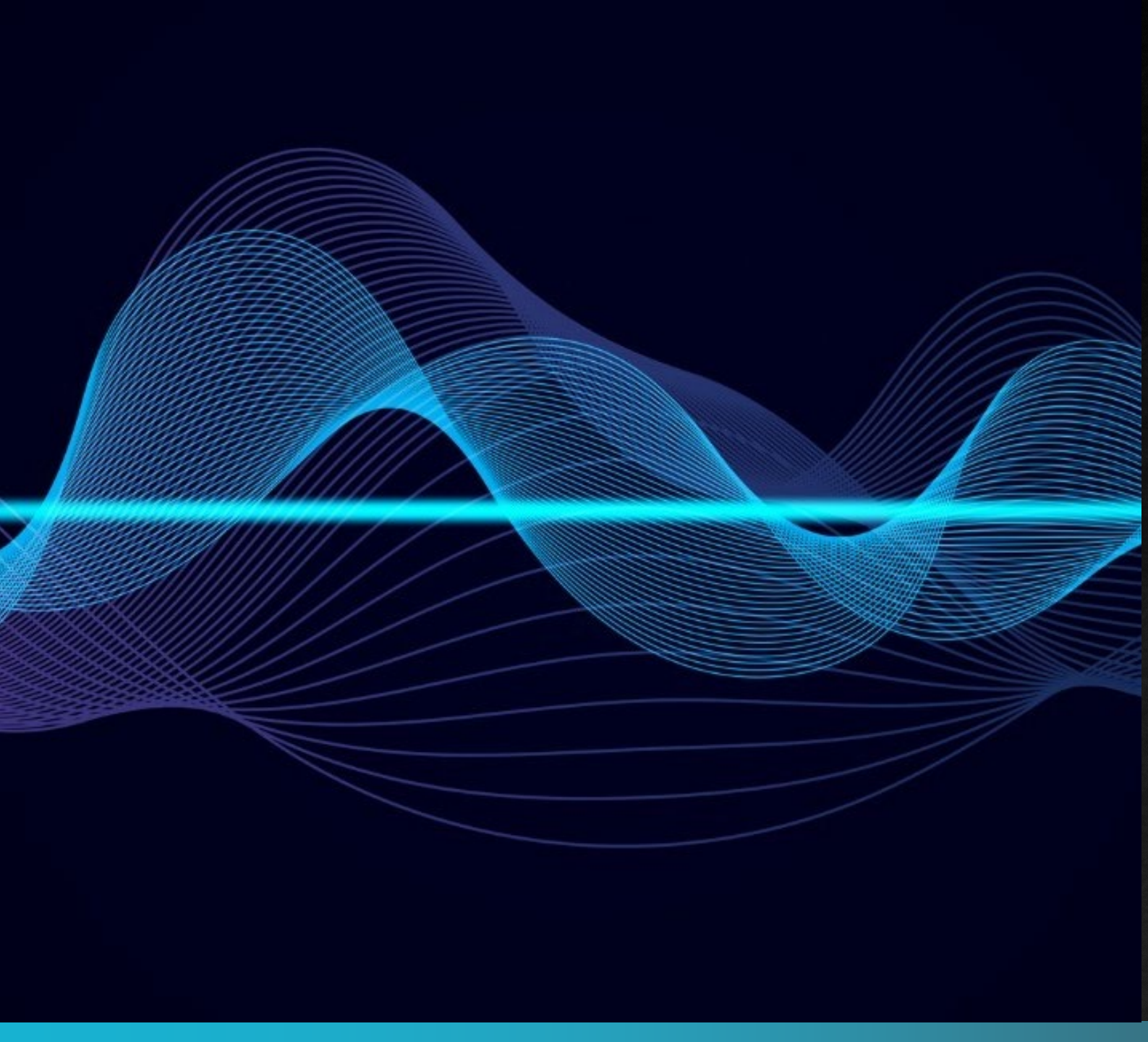


Computational Physics

PHYS 6260

Parallel Programming MPI / GPU

Announcements:

- HW6: Posted, due Friday 3/1
- Project proposal: due Friday 3/8

Lecture Outline

We will cover these topics

- Finishing up MPI
- GPU acceleration
  - OpenACC
  - pyCUDA

Weak versus Strong Scaling

- In assessing the parallel performance of code, there are two common methods
- Strong scaling: keep the problem size fixed and increase the number of cores
  - Eventually you will become work-starved
  - Scaling will stop when communication and overhead dominate
- Weak scaling: increase the amount of work in proportion to the number of cores
  - Perfect scaling will result in the same “wall clock” time for all core counts

Parallel debugging

- There are parallel debuggers, but they are usually expensive
- It’s possible to spawn multiple gdb (GNU debugger) sessions, but this gets out of hand quickly – “mpirun -n 2 xterm -e gdb ./a.out”
- Print is still your friend
  - Run a small of a problem as possible on as few cores as necessary
- Some round off differences are to be expected from sums (different order of operation … numeric addition is not associative)

Hybrid parallelism

- To get good performance on current HPC platforms (>1k cores), you need to use hybrid parallelism
- OpenMP within a node / socket, MPI across nodes
- For example in our MPI relaxation coe, we could split the loops over each subdomain over multiple cores on a node using OpenMP
- Then we have MPI to communicate across nodes and OpenMP within the nodes
- The hybrid approach is often needed to get the best performance on big machines

GPUs

- GPU offloading can greatly accelerate computing
- Main issue: data needs to be transferred from the CPU/RAM across the PCIe bus (relatively slow) to the GPU
- Good performance requires that lots of work is done on the data to “pay” for the cost of the transfer

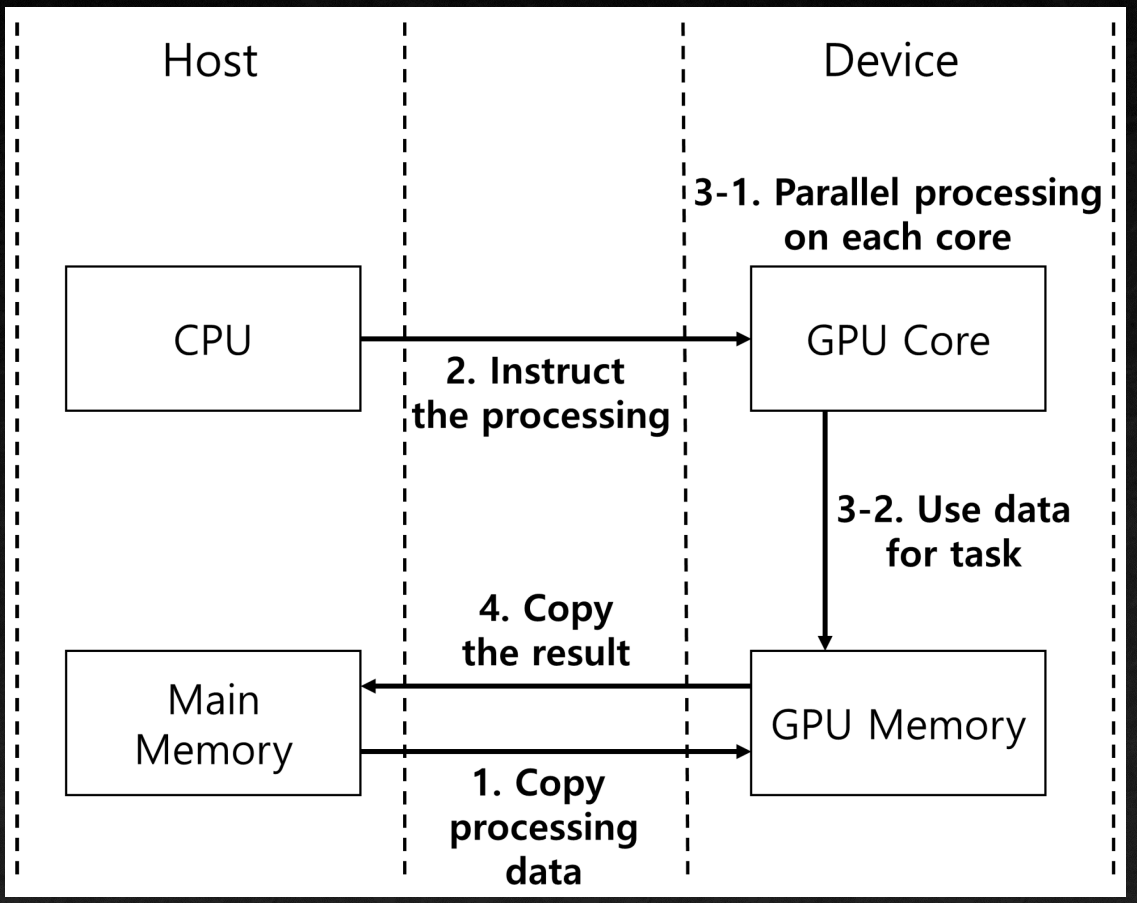
- GPUs work as SIMD parallel machines (vectorized)
  - The same instructions operate on all the data in lockstep
  - Branching (if-statements) is slower
- Best performance requires that you structure the code to be vectorized

GPUs

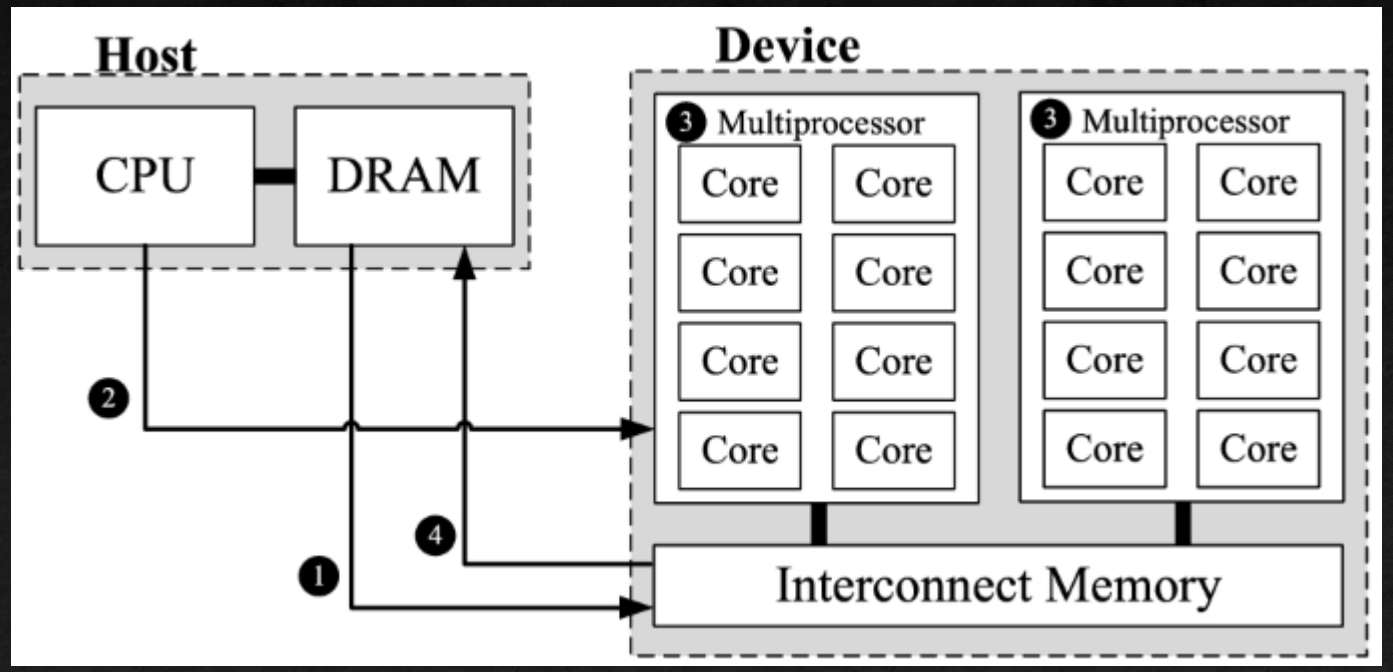
- GPU offloading can greatly accelerate computing
- Main issue: data needs to be transferred from the CPU/RAM across the PCIe bus (relatively slow) to the GPU
- Good performance requires that lots of work is done on the data to “pay” for the cost of the transfer

- GPUs work as SIMD parallel machines (vectorized)
  - The same instructions operate on all the data in lockstep
  - Branching (if-statements) is slower
- Best performance requires that you structure the code to be vectorized

GPUs: Hardware considerations



- Memory movement is experience and should be minimized



GPUs

- CUDA is popular but is not portable to non-nVidia GPUs and accelerators
- In my experience, directive-based frameworks are the best initial option to explore possible acceleration
- OpenMP supports offloading to GPUs
  - In GNU compilers since v5
  - Full slideset from OpenMP org
  - Now you have to copy the data to/from the GPU with the “target” and “map” directives

GPUs: OpenACC

- OpenACC is one such framework for offloading computing to GPUs
  - Looks like OpenMP
  - A big difference is that you need to explicitly add directives that control data movement from CPU → GPU
  - Word of warning: I spent a few hours trying to get OpenACC to work with the GNU compilers on Ubuntu and finally gave up. NVIDIA's HPC SDK worked for me, though!
- To offset the big cost of moving data from CPU → GPU → CPU
  - Need to perform a lot of computing on the GPU for each data movement
  - We can separately control what is copied to and from the GPU
- Let's look at a relaxation code with OpenACC and OpenMP

OpenACC Example: Relaxation

```
73
74  #pragma acc data copy(A[0:n][0:m]) create(Anew[0:n][0:m])
75      while ( error > tol && iter < iter_max )
76      {
77          error = 0.f;
78
79  #pragma omp parallel for shared(m, n, Anew, A) schedule(dynamic)
80
81  #pragma acc parallel loop reduction(max:error)
82  // #pragma acc parallel loop // without reduction(max:error) no computation
83          for( int j = 1; j < n-1; j++)
84          {
85         #pragma acc loop reduction(max:error)
86        // #pragma acc loop // without reduction(max:error) no computation
87           for( int i = 1; i < m-1; i++ )
88              {
89                  Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
90                                       + A[j-1][i] + A[j+1][i]);
91                  error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
92              }
93          }
94
```

To compile with NVHPC:
nvc –acc 15_relax.c

(see comment at top of code what worked with gcc in 2023 but doesn't in 2024 ☹)

OpenACC Example: Relaxation

We need a machine with a discrete GPU (i.e. separate card) to realize any performance gains

Usually these computers are desktops, gaming laptops, or HPC nodes with GPUs

Let’s test it out on my desktop. On Monday, you will test it on a GPU node of PACE-ICE

In my testing before class, I saw ~5x speedup when compared to OpenMP.

Python and CUDA: Numba and PyCUDA

- There are two popular python modules that wrap the CUDA API

PyCUDA

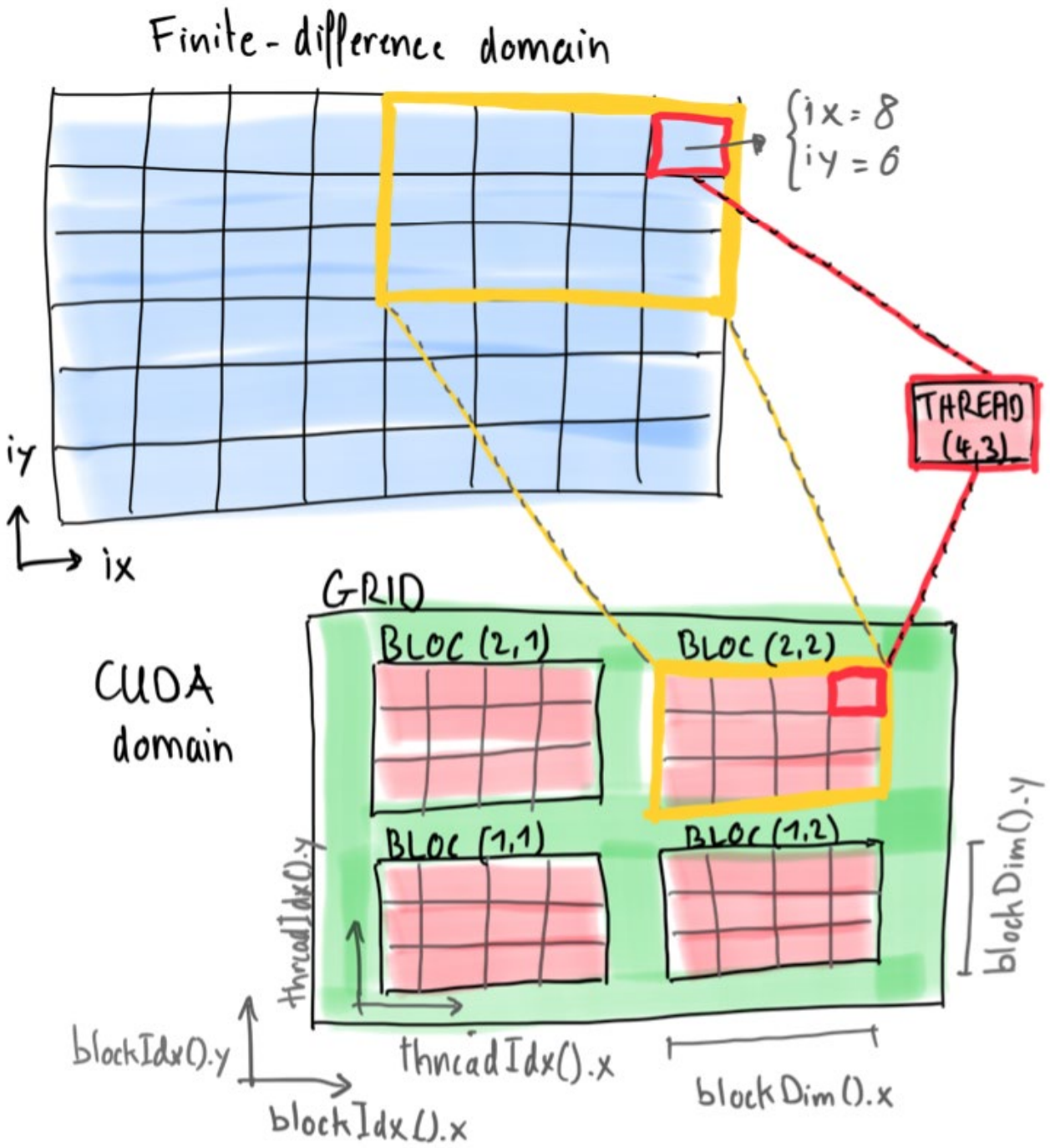
- Resembles similar operations as plain CUDA but without the C/C++ developmental overhead (declarations, types, etc)

Numba

- Module for “Just in Time” compilation of Python into C for faster code
- We used it in the Ising Model from the Monte Carlo Simulations lecture
- Numba has a CUDA module that allows for CUDA code generation
- Instiantiated with decorators
- We’ll first look at some Numba+CUDA code because it’s simpler

CUDA data hierarchy

- GPUs consist of many cores (>1000)
- Each can support a large number of threads
- Threads are grouped into “blocks”
  - Blocks can be 1D, 2D, or 3D
- Each “kernel” (i.e. function) call uses a “grid” of blocks
  - Grids can be 1D or 2D
- Programmer will specify the grid/block organization on each kernel call
  - Must be within the limits of the GPU



Numba + CUDA

- Need to specify threads/block and # of blocks
- A decorator compiles the function in CUDA
- It cannot return anything. The array must be modified in place.

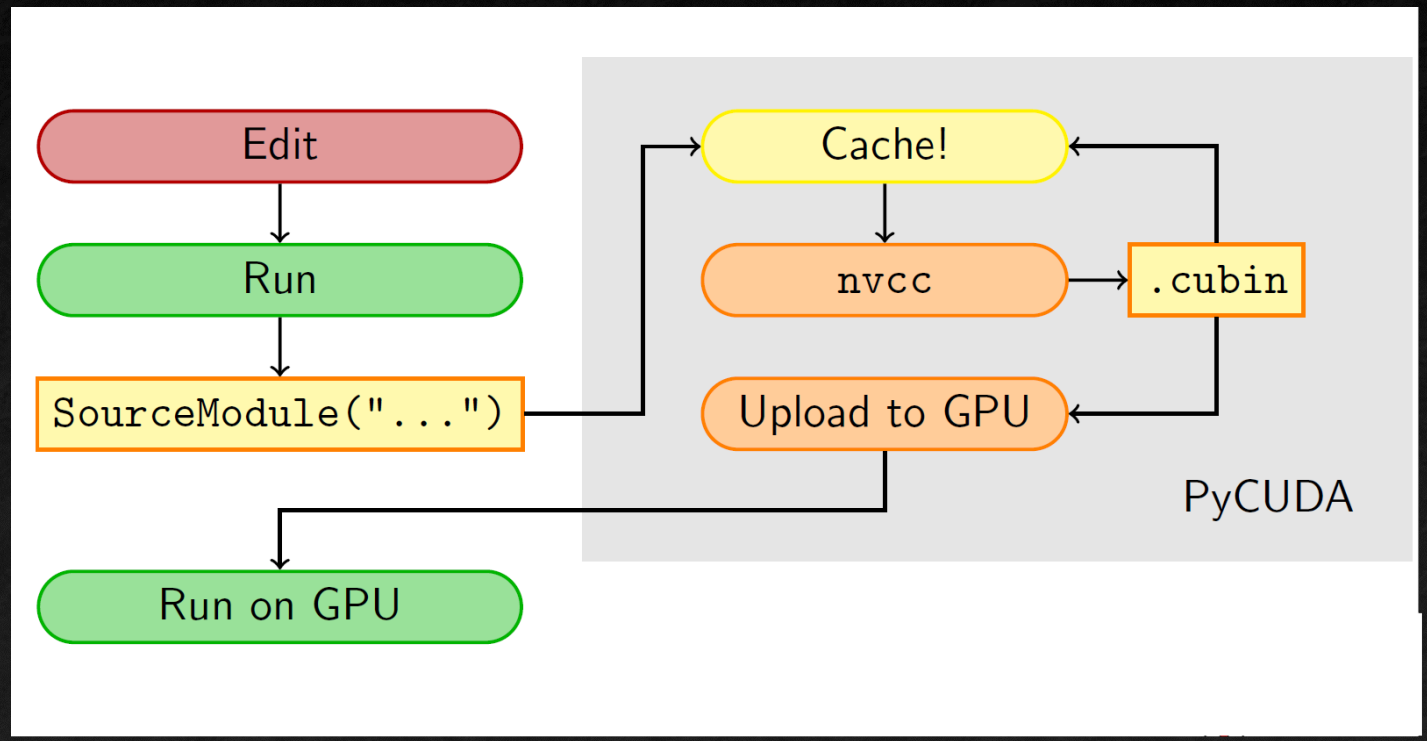
```
@cuda.jit
def compute_pi(rng_states, iterations, out):
    """Find the maximum value in values and store in result[0]"""
    thread_id = cuda.grid(1)

    # Compute pi by drawing random (x, y) points and finding what
    # fraction lie inside a unit circle
    inside = 0
    for i in range(iterations):
        x = xoroshiro128p_uniform_float32(rng_states, thread_id)
        y = xoroshiro128p_uniform_float32(rng_states, thread_id)
        if x**2 + y**2 <= 1.0:
            inside += 1

    out[thread_id] = 4.0 * inside / iterations
```

```
threads_per_block = 64
blocks = 4096
rng_states = create_xoroshiro128p_states(threads_per_block * blocks, seed=1)
out = np.zeros(threads_per_block * blocks, dtype=np.float32)
```

```
iter = 10000
start = time.perf_counter()
compute_pi[blocks, threads_per_block](rng_states, iter, out)
end = time.perf_counter()
```

PyCUDA: More low-level

- PyCUDA exposes all of CUDA
  - Arrays & textures
  - Memory transfers
  - OpenGL Interoperability
  - Device queries
  - Streams & Events
  - and so on
- PyCUDA philosophy
  - Provide complete access
  - Automatically manage resources
  - Provide abstractions
  - Allow interactive use
  - Check for and report errors
  - Integrate tightly with numpy



PyCUDA: Example code

```
mod = cuda.SourceModule("""
    __global__ void doublify(float *a)
    {
      int idx = threadIdx.x + threadIdx.y*4;
      a[idx] *= 2;
    }
    """)

func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```

Compute kernel

See Canvas for ZIP file or Teams for Google Drive link (to copy into your own gDrive and use in Google Colab).

- 0: Checking device details and pyCUDA is installed and works
- 1: Simple add (1 grid, 1 block = 1 element)
- 2: Parallel add (multiple grids, 1 block = 1 element)
- 3: Threaded parallel add (multiple grids, multiple blocks)
- 4: Threaded parallel multiply 2D array
- 5: Calculate distance in an element-by-element fashion
  - Uses ElementWise kernel instead of writing the native CUDA kernel

PyCUDA: Abstraction

```
import numpy
import pycuda.autoinit
import pycuda.gpuarray as gpuarray

a_gpu = gpuarray.to_gpu(
    numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
```