Your First GPU Kernel

Last updated on 2024-11-19 | Edit this page

OVERVIEW

Questions

- “How can I parallelize a Python application on a GPU?”
- “How to write a GPU program?”
- “What is CUDA?”

Objectives

- “Recognize possible data parallelism in Python code”
- “Understand the structure of a CUDA program”
- “Execute a CUDA program in Python using CuPy”
- “Measure the execution time of a CUDA kernel with CuPy”

Summing Two Vectors in Python

We start by introducing a program that, given two input vectors of the same size, stores the sum of the corresponding elements of the two input vectors into a third one.

PYTHON

```python
def vector_add(A, B, C, size):
    for item in range(0, size):
        C[item] = A[item] + B[item]
```

One of the characteristics of this program is that each iteration of the `for` loop is independent from the other iterations. In other words, we could reorder the iterations and still produce the same output, or even compute each iteration in parallel or on a different device, and still come up with the same output. These are the kind of programs that we would call *naturally parallel*, and they are perfect candidates for being executed on a GPU.

Summing Two Vectors in CUDA

While we could just use CuPy to run something equivalent to our `vector_add` on a GPU, our goal is to learn how to write code that can be executed by GPUs, therefore we now begin learning CUDA.

The CUDA-C language is a GPU programming language and API developed by NVIDIA. It is mostly equivalent to C/C++, with some special keywords, built-in variables, and functions.

We begin our introduction to CUDA by writing a small kernel, i.e. a GPU program, that computes the same function that we just described in Python.

C

```
extern "C"
__global__ void vector_add(const float * A, const float * B, float * C, const int size)
{
    int item = threadIdx.x;
    C[item] = A[item] + B[item];
}
```

CALLOUT

We are aware that CUDA is a proprietary solution, and that there are open-source alternatives such as OpenCL. However, CUDA is the most used platform for GPU programming and therefore we decided to use it for our teaching material.

Running Code on the GPU with CuPy

Before delving deeper into the meaning of all lines of code, and before starting to understand how CUDA works, let us execute the code on a GPU and check if it is correct or not. To compile the code and manage the GPU in Python we are going to use the interface provided by CuPy.

PYTHON

```
import cupy

# size of the vectors
size = 1024

# allocating and populating the vectors
a_gpu = cupy.random.rand(size, dtype=cupy.float32)
b_gpu = cupy.random.rand(size, dtype=cupy.float32)
c_gpu = cupy.zeros(size, dtype=cupy.float32)

# CUDA vector_add
vector_add_cuda_code = r'''
extern "C"
__global__ void vector_add(const float * A, const float * B, float * C, const int size)
{
    int item = threadIdx.x;
    C[item] = A[item] + B[item];
}
'''
vector_add_gpu = cupy.RawKernel(vector_add_cuda_code, "vector_add")

vector_add_gpu((1, 1, 1), (size, 1, 1), (a_gpu, b_gpu, c_gpu, size))
```

And to be sure that the CUDA code does exactly what we want, we can execute our sequential Python code and compare the results.

PYTHON

```python
import numpy as np

a_cpu = cupy.asnumpy(a_gpu)
b_cpu = cupy.asnumpy(b_gpu)
c_cpu = np.zeros(size, dtype=np.float32)

vector_add(a_cpu, b_cpu, c_cpu, size)

# test
if np.allclose(c_cpu, c_gpu):
    print("Correct results!")
```

OUTPUT

```
Correct results!
```

Understanding the CUDA Code

We can now move back to the CUDA code and analyze it line by line to highlight the differences between CUDA-C and standard C.

C

```c
__global__ void vector_add(const float * A, const float * B, float * C, const int size)
```

This is the definition of our CUDA `vector_add` function. The `__global__` keyword is an execution space identifier, and it is specific to CUDA. What this keyword means is that the defined function will be able to run on the GPU, but can also be called from the host (in our case the Python interpreter running on the CPU). All of our kernel definitions will be preceded by this keyword.

Other execution space identifiers in CUDA-C are `__host__`, and `__device__`. Functions annotated with the `__host__` identifier will run on the host, and be only callable from the host, while functions annotated with the `__device__` identifier will run on the GPU, but can only be called from the GPU itself. We are not going to use these identifiers as often as `__global__`.

The following table offers a recapitulation of the keywords we just introduced.

Keyword	Description
`__global__`	the function is visible to the host and the GPU, and runs on the GPU
`__host__`	the function is visible only to the host, and runs on the host
`__device__`	the function is visible only to the GPU, and runs on the GPU

The following is the part of the code in which we do the actual work.

C

```c
int item = threadIdx.x;
C[item] = A[item] + B[item];
```

As you may see, it looks similar to the innermost loop of our `vector_add` Python function, with the main difference being in how the value of the `item` variable is evaluated.

In fact, while in Python the content of `item` is the result of the `range` function, in CUDA we are reading a special variable, i.e. `threadIdx`, containing a triplet that indicates the id of a thread inside a three-dimensional CUDA block. In this particular case we are working on a one dimensional vector, and therefore only interested in the first dimension, that is stored in the `x` field of this variable.

CHALLENGE: LOOSE THREADS

We know enough now to pause for a moment and do a little exercise. Assume that in our `vector_add` kernel we replace the following line:

```c
int item = threadIdx.x;
```

With this other line of code:

```c
int item = 1;
```

What will the result of this change be?

1. Nothing changes
2. Only the first thread is working
3. Only `C[1]` is written
4. All elements of `C` are zero

Solution

The correct answer is number 3, only the element `C[1]` is written, and we do not even know by which thread!

Computing Hierarchy in CUDA

In the previous example we had a small vector of size 1024, and each of the 1024 threads we generated was working on one of the element.

What would happen if we changed the size of the vector to a larger number, such as 2048? We modify the value of the variable size and try again.

PYTHON

```python
# size of the vectors
size = 2048

# allocating and populating the vectors
a_gpu = cupy.random.rand(size, dtype=cupy.float32)
b_gpu = cupy.random.rand(size, dtype=cupy.float32)
c_gpu = cupy.zeros(size, dtype=cupy.float32)

# CUDA vector_add
vector_add_gpu = cupy.RawKernel(r'''
extern "C"
__global__ void vector_add(const float * A, const float * B, float * C, const int size)
{
    int item = threadIdx.x;
    C[item] = A[item] + B[item];
}
''', "vector_add")

vector_add_gpu((1, 1, 1), (size, 1, 1), (a_gpu, b_gpu, c_gpu, size))
```

This is how the output should look like when running the code in a Jupyter Notebook:

OUTPUT

```
---------------------------------------------------------------------------

CUDADriverError                           Traceback (most recent call last)

<ipython-input-4-a26bc8acad2fin <module>()
     19 ''', "vector_add")
     20
---21 vector_add_gpu((1, 1, 1), (size, 1, 1), (a_gpu, b_gpu, c_gpu, size))
     22
     23 print(c_gpu)

cupy/core/raw.pyx in cupy.core.raw.RawKernel.__call__()

cupy/cuda/function.pyx in cupy.cuda.function.Function.__call__()

cupy/cuda/function.pyx in cupy.cuda.function._launch()

cupy_backends/cuda/api/driver.pyx in cupy_backends.cuda.api.driver.launchKernel()

cupy_backends/cuda/api/driver.pyx in cupy_backends.cuda.api.driver.check_status()

CUDADriverError: CUDA_ERROR_INVALID_VALUE: invalid argument
```

The reason for this error is that most GPUs will not allow us to execute a block composed of more than 1024 threads. If we look at the parameters of our functions we see that the first two parameters are two triplets.

PYTHON

```python
vector_add_gpu((1, 1, 1), (size, 1, 1), (a_gpu, b_gpu, c_gpu, size))
```

The first triplet specifies the size of the CUDA **grid**, while the second triplet specifies the size of the CUDA **block**. The grid is a three-dimensional structure in the CUDA programming model and it represent the organization of a whole kernel execution. A grid is made of one or more independent blocks, and in the case of our previous snippet of code we have a grid composed by a single block `(1, 1, 1)`. The size of this block is specified by the second triplet, in our case `(size, 1, 1)`. While blocks are independent of each other, the thread composing a block are not completely independent, they share resources and can also communicate with each other.

To go back to our example, we can modify che grid specification from `(1, 1, 1)` to `(2, 1, 1)`, and the block specification from `(size, 1, 1)` to `(size // 2, 1, 1)`. If we run the code again, we should now get the expected output.

We already introduced the special variable `threadIdx` when introducing the `vector_add` CUDA code, and we said it contains a triplet specifying the coordinates of a thread in a thread block. CUDA has other variables that are important to understand the coordinates of each thread and block in the overall structure of the computation.

These special variables are `blockDim`, `blockIdx`, and `gridDim`, and they are all triplets. The triplet contained in `blockDim` represents the size of the calling thread's block in three dimensions. While the content of `threadIdx` is different for each thread in the same block, the content of `blockDim` is the same because the size of the block is the same for all threads. The coordinates of a block in the computational grid are contained in `blockIdx`, therefore the content of this variable will be the same for all threads in the same block, but different for threads in different blocks. Finally, `gridDim` contains the size of the grid in three dimensions, and it is again the same for all threads.

The following table offers a recapitulation of the keywords we just introduced.

Keyword	Description
`threadIdx`	the ID of a thread in a block
`blockDim`	the size of a block, i.e. the number of threads per dimension
`blockIdx`	the ID of a block in the grid
`gridDim`	the size of the grid, i.e. the number of blocks per dimension

CHALLENGE: HIDDEN VARIABLES

Given the following snippet of code:

PYTHON

```python
size = 512
vector_add_gpu((4, 1, 1), (size, 1, 1), (a_gpu, b_gpu, c_gpu, size))
```

What is the content of the `blockDim` and `gridDim` variables inside the CUDA `vector_add` kernel?

Solution

The content of `blockDim` is `(512, 1, 1)` and the content of `gridDim` is `(4, 1, 1)`, for all threads.

What happens if we run the code that we just modified to work on an vector of 2048 elements, and compare the results with our CPU version?

PYTHON

```python
# size of the vectors
size = 2048

# allocating and populating the vectors
a_gpu = cupy.random.rand(size, dtype=cupy.float32)
b_gpu = cupy.random.rand(size, dtype=cupy.float32)
c_gpu = cupy.zeros(size, dtype=cupy.float32)
a_cpu = cupy.asnumpy(a_gpu)
b_cpu = cupy.asnumpy(b_gpu)
c_cpu = np.zeros(size, dtype=np.float32)

# CPU code
def vector_add(A, B, C, size):
    for item in range(0, size):
        C[item] = A[item] + B[item]

# CUDA vector_add
vector_add_gpu = cupy.RawKernel(r'''
extern "C"
__global__ void vector_add(const float * A, const float * B, float * C, const int size)
{
    int item = threadIdx.x;
    C[item] = A[item] + B[item];
}
''', "vector_add")

# execute the code
vector_add_gpu((2, 1, 1), (size // 2, 1, 1), (a_gpu, b_gpu, c_gpu, size))
vector_add(a_cpu, b_cpu, c_cpu, size)

# test
if np.allclose(c_cpu, c_gpu):
    print("Correct results!")
else:
    print("Wrong results!")
```

OUTPUT

```
Wrong results!
```

The results are wrong! In fact, while we increased the number of threads we launch, we did not modify the kernel code to compute the correct results using the new builtin variables we just introduced.

CHALLENGE: SCALING UP

In the following code, fill in the blank to work with vectors that are larger than the largest CUDA block (i.e. 1024).

C

```c
extern "C"
__global__ void vector_add(const float * A, const float * B, float * C, const int size)
{
   int item = ______________;
   C[item] = A[item] + B[item];
}
```

Solution

The correct answer is `(blockIdx.x * blockDim.x) + threadIdx.x`. The following code is the complete `vector_add` that can work with vectors larger than 1024 elements.

C

```c
extern "C"
__global__ void vector_add(const float * A, const float * B, float * C, const int size)
{
   int item = (blockIdx.x * blockDim.x) + threadIdx.x;
   C[item] = A[item] + B[item];
}
```

Vectors of Arbitrary Size

So far we have worked with a number of threads that is the same as the elements in the vector. However, in a real world scenario we may have to process vectors of arbitrary size, and to do this we need to modify both the kernel and the way it is launched.

CHALLENGE: MORE WORK THAN NECESSARY

We modified the `vector_add` kernel to include a check for the size of the vector, so that we only compute elements that are within the vector boundaries. However the code is not correct as it is written now. Can you rewrite the code to make it work?

```c
extern "C"
__global__ void vector_add(const float * A, const float * B, float * C, const int size)
{
   if ( item < size )
   {
      int item = (blockIdx.x * blockDim.x) + threadIdx.x;
   }
   C[item] = A[item] + B[item];
}
```

Solution

The correct way to modify the `vector_add` to work on vectors of arbitrary size is to first compute the coordinates of each thread, and then perform the sum only on elements that are within the vector boundaries, as shown in the following snippet of code.

```c
extern "C"
__global__ void vector_add(const float * A, const float * B, float * C, const int size)
{
   int item = (blockIdx.x * blockDim.x) + threadIdx.x;
   if ( item < size )
   {
      C[item] = A[item] + B[item];
   }
}
```

To test our changes we can modify the `size` of the vectors from 2048 to 10000, and execute the code again.

OUTPUT

```
---------------------------------------------------------------------------

CUDADriverError                           Traceback (most recent call last)

<ipython-input-20-00d938215d28in <module>()
     31
     32 # Execute the code
---33 vector_add_gpu((2, 1, 1), (size // 2, 1, 1), (a_gpu, b_gpu, c_gpu, size))
     34 vector_add(a_cpu, b_cpu, c_cpu, size)
     35

cupy/core/raw.pyx in cupy.core.raw.RawKernel.__call__()

cupy/cuda/function.pyx in cupy.cuda.function.Function.__call__()

cupy/cuda/function.pyx in cupy.cuda.function._launch()

cupy/cuda/driver.pyx in cupy.cuda.driver.launchKernel()

cupy/cuda/driver.pyx in cupy.cuda.driver.check_status()

CUDADriverError: CUDA_ERROR_INVALID_VALUE: invalid argument
```

This error is telling us that CUDA cannot launch a block with `size // 2` threads, because the maximum amount of threads in a kernel is 1024 and we are requesting 5000 threads.

What we need to do is to make grid and block more flexible, so that they can adapt to vectors of arbitrary size. To do that, we can replace the Python code to call `vector_add_gpu` with the following code.

PYTHON

```python
import math

grid_size = (int(math.ceil(size / 1024)), 1, 1)
block_size = (1024, 1, 1)

vector_add_gpu(grid_size, block_size, (a_gpu, b_gpu, c_gpu, size))
```

With these changes we always have blocks composed of 1024 threads, but we adapt the number of blocks so that we always have enough to threads to compute all elements in the vector. If we want to be able to easily modify the number of threads per block, we can even rewrite the code like the following:

PYTHON

```python
threads_per_block = 1024
grid_size = (int(math.ceil(size / threads_per_block)), 1, 1)
block_size = (threads_per_block, 1, 1)

vector_add_gpu(grid_size, block_size, (a_gpu, b_gpu, c_gpu, size))
```

So putting this all together in a full snippet we can execute the code again.

PYTHON

```python
vector_add_cuda_code = r'''
extern "C"
__global__ void vector_add(const float * A, const float * B, float * C, const int size)
{
   int item = (blockIdx.x * blockDim.x) + threadIdx.x;
   if ( item < size )
   {
      C[item] = A[item] + B[item];
   }
}
'''
vector_add_gpu = cupy.RawKernel(vector_add_cuda_code, "vector_add")

threads_per_block = 1024
grid_size = (int(math.ceil(size / threads_per_block)), 1, 1)
block_size = (threads_per_block, 1, 1)

vector_add_gpu(grid_size, block_size, (a_gpu, b_gpu, c_gpu, size))

if np.allclose(c_cpu, c_gpu):
    print("Correct results!")
else:
    print("Wrong results!")
```

OUTPUT

```
Correct results!
```

CHALLENGE: COMPUTE PRIME NUMBERS WITH CUDA

Given the following Python code, similar to what we have seen in the previous episode about Numba, write the missing CUDA kernel that computes all the prime numbers up to a certain upper bound.

PYTHON

```
import numpy as np
import cupy
import math
from cupyx.profiler import benchmark

# CPU version
def all_primes_to(upper : int, prime_list : list):
    for num in range(0, upper):
        prime = True
        for i in range(2, (num // 2) + 1):
            if (num % i) == 0:
                prime = False
                break
        if prime:
            prime_list[num] = 1

upper_bound = 100_000
all_primes_cpu = np.zeros(upper_bound, dtype=np.int32)

# GPU version
check_prime_gpu_code = r'''
extern "C"
__global__ void all_primes_to(int size, int * const all_prime_numbers)
{
   for ( int number = 0; number < size; number++ )
   {
       int result = 1;
       for ( int factor = 2; factor <= number / 2; factor++ )
       {
           if ( number % factor == 0 )
           {
               result = 0;
               break;
           }
       }
>
       all_prime_numbers[number] = result;
   }
}
'''
# Allocate memory
all_primes_gpu = cupy.zeros(upper_bound, dtype=cupy.int32)

# Setup the grid
all_primes_to_gpu = cupy.RawKernel(check_prime_gpu_code, "all_primes_to")
grid_size = (int(math.ceil(upper_bound / 1024)), 1, 1)
block_size = (1024, 1, 1)

# Benchmark and test
%timeit -n 1 -r 1 all_primes_to(upper_bound, all_primes_cpu)
execution_gpu = benchmark(all_primes_to_gpu, (grid_size, block_size, (upper_bound, all_primes_gpu)), n_repeat=10)
gpu_avg_time = np.average(execution_gpu.gpu_times)
print(f"{gpu_avg_time:.6f} s")
>
if np.allclose(all_primes_cpu, all_primes_gpu):
    print("Correct results!")
```

```
else:
    print("Wrong results!")
```

There is no need to modify anything in the code, except the body of the CUDA `all_primes_to` inside the `check_prime_gpu_code` string, as we did in the examples so far.

Be aware that the provided CUDA code is a direct port of the Python code, and therefore very slow. If you want to test it, user a lower value for `upper_bound`.

Solution

One possible solution for the CUDA kernel is provided in the following code.

C

```c
extern "C"
__global__ void all_primes_to(int size, int * const all_prime_numbers)
{
    int number = (blockIdx.x * blockDim.x) + threadIdx.x;
    int result = 1;

    if ( number < size )
    {
        for ( int factor = 2; factor <= number / 2; factor++ )
        {
            if ( number % factor == 0 )
            {
                result = 0;
                break;
            }
        }

        all_prime_numbers[number] = result;
    }
}
```

The outermost loop in Python is replaced by having each thread testing for primeness a different number of the sequence. Having one number assigned to each thread via its ID, the kernel implements the innermost loop the same way it is implemented in Python.

KEY POINTS

- “Precede your kernel definition with the `__global__` keyword”
- “Use built-in variables `threadIdx`, `blockIdx`, `gridDim` and `blockDim` to identify each thread”