Introduction to CUDA Python with Numba

08 September 2024 05:47

Part-1

"Introduction to CUDA Python with Numba refers to using the Numba library to write Python code that can run on NVIDIA GPUs using CUDA (Compute Unified Device Architecture) technology. Here's what this entails:

Overview of CUDA Python with Numba

1. **CUDA:**
   - **CUDA** is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows developers to use NVIDIA GPUs for general-purpose processing (GPGPU), which can significantly accelerate computation-heavy tasks.
   - CUDA enables you to write code that can leverage the massively parallel architecture of modern GPUs, allowing for faster execution of tasks compared to traditional CPU-based computation.
2. **Numba:**
   - **Numba** is an open-source just-in-time compiler for Python that translates a subset of Python and NumPy code into optimized machine code at runtime using LLVM (Low-Level Virtual Machine). It can be used to accelerate numerical computations by compiling Python code to run on CPUs and GPUs.
   - Numba provides support for CUDA, allowing you to write GPU-accelerated Python code without needing to manually manage CUDA C++ code
3. **CUDA Support:**
   - With Numba's CUDA support, you can write CUDA kernels in Python. These kernels are functions that run on the GPU and are executed by multiple threads in parallel. Numba handles the complexities of interacting with CUDA, making it easier to write GPU-accelerated code.

(Q) What is Accelerated computing ??
Ans) **Accelerated computing** refers to the use of specialized hardware or software to speed up computing tasks beyond what is achievable with traditional general-purpose processors (CPUs). This approach leverages various types of processors and technologies to perform specific types of computations more efficiently, thereby improving performance and reducing execution time for various applications.

By the time you complete this section you will be able to:

- **Use Numba to compile Python functions for the CPU.**
- **Understand how Numba compiles Python functions.**
- **GPU accelerate NumPy ufuncs.**
- **GPU accelerate hand-written vectorized functions.**
- **Optimize data transfers between the CPU host and GPU device.**

What is Numba?

Numba is a **just-in-time**, **type-specializing**, **function compiler** for accelerating **numerically-focused** Python for either a CPU or GPU. That's a long list, so let's break down those terms:

- **function compiler**: Numba compiles Python functions, not entire applications, and not parts of functions. Numba does not replace your Python interpreter, but is just another Python module that can turn a function into a (usually) faster function.
- **type-specializing**: Numba speeds up your function by generating a specialized implementation for the specific data types you are using. Python functions are designed to operate on generic data types, which makes them very flexible, but also very slow as it store their overhead in the memory which contains the information about the datatype and all which make them slow. In practice, you only will call a function with a small number of argument types, so Numba will generate a fast implementation for each set of types.
- **just-in-time**: Numba translates functions when they are first called. This ensures the compiler knows

what argument types you will be using. This also allows Numba to be used interactively in a Jupyter notebook just as easily as a traditional application.

- **numerically-focused**: Currently, Numba is focused on numerical data types, like int, float, and complex. There is very limited string processing support, and **many string use cases are not going to work well on the GPU**. To get best results with Numba, you will likely be using **NumPy arrays.**

What is numpy and how it is so different from the list in python

NumPy arrays and **Python lists** are both ways to store collections of items, but they have some important differences that make NumPy arrays particularly useful for numerical and scientific computing.

Key Features:

- **Homogeneous Data:** All elements in a NumPy array must be of the same data type (e.g., all integers or all floats).
- **Multidimensional:** Arrays can be one-dimensional (vectors), two-dimensional (matrices), or higher dimensions.
- **Efficient Storage and Computation:** NumPy arrays use less memory and are faster for numerical computations compared to Python lists because they are implemented in C and optimized for performance.
- **Vectorized Operations:** NumPy supports vectorized operations, meaning you can perform operations on entire arrays without using explicit loops, leading to faster and more concise code.

Aside: CUDA C/C++ vs. Numba vs. pyCUDA

By no means is Numba the only way to program with CUDA. By far the most common way to program in CUDA is with the CUDA C/C++ language extensions. With regards to Python, pyCUDA is, in addition to Numba, an alternative to GPU accelerating Python code. We will remained focused on Numba throughout this course, but a quick comparison of the three options just named is worth a mention before we get started, just for a little context.

First Steps: Compile for the CPU

If you recall **Numba can be used to optimize code for either a CPU or GPU**. As an introduction, and before moving onto GPU acceleration, let's write our first Numba function and compile it for the **CPU**. In doing so we will get an easy entrance into Numba syntax, and will also have an opportunity a little later on to compare the performance of CPU optimized Numba code to GPU acclerated Numba code.
The Numba compiler is typically enabled by applying a **function decorator** to a Python function. **Decorators are function modifiers that transform the Python functions they decorate, using a very simple syntax. Here we will use Numba's CPU compilation decorator @jit:**

What is Function Decorator ??

In Python, **function decorators** are a powerful feature that allows you to modify or enhance functions or methods without changing their actual code. They are a type of higher-order function, which means **they take another function as an argument and extend or alter its behavior.**

EXAMPLE

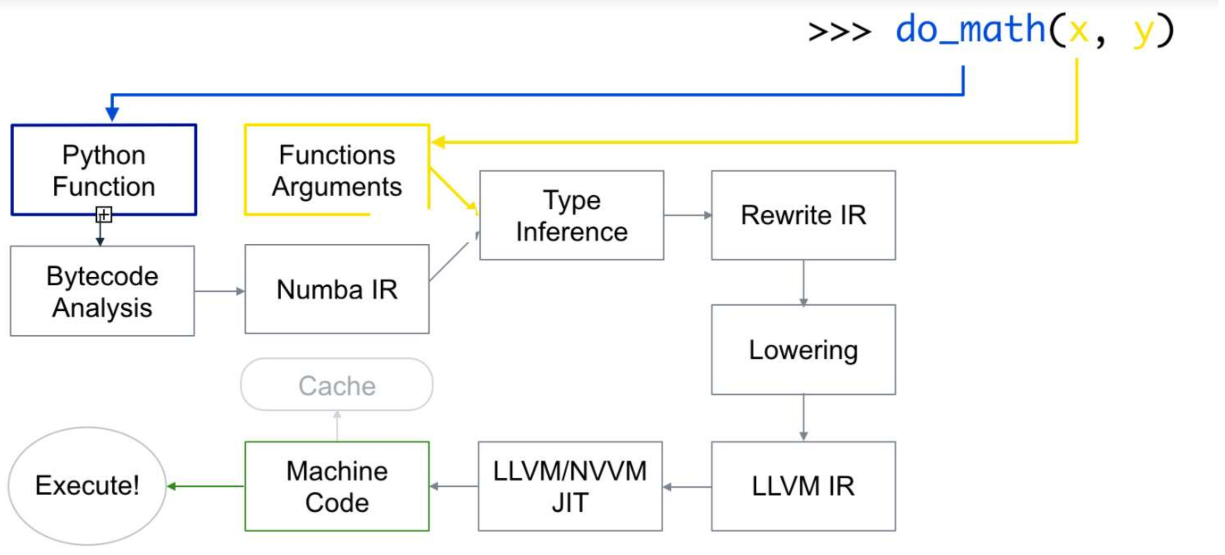
```
@decorator_name
def my_function():
  pass

#USING FUNCTION DECORATOR IT IS EQUIVALENT TO

def my_function():
  pass

my_function = decorator_name(my_function)
```

HOW NUMBA EXECUTE THE CODE



WHY inspect_types() function is important ??

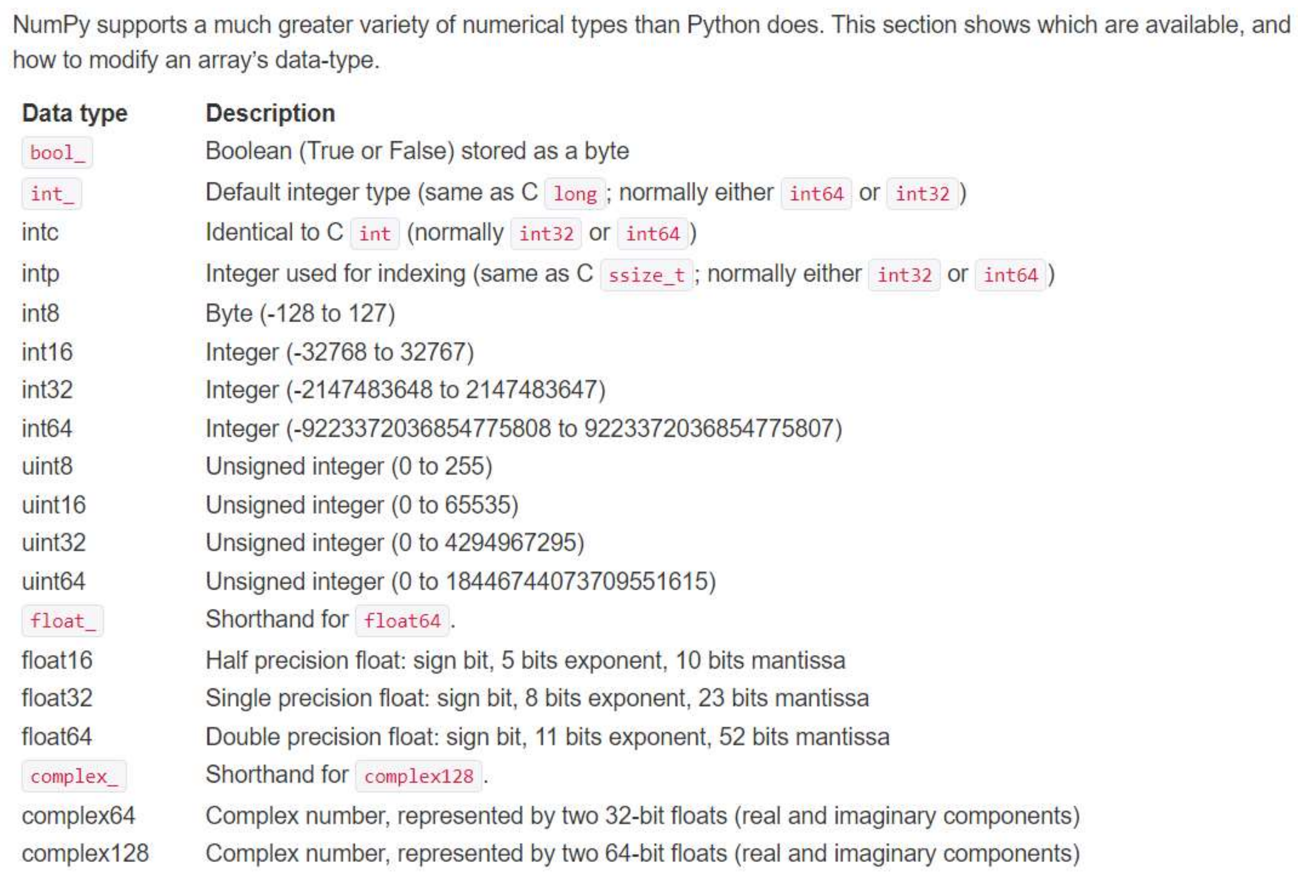
The inspect_types() function in Numba is a useful tool for understanding how Numba has interpreted and compiled the data types of the variables in your function. Here's a detailed explanation of what inspect_types() does and why it's important

What is Type Inference?

Type inference is the process by which a compiler or interpreter determines the data types of expressions and variables without explicit type annotations from the programmer. This means that the system can deduce what types are being used based on the operations and values in the code and it is important so that the effective machine code can be generated by knowing the data types as higher data types may hinder the performance if it is used at the place where it is not needed .

How Data Types Are Decided

1. **Automatic Type Inference:**
    - **Type Inference:** Numba automatically infers data types based on the function's input arguments and operations. For example, if you pass a float64 to a function, Numba will infer that the computations inside the function should use float64.
    - **Intermediate Representation:** During the compilation process, **Numba converts Python objects into its own Intermediate Representation (IR), which is then used to generate machine code.** The IR helps Numba understand what data types are used and how to optimize for them.
2. **Default Behavior:**
    - **Python to Numba Types:** Numba translates Python types to its own types **based on the Python objects passed to functions**. For instance, Python's float corresponds to float64, and Python’s int corresponds to int32 or int64, depending on the platform.
3. **GPU Specifics:**
    - **CUDA Compatibility:** For CUDA-enabled GPUs, **Numba defaults to float32 for single-precision operations and float64 for double-precision operations**. GPUs often have hardware optimizations for float32 operations, so **it's common to use float32 for better performance, unless higher precision is needed**.



NumPy supports a much greater variety of numerical types than Python does. This section shows which are available, and how to modify an array's data-type.

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
intc	Identical to C `int` (normally `int32` or `int64`)
intp	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

Numba Compilation Modes

Numba provides two main compilation modes: **object mode** and **nopython mode**. Each mode handles Python code differently, and understanding these modes helps in writing efficient code.

1. Object Mode

- **Definition:**
  - **Object mode** is the default mode in Numba when it is unable to fully compile the Python code into efficient machine code. In this mode, Numba **does not perform type specialization** and instead **falls back to a more generic, less optimized form of execution.**
- **Behavior:**
  - **Fallback:** When Numba encounters Python code that it cannot compile into machine code (such as **code that uses unsupported Python data structures like dictionaries)**, it uses object mode to handle the execution.
  - **Type Inference:** Object mode does **not enforce strict type inference**. This means Numba can still execute the function but **without the performance benefits of type specialization.**
    At conclusion it means that if the code is not converted to effective machine code due to un supported data types such as dictionaries then it will fall back for more generalize approach and execute the function without any specialization and give more general execution
- **Example:**

```
python
from numba import jit

@jit
def cannot_compile(x):
    return x['key']

# This will execute in object mode
```

```
result = cannot_compile(dict(key='value'))
```

- **Outcome:** In this example, Numba falls back to object mode because dictionaries are not fully supported. The function executes without errors, but it does not benefit from optimizations.

2. Nopython Mode

- **Definition:**
  - **Nopython mode** is a stricter mode where Numba attempts to compile the Python code into machine code without relying on Python objects at runtime. It is designed to achieve the **highest performance by avoiding Python's dynamic features**.
- **Behavior:**
  - **Strict Compilation:** Numba will only compile code if it can convert all Python objects and operations into machine code. **If it encounters unsupported types or operations, it will raise an error.**
  - **Error Handling:** If the code cannot be fully compiled in nopython mode, Numba will throw an exception, indicating what part of the code is problematic.
- **Example:**

```
python
from numba import jit

@jit(nopython=True)
def cannot_compile(x):
    return x['key']

# This will raise an exception because dictionaries are not supported
result = cannot_compile(dict(key='value'))
```

  - **Outcome:** Numba raises an exception because dictionaries are not supported in nopython mode. This helps you identify and correct issues in your code.

Numba provides another decorator njit which is an alias for jit(nopython=True)
@njit: This is a shorthand decorator provided by Numba for convenience. It stands for "No Python JIT" and is used to compile Python functions with Numba's Just-In-Time (JIT) compiler in "nopython" mode. @numba(nopython = True) is the explicit form of the decorator.

Note : Before moving further revise some basics function of the numpy lib such reshape shape random vectorize concept etc

Familiarity with NumPy ufuncs is a prerequisite of this course, but in case you are unfamiliar with them, or in case it has been a while, here is a very brief introduction. If, at the end of this brief introduction, you don't feel comfortable with the basic NumPy mechanisms for array creation and ufuncs, consider the ~1 hour NumPy Quickstart Tutorial.
NumPy has the concept of universal functions ("ufuncs"), which are functions that can take NumPy arrays of varying dimensions, or scalars, and operate on them element-by-element.
As an example we'll use the NumPy add ufunc to demonstrate the basic ufunc mechanism:

In [12]:

```
import numpy as np

a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])

np.add(a, b) # Returns a new NumPy array resulting from adding every element in `a` to every element in `b`
```

Out[12]:

```
array([11, 22, 33, 44])
```

Ufuncs also can combine scalars with arrays:

In [13]:

```
np.add(a, 100) # Returns a new NumPy array resulting from adding 100 to every element in `a`
```

Out[13]:

```
array([101, 102, 103, 104])
```

Arrays of different, but compatible dimensions can also be combined via a technique called *broadcasting*. The lower dimensional array will be replicated to match the dimensionality of the higher dimensional array. If needed, check out the docs for numpy.arange and numpy.ndarray.reshape, both will be used several times throughout this course:

In [14]:

```
c = np.arange(4*4).reshape((4,4))
print('c:', c)

np.add(b, c)
```

```
c: [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

Out[14]:

```
array([[10, 21, 32, 43],
       [14, 25, 36, 47],
       [18, 29, 40, 51],
       [22, 33, 44, 55]])
```

Making ufuncs for the GPU

Numba has the ability to create *compiled* ufuncs, typically a not-so-straighforward process involving C code. With Numba you simply implement a scalar function to be performed on all the inputs, decorate it with @vectorize, and Numba will figure out the broadcast rules for you. For those of you familiar with NumPy's vectorize, Numba's vectorize decorator will be very familiar.

Vectorization in NumPy refers to the process of applying operations on entire arrays or large blocks of data at once, **rather than using explicit loops to perform operations element by elemen**t. This approach leverages low-level optimizations and parallelism provided by NumPy

In this very first example we will use the @vectorize decorator to compile and optimize a ufunc for the **CPU**.

In [ ]:

```
from numba import vectorize

@vectorize
def add_ten(num):
    return num + 10 # This scalar operation will be performed on each element
```

In [ ]:

```
nums = np.arange(10)
add_ten(nums) # pass the whole array into the ufunc, it performs the operation on each element
```

We are generating a ufunc that uses CUDA on the GPU with the addition of giving an explicit type signature and setting the target attribute. The type signature argument describes what types to use both for the ufuncs arguments and return value:

return_value_type(argument1_value_type, argument2_value_type, ...)
Please see the Numba docs for more on available types, as well as for additional information on writing ufuncs with more than one signature
Here is a simple example of a ufunc that will be compiled for a CUDA enabled GPU device. It expects two int64 values and return also an int64 value:

In [ ]:

```
@vectorize(['int64(int64, int64)'], target='cuda') # Type signature and target are required for the GPU
def add_ufunc(x, y):
    return x + y
```

In [ ]:

```
add_ufunc(a, b)
```

For such a simple function call, a lot of things just happened! Numba just automatically:

- Compiled a CUDA kernel to execute the ufunc operation in parallel over all the input elements.
- Allocated GPU memory for the inputs and the output.
- Copied the input data to the GPU.
- Executed the CUDA kernel (GPU function) with the correct kernel dimensions given the input sizes.
- Copied the result back from the GPU to the CPU.

- Returned the result as a NumPy array on the host.

Compared to an implementation in C, the above is remarkably more concise.
You might be wondering how fast our simple example is on the GPU? Let's see:

In [ ]:

```
%timeit np.add(b, c)   # NumPy on CPU
```

In [ ]:

```
%timeit add_ufunc(b, c) # Numba on GPU
```

Wow, the GPU is *a lot slower* than the CPU?? For the time being this is to be expected because we have (deliberately) misused the GPU in several ways in this example. How we have misused the GPU will help clarify what kinds of problems are well-suited for GPU computing, and which are best left to be performed on the CPU:

- **Our inputs are too small**: the GPU achieves performance through parallelism, operating on thousands of values at once. Our test inputs have only 4 and 16 integers, respectively. We need a much larger array to even keep the GPU busy.
- **Our calculation is too simple**: Sending a calculation to the GPU involves quite a bit of overhead compared to calling a function on the CPU. If our calculation does not involve enough math operations (often called "arithmetic intensity"), then the GPU will spend most of its time waiting for data to move around.
- **We copy the data to and from the GPU**: While in some scenarios, paying the cost of copying data to and from the GPU can be worth it for a single function, often it will be preferred to to run several GPU operations in sequence. In those cases, it makes sense to send data to the GPU and keep it there until all of our processing is complete.
- **Our data types are larger than necessary**: Our example uses int64 when we probably don't need it. Scalar code using data types that are 32 and 64-bit run basically the same speed on the CPU, and for integer types the difference may not be drastic, but 64-bit floating point data types may have a significant performance cost on the GPU, depending on the GPU type. Basic arithmetic on 64-bit floats can be anywhere from 2x (Pascal-architecture Tesla) to 24x (Maxwell-architecture GeForce) slower than 32-bit floats. If you are using more modern GPUs (Volta, Turing, Ampere), then this could be far less of a concern. NumPy defaults to 64-bit data types when creating arrays, so it is important to set the dtype attribute or use the ndarray.astype() method to pick 32-bit types when you need them.

Given the above, let's try an example that is faster on the GPU by performing an operation with much greater arithmetic intensity, on a much larger input, and using a 32-bit data type.
Please note: Not all NumPy code will work on the GPU, and, as in the following example, we will need to use the math library's pi and exp instead of NumPy's. Please see the Numba docs for extensive coverage of NumPy support on the GPU.

IMP POINT :- Note that for the CUDA target, we need to use the scalar functions from the math module, not NumPy

numba.cuda.jit Overview

In Numba, the @numba.cuda.jit decorator is used to compile Python functions into GPU kernels, which can then be executed on a CUDA-capable GPU. This is different from typical Numba @jit functions that are optimized for CPU execution.

What is a GPU Kernel?

A GPU kernel is a function that runs on the GPU, and it is typically designed to perform operations on large datasets in parallel. When using @numba.cuda.jit, you're essentially writing code that will be executed on the GPU rather than on the CPU.

GPU-Only Functions

In the context of your explanation:

1. **Element-Wise Operations**: These are common operations that can be efficiently parallelized across many elements. For such operations, you use @numba.vectorize or @numba.guvectorize to create ufuncs (universal functions) that can process arrays element-wise.
2. **Non-Element-Wise Operations**: For operations that are not inherently element-wise or that involve more complex logic, you might need to use @numba.cuda.jit. This allows you to define a function that will be executed on the GPU, but doesn’t necessarily have to operate on arrays element-wise. For example, you might have a function that processes a block of data or performs some computation that involves more than simple element-wise operations.

The device=True Argument

The device=True argument in @numba.cuda.jit specifies that the decorated function is a "device function." Device functions:

- **Can Only Be Called From GPU Code**: Functions decorated with @numba.cuda.jit(device=True) can only be invoked from within other GPU kernels or device functions. They cannot be called from CPU code or from the host side of the program.
- **Cannot Be Called Directly From Host**: This is crucial because it helps to separate the code that runs on the GPU from the code that runs on the CPU. Device functions are meant to be used internally by other GPU code but are not directly accessible from the CPU code.

```
from numba import cuda

# Device function - can only be called from GPU code
@cuda.jit(device=True)
def polar_to_cartesian(r, theta):
    x = r * cuda.cos(theta)
    y = r * cuda.sin(theta)
    return x, y

# Kernel function - can be called from CPU code
@cuda.jit
def kernel_function(data):
    idx = cuda.grid(1)
    if idx < len(data):
        r, theta = data[idx]
        x, y = polar_to_cartesian(r, theta)
        # Do something with x, y

# CPU-side code to launch the kernel
import numpy as np
from numba import cuda

data = np.array([(1.0, 0.5), (2.0, 1.5), (3.0, 2.5)], dtype=np.float32)
d_data = cuda.to_device(data)

kernel_function[1, len(data)](d_data)
```

Allowed Python on the GPU

Compared to Numba on the CPU (which is already limited), Numba on the GPU has more limitations. Supported Python includes:

- if/elif/else
- while and for loops
- Basic math operators
- Selected functions from the math and cmath modules
- Tuples

Managing GPU Memory

Managing GPU memory efficiently is crucial for optimizing performance when using GPUs for computation. The automatic data transfer between the CPU (host) and GPU (device) can become a **bottleneck** if not handled carefully. Here's a detailed explanation of **managing GPU memory, focusing on the use of CUDA Device Arrays and minimizing unnecessary data transfers:**

Key Concepts

1. **Implicit Data Transfer**:
    - **Automatic Handling**: By default, Numba handles data transfer between the CPU and GPU automatically when using functions like @vectorize with the target='cuda' option.
    - **Time Overhead**: While this automatic transfer is convenient, it introduces overhead. Transferring data between the CPU and GPU can be time-consuming, especially for large

datasets.

2. **Minimizing Data Transfers**:
   - **CUDA Best Practices**: It is crucial to minimize the number of data transfers to and from the GPU. Frequent transfers can negate the benefits of GPU acceleration. Instead, try to keep data on the GPU as long as possible and perform all necessary computations there before transferring the final results back to the CPU.

Managing GPU Memory with Numba

1. **Using CUDA Device Arrays**:
   - **Creating Device Arrays**: Use cuda.to_device() to transfer data from the CPU to the GPU and create device arrays. These arrays reside in GPU memory and do not automatically transfer data back to the CPU.
   - **Output Arrays**: To avoid unnecessary data transfers when using GPU functions, create an output array on the GPU using cuda.device_array().
2. **Example Code Explanation**:

```
python
import numpy as np
from numba import vectorize, cuda

# Define a vectorized CUDA function for addition
@vectorize(['float32(float32, float32)'], target='cuda')
def add_ufunc(x, y):
    return x + y

# Initialize host arrays
n = 100000
x = np.arange(n).astype(np.float32)
y = 2 * x

# Baseline performance with host arrays
%timeit add_ufunc(x, y)  # Measures time taken with automatic data transfer

# Transfer arrays to the GPU
x_device = cuda.to_device(x)
y_device = cuda.to_device(y)

# Print device array information
print(x_device)
print(x_device.shape)
print(x_device.dtype)

# Perform addition with device arrays and measure time
%timeit add_ufunc(x_device, y_device)
```

   Explanation:
   - **@vectorize Function**: The add_ufunc function is defined to run on the GPU, performing element-wise addition.
   - **Baseline Timing**: Measures the time taken for operations using host arrays, including data transfer to and from the GPU.
   - **Device Arrays**: x_device and y_device are created on the GPU, reducing the need for data transfer during computations.
   - **Timing with Device Arrays**: This measures the performance of the GPU function when using device arrays, bypassing some of the overhead associated with automatic transfers.
3. **Creating and Using Output Device Arrays**:
   - **Creating Output Array**:

     python
     Copy code

out_device = cuda.device_array(shape=(n,), dtype=np.float32) # Allocate device array
  - This creates an uninitialized device array for storing results.
- **Using the out Argument**:

  python
  Copy code
  %timeit add_ufunc(x_device, y_device, out=out_device)
  - By specifying out=out_device, you direct the add_ufunc function to write the results directly to out_device, avoiding unnecessary intermediate data transfers.
- **Copying Data Back**:

  python
  Copy code
  out_host = out_device.copy_to_host()
  print(out_host[:10])
  - Transfer the results from GPU memory to CPU memory when needed.

Frequently use operation

@jit
@njit
@vectorize
@cuda.jit(device = True)
@vectorize(['return_type(type1 , type2)'] , target = 'cuda')
x_device_array = cuda.to_device(numpy_array)
array = cuda.device_array(shape(n,) , dtype = np.datatype)
%timeit add_ufunc(x_device, y_device, out=out_device)

```
# Modify the body of this function to optimize data transfers and therefore speed up
performance.
# As a constraint, even after you move work to the GPU, make this function return a host array.
def create_hidden_layer(n, greyscales, weights, exp, normalize, weigh, activate ,):
  # Allocate device arrays for inputs
   x_greyscales = cuda.to_device(greyscales)
   x_weights = cuda.to_device(weights)

   # Allocate device array for intermediate results
   normalized_device = cuda.device_array(n, dtype=np.float32)
   weighted_device = cuda.device_array(n, dtype=np.float32)

   normalized = normalize(x_greyscales)
   weighted = weigh(normalized, x_weights)
   activated = activate(x_weights)

   activated_host = activated_device.copy_to_host()

   # The assessment mechanism will expect `activated` to be a host array, so,
   # even after you refactor this code to run on the GPU, make sure to explicitly copy
   # `activated` back to the host.
   return activated
```

PART-02

Custom CUDA Kernels in Python with Numba

Custom CUDA kernels are **user-defined functions written to execute on a GPU** using the CUDA programming model. These kernels allow developers to harness the parallel processing power of GPUs for a wide range of computations that go beyond simple element-wise operations. Custom kernels **provide flexibility and control over the parallel execution and memory management of tasks.**

Objectives

By the time you complete this section you will be able to:

- Write custom CUDA kernels in Python and launch them with an execution configuration.
- Utilize grid stride loops for working in parallel over large data sets and leveraging memory coalescing.
- Use atomic operations to avoid race conditions when working in parallel

The Need for Custom Kernels

Ufuncs are fantastically elegant, and for any scalar operation that ought to be performed **element wise on data**, ufuncs are likely the right tool for the job.
As you are well aware, there are many, if not more, classes of **problems that cannot be solved by applying the same function to each element of a data set**. Consider, **for example, any problem that requires access to more than one element of a data structure in order to calculate its output**, like stencil algorithms, or any problem that cannot be expressed by a **one input value to one output value mapping**, such as a reduction. **Many of these problems are still inherently parallelizable, but cannot be expressed by a ufunc**.

Introduction to CUDA Kernels

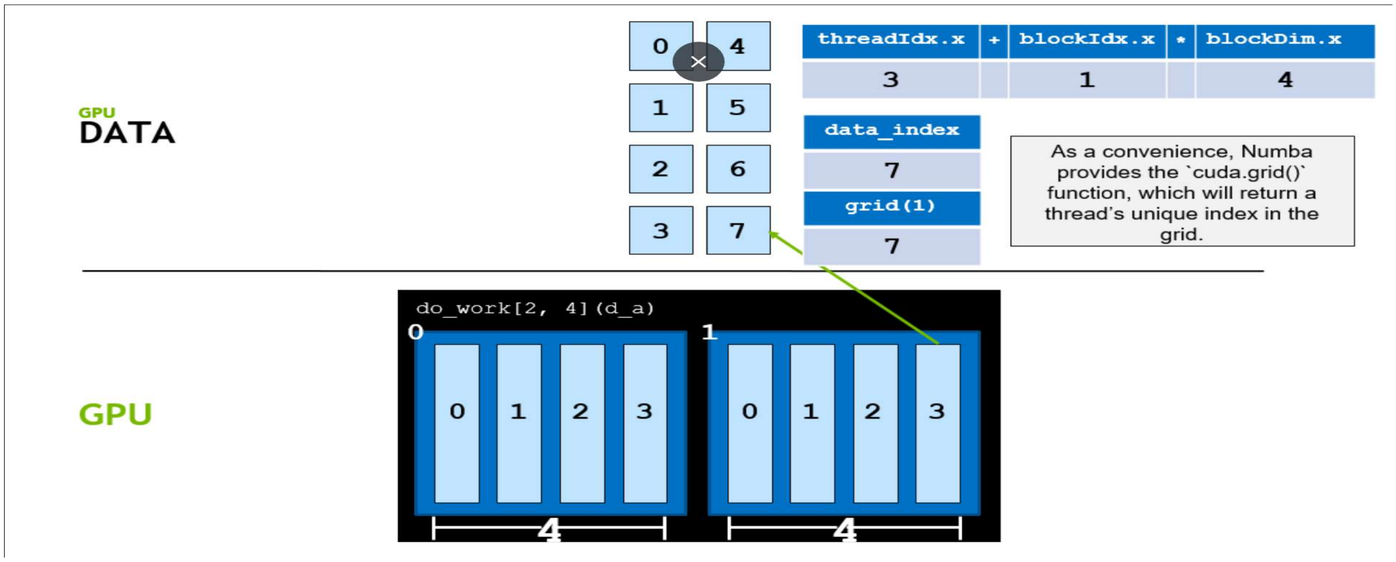
When programming in CUDA, developers write functions for the GPU called **kernels**, which are executed, or in CUDA parlance, **launched**, on the GPU's many cores in parallel **threads**. When kernels are launched, programmers use a special syntax, called an **execution configuration** (also called a launch configuration) to describe the parallel execution's configuration.
The following slides (which will appear after executing the cell below) give a high level introduction to how CUDA kernels can be created to work on large datasets in parallel on the GPU device. Work through the slides and then you will begin writing and executing your own custom CUDA kernels, using the ideas presented in the slides.

In the context of CUDA and parallel computing, a **thread** is a fundamental unit of execution. Here's a detailed explanation of what a thread is and how it functions within the CUDA programming model:

What is a Thread in CUDA?

1. **Definition**:
   - A thread is an **individual sequence of execution instruction**s that runs independently from other threads. In CUDA, threads **execute kernel code on the GPU**. Each thread operates on a unique subset of data and performs computations in parallel with other threads.
2. **Creation and Execution**:
   - When you launch a CUDA kernel, **you specify how many threads will execute the kernel code**. These threads are **created in a grid of block**s, and each thread executes a copy of the kernel code.
3. **Identification**:
   - Threads are identified by their indices within the block and grid. CUDA provides built-in variables to access these indices:
     - threadIdx: The index of the thread within its block.
     - blockIdx: The index of the block within the grid.
     - blockDim: The dimensions (number of threads) of the block.
     - gridDim: The dimensions (number of blocks) of the grid.

When programming in CUDA, developers write functions for the GPU called **kernels**, which are executed, or in CUDA parlance, **launched**, on the GPU's many cores in parallel **threads**. When kernels are launched, programmers use a special syntax, called an **execution configuration** (also called a launch configuration) to describe the parallel execution's configuration


GPU DATA
0
4
1
5
2
6
3
7
threadIdx.x
+
blockIdx.x
*
blockDim.x
3
1
4
data_index
7
grid(1)
7
As a convenience, Numba provides the `cuda.grid()` function, which will return a thread's unique index in the grid.
GPU
do_work[2, 4](d_a)
0
1
0
1
2
3
0
1
2
3
4
4

NOTE :- READ THE PPT FOR THE BETTER UNDERSTANDING

A First CUDA Kernel

Let's start with a concrete, and very simple example by rewriting our addition function for 1D NumPy arrays. CUDA kernels are compiled using the numba.cuda.jit **decorator.** numba.cuda.jit **is not to be confused with the** numba.jit **decorator you've already learned which optimizes functions for the CPU.**

The function cuda.synchronize() in CUDA programming is used to synchronize the CPU and GPU, ensuring that all CUDA operations on the GPU are completed before the program continues

Note :- Refer to the code first in the note book 2

Exercise: Tweak the Code
Make the following minor changes to the code above to see how it affects its execution. Make educated guesses about what will happen before running the code:

- Decrease the threads_per_block variable
- Decrease the blocks_per_grid variable
- Increase the threads_per_block and/or blocks_per_grid variables
- Remove or comment out the cuda.synchronize() call

Results
In the example above, because the kernel is written so that each thread works on exactly one data element, it is essential for the number of threads in the grid equal the number of data elements. By **reducing the number of threads in the grid**, either by reducing the number of blocks, and/or reducing the number of threads per block, there are elements where work is left undone and thus we can see in the output that the elements toward the end of the d_out array did not have any values added to it. If you edited the execution configuration by reducing the number of threads per block, then in fact there are other elements through the d_out array that were not processed. **Increasing the size of the grid** in fact creates issues with out of bounds memory access. This error will not show in your code presently, but later in this section you will learn how to expose this error using cuda-memcheck and debug it.
You might have expected that **removing the synchronization point** would have resulted in a print showing that no or less work had been done. This is a reasonable guess since without a synchronization point the CPU will work asynchronously while the GPU is processing. The detail to learn here is that memory copies carry implicit synchronization, making the call to cuda.synchronize above unnecessary.

An Aside on Hiding Latency and Execution Configuration Choices

Key Concepts

1. **Streaming Multiprocessors (SMs)**:
    - **SMs** are the fundamental units within a CUDA-enabled GPU where the actual computation happens. Each GPU die (the physical chip) contains multiple SMs.
    - Each SM has its own set of CUDA cores, which are the processors that execute individual threads.
2. **CUDA Cores**:
    - CUDA cores are the processing units within an SM that perform the actual computation tasks. Each core executes one thread.
3. **Blocks and Warps**:
    - **Blocks**: When a kernel is launched, it is divided into blocks of threads. Each block is assigned to an SM.
    - **Warps**: Within an SM, threads are further organized into warps. A warp is a group of 32 threads that execute the same instruction simultaneously. This is the smallest unit of execution in CUDA.

Execution Flow

1. **Kernel Launch**:
    - When a CUDA kernel is launched, it is divided into a grid of blocks. Each block is scheduled to run on an SM.
    - Each SM can handle multiple blocks simultaneously, depending on the number of resources (like registers and shared memory) available.
2. **Warp Scheduling**:
    - An SM executes warps of threads. If a warp is executing an instruction that takes multiple clock cycles, the SM can continue to issue new instructions to other warps that are ready to execute. This helps in hiding latency (delay) caused by long instructions.
    - This means that if some threads in a warp are waiting for data (causing a delay), the SM can switch to another warp and continue processing.
3. **Hiding Latency**:
    - Because of large register files on SMs, switching between different warps does not incur a significant performance penalty. The SM can keep other warps busy while waiting for the completion of instructions in other warps, effectively hiding the latency.

Optimizing Performance

1. **Block Size**:
    - The number of threads per block should be a multiple of 32, the size of a warp, to ensure efficient execution. Typical block sizes range from 128 to 512 threads per block.
    - This alignment helps maximize the utilization of CUDA cores and minimizes idle times due to warp divergence (when threads within a warp take different execution paths).
2. **Grid Size**:
    - The grid size should be large enough to ensure that all available SMs are utilized. A common heuristic is to have 2x to 4x the number of blocks as there are SMs on the GPU. For instance, if a GPU has 20 SMs, launching 40 to 80 blocks could be ideal.
    - However, launching too many blocks can increase kernel launch overhead and reduce performance. Therefore, for very large datasets, the number of threads in the grid should be adjusted to avoid excessive kernel launch overhead.
3. **Handling Large Datasets**:
    - When dealing with large datasets, the grid size should be chosen to balance between providing enough work to keep the SMs busy and minimizing the overhead from managing too many blocks.
    - Techniques like grid stride loops can be used to handle large datasets efficiently by having each thread work on multiple data elements.

Deciding the very best size for the CUDA thread grid is a complex problem, and depends on both the algorithm and the specific GPU's compute capability, but here are some very rough heuristics that we tend to follow and which can work well for getting started:

- The size of a block should be a multiple of 32 threads (the size of a warp), with typical block sizes between 128 and 512 threads per block.
- The size of the grid should ensure the full GPU is utilized where possible. Launching a grid where the number of blocks is 2x-4x the number of SMs on the GPU is a good starting place. Something in the range of 20 - 100 blocks is usually a good starting point.
- **The CUDA kernel launch overhead does increase with the number of blocks, so when the input size is very large we find it best not to launch a grid where the number of threads equals the number of input elements, which would result in a tremendous number of blocks. Instead we use a pattern to which we will now turn our attention for dealing with large inputs.**
- **With the big data set the overhead to manage those blocks by the kernel will be high so it will affect the overall performance of the GPU**

why Block Size Should Be a Multiple of 32 Threads

1. **Efficiency and Performance**:
   - **Warp Execution**: Since the GPU executes threads in groups of 32 (warps), if your block size is not a multiple of 32, some threads within a warp might be idle if the number of threads in the block isn't evenly divisible by 32. For example, if you have a block with 50 threads, you'll have 2 full warps (32 * 2 = 64 threads) and 18 threads that don't fit into a full warp. This means the GPU will waste resources on the partially filled warp.
   - **Memory Coalescing**: Memory accesses are more efficient when threads in a warp access contiguous memory locations. Aligning block sizes with warp size helps in achieving better memory coalescing, reducing memory access overhead and improving overall performance.
2. **Typical Block Sizes**:
   - **128 to 512 Threads**: This range is often recommended because:
     - **128 Threads**: This size is a multiple of 32 (4 warps) and is small enough to fit within the limited resources (like shared memory) of a block while providing good warp utilization.
     - **512 Threads**: This size is a multiple of 32 (16 warps), and it can efficiently utilize the resources of the GPU, especially when the GPU has more shared memory and registers available.

Working on Largest Datasets with Grid Stride Loops

Let's refactor the add_kernel above to utilize a grid stride loop so that we can launch it to work on larger data sets flexibly while incurring the benefits of global **memory coalescing**, which allows parallel threads to access memory in contiguous chunks, a scenario which the GPU can leverage to reduce the total number of memory operations:

In [ ]:

```python
from numba import cuda

@cuda.jit
def add_kernel(x, y, out):


    start = cuda.grid(1)

    # This calculation gives the total number of threads in the entire grid
    stride = cuda.gridsize(1)   # 1 = one dimensional thread grid, returns a single value.
                                # This Numba-provided convenience function is equivalent to
                                # `cuda.blockDim.x * cuda.gridDim.x`

    # This thread will start work at the data element index equal to that of its own
    # unique index in the grid, and then, will stride the number of threads in the grid each
    # iteration so long as it has not stepped out of the data's bounds. In this way, each
    # thread may work on more than one data element, and together, all threads will work on
    # every data element.
    for i in range(start, x.shape[0], stride):
        # Assuming x and y inputs are same length
    out[i] = x[i] + y[i]
```

In [ ]:

```
import numpy as np

n = 100000 # This is far more elements than threads in our grid
x = np.arange(n).astype(np.int32)
y = np.ones_like(x)

d_x = cuda.to_device(x)
d_y = cuda.to_device(y)
d_out = cuda.device_array_like(d_x)

threads_per_block = 128
blocks_per_grid = 30
```

In [ ]:

```
add_kernel[blocks_per_grid, threads_per_block](d_x, d_y, d_out)
print(d_out.copy_to_host()) # Remember, memory copy carries implicit synchronization
```

Atomic Operations and Avoiding Race Conditions

CUDA, like many general purpose parallel execution frameworks, makes it possible to have race conditions in your code. A race condition in CUDA arises when threads read to or write from a memory location that might be modified by another independent thread. Generally speaking, you need to worry about:

- read-after-write hazards: One thread is reading a memory location at the same time another thread might be writing to it.
- write-after-write hazards: Two threads are writing to the same memory location, and only one write will be visible when the kernel is complete.

A common strategy to avoid both of these hazards is to organize your CUDA kernel algorithm such that each thread has exclusive responsibility for unique subsets of output array elements, and/or to never use the same array for both input and output in a single kernel call. (Iterative algorithms can use a double-buffering strategy if needed, and switch input and output arrays on each iteration.)

However, there are many cases where different threads need to combine results. Consider something very simple, like: "every thread increments a global counter." Implementing this in your kernel requires each thread to:

1. Read the current value of a global counter.
2. Compute counter + 1.
3. Write that value back to global memory.

However, there is no guarantee that another thread has not changed the global counter between steps 1 and 3. To resolve this problem, CUDA provides **atomic operations** which will read, modify and update a memory location in one, indivisible step. Numba supports several of these functions, described here.

Let's make our thread counter kernel:

>