976  1.12504506  1.51281714]
 [-0.37920788 -0.59378809  1.36134958  1.56078029]
 [ 0.14413041 -1.46224082  0.60812396  1.43176913]
 [ 0.78825873  0.31750482  1.10785341 -0.22268796]]
```

It worked! That completes our walkthrough. Thankfully, PyCuda takes over from here and does all the cleanup for you, so you're done. Stick around for some bonus material in the next section, though.

(You can find the code for this demo as `examples/demo.py` in the PyCuda source distribution.)

Shortcuts for Explicit Memory Copies

The `pycuda.driver.In`, `pycuda.driver.Out`, and `pycuda.driver.InOut` argument handlers can simplify some of the memory transfers. For example, instead of creating *a_gpu*, if replacing *a* is fine, the following code can be used:

```
func(cuda.InOut(a), block=(4, 4, 1))
```

Prepared Invocations

Function invocation using the built-in `pycuda.driver.Function.__call__()` method incurs overhead for type identification (see *Device Interface Reference Documentation*). To achieve the same effect as above without this overhead, the function is bound to argument types (as designated by Python's standard library `struct` module), and then called. This also avoids having to assign explicit argument sizes using the *numpy.number* classes:

```
func.prepare("P", block=(4,4,1))
func.prepared_call((1, 1), a_gpu)
```

1.2.4 Bonus: Abstracting Away the Complications

Using a `pycuda.gpuarray.GPUArray`, the same effect can be achieved with much less writing:

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
```

```
a_gpu = gpuarray.to_gpu(numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
```

1.2.5 Advanced Topics

Structures

(contributed by Nicholas Tung, find the code in `examples/demo_struct.py`)

Suppose we have the following structure, for doubling a number of variable length arrays:

```
mod = cuda.SourceModule("""
    struct DoubleOperation {
        int datalen, __padding; // so 64-bit ptrs can be aligned
        float *ptr;
    };

    __global__ void double_array(DoubleOperation *a) {
        a = &a[blockIdx.x];
        for (int idx = threadIdx.x; idx < a->datalen; idx += blockDim.x) {
            a->ptr[idx] *= 2;
        }
    }
    """)
```

Each block in the grid (see CUDA documentation) will double one of the arrays. The *for* loop allows for more data elements than threads to be doubled, though is not efficient if one can guarantee that there will be a sufficient number of threads. Next, a wrapper class for the structure is created, and two arrays are instantiated:

```
class DoubleOpStruct:
    mem_size = 8 + numpy.intp(0).nbytes
    def __init__(self, array, struct_arr_ptr):
        self.data = cuda.to_device(array)
        self.shape, self.dtype = array.shape, array.dtype
        cuda.memcpy_htod(int(struct_arr_ptr), numpy.int32(array.size))
        cuda.memcpy_htod(int(struct_arr_ptr) + 8, numpy.intp(int(self.data)))
    def __str__(self):
        return str(cuda.from_device(self.data, self.shape, self.dtype))

struct_arr = cuda.mem_alloc(2 * DoubleOpStruct.mem_size)
do2_ptr = int(struct_arr) + DoubleOpStruct.mem_size

array1 = DoubleOpStruct(numpy.array([1, 2, 3], dtype=numpy.float32), struct_arr)
array2 = DoubleOpStruct(numpy.array([0, 4], dtype=numpy.float32), do2_ptr)
print("original arrays", array1, array2)
```

This code uses the `pycuda.driver.to_device()` and `pycuda.driver.from_device()` functions to allocate and copy values, and demonstrates how offsets to an allocated block of memory can be used. Finally, the code can be executed; the following demonstrates doubling both arrays, then only the second:

```
func = mod.get_function("double_array")
func(struct_arr, block = (32, 1, 1), grid=(2, 1))
```

```
print("doubled arrays", array1, array2)

func(numpy.intp(do2_ptr), block = (32, 1, 1), grid=(1, 1))
print("doubled second only", array1, array2, "\n")
```

1.2.6 Where to go from here

Once you feel sufficiently familiar with the basics, feel free to dig into the *Device Interface Reference Documentation*. For more examples, check the in the `examples/` subdirectory of the distribution. This folder also contains several benchmarks to see the difference between GPU and CPU based calculations. As a reference for how stuff is done, PyCuda's test suite in the `test/` subdirectory of the distribution may also be of help.

1.3 Device Interface Reference Documentation

1.3.1 Error Reporting

exception Error
Base class of all PyCuda errors.

exception CompileError
Thrown when `SourceModule` compilation fails.

exception MemoryError
Thrown when `mem_alloc()` or related functionality fails.

exception LogicError
Thrown when PyCuda was confronted with a situation where it is likely that the programmer has made a mistake. `LogicErrors` do not depend on outer circumstances defined by the run-time environment.

Example: CUDA was used before it was initialized.

exception LaunchError
Thrown when kernel invocation has failed. (Note that this will often be reported by the next call after the actual kernel invocation.)

exception RuntimeError
Thrown when a unforeseen run-time failure is encountered that is not likely due to programmer error.

Example: A file was not found.

1.3.2 Constants

class ctx_flags()
Flags for `Device.make_context()`. CUDA 2.0 and above only.

SCHED_AUTO
If there are more contexts than processors, yield, otherwise spin while waiting for CUDA calls to complete.

SCHED_SPIN
Spin while waiting for CUDA calls to complete.

SCHED_YIELD
Yield to other threads while waiting for CUDA calls to complete.

SCHED_MASK
Mask of valid flags in this bitfield.

SCHED_FLAGS_MASK
Mask of valid scheduling flags in this bitfield.

class **device_attribute**()

MAX_THREADS_PER_BLOCK

MAX_BLOCK_DIM_X

MAX_BLOCK_DIM_Y

MAX_BLOCK_DIM_Z

MAX_GRID_DIM_X

MAX_GRID_DIM_Y

MAX_GRID_DIM_Z

TOTAL_CONSTANT_MEMORY

WARP_SIZE

MAX_PITCH

CLOCK_RATE

TEXTURE_ALIGNMENT

GPU_OVERLAP

MULTIPROCESSOR_COUNT
CUDA 2.0 and above only.

SHARED_MEMORY_PER_BLOCK
Deprecated as of CUDA 2.0. See below for replacement.

MAX_SHARED_MEMORY_PER_BLOCK
CUDA 2.0 and above only.

REGISTERS_PER_BLOCK
Deprecated as of CUDA 2.0. See below for replacement.

MAX_REGISTERS_PER_BLOCK
CUDA 2.0 and above only.

class **array_format**()

UNSIGNED_INT8

UNSIGNED_INT16

UNSIGNED_INT32

SIGNED_INT8

SIGNED_INT16

SIGNED_INT32

HALF

FLOAT

class **address_mode**()

WRAP

CLAMP

MIRROR

class **filter_mode**()

POINT

LINEAR

class **memory_type**()

HOST

DEVICE

ARRAY

1.3.3 Devices and Contexts

get_version()
Obtain the version of CUDA against which PyCuda was compiled. Returns a 3-tuple of integers as *(major, minor, revision)*.

init(*flags=0*)
Initialize CUDA.

> **Warning:** This must be called before any other function in this module.

See also `pycuda.autoinit`.

class **Device**(*number*)
A handle to the *number*'th CUDA device. See also `pycuda.autoinit`.

static **count**()
Return the number of CUDA devices found.

name()
Return the name of this CUDA device.

compute_cabability()
Return a 2-tuple indicating the compute capability version of this device.

total_memory()
Return the total amount of memory on the device in bytes.

get_attribute(*attr*)
Return the (numeric) value of the attribute *attr*, which may be one of the `device_attribute` values.

get_attributes()
Return all device attributes in a `dict`, with keys from `device_attribute`.

make_context(*flags=ctx_flags.SCHED_AUTO*)
Create a `Context` on this device, with flags taken from the `ctx_flags` values.

Also make the newly-created context the current context.

__hash__()

__eq__()

__ne__()

class **Context**()
An equivalent of a UNIX process on the compute device. Create instances of this class using `Device.make_context()`. See also `pycuda.autoinit`.

detach()
Decrease the reference count on this context. If the reference count hits zero, the context is deleted.

push()
Make *self* the active context, pushing it on top of the context stack. CUDA 2.0 and above only.

pop()
Remove *self* from the top of the context stack, deactivating it. CUDA 2.0 and above only.

static **get_device**()
Return the device that the current context is working on.

static **synchronize**()
Wait for all activity in the current context to cease, then return.

1.3.4 Concurrency and Streams

class Stream(*flags=0*)
A handle for a queue of operations that will be carried out in order.

synchronize()
Wait for all activity on this stream to cease, then return.

is_done()
Return *True* iff all queued operations have completed.

class Event(*flags=0*)

record()
Insert a recording point for *self* into the global device execution stream.

record_in_stream(*stream*)
Insert a recording point for *self* into the `Stream` *stream*

synchronize()
Wait until the device execution stream reaches this event.

query()
Return *True* if the device execution stream has reached this event.

time_since(*event*)
Return the time in milliseconds that has passed between *self* and *event*.

time_till(*event*)
Return the time in milliseconds that has passed between *event* and *self*.

1.3.5 Memory

Global Device Memory

mem_get_info()
Return a tuple *(free, total)* indicating the free and total memory in the current context, in bytes.

mem_alloc(*bytes*)
Return a `DeviceAllocation` object representing a linear piece of device memory.

to_device(*buffer*)
Allocate enough device memory for *buffer*, which adheres to the Python `buffer` interface. Copy the contents of *buffer* onto the device. Return a `DeviceAllocation` object representing the newly-allocated memory.

from_device(*devptr, shape, dtype, order="C"*)
Make a new `numpy.ndarray` from the data at *devptr* on the GPU, interpreting them using *shape*, *dtype* and *order*.

from_device_like(*devptr, other_ary*)
Make a new `numpy.ndarray` from the data at *devptr* on the GPU, interpreting them as having the same shape, dtype and order as *other_ary*.

mem_alloc_pitch(*width, height, access_size*)
Allocates a linear piece of device memory at least *width* bytes wide and *height* rows high that an be accessed using a data type of size *access_size* in a coalesced fashion.

Returns a tuple *(dev_alloc, actual_pitch)* giving a `DeviceAllocation` and the actual width of each row in bytes.

class **DeviceAllocation**()
An object representing an allocation of linear device memory. Once this object is deleted, its associated device memory is freed.

Objects of this type can be cast to `int` to obtain a linear index into this `Context`'s memory.

free()
Release the held device memory now instead of when this object becomes unreachable. Any further use of the object is an error and will lead to undefined behavior.

Pagelocked Host Memory

pagelocked_empty(*shape, dtype, order="C"*)
Allocate a pagelocked `numpy.ndarray` of *shape*, *dtype* and *order*. For the meaning of these parameters, please refer to the `numpy` documentation.

pagelocked_zeros(*shape, dtype, order="C"*)
Allocate a pagelocked `numpy.ndarray` of *shape*, *dtype* and *order* that is zero-initialized.

For the meaning of these parameters, please refer to the `numpy` documentation.

pagelocked_empty_like(*array*)
Allocate a pagelocked `numpy.ndarray` with the same shape, dtype and order as *array*.

pagelocked_zeros_like(*array*)
Allocate a pagelocked `numpy.ndarray` with the same shape, dtype and order as *array*. Initialize it to 0.

The `numpy.ndarray` instances returned by these functions have an attribute *base* that references an object of type

class **HostAllocation**()
An object representing an allocation of pagelocked host memory. Once this object is deleted, its associated device memory is freed.

free()
Release the held memory now instead of when this object becomes unreachable. Any further use of the object (or its associated `numpy` array) is an error and will lead to undefined behavior.

Arrays and Textures

class **ArrayDescriptor**()

width

height

format
A value of type `array_format`.

num_channels

class ArrayDescriptor3D()

width

height

depth

format
A value of type array_format. CUDA 2.0 and above only.

num_channels

class Array(*descriptor*)
A 2D or 3D memory block that can only be accessed via texture references.

descriptor can be of type ArrayDescriptor or ArrayDescriptor3D.

free()
Release the array and its device memory now instead of when this object becomes unreachable. Any further use of the object is an error and will lead to undefined behavior.

get_descriptor()
Return a ArrayDescriptor object for this 2D array, like the one that was used to create it.

get_descriptor_3d()
Return a ArrayDescriptor3D object for this 3D array, like the one that was used to create it. CUDA 2.0 and above only.

class TextureReference()
A handle to a binding of either linear memory or an Array to a texture unit.

set_array(*array*)
Bind *self* to the Array *array*.

As long as *array* remains bound to this texture reference, it will not be freed–the texture reference keeps a reference to the array.

set_address(*devptr, bytes*)
Bind *self* to the a chunk of linear memory starting at the integer address *devptr*, encompassing a number of *bytes*.

Unlike for Array objects, no life support is provided for linear memory bound to texture references.

set_format(*fmt, num_components*)
Set the texture to have array_format *fmt* and to have *num_components* channels.

set_address_mode(*dim, am*)
Set the address mode of dimension *dim* to *am*, which must be one of the address_mode values.

set_flags(*flags*)
Set the flags to a combination of the *TRSF_XXX* values.

get_array()
Get back the Array to which *self* is bound.

get_address_mode(*dim*)

get_filter_mode()

get_format()
Return a tuple *(fmt, num_components)*, where *fmt* is of type array_format, and *num_components* is the number of channels in this texture.

(Version 2.0 and above only.)

get_flags()

TRSA_OVERRIDE_FORMAT

TRSF_READ_AS_INTEGER

TRSF_NORMALIZED_COORDINATES

TR_DEFAULT

matrix_to_array(*matrix, order*)
Turn the two-dimensional `numpy.ndarray` object *matrix* into an `Array`. The *order* argument can be either *"C"* or *"F"*. If it is *"C"*, then *tex2D(x,y)* is going to fetch *matrix[y,x]*, and vice versa for for *"F"*.

make_multichannel_2d_array(*matrix, order*)
Turn the three-dimensional `numpy.ndarray` object *matrix* into an 2D `Array` with multiple channels.

Depending on *order*, the *matrix*'s shape is interpreted as

- *height, width, num_channels* for *order == "C"*,
- *num_channels, width, height* for *order == "F"*.

Initializing Device Memory

memset_d8(*dest, data, count*)

memset_d16(*dest, data, count*)

memset_d32(*dest, data, count*)

Note: *count* is the number of elements, not bytes.

memset_d2d8(*dest, pitch, data, width, height*)

memset_d2d16(*dest, pitch, data, width, height*)

memset_d2d32(*dest, pitch, data, width, height*)

Unstructured Memory Transfers

memcpy_htod(*dest, src, stream=None*)
Copy from the Python buffer *src* to the device pointer *dest* (an `int` or a `DeviceAllocation`). The size of the copy is determined by the size of the buffer.

Optionally execute asynchronously, serialized via *stream*. In this case, *src* must be page-locked.

memcpy_dtoh(*dest, src, stream=None*)
Copy from the device pointer *src* (an `int` or a `DeviceAllocation`) to the Python buffer *dest*. The size of the copy is determined by the size of the buffer.

Optionally execute asynchronously, serialized via *stream*. In this case, *dest* must be page-locked.

memcpy_dtod(*dest, src, size*)

memcpy_dtoa(*ary, index, src, len*)

memcpy_atod(*dest, ary, index, len*)

memcpy_htoa(*ary, index, src*)

memcpy_atoh(*dest, ary, index*)

memcpy_atoa(*dest, dest_index, src, src_index, len*)

Structured Memory Transfers

class Memcpy2D()

src_x_in_bytes
X Offset of the origin of the copy. (initialized to 0)

src_y
Y offset of the origin of the copy. (initialized to 0)

src_pitch
Size of a row in bytes at the origin of the copy.

set_src_host(*buffer*)
Set the *buffer*, which must be a Python object adhering to the buffer interface, to be the origin of the copy.

set_src_array(*array*)
Set the `Array` *array* to be the origin of the copy.

set_src_device(*devptr*)
Set the device address *devptr* (an `int` or a `DeviceAllocation`) as the origin of the copy.

dst_x_in_bytes
X offset of the destination of the copy. (initialized to 0)

dst_y
Y offset of the destination of the copy. (initialized to 0)

dst_pitch
Size of a row in bytes at the destination of the copy.

set_dst_host(*buffer*)
Set the *buffer*, which must be a Python object adhering to the buffer interface, to be the destination of the copy.

set_dst_array(*array*)
Set the `Array` *array* to be the destination of the copy.

set_dst_device(*devptr*)
Set the device address *devptr* (an `int` or a `DeviceAllocation`) as the destination of the copy.

width_in_bytes
Number of bytes to copy for each row in the transfer.

height
Number of rows to copy.

__call__(*[aligned=True]*)
Perform the specified memory copy, waiting for it to finish. If *aligned* is *False*, tolerate misalignment that may lead to severe loss of copy bandwidth.

__call__(*stream*)
Perform the memory copy asynchronously, serialized via the `Stream` *stream*. Any host memory involved in the transfer must be page-locked.

class Memcpy3D()
`Memcpy3D` has the same members as `Memcpy2D`, and additionally all of the following:

src_height
Ignored when source is an `Array`. May be 0 if Depth==1.

src_z
Z offset of the origin of the copy. (initialized to 0)

dst_height
Ignored when destination is an `Array`. May be 0 if Depth==1.

dst_z
Z offset of the destination of the copy. (initialized to 0)

depth

`Memcpy3D` is supported on CUDA 2.0 and above only.

1.3.6 Code on the Device: Modules and Functions

class Module()

Handle to a CUBIN module loaded onto the device. Can be created with `module_from_file()` and `module_from_buffer()`.

get_function(*name*)

Return the `Function` *name* in this module.

> **Warning:** While you can obtain different handles to the same function using this method, these handles all share the same state that is set through the `set_XXX` methods of `Function`. This means that you can't obtain two different handles to the same function and `Function.prepare()` them in two different ways.

get_global(*name*)

Return the device address of the global *name* as an `int`.

get_texref(*name*)

Return the `TextureReference` *name* from this module.

module_from_file(*filename*)

Create a `Module` by loading the CUBIN file *filename*.

module_from_buffer(*buffer*)

Create a `Module` by loading a CUBIN from *buffer*, which must support the Python buffer interface. (For example, `str` and `numpy.ndarray` do.)

class Function()

Handle to a *__global__* function in a `Module`. Create using `Module.get_function()`.

__call__(*arg1, ..., argn, block=block_size, [grid=(1, 1), [stream=None, [shared=0, [texrefs=, [], [time_kernel=False]]]]]*)

Launch *self*, with a thread block size of *block*. *block* must be a 3-tuple of integers.

arg1 through *argn* are the positional C arguments to the kernel. See `param_set()` for details. See especially the warnings there.

grid specifies, as a 2-tuple, the number of thread blocks to launch, as a two-dimensional grid. *stream*, if specified, is a `Stream` instance serializing the copying of input arguments (if any), execution, and the copying of output arguments (again, if any). *shared* gives the number of bytes available to the kernel in *extern __shared__* arrays. *texrefs* is a `list` of `TextureReference` instances that the function will have access to.

The function returns either *None* or the number of seconds spent executing the kernel, depending on whether *time_kernel* is *True*.

This is a convenience interface that can be used instead of the `param_*()` and `launch_*()` methods below. For a faster (but mildly less convenient) way of invoking kernels, see `prepare()` and `prepared_call()`.

param_set(*arg1, ... argn*)

Set up *arg1* through *argn* as positional C arguments to *self*. They are allowed to be of the following types:

- Subclasses of `numpy.number`. These are sized number types such as `numpy.uint32` or `numpy.float32`.
- `DeviceAllocation` instances, which will become a device pointer to the allocated memory.
- Instances of `ArgumentHandler` subclasses. These can be used to automatically transfer `numpy` arrays onto and off of the device.
- Objects supporting the Python `buffer` interface. These chunks of bytes will be copied into the parameter space verbatim.
- `GPUArray` instances.

Warning: You cannot pass values of Python's native `int` or `float` types to param_set. Since there is no unambiguous way to guess the size of these integers or floats, it complains with a `TypeError`.

Note: This method has to guess the types of the arguments passed to it, which can make it somewhat slow. For a kernel that is invoked often, this can be inconvenient. For a faster (but mildly less convenient) way of invoking kernels, see `prepare()` and `prepared_call()`.

set_block_shape(*x, y, z*)
Set the thread block shape for this function.

set_shared_size(*bytes*)
Set *shared* to be the number of bytes available to the kernel in *extern __shared__* arrays.

param_set_size(*bytes*)
Size the parameter space to *bytes*.

param_seti(*offset, value*)
Set the integer at *offset* in the parameter space to *value*.

param_setf(*offset, value*)
Set the float at *offset* in the parameter space to *value*.

param_set_texref(*texref*)
Make the `TextureReference` texref available to the function.

launch()
Launch a single thread block of *self*.

launch_grid(*width, height*)
Launch a width*height grid of thread blocks of *self*.

launch_grid_async(*width, height, stream*)
Launch a width*height grid of thread blocks of *self*, sequenced by the `Stream` *stream*.

prepare(*arg_types, block, shared=None, texrefs=, []*)
Prepare the invocation of this function by

- setting up the argument types as *arg_types*. *arg_types* is expected to be an iterable containing type characters understood by the `struct` module or `numpy.dtype` objects.
- setting the thread block shape for this function to *block*.
- Registering the texture references *texrefs* for use with this functions. The `TextureReference` objects in *texrefs* will be retained, and whatever these references are bound to at invocation time will be available through the corresponding texture references within the kernel.

Return *self*.

prepared_call(*grid, *args*)
Invoke *self* using `launch_grid()`, with *args* and a grid size of *grid*. Assumes that `prepare()` was called on *self*. The texture references given to `prepare()` are set up as parameters, as well.

prepared_timed_call(*grid, stream, *args*)
Invoke *self* using `launch_grid()`, with *args* and a grid size of *grid*. Assumes that `prepare()` was called on *self*. The texture references given to `prepare()` are set up as parameters, as well.

Return a 0-ary callable that can be used to query the GPU time consumed by the call, in seconds. Once called, this callable will block until completion of the invocation.

prepared_async_call(*grid, stream, *args*)
Invoke *self* using `launch_grid_async()`, with *args* and a grid size of *grid*, serialized into the `pycuda.driver.Stream` *stream*. If *stream* is None, do the same as `prepared_call()`. Assumes that `prepare()` was called on *self*. The texture references given to `prepare()` are set up as parameters, as well.

Return a 0-ary callable that can be used to query the GPU time consumed by the call, in seconds. Once called, this callable will block until completion of the invocation.

lmem
The number of bytes of local memory used by this function. Only available if this function is part of a `SourceModule`.

smem
The number of bytes of shared memory used by this function. Only available if this function is part of a `SourceModule`.

registers
The number of 32-bit registers used by this function. Only available if this function is part of a `SourceModule`.

class ArgumentHandler(*array*)

class In(*array*)
Inherits from `ArgumentHandler`. Indicates that `buffer` *array* should be copied to the compute device before invoking the kernel.

class Out(*array*)
Inherits from `ArgumentHandler`. Indicates that `buffer` *array* should be copied off the compute device after invoking the kernel.

class InOut(*array*)
Inherits from `ArgumentHandler`. Indicates that `buffer` *array* should be copied both onto the compute device before invoking the kernel, and off it afterwards.

class SourceModule(*source, nvcc="nvcc", options=, [], keep=False, no_extern_c=False, arch=None, code=None, cache_dir=None*)
Create a `Module` from the CUDA source code *source*. The Nvidia compiler *nvcc* is assumed to be on the **PATH** if no path to it is specified, and is invoked with *options* to compile the code. If *keep* is *True*, the compiler output directory is kept, and a line indicating its location in the file system is printed for debugging purposes.

Unless *no_extern_c* is *True*, the given source code is wrapped in *extern "C" { ... }* to prevent C++ name mangling.

arch and *code* specify the values to be passed for the `-arch` and `-code` options on the **nvcc** command line. If *arch* is *None*, it defaults to the current context's device's compute capability. If *code* is *None*, it will not be specified.

cache_dir gives the directory used for compiler caching. It has a sensible per-user default. If it is set to *False*, caching is disabled.

This class exhibits the same public interface as `Module`, but does not inherit from it.

1.4 Built-in Utilities

1.4.1 Automatic Initialization

This module, when imported, automatically performs all the steps necessary to get CUDA ready for submission of compute kernels. When imported, this module will automatically initialize CUDA and create a `pycuda.driver.Context` on the device.

device
An instance of `pycuda.driver.Device` that was used for automatic initialization. The appropriate device is found by calling `pycuda.tools.get_default_device()`.

context
A default-constructed instance of `pycuda.driver.Context` on `device`.

1.4.2 Choice of Device

get_default_device(*default=0*)
Return a `pycuda.driver.Device` instance chosen according to the following rules:

- If the environment variable **CUDA_DEVICE** is set, its integer value is used as the device number.
- If the file `.cuda-device` is present in the user's home directory, the integer value of its contents is used as the device number.
- Otherwise, *default* is used as the device number.

1.4.3 Device Metadata and Occupancy

class DeviceData(*dev=None*)
Gives access to more information on a device than is available through `pycuda.driver.Device.get_attribute()`. If *dev* is *None*, it defaults to the device returned by `pycuda.driver.Context.get_device()`.

max_threads

warp_size

warps_per_mp

thread_blocks_per_mp

registers

shared_memory

smem_granularity
The number of threads that participate in banked, simultaneous access to shared memory.

align_bytes(*word_size=4*)
The distance between global memory base addresses that allow accesses of word-size *word_size* bytes to get coalesced.

align(*bytes, word_size=4*)
Round up *bytes* to the next alignment boundary as given by `align_bytes()`.

align_words(*word_size*)
Return *self.align_bytes(word_size)/word_size*, while checking that the division did not yield a remainder.

align_dtype(*elements, dtype_size*)
Round up *elements* to the next alignment boundary as given by `align_bytes()`, where each element is assumed to be *dtype_size* bytes large.

static **make_valid_tex_channel_count**(*size*)
Round up *size* to a valid texture channel count.

class OccupancyRecord(*devdata, threads, shared_mem=0, registers=0*)
Calculate occupancy for a given kernel workload characterized by

- thread count of *threads*
- shared memory use of *shared_mem* bytes
- register use of *registers* 32-bit registers

tb_per_mp
How many thread blocks execute on each multiprocessor.

limited_by
What `tb_per_mp` is limited by. One of *"device"*, *"warps"*, *"regs"*, *"smem"*.

warps_per_mp
How many warps execute on each multiprocessor.

occupancy
A *float* value between 0 and 1 indicating how much of each multiprocessor's scheduling capability is occupied by the kernel.

1.4.4 Memory Pools

The functions `pycuda.driver.mem_alloc()` and `pycuda.driver.pagelocked_empty()` can consume a fairly large amount of processing time if they are invoked very frequently. For example, code based on `pycuda.gpuarray.GPUArray` can easily run into this issue because a fresh memory area is allocated for each intermediate result. Memory pools are a remedy for this problem based on the observation that often many of the block allocations are of the same sizes as previously used ones.

Then, instead of fully returning the memory to the system and incurring the associated reallocation overhead, the pool holds on to the memory and uses it to satisfy future allocations of similarly-sized blocks. The pool reacts appropriately to out-of-memory conditions as long as all memory allocations are made through it. Allocations performed from outside of the pool may run into spurious out-of-memory conditions due to the pool owning much or all of the available memory.

Device-based Memory Pool

class PooledDeviceAllocation()
An object representing a `DeviceMemoryPool`-based allocation of linear device memory. Once this object is deleted, its associated device memory is freed. `PooledDeviceAllocation` instances can be cast to `int` (and `long`), yielding the starting address of the device memory allocated.

free()
Explicitly return the memory held by *self* to the associated memory pool.

__len__()
Return the size of the allocated memory in bytes.

class DeviceMemoryPool()
A memory pool for linear device memory as allocated using `pycuda.driver.mem_alloc()`. (see *Memory Pools*)

held_blocks
The number of unused blocks being held by this pool.

active_blocks
The number of blocks in active use that have been allocated through this pool.

allocate(*size*)
Return a `PooledDeviceAllocation` of *size* bytes.

free_held()
Free all unused memory that the pool is currently holding.

stop_holding()
Instruct the memory to start immediately freeing memory returned to it, instead of holding it for future allocations. Implicitly calls `free_held()`. This is useful as a cleanup action when a memory pool falls out of use.

Memory Pool for pagelocked memory

class PooledHostAllocation()
An object representing a `PageLockedMemoryPool`-based allocation of linear device memory. Once this

object is deleted, its associated device memory is freed.

free()
Explicitly return the memory held by *self* to the associated memory pool.

__len__()
Return the size of the allocated memory in bytes.

class PageLockedMemoryPool()
A memory pool for pagelocked host memory as allocated using `pycuda.driver.pagelocked_empty()`. (see *Memory Pools*)

held_blocks
The number of unused blocks being held by this pool.

active_blocks
The number of blocks in active use that have been allocated through this pool.

allocate(*shape, dtype, order="C"*)
Return an uninitialized ("empty") `numpy.ndarray` with the given *shape*, *dtype*, and *order*. This array will be backed by a `PooledHostAllocation`, which can be found as the `.base` attribute of the array.

free_held()
Free all unused memory that the pool is currently holding.

stop_holding()
Instruct the memory to start immediately freeing memory returned to it, instead of holding it for future allocations. Implicitly calls `free_held()`. This is useful as a cleanup action when a memory pool falls out of use.

1.5 The `GPUArray` Array Class

class GPUArray(*shape, dtype, stream=None*)
A `numpy.ndarray` work-alike that stores its data and performs its computations on the compute device. *shape* and *dtype* work exactly as in `numpy`. Arithmetic methods in `GPUArray` support the broadcasting of scalars. (e.g. *array+5*) If the `pycuda.driver.Stream` *stream* is specified, all computations on *self* are sequenced into it.

gpudata
The `pycuda.driver.DeviceAllocation` instance created for the memory that backs this `GPUArray`.

shape
The tuple of lengths of each dimension in the array.

dtype
The numpy `numpy.dtype` of the items in the GPU array.

size
The number of meaningful entries in the array. Can also be computed by multiplying up the numbers in `shape`.

mem_size
The total number of entries, including padding, that are present in the array. Padding may arise for example because of pitch adjustment by `pycuda.driver.mem_alloc_pitch()`.

nbytes
The size of the entire array in bytes. Computed as `size` times `dtype.itemsize`.

set(*ary, stream=None*)
Transfer the contents the `numpy.ndarray` object *ary* onto the device, optionally sequenced on *stream*. *ary* must have the same dtype and size (not necessarily shape) as *self*.

get(*ary=None, stream=None, pagelocked=False*)
Transfer the contents of *self* into *ary* or a newly allocated `numpy.ndarray`. If *ary* is given, it must have the right size (not necessarily shape) and dtype. If it is not given, *pagelocked* specifies whether the new array is allocated page-locked.

mul_add(self, selffac, other, otherfac, add_timer=None):()
Return *selffac*self + otherfac*other*. *add_timer*, if given, is invoked with the result from `pycuda.driver.Function.prepared_timed_call()`.

__add__(*other*)

__sub__(*other*)

__iadd__(*other*)

__isub__(*other*)

__neg__(*other*)

__mul__(*other*)

__div__(*other*)

__rdiv__(*other*)

__pow__(*other*)

__abs__()
Return a `GPUArray` containing the absolute value of each element of *self*.

fill(*scalar*)
Fill the array with *scalar*.

bind_to_texref(*texref*)
Bind *self* to the `TextureReference` *texref*.

1.5.1 Constructing `GPUArray` Instances

to_gpu(*ary, stream=None*)
Return a `GPUArray` that is an exact copy of the `numpy.ndarray` instance *ary*. Optionally sequence on *stream*.

empty(*shape, dtype, stream*)
A synonym for the `GPUArray` constructor.

zeros(*shape, dtype, stream*)
Same as `empty()`, but the `GPUArray` is zero-initialized before being returned.

empty_like(*other_ary*)
Make a new, uninitialized `GPUArray` having the same properties as *other_ary*.

zeros_like(*other_ary*)
Make a new, zero-initialized `GPUArray` having the same properties as *other_ary*.

arange(*start, stop, step, dtype=numpy.float32*)
Create a `GPUArray` filled with numbers spaced *step* apart, starting from *start* and ending at *stop*.

For floating point arguments, the length of the result is *ceil((stop - start)/step)*. This rule may result in the last element of the result being greater than *stop*.

1.5.2 Elementwise Functions on `GPUArrray` Instances

The `pycuda.cumath` module contains elementwise workalikes for the functions contained in `math`.

Rounding and Absolute Value

fabs(*array*)

ceil(*array*)

floor(*array*)

General Transcendental Functions

exp(*array*)

log(*array*)

log10(*array*)

sqrt(*array*)

Trigonometric Functions

sin(*array*)

cos(*array*)

tan(*array*)

asin(*array*)

acos(*array*)

atan(*array*)

Hyperbolic Functions

sinh(*array*)

cosh(*array*)

tanh(*array*)

Floating Point Decomposition and Assembly

fmod(*arg, mod*)
Return the floating point remainder of the division *arg/mod*, for each element in *arg* and *mod*.

frexp(*arg*)
Return a tuple *(significands, exponents)* such that *arg == significand * 2**exponent*.

ldexp(*significand, exponent*)
Return a new array of floating point values composed from the entries of *significand* and *exponent*, paired together as *result = significand * 2**exponent*.

modf(*arg*)
Return a tuple *(fracpart, intpart)* of arrays containing the integer and fractional parts of *arg*.

1.5.3 Generating Arrays of Random Numbers

rand(*shape, dtype=numpy.float32*)
Return an array of *shape* filled with random values of *dtype* in the range [0,1).

1.5.4 Single-pass Expression Evaluation

Warning: The following functionality is included in this documentation in the hope that it may be useful, but its interface may change in future revisions. Feedback is welcome.

Evaluating involved expressions on `GPUArray` instances can be somewhat inefficient, because a new temporary is created for each intermediate result. The functionality in the module `pycuda.elementwise` contains tools to help generate kernels that evaluate multi-stage expressions on one or several operands in a single pass.

class `ElementwiseKernel`(*arguments, operation, name="kernel", keep=False, options=, []*)

Generate a kernel that takes a number of scalar or vector *arguments* and performs the scalar *operation* on each entry of its arguments, if that argument is a vector.

arguments is specified as a string formatted as a C argument list. *operation* is specified as a C assignment statement, without a semicolon. Vectors in *operation* should be indexed by the variable *i*.

name specifies the name as which the kernel is compiled, *keep* and *options* are passed unmodified to `pycuda.driver.SourceModule`.

`__call__`(**args*)

Invoke the generated scalar kernel. The arguments may either be scalars or `GPUArray` instances.

Here's a usage example:

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
import numpy
from pycuda.curandom import rand as curand

a_gpu = curand((50,))
b_gpu = curand((50,))

from pycuda.elementwise import ElementwiseKernel
lin_comb = ElementwiseKernel(
        "float a, float *x, float b, float *y, float *z",
        "z[i] = a*x[i] + b*y[i]",
        "linear_combination")

c_gpu = gpuarray.empty_like(a_gpu)
lin_comb(5, a_gpu, 6, b_gpu, c_gpu)

import numpy.linalg as la
assert la.norm((c_gpu - (5*a_gpu+6*b_gpu)).get()) < 1e-5
```

(You can find this example as `examples/demo_elementwise.py` in the PyCuda distribution.)

1.6 Metaprogramming with PyCuda

In 'conventional' programming, one writes a program that accomplishes a task. In *metaprogramming*, one writes a program *that writes a program* that accomplishes a task.

That sounds pretty complicated–so first of all, we'll look at why it may be a good idea nonetheless.

1.6.1 Why Metaprogramming?

Automated Tuning

A sizable part of a CUDA programmer's time is typically spent tuning code. This tuning answers questions like:

- What's the optimal number of threads per block?
- How much data should I work on at once?
- What data should be loaded into shared memory, and how big should the corresponding blocks be?

If you are lucky, you'll be able to find a pattern in the execution time of your code and come up with a heuristic that will allow you to reliably pick the fastest version. Unfortunately, this heuristic may become unreliable or even fail entirely with new hardware generations. The solution to this problem that PyCuda tries to promote is:

> Forget heuristics. Benchmark at run time and use whatever works fastest.

This is an important advantage of PyCuda over the CUDA runtime API: It lets you make these decisions *while your code is running*. A number of prominent computing packages make use of a similar technique, among them ATLAS and FFTW. And while those require rather complicated optimization driver routines, you can drive PyCuda from the comfort of Python.

Data Types

Your code may have to deal with different data types at run time. It may, for example, have to work on both single and double precision floating point numbers. You could just precompile versions for both, but why? Just generate whatever code is needed right *when* it is needed.

Specialize Code for the Given Problem

If you are writing a library, then your users will ask your library to perform a number of tasks. Imagine how liberating it would be if you could generate code purposely for the problem you're being asked to solve, instead of having to keep code unnecessarily generic and thereby slow. PyCuda makes this a reality.

Constants are Faster than Variables

If your problem sizes vary from run to run, but you perform a larger number of kernel invocations on data of identical size, you may want to consider compiling data size into your code as a constant. This can have significant performance benefits, resulting mainly from decreased fetch times and less register pressure. In particular, multiplications by constans are much more efficiently carried out that general variable-variable multiplications.

Loop Unrolling

The CUDA programming guide says great things about **nvcc** and how it will unroll loops for you. As of Version 2.1, that's simply not true, and `#pragma unroll` is simply a no-op, at least according to my experience. With metaprogramming, you can dynamically unroll your loops to the needed size in Python.

1.6.2 Metaprogramming using a Templating Engine

If your metaprogramming needs are rather simple, perhaps the easiest way to generate code at run time is through a templating engine. Many templating engines for Python exist, two of the most prominent ones are Jinja 2 and Cheetah.

The following is a simple metaprogram that performs vector addition on configurable block sizes. It illustrates the templating-based metaprogramming technique:

```
from jinja2 import Template

tpl = Template("""
    typedef {{ type_name }} value_type;

    __global__ void add(value_type *result, value_type *op1, value_type *op2)
    {
      int idx = threadIdx.x + {{ thread_block_size }} * {{block_size}} * blockIdx.x;

      #for i in range(block_size)
          #set offset = i*thread_block_size
          result[idx + {{ offset }}] = op1[idx + {{ offset }}] + op2[idx + {{ offset }}];
      #endfor
    }
    """,
    line_statement_prefix="#")

rendered_tpl = tpl.render(
    type_name="float",
    block_size=block_size,
    thread_block_size=thread_block_size)

mod = cuda.SourceModule(rendered_tpl)
```

This snippet in a working context can be found in `examples/demo_meta_template.py`.

1.6.3 Metaprogramming using `codepy`

For more complicated metaprograms, it may be desirable to have more programmatic control over the assembly of the source code than a templating engine can provide. The `codepy` package provides a means of generating CUDA source code from a Python data structure.

The following example demonstrates the use of `codepy` for metaprogramming. It accomplishes exactly the same as the above program:

```
from codepy.cgen import FunctionBody, FunctionDeclaration, \
        Typedef, POD, Value, Pointer, Module, Block, Initializer, Assign

from codepy.cgen.cuda import CudaGlobal
mod = Module([
    Typedef(POD(dtype, "value_type")),
    FunctionBody(
        CudaGlobal(FunctionDeclaration(
            Value("void", "add"),
            [Pointer(POD(dtype, name)) for name in ["result", "op1", "op2"]]
        )),
        Block([
            Initializer(
                POD(numpy.int32, "idx"),
                "threadIdx.x + %d*blockIdx.x" % (thread_block_size*block_size)),
```

```
            ]+[
            Assign("result[idx+%d]" % (o*thread_block_size),
                "op1[idx+%d] + op2[idx+%d]" % (
                    o*thread_block_size,
                    o*thread_block_size))
            for o in range(block_size)
            ])
        )
    ])

mod = cuda.SourceModule(mod)
```

This snippet in a working context can be found in `examples/demo_meta_codepy.py`.

1.7 Frequently Asked Questions

1.7.1 How about multiple GPUs?

Two ways:

- Allocate two contexts, juggle (`pycuda.driver.Context.push()` and `pycuda.driver.Context.pop()`) them from that one process.
- Work with several threads. As of Version 0.90.2, PyCuda will actually release the GIL while it is waiting for CUDA operations to finish.

1.7.2 My program terminates after a launch failure. Why?

You're probably seeing something like this:

```
Traceback (most recent call last):
  File "fail.py", line 32, in <module>
    cuda.memcpy_dtoh(a_doubled, a_gpu)
RuntimeError: cuMemcpyDtoH failed: launch failed
terminate called after throwing an instance of 'std::runtime_error'
  what():  cuMemFree failed: launch failed
zsh: abort      python fail.py
```

What's going on here? First of all, recall that launch failures in CUDA are asynchronous. So the actual traceback does not point to the failed kernel launch, it points to the next CUDA request after the failed kernel.

Next, as far as I can tell, a CUDA context becomes invalid after a launch failure, and all following CUDA calls in that context fail. Now, that includes cleanup (see the `cuMemFree` in the traceback?) that PyCuda tries to perform automatically. Here, a bit of PyCuda's C++ heritage shows through. While performing cleanup, we are processing an exception (the launch failure reported by `cuMemcpyDtoH`). If another exception occurs during exception processing, C++ gives up and aborts the program with a message.

In principle, this could be handled better. If you're willing to dedicate time to this, I'll likely take your patch.

1.7.3 Are the CUBLAS APIs available via PyCuda?

No. I would be more than happy to make them available, but that would be mostly either-or with the rest of PyCuda, because of the following sentence in the CUDA programming guide:

> [CUDA] is composed of two APIs:
>
> - A low-level API called the CUDA driver API,
> - A higher-level API called the CUDA runtime API that is implemented on top of the CUDA driver API.
>
> These APIs are mutually exclusive: An application should use either one or the other.

PyCuda is based on the driver API. CUBLAS uses the high-level API. Once *can* violate this rule without crashing immediately. But sketchy stuff does happen. Instead, for BLAS-1 operations, PyCuda comes with a class called `pycuda.gpuarray.GPUArray` that essentially reimplements that part of CUBLAS.

If you dig into the history of PyCuda, you'll find that, at one point, I did have rudimentary CUBLAS wrappers. I removed them because of the above issue. If you would like to make CUBLAS wrappers, feel free to use these rudiments as a starting point. That said, Arno Pähler's python-cuda has complete `ctypes`-based wrappers for CUBLAS. I don't think they interact natively with numpy, though.

1.7.4 I've found some nice undocumented function in PyCuda. Can I use it?

Of course you can. But don't come whining if it breaks or goes away in a future release. Being open-source, neither of these two should be show-stoppers anyway, and we welcome fixes for any functionality, documented or not.

The rule is that if something is documented, we will in general make every effort to keep future version backward compatible with the present interface. If it isn't, there's no such guarantee.

1.7.5 I have <insert random compilation problem> with gcc 4.1 or older. Help!

Try adding:

```
CXXFLAGS = ['-DBOOST_PYTHON_NO_PY_SIGNATURES']
```

to your `pycuda/siteconf.py` or `$HOME/.aksetup-defaults.py`.

1.8 User-visible Changes

1.8.1 Version 0.92

Note: If you're upgrading from prior versions, you may delete the directory `$HOME/.pycuda-compiler-cache` to recover now-unused disk space.

Note: During this release time frame, I had the honor of giving a talk on PyCuda for a class that a group around Nicolas Pinto was teaching at MIT. If you're interested, the slides for it are available.

Warning: Version 0.92 is currently a release candidate and therefore has a somewhat higher likelihood of bugs.

- Make `pycuda.tools.DeviceMemoryPool` official functionality, after numerous improvements. Add `pycuda.tools.PageLockedMemoryPool` for pagelocked memory, too.
- Properly deal with automatic cleanup in the face of several contexts.
- Fix compilation on Python 2.4.
- Fix 3D arrays. (Nicolas Pinto)

- Improve error message when **nvcc** is not found.
- Automatically run Python GC before throwing out-of-memory errors.
- Allow explicit release of memory using `pycuda.driver.DeviceAllocation.free()`, `pycuda.driver.HostAllocation.free()`, `pycuda.driver.Array.free()`, `pycuda.tools.PooledDeviceAllocation.free()`, `pycuda.tools.PooledHostAllocation.free()`.
- Make configure switch `./configure.py --cuda-trace` to enable API tracing.
- Add a documentation chapter and examples on *Metaprogramming with PyCuda*.
- Add `pycuda.gpuarray.empty_like()` and `pycuda.gpuarray.zeros_like()`.
- Add and document `pycuda.gpuarray.GPUArray.mem_size` in anticipation of stride/pitch support in `pycuda.gpuarray.GPUArray`.
- Merge Jozef Vesely's MD5-based RNG.
- Document `pycuda.driver.from_device()` and `pycuda.driver.from_device_like()`.
- Add `pycuda.elementwise.ElementwiseKernel`.
- Various documentation improvements. (many of them from Nicholas Tung)
- Move PyCuda's compiler cache to the system temporary directory, rather than the users home directory.

1.8.2 Version 0.91

- Add support for compiling on CUDA 1.1. Added version query `pycuda.driver.get_version()`. Updated documentation to show 2.0-only functionality.
- Support for Windows and MacOS X, in addition to Linux. (Gert Wohlgemuth, Cosmin Stejerean, Znah on the Nvidia forums, and David Gadling)
- Support more arithmetic operators on `pycuda.gpuarray.GPUArray`. (Gert Wohlgemuth)
- Add `pycuda.gpuarray.arange()`. (Gert Wohlgemuth)
- Add `pycuda.curandom`. (Gert Wohlgemuth)
- Add `pycuda.cumath`. (Gert Wohlgemuth)
- Add `pycuda.autoinit`.
- Add `pycuda.tools`.
- Add `pycuda.tools.DeviceData` and `pycuda.tools.OccupancyRecord`.
- `pycuda.gpuarray.GPUArray` parallelizes properly on GTX200-generation devices.
- Make `pycuda.driver.Function` resource usage available to the program. (See, e.g. `pycuda.driver.Function.registers`.)
- Cache kernels compiled by `pycuda.driver.SourceModule`. (Tom Annau)
- Allow for faster, prepared kernel invocation. See `pycuda.driver.Function.prepare()`.
- Added memory pools, at `pycuda.tools.DeviceMemoryPool` as experimental, undocumented functionality. For some workloads, this can cure the slowness of `pycuda.driver.mem_alloc()`.
- Fix the *memset* family of functions.

- Improve *Error Reporting*.
- Add *order* parameter to `pycuda.driver.matrix_to_array()` and `pycuda.driver.make_multichannel_2d_array()`.

1.9 Acknowledgments

- Gert Wohlgemuth ported PyCuda to MacOS X and contributed large parts of `pycuda.gpuarray.GPUArray`.
- Znah on the Nvidia forums contributed fixes for Windows XP.
- Cosmin Stejerean provided multiple patches for PyCuda's build system.
- Tom Annau contributed an alternative SourceModule compiler cache as well as Windows build insight.
- Nicholas Tung improved PyCuda's documentation.
- Jozef Vesely contributed a massively improved random number generator derived from the RSA Data Security, Inc. MD5 Message Digest Algorithm.

1.10 Licensing

Note that this guide will not explain CUDA programming and technology. Please refer to Nvidia's programming documentation for that.

PyCuda also has its own web site, where you can find updates, new versions, documentation, and support.

CHAPTER

TWO

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

P

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

Z