Workshop: School of Data Science and Computation Thinking, Stellenbosch University

Introduction to Parallel Computation with GPUs: Programming with JAX, NUMBA, THRUST, and CUDA

Martin BUCHER
(bucher@sun.ac.za)

LECTURE 8 (13 March 2025)

Course dates (4-5pm SAST):

Tu 18 Feb	Th 20 Feb
Tu 25 Feb	Th 27 Feb
Tu 4 March	Th 6 March
Tu 11 March	Th 13 March

Zoom link:

`https://u-paris.zoom.us/j/82740807191?pwd=8yAGlP3R6VKbsjNCIeRjJU3u0AQtU6.1`

Course webpage:

`http://www.sun.ac.za/english/data-science-and-computational-thinking/parallelcomputation`
`https://github.com/martinabucher/cudaCourse`

Lecture 8 Outline

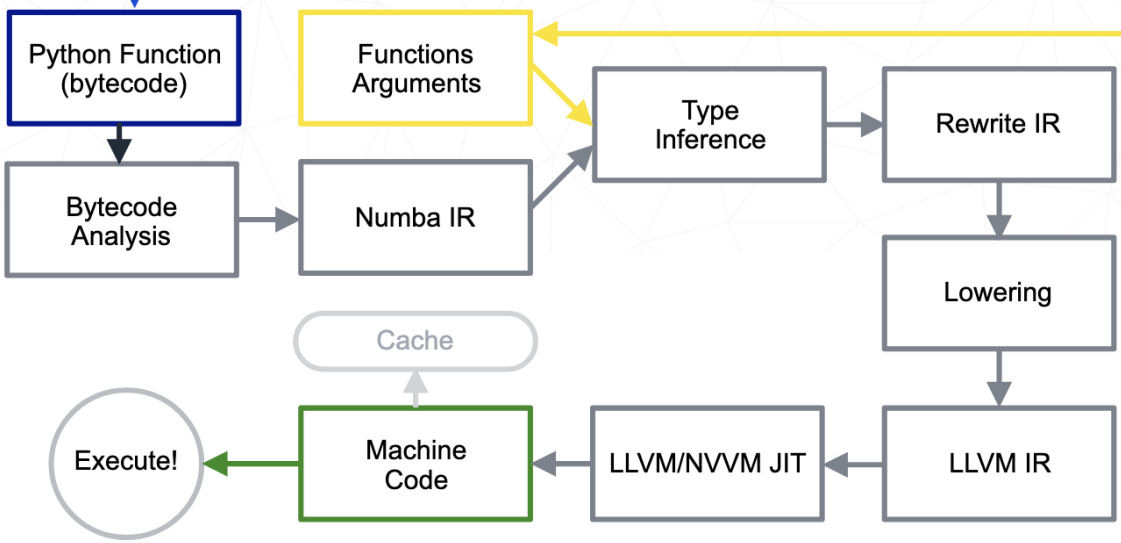
1. More More Details on NUMBA JIT
2. CUDA NUMBA: A Simplest Example
3. CUDA NUMBA Transferring Data
4. CUDA NUMBA Kernel Function Launches
5. Heat Equation Example
6. Cooperative Groups
7. JAX
8. GIL (Global Interpreter Lock) Revisited
9. Python mulitprocessing module
10. MPI (Message Passing Interface)
11. OpenMP
12. Indirection Quote

A simplest NUMBA JIT example

```
# https://numba.pydata.org/numba-doc/dev/user/5minguide.html
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

# Set "nopython" mode for best performance, equivalent to @njit
@jit(nopython=True)
def go_fast(a):
    # Function is compiled to machine code when called the first time
    trace = 0.0
    for i in range(a.shape[0]):   # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace              # Numba likes NumPy broadcasting

print(go_fast(x))
```

How does Numba work?

```
@jit
def do_math(a, b):
    ...
>>> do_math(x, y)
```



From talk by Stan Seibert, "Numba: A Compiler for Python Functions", (2018)

`https://indico.cern.ch/event/709711/contributions/2915722/attachments/1638199/2614603/2018_04_23_Numba_DIANA_v2.pdf`

How JIT works in NUMBA: Some Simple Examples

Lazy compilation

[Here lazy means that whatever work is to be done is done at the last minute.]
The recommended way to use the jit decorator is to let Numba decide when and how to optimize:
from numba import jit

```
@jit
def f(x, y):
    # A somewhat trivial example
    return x + y
```

In this mode, compilation will be deferred until the first function execution. Numba infers the argument types at call time and generates optimized code based on this information. Numba can also compile separate specializations depending on the input types. For example, calling the f() function above with integer or complex numbers will generate different code paths:

```
f(1, 2)
3


f(1j, 2)
(2+1j)
```

This page is worth reading very carefully.

`https://numba.readthedocs.io/en/stable/user/jit.html`

`https://en.wikipedia.org/wiki/Just-in-time_compilation`

More Simple Examples

Eager compilation (do every you are able to do right away)
You can also tell Numba the function signature (i.e., input and output types) you are expecting. The function f() would now look like:

```
from numba import jit, int32

@jit(int32(int32, int32))
def f(x, y):
    # A somewhat trivial example
    return x + y
```

A machine code version will be compiled and cached for those input types straightaway.

Caching of compiled functions has several known limitations:

- The caching of compiled functions is not performed on a function-by-function basis. The cached function is the main jit function, and all secondary functions (those called by the main function) are incorporated in the cache of the main function.
- Cache invalidation fails to recognize changes in functions defined in a different file. This means that when a main jit function calls functions that were imported from a different module, a change in those other modules will not be detected and the cache will not be updated. This carries the risk that "old" function code might be used in the calculations.
- Global variables are treated as constants. The cache will remember the value of the global variable at compilation time. On cache load, the cached function will not rebind to the new value of the global variable.

  [https://numba.readthedocs.io/en/stable/user/jit.html]

Caching of compiled functions has several known limitations:

Commentary:

- The default is not to cache functions. This is the safe route absent further information.
- In functional programming, there is much talk about so-called pure functions. A pure function has no side-effects. The function depends only on its input arguments. It does not read its environment (eg through global variable). Nor may it write to global variables. If a pure function is given the input it will always produce the same output. This is not generally true in computer programming. matplotlib for example has a very complicated environment which often is invisible, except when one is not happy with the defaults, and can tweak the environmental variables. It is easy to produce efficient code for pure functions. The compiler need look no further than the function body and does not have to guard against hidden dependencies.
- Even if a function needs to be recompiled at each use, in many cases considerable speedup can be obtained by compiling with the JIT compiler, for example if there are long for loops and often traversed control structures.

When is Numba a Good Idea?

- Numerical algorithms
- Data is in the form of NumPy arrays, or (more broadly) flat data buffers
- Performance bottleneck is a handful of well encapsulated functions
- Example use cases:
  - Compiling user-defined functions to call from another algorithm (like an optimizer)
  - Creating "missing" NumPy/SciPy functions (librosa)
  - Rapidly prototyping GPU algorithms (FBPIC)
  - Constructing specialized Python compilers (HPAT, OAMap)

From talk by Stan Seibert, "Numba: A Compiler for Python Functions", (2018)

`https://indico.cern.ch/event/709711/contributions/2915722/attachments/1638199/2614603/2018_04_23_Numba_DIANA_v2.pdf`

A Simplest CUDA NUMBA Example: Vector Addition

See: https://numba.readthedocs.io/en/stable/cuda/overview.html

```
import numpy as np
from numba import cuda

# The code below is like an @jit decorated function except
# that it creates a CUDA kernel function

@cuda.jit
def f(a, b, c):
    # like threadIdx.x + (blockIdx.x * blockDim.x)
    tid = cuda.grid(1)
    size = len(c)
    if tid < size:
        c[tid] = a[tid] + b[tid]

N = 100000
# The following three lines transfers Python objects to the device
a = cuda.to_device(np.random.random(N))
b = cuda.to_device(np.random.random(N))
c = cuda.device_array_like(a)

# The following is one way to make a CUDA NUMBA kernel function call
f.forall(len(a))(a, b, c)
# The result is on the device and must be copied back before printing
print(c.copy_to_host())
```

Commentary

- This decorator asks to compile the code. But eager evaluation of the decorator is not possible because the type of the inputs are not yet known.

```
@cuda.jit
```

- One needs to extract the thread identity:

```
# like threadIdx.x + (blockIdx.x * blockDim.x)
tid = cuda.grid(1)
```

- The number of threads does not in general correspond to the amount of work to be done, so some sort of a guard is needed.

```
    if tid < size:
        ....
```

- Half the statement is on the host, the other half in on the device:

```
a = cuda.to_device(np.random.random(N))
b = cuda.to_device(np.random.random(N))
c = cuda.device_array_like(a)
```

- `# Here <<<num_blocks,threads_per_block>> is automatically chosen:`
  `# The following one way to make a CUDA NUMBA kernel function call`
  `f.forall(len(a))(a, b, c)`

- Again half is on the device, and half on the host:

```
# The result is on the device and must be copied back before printing
print(c.copy_to_host())
```

Some variations on a NUMBA CUDA kernel call with more control
(number of blocks and number of threads per block are specified)

```
# Enough threads per block for several warps per block

nthreads = 256
# Enough blocks to cover the entire vector depending on its length
nblocks = (len(a) // nthreads) + 1
f[nblocks, nthreads](a, b, c)
print(c.copy_to_host())
```

https://numba.readthedocs.io/en/0.60.0/cuda-reference/kernel.html

Kernel function call in CUDA NUMBA

```
configured = func[griddim, blockdim, stream, sharedmem]
configured(x, y, z)

func[griddim, blockdim, stream, sharedmem](x, y, z)
```

Syntax resembles launch configuration in CUDA C/C++:

```
func<<<griddim, blockdim, sharedmem, stream>>>(x, y, z)
```

forall kernel call construction

```
forall(ntasks, tpb=0, stream=0, sharedmem=0)
    Returns a 1D-configured dispatcher for a given number of tasks.
    This assumes that:
        the kernel maps the Global Thread ID cuda.grid(1) to tasks on a 1-1 bas
        the kernel checks that the Global Thread ID is upper-bounded by ntasks,
    Parameters
            ntasks { The number of tasks.
            tpb { The size of a block. An appropriate value is chosen if this p
            stream { The stream on which the configured dispatcher will be laun
            sharedmem { The number of bytes of dynamic shared memory required b
    Returns
        A configured dispatcher, ready to launch on a set of arguments.
```

One-Dimensional Heat Equation

$$\frac{\partial u}{\partial t} = K \frac{\partial^2 u}{\partial x^2}$$

δ-function source as initial condition at $t = 0$ spreads into a Gaussian. Solution takes the form

$$u(x,t) = \frac{1}{\sqrt{2\pi t^2}} e^{-\alpha \frac{(x-x_0)^2}{2t}}$$

when α is proportional to K.

```
@cuda.jit
def solve_heat_equation(buf_0, buf_1, timesteps, k):
    i = cuda.grid(1)
    # Don't continue if our index is outside the domain
    if i >= len(buf_0):
        return
    # Prepare to do a grid-wide synchronization later
    grid = cuda.cg.this_grid()
    # CONTINUED ON NEXT PAGE
```

```
# CONTINUED ON NEXT PAGE
for step in range(timesteps):
    # Select the buffer from the previous timestep
    if (step % 2) == 0:
        data = buf_0
        next_data = buf_1
    else:
        data = buf_1
        next_data = buf_0
    # Get the current temperature associated with this point
    curr_temp = data[i]
    # Apply formula from finite difference equation
    if i == 0:
        # Left wall is held at T = 0
        next_temp = curr_temp + k * (data[i + 1] - (2 * curr_temp))
    elif i == len(data) - 1:
        # Right wall is held at T = 0
        next_temp = curr_temp + k * (data[i - 1] - (2 * curr_temp))
    else:
        # Interior points are a weighted average of their neighbors
        next_temp = curr_temp + k * ( data[i - 1] - (2 * curr_temp) + data[
        )
    # Write new value to the next buffer
    next_data[i] = next_temp
    # Wait for every thread to write before moving on
    grid.sync()
```

Cooperative Groups

- In any earlier lecture I wrongly told you that synchronization is possible only at the block level (because of hardware limitations).
- I read this on the web, and it was probably true at one time. CUDA is a moving target and not particularly well documented. Old posts on eg Stack Exchange must be treated with some scepticism.
- Thread synchronisation is possible at many levels: entire grid, cluster, block, and sub-block.
- grid level synchronization is limited to grids that can simultaneously fit on the device. It will not work on larger grids.

`https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf Chapter 11`
`https://developer.nvidia.com/blog/cooperative-groups/`

JAX: A Python Package for AI

One of the big pluses is automatic differentiation. Below is a simplest example.

```
# imports
from jax import grad
import jax.numpy as jnp

# define the logistic function
def logistic(x):
    return jnp.exp(x) / (jnp.exp(x) + 1)

# obtain the gradient function of the logistic function
grad_logistic = grad(logistic)

# evaluate the gradient of the logistic function at x = 1
grad_log_out = grad_logistic(1.0)
print(grad_log_out)
```

In most languages the differentiation has to be coded in by hand. This is time-consuming and error prone.

https://docs.jax.dev/en/latest/

This functionality however comes at a price.

Differentiation

- With the theory of distributions formalized Laurent Schwarz in the 1950s almost any function can be differentiated.
- Considered the step function

$$\theta(t) = \begin{cases} 1, & t > 0, \\ 0, & t \leq 0. \end{cases}$$

- $\dfrac{d\theta}{dt} = \delta(t)$ Dirac delta function
- But how would this be represented on a computer?

Why can't we just JIT everything?

Adapted from: https://docs.jax.dev/en/latest/jit-compilation.html

From the example above, you wonder why not apply `jax.jit()` to every function.

To understand why this is a bad idea, we show some cases where JIT fails:

```
# Condition on value of x.
def f(x):
  if x > 0:
    return x
  else:
    return 2 * x
jax.jit(f)(10)  # Raises an error
```

This would trivially compile with NUMBA JIT.

JAX JIT Limitations

- Functions must be **pure** (no side effects, no reading of global variables) and use certain function primitives in order to be easily analyzable.
- This restriction is reminiscent of **pure functional programming** languages, like **Haskell**, where all functions must be pure. Inherently impure functions (like print) must be implemented using **monoids**. For general purpose programming this leads to what looks like incredibly inefficient code (variables cannot change their value—they are like void functions with the right type). But nonetheless the infinite wisdom of the compiler—aided by 'purity', which makes functions easy to analyze— supposedly allows competitively fast code to be generated.
- Arrays are immutable. From point of view of generating fast code, this appears really weird but is needed to allow automatic differentiation.
- The philosophy is that many codes need optimization only in a few localized spots where there are no complicated data structures, so these limitations are thus not problematic.

PEP 703: Making the Global Interpreter Lock Optional in CPython

`https://peps.python.org/pep-0703/`

Abstract

CPython's global interpreter lock ("GIL") prevents multiple threads from executing Python code at the same time. The GIL is an obstacle to using multi-core CPUs from Python efficiently. This PEP proposes adding a build configuration (–disable-gil) to CPython to let it run Python code without the global interpreter lock and with the necessary changes needed to make the interpreter thread-safe.

`https://docs.python.org/3/glossary.html#term-global-interpreter-lock`

Parallel Python Module

Table of Contents

`multiprocessing` — Process-based parallelism

Source code: Lib/multiprocessing/

Availability: not Android, not iOS, not WASI.

This module is not supported on mobile platforms or WebAssembly platforms.

Introduction

`multiprocessing` is a package that supports spawning processes using an API similar to the `threading` module. The `multiprocessing` package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the `multiprocessing` module allows the programmer to fully leverage multiple processors on a given machine. It runs on both POSIX and Windows.

- This is a good way to launch jobs on a multi-thread machine, especially when there is strong data parallelism.
- Many parallel jobs in native python code can be launched.
- This package gets around the GIL without user intervention.
- Not all python packages do this. Threads can be run sequentially, and for some programming this is useful. "Separation of concerns."

MPI (Message Passing Interface)

https://arcb.csc.ncsu.edu/~mueller/cluster/mpi.guide.pdf

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv) {
    int my_rank;        /* Rank of process */
    int p;              /* Number of processes */
    int source;         /* Rank of sender */
    int dest;           /* Rank of receiver */
    int tag = 50;       /* Tag for messages */
    char message[100];  /* Storage for the message */
    MPI_Status status;  /* Return status for receive */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank != 0) {
        sprintf(message, "Greetings from process %d!",
            my_rank);
        dest = 0;
        /* Use strlen(message)+1 to include '\0' */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest,
            tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
```

```
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize();
} /* main */
```

Message Passing Interface Philosophy

- No special hardware assumed but just the ability to pass explicit messages. Could be through special-purpose, high-speed interconnects, or through the (very slow) internet.
- The 1990s saw a rivalry between very expensive special purpose hardware (supercomputers) and commodity computers connected together. The commodity approach eventually won out.
- Some people hate MPI. They say it is too hard to program and prefer automatic parallelization.
- But MPI follows well-defined orders. It does what you tell it to.
- OpenMP often does not take the hints you give and won't always speed up your code. You are left wondering why—and have little control. The easy parallelism comes at a price.

How to Compile MPI Programs (depends on local environment)

Here `mpicc` is an extension or front-end to `gcc` in the same spirit as `nvcc` is a front-end and extension of a C compiler.

```
$mpicc -Wall -g -o my_mpi_prog my_mpi_prog.c
```

A special run command must be used to set the number of threads

```
$mpirun -np 32 ./your_mpi_binary
```

MPI is in general used on large shared clusters using a queuing system. The above shows how to run interactively.

OpenMP Example

```
int main(int argc, char **argv)
{
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

OpenMP Syntax

- Hints are given in terms of #pragma directives. These look like C pre-processor directives, which which the compiler might or might not follow.
- The assumption is that the compiler is infinitely clever.

A nice quote worth thinking about:

> *All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection.*

—David Wheeler (?)
https://news.ycombinator.com/item?id=9115021