UPPSALA UNIVERSITET

Going beyond TensorFlow

Carl Nettelblad

2020-06-02

UPPSALA UNIVERSITET

What did we do yesterday?

- General comparison CPU/GPU
- Why are GPUs good?
- Why are they hard to use?

- The performance span between a modestly bad and a really good implementation of the same underlying algorithm is much larger on GPU
  - The extreme parallelism even challenges your choice of algorithm

UPPSALA UNIVERSITET

TensorFlow

- Framework for deep learning
- Or expressing other numerical operations

- TensorFlow is not optimally performant
- And not every problem is reasonably expressed as computational graph
  - While many are

- Talking more about programming models today

UPPSALA UNIVERSITET

Programming early on

- The first 3D-accelerated video cards were fixed-function
  - Compute coordinates for triangles
  - Compute texture coordinates for those triangles
- Roughly in 2001, programmable *shaders* were introduced
- Pixel shaders
  - Run for computing the actual pixel value (RGBA + Z) for each pixel
  - Read from one or multiple textures
  - Can render into another texture
- Vertex shader
  - Receive coordinates for each vertex in a geometry
  - Modify coordinates

UPPSALA UNIVERSITET

How to program

- Up until this point, the video cards were extremely opaque
  - They did just one thing
  - And *how* they did that allowed a fair deal of flexibility
  - Exact implementation of rendering, including arithmetics, rounding, could vary
  - Vastly different chip architectures, and continuing differences over generations from the same vendor

UPPSALA UNIVERSITET

How to program

- Two main routes:
  - Some kind of assembly-like language
  - Something closer to a high-level (C-like) language
- A standardization of an assembly language (ARB) was created
  - But most features were never added there
  - A slightly more high-level representation of the operations can be *easier* to optimize successfully for a wider variety of hardware platforms

UPPSALA UNIVERSITET

Cg/HLSL and GLSL

- Two families of languages, Cg originally Nvidia, then DirectX through HLSL
- GLSL in OpenGL
  - Both C-like
  - “Any” card will support both

- Very limited early on, especially for pixel shaders
- More features added over time
- Loops, more memory, control flow
- But
  - No pointers, no real call stack, limited synchronization

UPPSALA UNIVERSITET

How code is sent to the GPU with shaders

- When you compile your program, you don't know what architecture it will be running on
- Therefore, shaders are distributed as source code
- The HLSL or GLSL program is sent to the driver as a string at runtime
  - Only then compiled by the driver
  - Cache files sometimes used

UPPSALA UNIVERSITET

GPGPU

- Quite early on, people realized that you could do non-graphical math using GPUs
- If your matrices are textures, you can implement matrix multiplication
  - Possibly in many passes
  - But multi-pass rendering was a thing for “real” graphics as well
- More complex operations in vertex shaders

- There were successful proof-of-concept papers, no widespread adoption, during most of the 00s

UPPSALA UNIVERSITET

Interconnects

- Yesterday, we talked about memory bandwidth
  - System slow
  - GPU fast
- But also memory size
  - System large
  - GPU small
    - Especially per thread
- How are things transferred into GPU memory?

UPPSALA UNIVERSITET

Interconnects

- A typical GPU is connected using a PCI Express bus
  - 16 combined links, or 16x
- In Snowy, these are of generation PCIe 2.0, giving 8 GB/s
  - Most recent machines would have PCIe 3.0, 16 GB/s
  - PCIe 4.0 is becoming common now, 32 GB/s
  - Two more doublings in the roadmap after slow progress during the 2010s
- This is *slower* than system memory
  - Much slower than video memory

UPPSALA UNIVERSITET

Latency/arbitration

- Bandwidth is not the only issue
- System memory is controlled by the CPU
- Every request has higher latency compared to CPU accessing main memory or GPU accessing video memory

- Also a matter of synchronization
  - If you tell the CPU “copy this data to the GPU” what needs to happen is something like:
    - The data has to be fully written (remember out-of-order execution etc)
    - The OS has to make sure that the memory will not be paged out (pinning)
    - A command has to be sent to the GPU “fetch this data”
    - The GPU has to respond to that command, asking for data in suitable chunks

UPPSALA UNIVERSITET

Unifying memory

- A memory address really only makes sense within a process
  - Modern GPUs support virtual memory as well, with address mappings specific to execution context
  - Special case:
    - Make CPU and GPU map at least some addresses to the same areas

UPPSALA UNIVERSITET

Unified memory – why?

- Not having to specify copying of data can be “nice”
  - The copying still has to happen
  - Ideally, you want copying to happen before data is needed anyway
- More important
  - Complex data structures
    - Any data structure using pointers can be moved transparently if the address space is shared

UPPSALA UNIVERSITET

Unified memory – not so fast

- Compiler and OS support has been patchy
  - On most platforms, what you can get is to have a special function call for allocating memory in the unified address space
    - This won't include local variables on the stack
    - It won't include general memory allocated using `new` or `malloc` or similar
      - Although there are hacks to reroute those
  - In general, all unified memory will be pinned
    - Not possible to swap to disk
    - Not a huge problem in HPC, possible issue in a more general desktop/laptop setting

UPPSALA UNIVERSITET

Interconnects between GPUs

- If you just buy a bunch of GPUs and put them in a server, the GPUs can only communicate using PCIe
  - And PCIe wasn't even intended for heavy cross-device communication
  - All communication needs to be routed through the PCIe controller
    - Sometimes integrated into the CPU these days

UPPSALA UNIVERSITET

Interconnects between GPUs

- Vendors have included proprietary systems for communication between GPUs
  - Nvidia NVLink being the most important
  - And separate NVSwitches
    - Allowing any GPU in a machine to access any other with low latency and speeds > 100 GB/s
    - Brings a multi-GPU solution closer to “a single very large GPU” for the programmer

UPPSALA UNIVERSITET

Pre-lab 2

- What did the code do?
- Which version was the fastest?
- Why?

UPPSALA UNIVERSITET

Pre-lab 2

- Do you have any ideas for how to speed up the compiled code?

- Do you have any ideas for why the Python code was slow?

UPPSALA UNIVERSITET

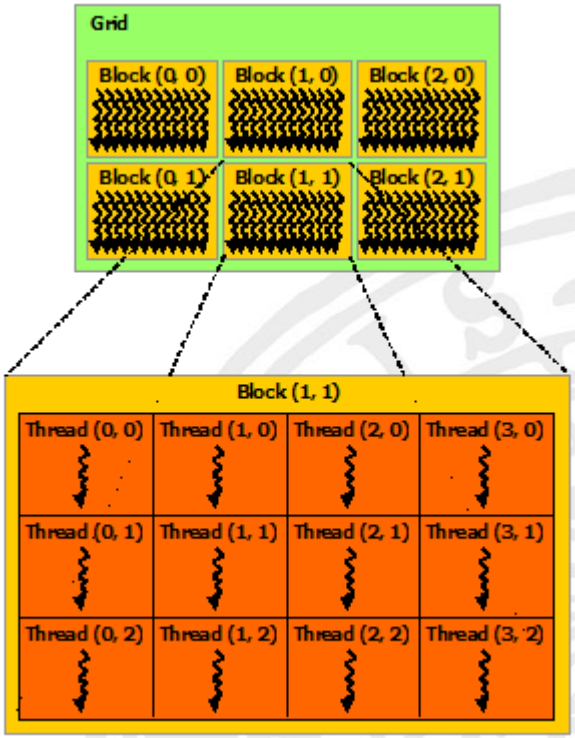
SIMT

- Single instruction, multiple threads
- The basic inner operation of all modern GPUs
- In practice: a compute kernel (function) is executed over a range of indices

UPPSALA UNIVERSITET

SIMT

- The closest threads are run in tandem
  - On Nvidia GPUs 32 threads form a *warp*
  - The most efficient execution happens when all 32 run the same instruction at the same time
  - A larger group of threads form a *block*
  - The total kernel invocation puts several blocks in a *grid*



UPPSALA UNIVERSITET

Memory

- Local memory
  - Per thread
  - Small, fast
- Shared memory
  - Per block
  - Small, fast
  - Same physical space as L1 cache in more recent chips
- Global memory
  - General GPU memory (or unified CPU memory)
  - Slow, large
  - Main way to communicate across blocks
  - Can enter L1 and L2 cache

UPPSALA UNIVERSITET

Memory (less important)

- Texture memory
  - Special buffers
  - Used to offer large benefits in irregular 2D and 3D accesses
  - Also offers “free” linear interpolation
    - At 8-bit resolution
  - Not so much anymore
- Constant memory
  - Stored in device memory, but separate caching based on the fact that they do not change
  - Far less important these days due to more efficient general caching

UPPSALA UNIVERSITET

Memory

- Reads/writes for global memory and L2 cache can be up to 128 bytes
- The most efficient thing is if each thread in a warp accesses 4 bytes within the same 128-byte region
  - Extremely different from threads on a CPU, where accesses to memory in the same cache line amounts to “false sharing”
- For L1 and shared memory, threads within the same warp should access *different* addresses in the same cycle

UPPSALA UNIVERSITET

Not really SIMT

- Most execution resources are independent per thread
- AMD chips also have “scalar instructions”
  - Executed for the full wave (corresponding to warp)
  - Including some more rare trigonometry
- Nvidia chips have less FP64 resources
  - Only 2 per 64 FP32 cores on our chips
  - 32 per 64 on the beefier ones
  - 16 cores for trigonometry
- Limited number of *tensor cores* per warp
  - These can do extremely fast int1, int4, int8, fp16 math for small matrix-vector and matrix-matrix multiplications
  - Kind of like SIMD within the SIMT

UPPSALA UNIVERSITET

Synchronization

- On-GPU synchronization can be at different levels
  - Within-warp
    - Special instructions for exchanging values, performing a joint and/or/count operation
    - Ensure all threads reaching a specific instruction
      - `__syncwarp`
  - Within-block
    - Special instructions
      - `__syncthreads`
  - To shared/global memory
    - Atomics

UPPSALA UNIVERSITET

Atomics

- Perform a combined read/write operation
  - Addition, subtraction, exchange, minimum, maximum, increase, decrease, binary and, binary or, binary xor, compare-and-swap
  - Add and sub up to double precision float
  - Many others for ints up to 64 bits
- Generality and performance greatly improved in successive generations

UPPSALA UNIVERSITET

Synchronization

- The GPU is inherently not synchronized to the CPU instruction stream
  - A kernel does not return anything
  - Some frameworks include the step to wait for the kernel, some explicitly allow you to write useful code to run on the CPU in the meantime
  - Common flows:
    - Feed data for iteration *i* + *1* while kernel for iteration *i* is running
    - Perform algorithm section that is only CPU code while GPU is doing another section

UPPSALA UNIVERSITET

Three models closer

- OpenMP Target
- Thrust
- CUDA

UPPSALA UNIVERSITET

OpenMP Target

- OpenMP is a way to express thread-parallelism in a compiled environment
- Standardized
- The main concept is that you add OpenMP pragmas
  - The same code compiled without OpenMP support will then be an ordinary serial program
- If you care at all about performance in compiled languages, not knowing OpenMP is stupid

UPPSALA UNIVERSITET

What's different in OpenMP target?

- Classic OpenMP centers on using multiple threads within the same device
  - The original main thread is running on an identical CPU, with an identical memory address space, as the other threads
  - All those threads can communicate freely
    - Although, in practice, some have more shared resources than others

UPPSALA UNIVERSITET

What's different in OpenMP target?

- Classic OpenMP centers on using multiple threads within the same device
  - The original main thread is running on an identical CPU, with an identical memory address space, as the other threads
  - All those threads can communicate freely
    - Although, in practice, some have more shared resources than others

MAIN (1)	2
3	4
5	6
7	8

9	10
11	12
13	14
15	16

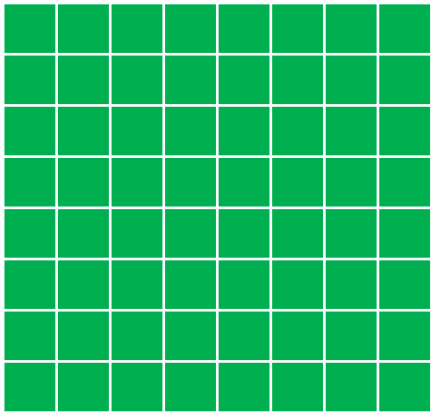
UPPSALA UNIVERSITET

What's different?

- OpenMP Target allows offloading computations to a different unit, with an independent memory space

MAIN (1)	2
3	4
5	6
7	8

9	10
11	12
13	14
15	16



UPPSALA UNIVERSITET

Different levels of execution

- OpenMP generally focuses on one type of parallelism
  - But we have seen that most GPUs have three
    - Grid, block, wave in Nvidia jargon

- In OpenMP Target, the grid level is represented as “teams”
  - The team set is the grid
  - Each block contains one OpenMP master thread
  - That thread can then denote parallel sections

UPPSALA UNIVERSITET

Access to data

- In standard OpenMP, it's OK to read from data directly
  - Variables can be explicitly set as `private`, `shared` etc (private implies that data is copied to each daughter thread)
- In OpenMP Target, *any* data needs to be copied

- From the prelab:

```
#pragma omp target teams \
map(to:origdata[:elems*side*side]) \
map(from:divergence[:elems*elems])
```

- Copy specified range of array *to* the GPU before the block starts
- Copy specified range of another array *from* the GPU at the end

UPPSALA UNIVERSITET

Loops

- `omp distribute`
  - Run loop over all teams
- `omp parallel for`
  - Run loop over all threads in team
- `omp distribute parallel for`
  - Run with maximum parallelism

UPPSALA UNIVERSITET

Loops

- `collapse(n)`
  - Use the same interpretation for several nested loops
- `reduction`
  - Specify variables that are to be summed/maximized or combined in some other way between threads
- `schedule`
  - Specify a scheduling regime

UPPSALA UNIVERSITET

Pros/cons

Pros

- OpenMP Target lets you stay in the normal language
- The same code can also be a performant CPU parallelization
- No need to really go into warp scheduling details

Cons

- Trying to sculpt with oven mittens
  - If the compiler fails to see a nice parallelization layout, it's hard to convince it to change
  - No/limited access to more advanced synchronization/reduction features

UPPSALA UNIVERSITET

Thrust

- Template-based library
  - Probably not your cup of tea if you have never used C++ templates before
  - Uses iterators and functors to express logic
- In the C++ standard, there is

```
std::accumulate(vec.begin(), vec.end(), 0,
    std::plus<int>());
```

- In Thrust, there is

```
thrust::reduce(vec.begin(), vec.end(), 0,
    thrust::plus<int>());
```

UPPSALA UNIVERSITET

Thrust

- Can execute on host, can execute on device
  - Host framework not top-notch these days
  - But reasonable, with thread parallelism
- Efficient tools for sorting and other operations that are hard to write massively parallel
- Fixed function, but allows for “kernel fusion”
  - We compute the differences between two digits and square the result when it’s needed
  - In `numpy`, this might have been
    - `diff = sum((A-B)**2)`
    - In many cases, that amounts to creating a temporary `A-B`, another temporary containing the elementwise squares, and then summing
    - Looping over the same data three times!

UPPSALA UNIVERSITET

Iterators

- Iterators are objects that can return elements
- Like pointers
- But Thrust brings
  - `transform_iterator`
    - Accept elements from one iterator and call a functor on each of them (like creating a “virtual” array where element is squared)
  - `zip_iterator`
    - Combine elements from two iterators into tuples of elements
  - `constant_iterator`
    - Return the same element over and over again
  - `counting_iterator`
    - Return a fixed sequence of numbers, like `range` in Python
- The algorithm doesn’t really care what kind of iterator you give it

UPPSALA UNIVERSITET

Functors

- An object that can be called as function
  - Classic way is to overload the `operator()` in C++
- You can also do it using Lambda expressions

```
thrust::reduce(vec.begin(), vec.end(), 0,
      [] (int a, int b) { return a * b; });
```

- Returns the product of all elements. With transform iterators and hook-in points for functors, Thrust is very flexible.
- Used in `thrustmax.cu` to compute the maximum of the divergences, without storing them in an array first.

UPPSALA UNIVERSITET

Pros/cons

- Provides efficient algorithms for many tasks that are hard
  - Complex reductions, sorts, scans, …
- Plugin your own algorithms and data structures
- Can co-exist with “raw” CUDA code
- Somewhat efficient CPU implementation allows shared source code

- Awkward if you are not into the iterator style of the C++ STL
- Performance might not improve a lot over OpenMP Target if you are mostly doing loops anyway

UPPSALA UNIVERSITET

CUDA

- A modified C++ programming language
  - Used to be very C-centric, now far more C++-inclined

- A function can be tagged as
  - `__global__`, the start of a kernel
  - `__device__`, a function called from a kernel on the device
  - `__host__`, a function explicitly running on the host

UPPSALA UNIVERSITET

Grids and blocks

- Here we see the grids and thread blocks clearly!
- Implicit global variables `threadIdx`, `blockDim`, `blockIdx`, `gridDim`
- Each of these are instances of a special `dim3` struct, with fields `x`, `y`, `z`.
  - `x` is the innermost, adjacent threads in `x` form a warp. `x` often chosen to be a multiple of 32.
- Common patterns include special logic for thread `0`, computing an index from the combination of the thread and block index (both present in `cuda.cu` in prelab 2)

UPPSALA UNIVERSITET

Special variables

A variable can be declared as:

- `__shared__`
  - Then it's shared over a block
  - Reads and writes to the same variable might require calls to `__syncthreads`
  - It's important that if one thread calls `__syncthreads`, all other threads do so as well
- `__managed__`
  - Accessible directly from GPU and CPU code alike
- `__device__`
  - In device memory only, special function calls needed for copying from host.

UPPSALA UNIVERSITET

Note on unified memory

- Even if you have unified memory, you can “advise” CUDA that you will need a piece of data on one device, or the other
- Not necessary, can sometimes help performance

UPPSALA UNIVERSITET

CUB

- CUB is a library for doing warp-wide, block-wide, and device-wide operations
- Parts of it are used in the Cuda backend for Thrust
- Reduces the risk for errors for slightly more complex synchronizing operations
  - Simplify operations like “use all threads in the block to collaborate on loading data into shared memory”

UPPSALA UNIVERSITET

What can you do in CUDA?

- Allocate dynamic memory
- Do recursive function calls
- Use polymorphism with virtual methods
  - If the object is created on the GPU
- Call another CUDA kernel (with `-rdc=on` and `-lcudadevrt` during compilation)
- Some header-based C++ libraries can be verbatim included for use on GPUs, or very minor changes
- You can do *more* things reasonably efficiently in pure CUDA compared to OpenMP target

UPPSALA UNIVERSITET

What can you *not* do in CUDA?

- C++ exception handling
  - Throwing code will sometimes compile
    - But it will not throw properly
  - Catching code will never compile
- C++ exception handling is a very contentious feature in general and frequently frowned upon in high-performance CPU code
  - Disabling C++ exceptions in a similar way allows CPU compiler to do more optimizations as well…

UPPSALA UNIVERSITET

Other frameworks

- OpenCL
  - Standardized “alternative” to Cuda. Unfortunately much worse language support.
  - Started out from the HLSL/GLSL concept of “sending a string of source code to the driver”.
- SyCL
  - Attempt to bring sort of standard C++ to GPU. Awkward syntax, in my opinion. No mature open implementation.
    - Cuda has a relatively mature parallel implementation in the Clang compiler.
- OpenACC
  - Similar to OpenMP Target, but more heavily geared towards GPUs. Some proprietary implementations might be faster than OpenMP Target, both Clang and GCC base their OpenACC support on mapping it to OpenMP equivalents.
- Standard C++
  - Nvidia bought the compiler vendor PGI.
  - Promised feature in their HPC SDK 20.05 to have a compiler where standard STL features are executable on GPU.
    - Mostly removing the need for a pseudo-specific library like Thrust.
    - Improved support for corner cases for unified memory will help.

UPPSALA UNIVERSITET

My recommendation

- For now:
  - For open standards, go for OpenMP Target.
  - If you need to maximize performance, give Cuda a look
    - Possibly using CUB or Thrust when those map well, avoid rolling your own low-level code when you can.
- We will see languages standardizing the features needed to coordinate light threads properly, some in C++20, some in C++23.
  - These will benefit other heterogeneous architectures as well, to some extent. (C++ on FPGA?! High-level synthesis.)

UPPSALA UNIVERSITET

GPUs from Python

- The deep learning frameworks are on way to access GPUs from Python
- If those were the perfect way to express any algorithm, we would write all our CPU computations using them as well
  - So maybe they are not ideal, after all

UPPSALA UNIVERSITET

MinPy

- A library tying together MXNet support and a Numpy API

- Ideally, it's as simple as replacing

```
import numpy as np
```

with

```
import minpy as np
```

Some pitfalls exist when you're exchanging numpy arrays with other modules, and using certain features.

UPPSALA UNIVERSITET

GPU support

- Minpy will, by default, use MXNet when present
- So, in theory, you can just try running the same piece of code with `numpy` or `minpy` imported
- Important caveat
  - Functions on arrays that modify the array itself are generally not allowed
  - Whether that has a huge effect or not depends a lot on your coding style
- Many functions exist as method working on the array, and as separate functions in the `numpy` module

UPPSALA UNIVERSITET

Autograd

- MinPy also has autograd support. One of their examples:

```
from minpy.core import grad

# define a function: f(x) = 5*x^2 + 3*x - 2
def foo(x):
    return 5*(x**2) + 3*x - 2

# f(4) = 90
print(foo(4))

# get the derivative function by `grad`: f'(x) = 10*x + 3
d_foo = grad(foo)

# f'(4) = 43.0
print(d_foo(4))
```

UPPSALA UNIVERSITET

Autograd

- This works if the return value is a vector or matrix, rather than a scalar, as well.
- If you have a function taking multiple parameters, you need to specify the indices of the parameters to `grad`

```
def foo(x,y):
    return 5*(x**2) + 3*y - 2

d_foo = grad(foo, [0, 1])

print(d_foo(4,6))
```

- The related function `grad_and_loss` will create a function that returns a tuple of the gradient and the plain return value.

UPPSALA UNIVERSITET

afnumpy

- afnumpy is a package with a similar goal as minpy, but focusing solely on numpy equivalents for GPU.
  - Developed here in Uppsala, a wrapper around the ArrayFire package for the actual GPU computations.
  - Since no autograd etc is involved, the performance might sometimes be better.
- Far less of a community than MinPy, and even MinPy is pretty small.

- Huge selling point of both: Small changes to existing Python code!

UPPSALA UNIVERSITET

numba

- numba is a general package for compiling Python code
- It uses the LLVM compiler backend (which is also used for the Clang compiler) and converts Python code to something LLVM can understand.
- LLVM can generate Nvidia assembler.
  - As already mentioned, Clang can compile most Cuda code.
- Two main usage modes of numba for GPU:
  - ufunc-like objects
  - Cuda kernels in a Python subset

UPPSALA UNIVERSITET

General numba

- Just put `@jit` at the function you want to go fast

```
from numba import jit

@jit
def f(x, y):
    return x + y
```

- If you call a Python function numba doesn't know about (like one of your own non-`@jit` functions), it will suddenly be slow again. You can put `nopython=True` to get an error in that case.

```
from numba import jit

@jit(nopython=True)
def f(x, y):
    return x + y
```

UPPSALA UNIVERSITET

`nogil`, `fastmath`, `parallel`

- The standard Python implementation uses the global interpreter lock (GIL), basically meaning that only a single thread can run actual Python code at a time
  - Some libraries release this lock during large operations, allowing multiple threads to run efficiently
  - Releasing the lock in the jitted code allows for some synchronization errors that would not occur, or be very rare, in ordinary Python code.
- `fastmath` allows a set of speed-ups for floating point that compilers also tend to enable with flags with similar names, including things like allowing (a + b) + c be rearranged into a + (b + c), if that's beneficial.
- `parallel` instructs Numba to try to parallelize a single function if there is e.g. a long loop or a large array operation
  - You need to use `prange` rather than `range` to state that loop iterations are independent.
  - `@jit(nopython=True, parallel=True, nogil=True, fastmath=True)`

UPPSALA UNIVERSITET

Eager compilation

- You can precompile the code if you specify beforehand what types it is intended for

```
@jit(int32(int32, int32))
def f(x, y):
    return x + y
```

UPPSALA UNIVERSITET

ufuncs

- Numpy supports ufuncs, universal functions, which adhere to the overall conventions of numpy (including broadcasting) while iterating elementwise over one or multiple arrays
- These are normally written in some compiled language
- numba can support compiling an ordinary scalar function as a ufunc

```
@vectorize([int32(int32, int32),
            int64(int64, int64),
            float32(float32, float32),
            float64(float64, float64)])
def f(x, y):
    return x + y
```

UPPSALA UNIVERSITET

Vectorization targets

- Vectorize supports several targets

`@vectorize(target='...')`

- `cpu` – default, best for small data (1000 elements?)
- `parallel` – multiple CPU threads, for maybe 100000 elements
- `cuda` – transfer data to the GPU and back behind-the-scenes
- `roc` – transfer data to a supported AMD GPU in the same way

- ufuncs support many features out-of-the-box in numpy, like calling `myufunc.reduce(a,b)` in order to compute the ufunc over `a` and `b` and then sum the results

UPPSALA UNIVERSITET

Actual Cuda kernels in numba

- You can also write explicit Cuda kernels in the numba Cuda subset of Python directly
  - In these, you get `numba.cuda.threadIdx`, `numba.cuda.blockDim`, `numba.cuda.blockIdx`, `numba.cuda.gridDim`, just like in the C++ CUDA
  - Also two helpers `numba.cuda.grid` and `numba.cuda.gridsize` to get absolute indices, combining the grid and block dimensions

UPPSALA UNIVERSITET

Increment each element in an array by one

```
@cuda.jit
def increment_a_2D_array(an_array):
    x, y = cuda.grid(2)
    if x < an_array.shape[0] and y < an_array.shape[1]:
       an_array[x, y] += 1

threadsperblock = (16, 16)
blockspergrid_x = math.ceil(an_array.shape[0] / threadsperblock[0])
blockspergrid_y = math.ceil(an_array.shape[1] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)
increment_a_2D_array[blockspergrid, threadsperblock](an_array)
```

- We're using brackets rather than triple angle brackets for specifying grid and block size

UPPSALA UNIVERSITET

Data transfers and memory

- Array parameters to kernels are transferred automatically to and from the GPU.
- For more efficient code, explicit allocations can be made using `numba.cuda.device_array`
- The kernel can access shared memory using `numba.cuda.shared.array`
- Read/writes to the same position shared memory from different threads need `numba.cuda.syncthreads()`, just like in “real” CUDA

UPPSALA UNIVERSITET

Atomics

- A subset of atomics are supported, on indices within arrays, from `numba.cuda.atomic`
  - `add`
  - `compare_and_swap`
  - `max`
  - `min`

UPPSALA UNIVERSITET

Reduction

```
@cuda.reduce
def sum_reduce(a, b):
    return a + b

sum_reduce = cuda.reduce(lambda a, b: a + b)
```

- Both of these create a callable object that will accept a full array, and combine all elements in the specified reduction operation.
  - The reduction operation does not have to be addition, of course, but any Python code combining elements.

UPPSALA UNIVERSITET

Debugging

- Debugging Python code is far easier when it is normal Python
  - The numba Cuda support adds thread indices and other things that are simply not part of normal Python
  - numba provides a facility for running the kernel within the normal Python interpreter for this
- Involves setting the evinronment variable `NUMBA_ENABLE_CUDASIM` to 1 before loading numba

UPPSALA UNIVERSITET

What to recommend?

- Do you have an existing, fast, numpy code that you want to run on GPU?
  - Try `MinPy`.
- Do you have an existing Python code with performance problems?
  - Try `numba`, both on CPU and GPU!
  - Numba also provides support for the proprietary AMD GPU API in a similar manner

UPPSALA UNIVERSITET

Lab 2

- Try one of the frameworks, with a problem you choose yourself, or with the final Python code from prelab 2. You can also try to speed up one of the C++ versions.
- You can start out from the various commands used in prelab 1 and 2 to use them from a container.
- If you want to start a notebook with afnumpy, MinPy, MXNet, and numba, use `./notebook.py -- gpulibs`
- You can still add additional flags, including `--reservation=g2020014_02` before the double-dash.
- Make a really short write-up, preferably in a markdown file or a Jupyter notebook, on the algorithm, what you tried, and your results. E-mail it to me.

UPPSALA UNIVERSITET

Q&A