



Denglin Hamming™ V2

PyCUDA API

DL-DG/SW-048A-01

2023-12-15

Copyright©苏州登临科技有限公司，2019 - 2025，版权所有。

未经苏州登临科技有限公司事先书面同意，不得以任何形式或方式复制或传播本文件的任何部分。

商标和许可



和其它苏州登临科技有限公司的其它登临科技的图标为苏州登临科技有限公司的商标。本手册中提及的所有其他商标均为其各自所有者的财产。

通知

所购买的产品、服务和特性由苏州登临科技有限公司与客户签订的合同规定。本文件中描述的所有或部分产品、服务和特性可能不在采购范围或使用范围内。除非合同中另有规定，本文件中的所有声明、信息和建议均按“原样”提供，无任何明示或暗示的保证或陈述。

本手册中的信息如有更改，恕不另行通知。本文件在编制过程中已尽一切努力确保内容的准确性，本文件中的所有声明、信息和建议不构成任何明示或暗示的保证。

苏州登临科技有限公司

苏州工业园区扬富路11号南岸新地一期商务楼5号1101室，江苏，中国

<http://www.denglin.ai>

Email : support@denglin.ai

更新历史

版本	更新描述
01	第一次发布

章节目录

- 1 Version Queries
- 2 Error Reporting
- 3 Constants
- 4 Devices and Contexts
- 5 Concurrency and Streams
- 6 Memory
 - 6.1 Global Device Memory
 - 6.2 Pagedlocked Host Memory
 - 6.3 Arrays
 - 6.4 Initializing Device Memory
 - 6.5 Unstructured Memory Transfers
 - 6.6 Structured Memory Transfers
- 7 Code on the Device: Modules and Functions
- 8 PyCUDA Sample

1 Version Queries

`pycuda.VERSION`

Gives the numeric version of PyCUDA as a variable-length tuple of integers. Enables easy version checks such as:

```
pycuda.VERSION >= (0, 93)
```

`pycuda.VERSION_STATUS`

A text string such as `"rc4"` or `"beta"` qualifying the status of the release.

`pycuda.VERSION_TEXT`

The full release name (such as `"0.93rc4"`) in string form.

2 Error Reporting

`pycuda.driver.Error`

Base exception class of all PyCUDA errors.

`pycuda.driver.CompileError`

Thrown when `pycuda.compiler.SourceModule` compilation fails.

Parameters:

- msg
- command_line
- stdout
- stderr

3 Constants

class `pycuda.driver.ctx_flags`

Flags for `Device.make_context()`.

Attributes:

- `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`
Field to other threads while waiting for CUDA calls to complete.
- `SCHED_MASK`
Mask of valid scheduling flags in this bitfield.
- `SCHED_BLOCKING_SYNC`
Use blocking synchronization.
- `MAP_HOST`
Support mapped pinned allocations.
- `LMEM_RESIZE_TO_MAX`
Keep local memory allocation after launch.
- `FLAGS_MASK`
Mask of valid flags in this bitfield.

class `pycuda.driver.event_flags`

Flags for class `Event`.

- `DEFAULT`
- `BLOCKING_SYNC`
- `DISABLE_TIMING`
- `INTERPROCESS`

class `pycuda.driver.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
- MAX_SHARED_MEMORY_PER_BLOCK
- MAX_REGISTERS_PER_BLOCK
- KERNEL_EXEC_TIMEOUT
- INTEGRATED
- CAN_MAP_HOST_MEMORY
- COMPUTE_MODE
See `compute_mode`.
- CONCURRENT_KERNELS
- MAXIMUM_TEXTURE1D_WIDTH
- MAXIMUM_TEXTURE2D_WIDTH
- MAXIMUM_TEXTURE2D_HEIGHT
- MAXIMUM_TEXTURE3D_WIDTH
- MAXIMUM_TEXTURE3D_HEIGHT
- MAXIMUM_TEXTURE3D_DEPTH
- MAXIMUM_TEXTURE2D_ARRAY_WIDTH
- MAXIMUM_TEXTURE2D_ARRAY_HEIGHT
- MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES
version added: 0.94
- MAXIMUM_TEXTURE2D_LAYERED_WIDTH
- MAXIMUM_TEXTURE2D_LAYERED_HEIGHT
- MAXIMUM_TEXTURE2D_LAYERED_LAYERS
- MAXIMUM_TEXTURE1D_LAYERED_WIDTH
- MAXIMUM_TEXTURE1D_LAYERED_LAYERS
version added: 2011.1
- SURFACE_ALIGNMENT
version added: 0.94
- ECC_ENABLED

- PCI_BUS_ID
- PCI_DEVICE_ID
- TCC_DRIVER
- MEMORY_CLOCK_RATE
- GLOBAL_MEMORY_BUS_WIDTH
- L2_CACHE_SIZE
- MAX_THREADS_PER_MULTIPROCESSOR
- ASYNC_ENGINE_COUNT
- UNIFIED_ADDRESSING
- MAXIMUM_TEXTURE2D_GATHER_WIDTH
- MAXIMUM_TEXTURE2D_GATHER_HEIGHT
- MAXIMUM_TEXTURE3D_WIDTH_ALTERNATE
- MAXIMUM_TEXTURE3D_HEIGHT_ALTERNATE
- MAXIMUM_TEXTURE3D_DEPTH_ALTERNATE
- PCI_DOMAIN_ID
- TEXTURE_PITCH_ALIGNMENT
- MAXIMUM_TEXTURECUBEMAP_WIDTH
- MAXIMUM_TEXTURECUBEMAP_LAYERED_WIDTH
- MAXIMUM_TEXTURECUBEMAP_LAYERED_LAYERS
- MAXIMUM_SURFACE1D_WIDTH
- MAXIMUM_SURFACE2D_WIDTH
- MAXIMUM_SURFACE2D_HEIGHT
- MAXIMUM_SURFACE3D_WIDTH
- MAXIMUM_SURFACE3D_HEIGHT
- MAXIMUM_SURFACE3D_DEPTH
- MAXIMUM_SURFACE1D_LAYERED_WIDTH
- MAXIMUM_SURFACE1D_LAYERED_LAYERS
- MAXIMUM_SURFACE2D_LAYERED_WIDTH
- MAXIMUM_SURFACE2D_LAYERED_HEIGHT
- MAXIMUM_SURFACE2D_LAYERED_LAYERS
- MAXIMUM_SURFACECUBEMAP_WIDTH
- MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH
- MAXIMUM_SURFACECUBEMAP_LAYERED_LAYERS
- MAXIMUM_TEXTURE1D_LINEAR_WIDTH
- MAXIMUM_TEXTURE2D_LINEAR_WIDTH
- MAXIMUM_TEXTURE2D_LINEAR_HEIGHT

- MAXIMUM_TEXTURE2D_LINEAR_PITCH
- MAXIMUM_TEXTURE2D_MIPMAPPED_WIDTH
- MAXIMUM_TEXTURE2D_MIPMAPPED_HEIGHT
- COMPUTE_CAPABILITY_MAJOR
- COMPUTE_CAPABILITY_MINOR
- MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH
- STREAM_PRIORITIES_SUPPORTED
- GLOBAL_L1_CACHE_SUPPORTED
- LOCAL_L1_CACHE_SUPPORTED
- MAX_SHARED_MEMORY_PER_MULTIPROCESSOR
- MAX_REGISTERS_PER_MULTIPROCESSOR
- MANAGED_MEMORY
- MULTI_GPU_BOARD
- MULTI_GPU_BOARD_GROUP_ID
- HOST_NATIVE_ATOMIC_SUPPORTED
- SINGLE_TO_DOUBLE_PRECISION_PERF_RATIO
- PAGEABLE_MEMORY_ACCESS
- CONCURRENT_MANAGED_ACCESS
- COMPUTE_PREEMPTION_SUPPORTED
- CAN_USE_HOST_POINTER_FOR_REGISTERED_MEM
- MAX_SHARED_MEMORY_PER_BLOCK_OPTIN
- PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES
- IRECT_MANAGED_MEM_ACCESS_FROM_HOST
- HANDLE_TYPE_POSIX_FILE_DESCRIPTOR_SUPPORTED
- HANDLE_TYPE_WIN32_HANDLE_SUPPORTED
- HANDLE_TYPE_WIN32_KMT_HANDLE_SUPPORTED
- MAX_PERSISTING_L2_CACHE_SIZE
- MAX_BLOCKS_PER_MULTIPROCESSOR
- GENERIC_COMPRESSION_SUPPORTED
- READ_ONLY_HOST_REGISTER_SUPPORTED

class `pycuda.driver.pointer_attribute`

- CONTEXT
- MEMORY_TYPE
- DEVICE_POINTER
- HOST_POINTER

class `pycuda.driver.profiler_output_mode`

- KEY_VALUE_PAIR
- CSV

class `pycuda.driver.function_attribute`

Flags for `Function.get_attribute`.

- MAX_THREADS_PER_BLOCK
- SHARED_SIZE_BYTES
- CONST_SIZE_BYTES
- LOCAL_SIZE_BYTES
- NUM_REGS
- PTX_VERSION
- BINARY_VERSION
- CACHE_MODE_CA
- MAX_DYNAMIC_SHARED_SIZE_BYTES
- PREFERRED_SHARED_MEMORY_CARVEOUT
- MAX

class `pycuda.driver.func_cache`

See `Function.set_cache_config`.

- PREFER_NONE
- PREFER_SHARED
- PREFER_L1
- PREFER_EQUAL

class `pycuda.driver.shared_config`

See `Function.set_shared_config`.

- DEFAULT_BANK_SIZE
- FOUR_BYTE_BANK_SIZE
- EIGHT_BYTE_BANK_SIZE

class `pycuda.driver.array_format`

- UNSIGNED_INT8
- UNSIGNED_INT16
- UNSIGNED_INT32
- SIGNED_INT8
- SIGNED_INT16
- SIGNED_INT32
- HALF
- FLOAT

class `pycuda.driver.array3d_flags`

- 2DARRAY
Deprecated--use :attr: `LAYERED`.
- LAYERED
- SURFACE_LDST
- CUBEMAP TEXTURE_GATHER

class `pycuda.driver.address_mode`

- WRAP
- CLAMP
- MIRROR
- BORDER

class `pycuda.driver.filter_mode`

- POINT
- LINEAR

class `pycuda.driver.memory_type`

- HOST
- DEVICE
- ARRAY

class `pycuda.driver.compute_mode`

- DEFAULT
- PROHIBITED
- EXCLUSIVE_PROCESS

class `pycuda.driver.host_alloc_flags`

Flags to be used to allocate `pagelocked_memory`.

- PORTABLE
- DEVICEMAP
- WRITECOMBINED

class `pycuda.driver.limit`

Limit values for `Context.get_limit` and `Context.set_limit`.

- STACK_SIZE
- PRINTF_FIFO_SIZE
- MALLOC_HEAP_SIZE

登临科技保密材料

4 Devices and Contexts

`pycuda.driver.set_cluster_mask (mask)`

Config device cluster mask.

- mask
cluster_mask, int

`pycuda.driver.get_device ()`

Get current device id, return an int value, it's range are 0 to (get_device_count() - 1).

`pycuda.driver.set_device (device)`

Set device to be active device.

- device
device id, an int value.

`pycuda.driver.get_device_name (device)`

Return the device's name.

- device
device id, an int value.

`pycuda.driver.get_device_count ()`

Return the number of available devices.

`pycuda.driver.get_cluster_count (device)`

Return the number of cluster on the device .

- device
device id, an int value.

`pycuda.driver.get_version ()`

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

`pycuda.driver.init (flags=0)`

Initialize CUDA.

Warning:

This must be called before any other function in this module.

See also `pycuda.autoinit`.

class `pycuda.driver.Device(number)`

class `pycuda.driver.Device(pci_bus_id)`

A handle to the number'th CUDA device.

- `count()`
Return the number of CUDA devices found.
- `name()`
- `pci_bus_id()`
- `compute_capability()`
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.
All `device_attribute` values may also be directly read as (lower-case) attributes on the `Device` object itself, e.g. `dev.clock_rate`.
- `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.
- `retain_primary_context()`
Return the `Context` obtained by retaining the device's primary context, which is the one used by the CUDA runtime API. Unlike `Context.make_context`, the newly-created context is not made current.
- `can_access_peer(dev)`
- `__hash__()`
- `__eq__()`
- `__ne__()`

class `pycuda.driver.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.
- static `pop()`
Remove any context from the top of the context stack, deactivating it.
- 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.
- static `set_limit(limit, value)`
See `limit` for possible values of *limit*.
- static `get_limit(limit)`
See `limit` for possible values of *limit*.
- static `set_cache_config(cc)`
See `func_cache` for possible values of *cc*.
- static `get_cache_config()`
Return a value from `func_cache`.
- static `set_shared_config(sc)`
See `shared_config` for possible values of *sc*.
- static `get_shared_config()`
Return a value from `shared_config`.
- `get_api_version()`
Return an integer API version number.
- `enable_peer_access(peer, flags=0)`
- `disable_peer_access(peer, flags=0)`

5 Concurrency and Streams

class `pycuda.driver.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.
- `wait_for_event(evt)`
Enqueues a wait for the given `Event` instance.

class `pycuda.driver.Event (flags=0)`

An event is a temporal 'marker' in a `Stream` that allows taking the time between two events--such as the time required to execute a kernel. An event's time is recorded when the `Stream` has finished all tasks enqueued before the `record` call.

See `event_flags` for values for the *flags* parameter.

- `record (stream=None)`
Insert a recordinoint for *self* into the `Stream` *stream*. Return *self*.
- `synchronize()`
Wait until the device execution stream reaches this event. Return *self*.
- `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*. Use this method as `end.time_since(start)`. Note that this method will fail with an "invalid value" error if either of the events has not been reached yet. Use `synchronize` to ensure that the event has been reached.
- `time_till(event)`
Return the time in milliseconds that has passed between *event* and *self*. Use this method as `start.time_till(end)`. Note that this method will fail with an "invalid value" error if either of the events has not been reached yet. Use `synchronize` to ensure that the event has been reached.
- `ipc_handle()`
Return a `bytes` object representing an IPC handle to this event.
- static `from_ipc_handle(handle)`

6 Memory

6.1 Global Device Memory

`pycuda.driver.mem_get_info()`

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

`pycuda.driver.mem_alloc(bytes)`

Return a `DeviceAllocation` object representing a linear piece of device memory.

`pycuda.driver.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.

`pycuda.driver.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*.

`pycuda.driver.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*.

`pycuda.driver.mem_alloc_pitch(width, height, access_size)`

Allocates a linear piece of device memory at least *width* bytes wide and *height* rows high that can 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 `pycuda.driver.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.

- `as_buffer(size, offset=0)`

Return the pointer encapsulated by *self* as a Python buffer object, with the given *size* and, optionally, *offset*.

```
pycuda.driver.mem_get_ipc_handle (devptr)
```

Return an opaque `bytes` object representing an IPC handle to the device pointer *devptr*.

```
class pycuda.driver.IPCMemoryHandle (ipc_handle,
flags=ipc_mem_flags.LAZY_ENABLE_PEER_ACCESS)
```

Objects of this type can be used in the same ways as a `DeviceAllocation`.

- `close()`

```
class pycuda.driver.PointerHolderBase
```

A base class that facilitates casting to pointers within PyCUDA. This allows the user to construct custom pointer types that may have been allocated by facilities outside of PyCUDA proper, but still need to be objects to facilitate RAI. The user needs to supply one method to facilitate the pointer cast:

- `get_pointer()`

Return the pointer encapsulated by *self*.

- `as_buffer (size, offset=0)`

Return the pointer encapsulated by *self* as a Python buffer object, with the given *size* and, optionally, *offset*.

6.2 Pagelocked Host Memory

Pagelocked Allocation

```
pycuda.driver.pagelocked_empty (shape, dtype, order="C", mem_flags=0)
```

Allocate a pagelocked `numpy.ndarray` of *shape*, *dtype* and *order*.

mem_flags may be one of the values in `host_alloc_flags`.

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

```
pycuda.driver.pagelocked_zeros (shape, dtype, order="C", mem_flags=0)
```

Like `pagelocked_empty`, but initialized to zero.

```
pycuda.driver.pagelocked_empty_like (array, mem_flags=0)
```

```
pycuda.driver.pagelocked_zeros_like (array, mem_flags=0)
```

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

class `pycuda.driver.PageLockedHostAllocation`

Inherits from `HostPointer`.

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 `:mod:`numpy` array`) is an error and will lead to undefined behavior.
- `get_flags()`
Return a bit field of values from `host_alloc_flags`.

class `pycuda.driver.HostAllocation`

A deprecated name for `PageLockedHostAllocation`.

class `pycuda.driver.HostPointer`

Represents a page-locked host pointer.

- `get_device_pointer()`
Return a device pointer that indicates the address at which this memory is mapped into the device's address space.

6.3 Arrays

`pycuda.driver.ArrayDescriptor`

- `width`
- `height`
- `format`
A value of type `array_format`.
- `num_channels`

`pycuda.driver.Array(descriptor)`

A 2D memory block that can only be accessed via texture references.

descriptor can be of type `ArrayDescriptor`.

- `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 :class: `ArrayDescriptor` object for this 2D array, like the one that was used to create it.

- `handle()`

Return an :class: `int` representing the address in device memory where this array resides.

`pycuda.driver.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 `"F"`.

6.4 Initializing Device Memory

`pycuda.driver.memset_d8(dest, data, count)`

`pycuda.driver.memset_d16(dest, data, count)`

`pycuda.driver.memset_d32(dest, data, count)`

Fill array with *data*.

Note:

The count is the number of elements, not bytes.

`pycuda.driver.memset_d2d8(dest, pitch, data, width, height)`

`pycuda.driver.memset_d2d16(dest, pitch, data, width, height)`

`pycuda.driver.memset_d2d32(dest, pitch, data, width, height)`

Fill a two-dimensional array with *data*.

`pycuda.driver.memset_d8_async(dest, data, count, stream=None)`

`pycuda.driver.memset_d16_async(dest, data, count, stream=None)`

`pycuda.driver.memset_d32_async(dest, data, count, stream=None)`

Fill array with *data* asynchronously, optionally serialized via *stream*.

`pycuda.driver.memset_d2d8_async(dest, pitch, data, width, height, stream=None)`

`pycuda.driver.memset_d2d16_async(dest, pitch, data, width, height, stream=None)`

`pycuda.driver.memset_d2d32_async(dest, pitch, data, width, height, stream=None)`

Fill a two-dimensional array with *data* asynchronously, optionally serialized via *stream*.

6.5 Unstructured Memory Transfers

`pycuda.driver.memcpy_htod(dest, src)`

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.

`pycuda.driver.memcpy_htod_async(dest, src, stream=None)`

Copy from the Python buffer *src* to the device pointer *dest* (an `int` or a `DeviceAllocation`) asynchronously, optionally serialized via *stream*. The size of the copy is determined by the size of the buffer.

src must be page-locked memory, see, e.g. `pagelocked_empty`.

`pycuda.driver.memcpy_dtoh(dest, src)`

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.

`pycuda.driver.memcpy_dtoh_async(dest, src, stream=None)`

Copy from the device pointer *src* (an `int` or a `DeviceAllocation`) to the Python buffer *dest* asynchronously, optionally serialized via *stream*. The size of the copy is determined by the size of the buffer.

dest must be page-locked memory, see, e.g. `pagelocked_empty`.

`pycuda.driver.memcpy_dtod(dest, src, size)`

`pycuda.driver.memcpy_dtod_async(dest, src, size, stream=None)`

`pycuda.driver.memcpy_peer(dest, src, size, dest_context=None, src_context=None)`

`pycuda.driver.memcpy_peer_async(dest, src, size, dest_context=None, src_context=None, stream=None)`

`pycuda.driver.memcpy_dtoa(ary, index, src, len)`

`pycuda.driver.memcpy_atod(dest, ary, index, len)`

`pycuda.driver.memcpy_htoa(ary, index, src)`

`pycuda.driver.memcpy_atoh(dest, ary, index)`

`pycuda.driver.memcpy_atoa(dest, dest_index, src, src_index, len)`

6.6 Structured Memory Transfers

class `pycuda.driver.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.
- `set_src_unified(buffer)`
Same as `set_src_host`, except that *buffer* may also correspond to device memory.
Requires unified addressing.
- `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.
- `set_dst_unified(buffer)`
Same as `set_dst_host`, except that *buffer* may also correspond to device memory.
Requires unified addressing.
- `width_in_bytes`
Number of bytes to copy for each row in the transfer.
- `height`

Number of rows to copy.

- `__call__` ([aligned=False])

Perform the specified memory copy, waiting for it to finish. If *aligned* is *False*, tolerate device-side misalignment for device-to-device copies that may lead to 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.

登临科技保密材料

7 Code on the Device: Modules and Functions

class `pycuda.driver.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 a tuple `(device_ptr, size_in_bytes)` giving the device address and size of the global *name*.

The main use of this method is to find the address of pre-declared `__constant__` arrays so they can be filled from the host before kernel invocation.

`pycuda.driver.module_from_file(filename)`

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

`pycuda.driver.module_from_buffer(buffer, options=[], message_handler=None)`

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

PARAMETERS:

- **options** - A list of tuples `(jit_option, value)`.
- **message_handler** - A callable that is called with a arguments of `(compile_success_bool, info_str, error_str)` which allows the user to process error and warning messages from the PTX compiler.

class `pycuda.driver.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 tuple of up to three integer entries, the number of thread blocks to launch, as a multi-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`.

arg1 through *argn* 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 `:mod: 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.

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 `:exc: 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`.

Note:

grid with more than two dimensions.

- `prepare(arg_types, 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.
(In addition, PyCUDA understands 'F' and 'D' for single- and double precision floating point numbers.)
- 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, block, *args, shared_size=0)`

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

- `prepared_timed_call` (grid, block, *args, shared_size=0)

Invoke `self` using `launch_grid`, with `args`, a grid size of `grid`, and a block size of `block`. 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, block, stream, *args, shared_size=0)

Invoke `self` using `launch_grid_async`, with `args`, a grid size of `grid`, and a block size of `block`, 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.

- `get_attribute` (attr)

Return one of the attributes given by the `function_attribute` value `attr`.

All `function_attribute` values may also be directly read as (lower-case) attributes on the `Function` object itself, e.g. `func.num_regs`.

- `set_attribute` (attr, value)

Set one of the (settable) attributes given by the `function_attribute` value `attr` to `value`.

- `set_cache_config` (fc)

See `func_cache` for possible values of `fc`.

- `set_shared_config` (sc)

See `shared_config` for possible values of `sc`.

- `set_shared_size` (bytes)

Set *shared* to be the number of bytes available to the kernel in *extern_shared_* arrays.

Warning:

Deprecated as of version 2011.1.

- `set_block_shape` (x, y, z)

Set the thread block shape for this function.

Warning:

Deprecated as of version 2011.1.

- `param_set` (arg1, ... argn)

Set the thread block shape for this function.

Warning:

Deprecated as of version 2011.1.

- `param_set_size` (bytes)

Size the parameter space to *bytes*.

Warning:

Deprecated as of version 2011.1.

- `param_seti` (offset, value)

Set the integer at *offset* in the parameter space to *value*.

Warning:

Deprecated as of version 2011.1.

- `param_setf` (offset, value)

Set the float at *offset* in the parameter space to *value*.

Warning:

Deprecated as of version 2011.1.

- `launch` ()

Launch a single thread block of *self*.

Warning:

Deprecated as of version 2011.1.

- `launch_grid` (width, height)

Launch a width*height grid of thread blocks of *self*.

Warning:

Deprecated as of version 2011.1.

- `launch_grid_async` (width, height, stream)

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

Warning:

Deprecated as of version 2011.1.

class `pycuda.driver.ArgumentHandler (array)`

class `pycuda.driver.In (array)`

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

class `pycuda.driver.Out (array)`

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

class `pycuda.driver.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.

8 PyCUDA Sample

```
import pycuda

pycuda.VERSION_TEXT

device_mem = pycuda.driver.mem_alloc(100)
device_mem.free()
```