Making Stellar Popcorn Pancakes in the Name of Science

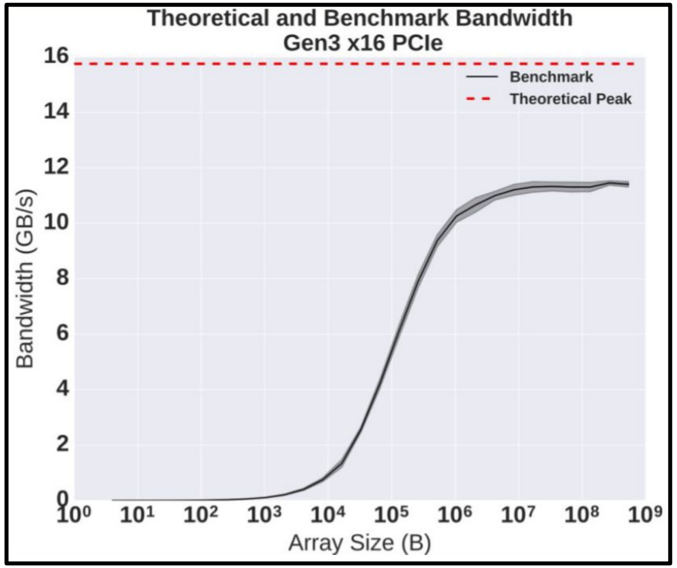
Roger Hatfull

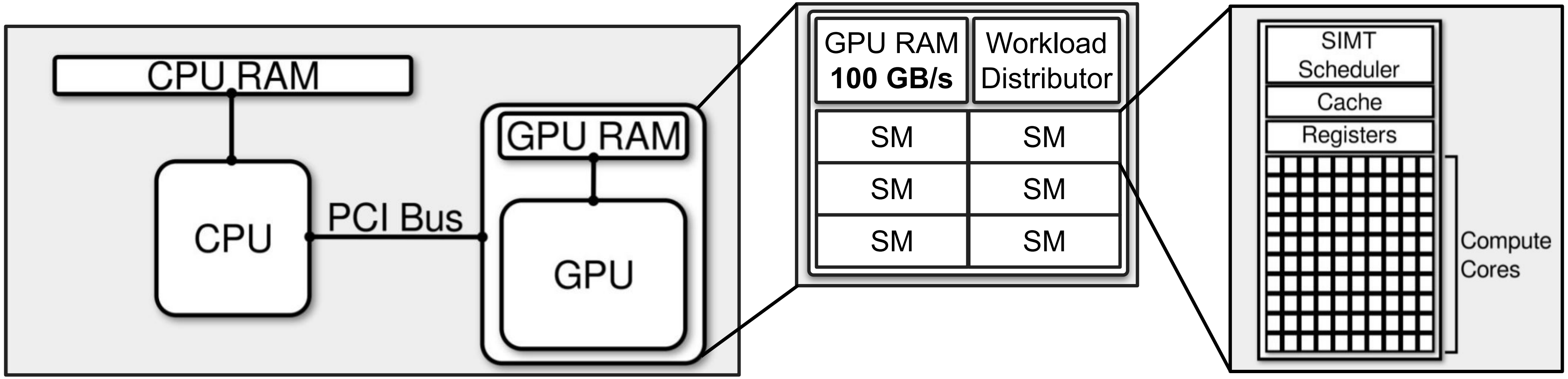
Code-along!

- Access examples at https://github.com/hatfullr/astroph-11042021
- Install Numba for Python
  - Install the CUDA Toolkit (https://developer.nvidia.com/cuda-downloads)
  - Using `pip` on x86/x86_64:
    1. Install pip if needed: `python3 -m ensurepip --upgrade`
       - windows: `py -m ensurepip --upgrade`
    2. `pip install numba`
  - Using Anaconda on x86/x86_64/POWER:
    1. `conda install numba` OR `conda update numba`
    2. `conda install cudatoolkit`
  - From source: follow instructions on the website (https://numba.pydata.org/numba-doc/latest/user/installing.html#installing-from-source)
- Check your GPU at any time with command `nvidia-smi`

What is a GPU? (Hardware Perspective)

- GPU (“Device”)
  - PCIe x16 Gen3 bandwidth (one way)
    - Theoretical = 15.75 GB/s
    - Practical < Theoretical
  - With NVLINK, theoretical = 35 GB/s





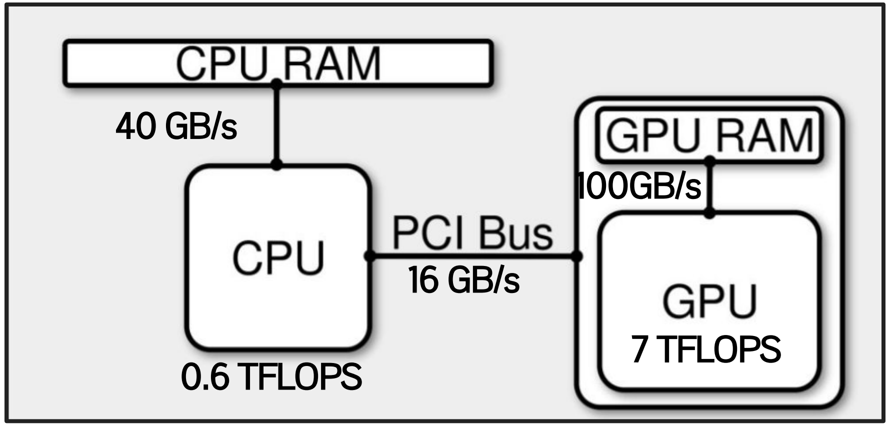
What is a GPU? (Hardware Perspective, contd.)

- FLOPS = FLoating point Operations per Second
- Single precision = “float” or “float32” (~ 7-8 decimal places)
- Double precision = “double” or “float64” (~ 16 decimal places)
- RTX 2070
  - Single precision: ~ 3.5 TFLOPS
  - Double precision: ~ 0.2 TFLOPS
- V100 Volta (Cedar, ComputeCanada)
  - Single precision: ~ 14 TFLOPS
  - Double precision: ~ 7 TFLOPS
- i9 9900K
  - Single precision: ~ 1.3 TFLOPS
  - Double precision: ~ 0.6 TFLOPS
- i7 9700K
  - Single precision: ~ 0.4 TFLOPS

Made for gaming

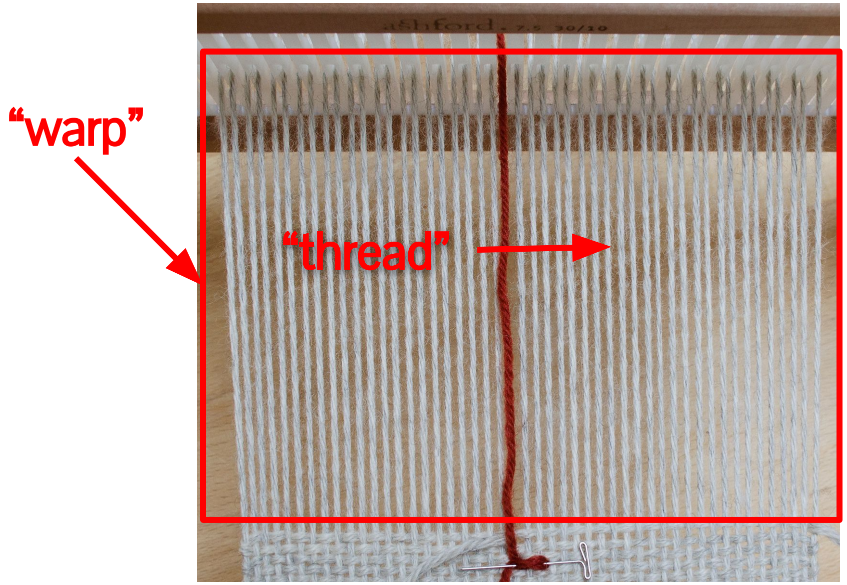
Made for science

The PCI Bus is the slowest part of a computer! Only transfer data between CPU and GPU when necessary.

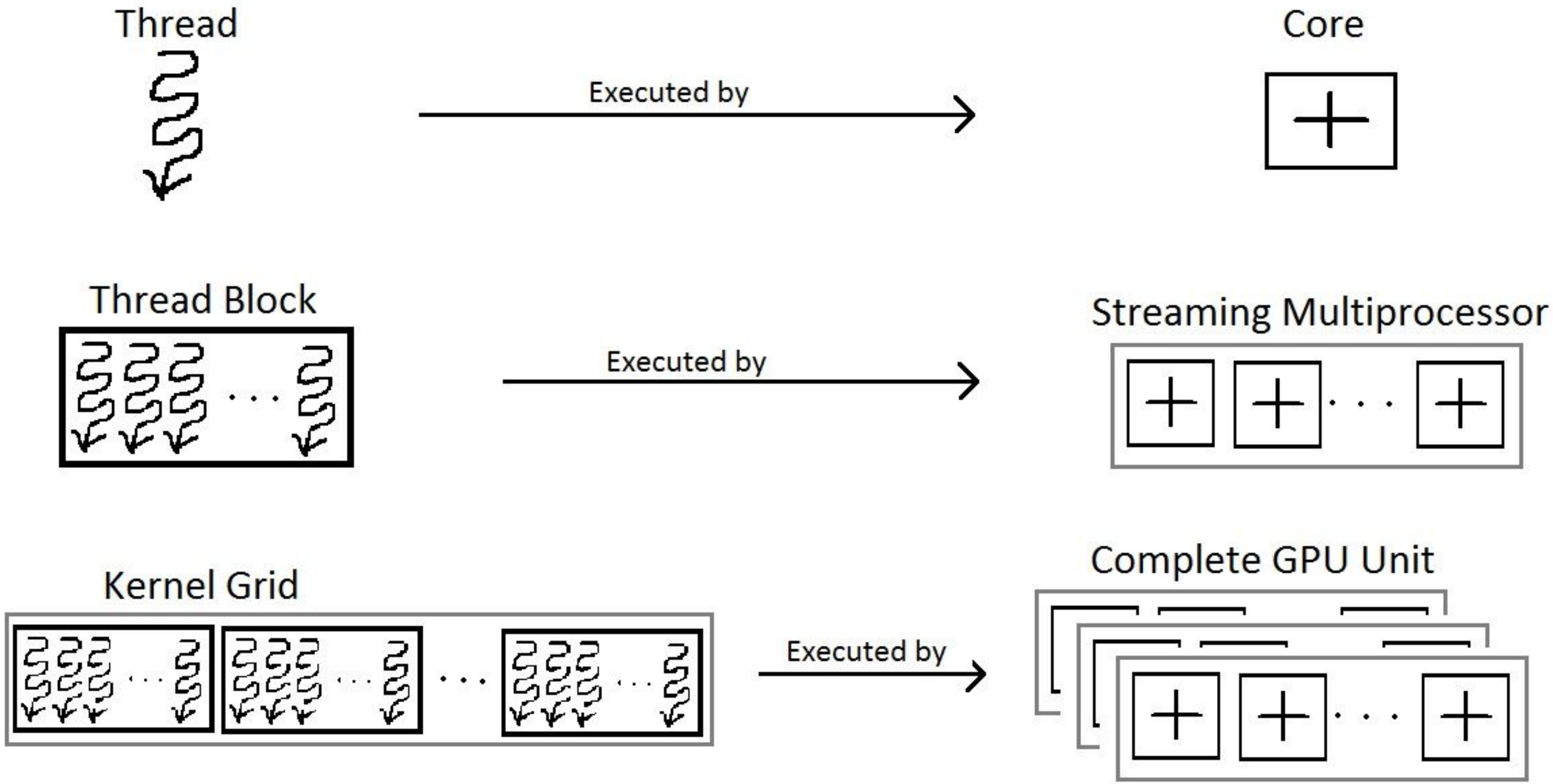


What is a GPU? (Software Perspective)

- Step 1: Understanding weaving
  - On a loom, threads are held in tension
    - All the threads are called a "warp"
- Step 2: Understanding GPUs
  - **Thread**: A single processor
  - **Warp**: A collection of 32 Threads
  - **Block**: A collection of Warps
  - **Grid**: A collection of Blocks
- You should request a number of Threads that is a multiple of 32
  - Otherwise, you might accidentally consume GPU resources without using them
- Sometimes it helps to abstract the Grid into multiple dimensions
  - For example: for an image, you can request a Grid with N x M Blocks
  - For example: for a 3D simulation, you can request a Grid with N x M x K Blocks
  - You can ask the GPU for the Block and Thread indices



Overview


Thread
Executed by
Core
Thread Block
Executed by
Streaming Multiprocessor
Kernel Grid
Executed by
Complete GPU Unit

example_simple.py

- ➤ Calculate `C = A + B`

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

N = 100000000
A = np.ones(N)
B = np.ones(N)
C = np.empty(N) # An empty array

device_A = cuda.to_device(A)
device_B = cuda.to_device(B)
device_C = cuda.to_device(C)

# Define a "device" (GPU) function:
@cuda.jit # This thing is called a "decorator"
def add(A,B,C):
    i = cuda.grid(1)
    if i < C.size:
        C[i] = A[i] + B[i]

threadsperblock=512
blockspergrid = N // threadsperblock + 1

add[blockspergrid,threadsperblock](
    device_A,
    device_B,
    device_C,
)
cuda.synchronize()

C = device_C.copy_to_host()

if all(C == 2):
    print("1+1=2. GPUs aren't so scary!")
else:
    print("Oh, no. 1+1!=2. GPUs are terrifying...")
```

example_simple.py

Create the host (CPU) arrays:

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

N = 100000000
A = np.ones(N)
B = np.ones(N)
C = np.empty(N)  # An empty array
```

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

N = 100000000
A = np.ones(N)
B = np.ones(N)
C = np.empty(N) # An empty array

device_A = cuda.to_device(A)
device_B = cuda.to_device(B)
device_C = cuda.to_device(C)

# Define a "device" (GPU) function:
@cuda.jit # This thing is called a "decorator"
def add(A,B,C):
    i = cuda.grid(1)
    if i < C.size:
        C[i] = A[i] + B[i]

threadsperblock=512
blockspergrid = N // threadsperblock + 1

add[blockspergrid,threadsperblock](
    device_A,
    device_B,
    device_C,
)
cuda.synchronize()

C = device_C.copy_to_host()

if all(C == 2):
    print("1+1=2. GPUs aren't so scary!")
else:
    print("Oh, no. 1+1!=2. GPUs are terrifying...")
```

example_simple.py

Send the arrays to the device (GPU):

```
device_A = cuda.to_device(A)
device_B = cuda.to_device(B)
device_C = cuda.to_device(C)
```

N. B. : Here we do not use `np.ascontiguousarray(A)`, but in other cases you might find this necessary. Numba will throw an error if it needs your arrays to be stored contiguously in memory. “Contiguous” means each element of an array is stored in one continuous block of memory, rather than in many various locations.

```
from numba import cuda
import numpy as np

N = 100000000
A = np.ones(N)
B = np.ones(N)
C = np.empty(N) # An empty array

device_A = cuda.to_device(A)
device_B = cuda.to_device(B)
device_C = cuda.to_device(C)

# Define a "device" (GPU) function:
@cuda.jit # This thing is called a "decorator"
def add(A,B,C):
    i = cuda.grid(1)
    if i < C.size:
        C[i] = A[i] + B[i]

threadsperblock=512
blockspergrid = N // threadsperblock + 1

add[blockspergrid,threadsperblock](
    device_A,
    device_B,
    device_C,
)
cuda.synchronize()

C = device_C.copy_to_host()

if all(C == 2):
    print("1+1=2. GPUs aren't so scary!")
else:
    print("Oh, no. 1+1!=2. GPUs are terrifying...")
```

`example_simple.py` (contd.)

Define a device (GPU) function:

```
# Define a "device" (GPU) function:
@cuda.jit # This thing is called a "decorator"
def add(A, B, C):
    # Get the position of the current thread in the
    # entire grid of blocks. We use "1" as input
    # because we are using a single dimension grid.
    i = cuda.grid(1)

    # Make sure this thread is one that is actually
    # being used in our calculations. The number of
    # threads we requested may not be a perfect
    # multiple of 32 (32 threads in a warp), so
    # this step just makes sure we are only using
    # the threads we actually requested.
    if i < C.size:
        C[i] = A[i] + B[i]
```

```
from numba import cuda
import numpy as np

N = 100000000
A = np.ones(N)
B = np.ones(N)
C = np.empty(N) # An empty array

device_A = cuda.to_device(A)
device_B = cuda.to_device(B)
device_C = cuda.to_device(C)

# Define a "device" (GPU) function:
@cuda.jit # This thing is called a "decorator"
def add(A,B,C):
    i = cuda.grid(1)
    if i < C.size:
        C[i] = A[i] + B[i]

threadsperblock=512
blockspergrid = N // threadsperblock + 1

add[blockspergrid,threadsperblock](
    device_A,
    device_B,
    device_C,
)
cuda.synchronize()

C = device_C.copy_to_host()

if all(C == 2):
    print("1+1=2. GPUs aren't so scary!")
else:
    print("Oh, no. 1+1!=2. GPUs are terrifying...")
```

example_simple.py (contd.)

Specify how many threads per block we will request:

```
# Define how many threads we would like to use per
# block that we request. This should always be a
# multiple of 32, which is the number of threads in
# a warp. If you instead requested 511 threads per
# block, then the GPU would still allocate 16 warps
# (16x32=512 threads) per block, but only 1 of the
# threads would receive no instructions at all.
threadsperblock=512
```

Specify how many blocks we will request:

```
# We request to use some number of blocks such that
# exactly 1 thread corresponds to 1 index of our A,
# B, and C arrays. We round the number up so that
# we don't accidentally request too few blocks.
blockspergrid = N // threadsperblock + 1
```

```
from numba import cuda
import numpy as np

N = 100000000
A = np.ones(N)
B = np.ones(N)
C = np.empty(N) # An empty array

device_A = cuda.to_device(A)
device_B = cuda.to_device(B)
device_C = cuda.to_device(C)

# Define a "device" (GPU) function:
@cuda.jit # This thing is called a "decorator"
def add(A,B,C):
    i = cuda.grid(1)
    if i < C.size:
        C[i] = A[i] + B[i]

threadsperblock=512
blockspergrid = N // threadsperblock + 1

add[blockspergrid,threadsperblock](
    device_A,
    device_B,
    device_C,
)
cuda.synchronize()

C = device_C.copy_to_host()

if all(C == 2):
    print("1+1=2. GPUs aren't so scary!")
else:
    print("Oh, no. 1+1!=2. GPUs are terrifying...")
```

`example_simple.py` (contd.)

Call the device (GPU) function:

```
add[blockspergrid,threadsperblock](
    device_A,
    device_B,
    device_C,
)
```

N. B. : Here we are communicating to the GPU how many threads total we would like to invoke with the syntax `[blockspergrid,threadsperblock]`.

If we use our host arrays `A`, `B`, and `C`, we will get a warning that says "overhead incurred!" as the kernel decides to automatically copy host arrays to the device.

Memory management is the key to speed!

If you are using arguments that are not arrays, then you do not need to copy them to the device.

```
from numba import cuda
import numpy as np

N = 100000000
A = np.ones(N)
B = np.ones(N)
C = np.empty(N) # An empty array

device_A = cuda.to_device(A)
device_B = cuda.to_device(B)
device_C = cuda.to_device(C)

# Define a "device" (GPU) function:
@cuda.jit # This thing is called a "decorator"
def add(A,B,C):
    i = cuda.grid(1)
    if i < C.size:
        C[i] = A[i] + B[i]

threadsperblock=512
blockspergrid = N // threadsperblock + 1

add[blockspergrid,threadsperblock](
    device_A,
    device_B,
    device_C,
)
cuda.synchronize()

C = device_C.copy_to_host()

if all(C == 2):
    print("1+1=2. GPUs aren't so scary!")
else:
    print("Oh, no. 1+1!=2. GPUs are terrifying...")
```

`example_simple.py` (contd.)

All the threads compute `C[i] = A[i] + B[i]` asynchronously, so before we copy our data from the device to the host, we need to make sure all the threads have finished first:

```
cuda.synchronize()
```

Finally, we copy our result from the device to the host (CPU):

```
C = device_C.copy_to_host()
```

and check the result for accuracy:

```
if all(C == 2):
    print("1+1=2. GPUs aren't so scary!" )
else:
    print("Oh, no. 1+1!=2. GPUs are terrifying..." )
```

Try a smaller value for `threadsperblock` if you encounter:

```
numba.cuda.cudadrv.driver.CudaAPIError: [1] Call
to cuLaunchKernel results in
CUDA_ERROR_INVALID_VALUE
```

```
from numba import cuda
import numpy as np

N = 100000000
A = np.ones(N)
B = np.ones(N)
C = np.empty(N) # An empty array

device_A = cuda.to_device(A)
device_B = cuda.to_device(B)
device_C = cuda.to_device(C)

# Define a "device" (GPU) function:
@cuda.jit # This thing is called a "decorator"
def add(A,B,C):
    i = cuda.grid(1)
    if i < C.size:
        C[i] = A[i] + B[i]

threadsperblock=512
blockspergrid = N // threadsperblock + 1

add[blockspergrid,threadsperblock](
    device_A,
    device_B,
    device_C,
)
cuda.synchronize()

C = device_C.copy_to_host()

if all(C == 2):
    print("1+1=2. GPUs aren't so scary!")
else:
    print("Oh, no. 1+1!=2. GPUs are terrifying...")
```

`example_reduce.py`

- Be careful: if 2 threads write to the same memory, you will get the wrong result
- "Reduce" is a fancy word for "adding many numbers together"
- Create array `A=[1, 2, 3, ..., N]`
- Compute sum(A) for N = 100,000

$$\sum_{i=1}^{N} i = \frac{1}{2} N(N+1) \quad = 5\,000\,050\,000$$

- Thread `i` is trying to add `A[i]` to `result[0]` and write the new value to `result[0]`
- Thread `i+1` will add to `result[0]` before thread i has finished!

```
@cuda.jit
def wrong_reduce(A, result):
    i = cuda.grid(1)
    if i < A.size:
        result[0] += A[i]
```

```
result[0] = 324,224
```

If you run `example_reduce.py`, you will get a different number than this. In fact, the result will be somewhat random.

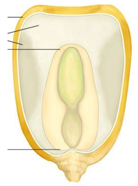
```
@cuda.jit
def correct_reduce(A, result):
    i = cuda.grid(1)
    if i < A.size:
        cuda.atomic.add(result, 0, A[i])
```

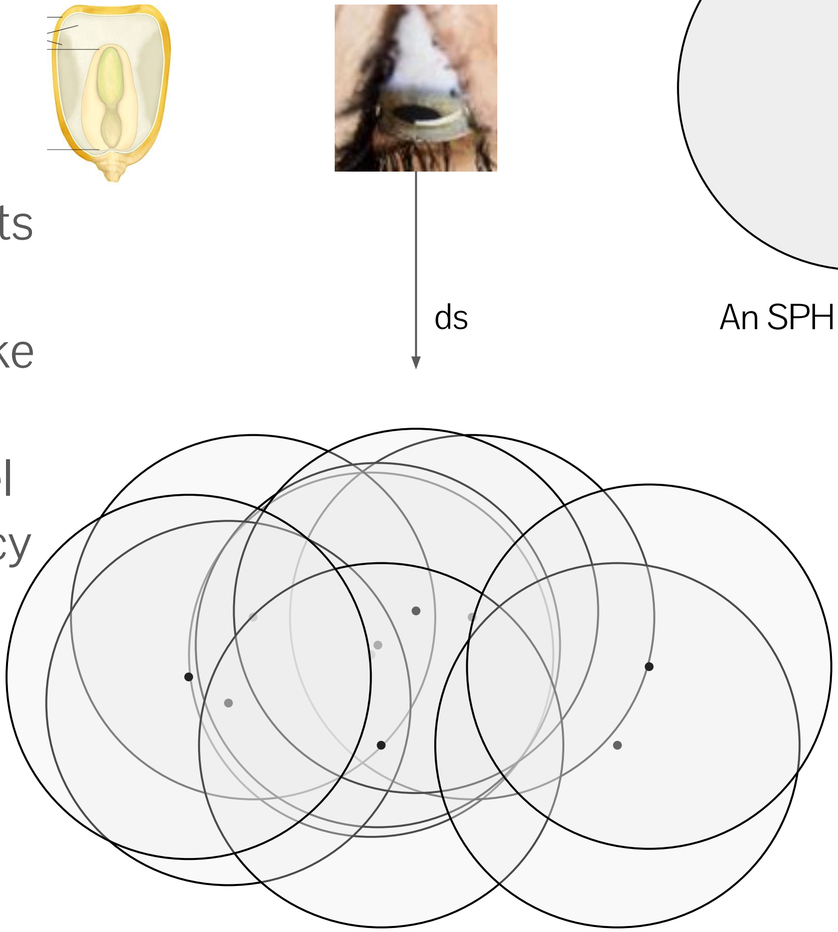
```
result[0] = 5000050000
```

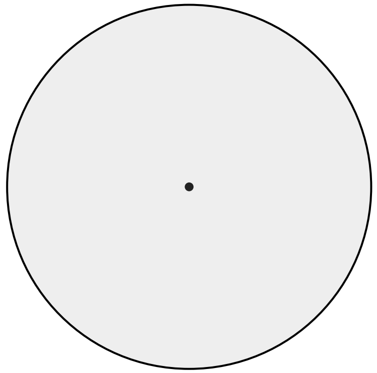
This will always be the value for `result[0]`.

Computing SPH Column Density using the GPU

- SPH particles have “kernels”
  - Represent the simulated fluid
- A corn kernel contains an embryo:
- When heated, the outer shell bursts from gas pressure and the endosperm foams outward to make popcorn
- In SPH, we can also change kernel sizes to ensure simulation accuracy
- Density > 0 inside a kernel
- Column density: $\Sigma \equiv \int \rho \, \mathrm{d}s$
- Compute by making a popcorn pancake!

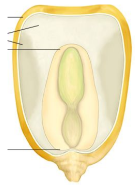


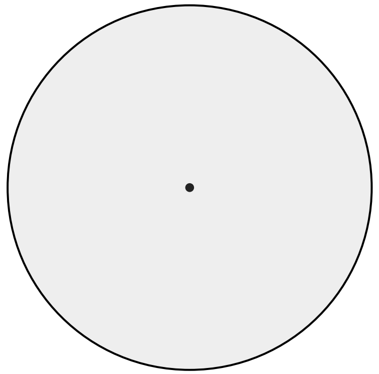


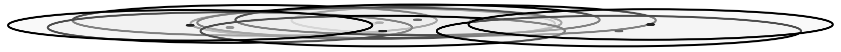

An SPH particle

Computing SPH Column Density using the GPU

- SPH particles have “kernels”
  - Represent the simulated fluid
- A corn kernel contains an embryo:
- When heated, the outer shell bursts from gas pressure and the endosperm foams outward to make popcorn
- In SPH, we can also change kernel sizes to ensure simulation accuracy
- Density > 0 inside a kernel
- Column density: $\Sigma \equiv \int \rho \, \mathrm{d}s$
- Compute by making a popcorn pancake!




An SPH particle



CPU vs GPU

- For 300,000 particles, compute $\Sigma \equiv \int \rho \, \mathrm{d}s$
  - For every pixel on the screen
- On CPU, it takes `449.307333` seconds
- On GPU, it takes `1.023861` seconds
- **439x speedup!**
- The image shows $\log_{10} \Sigma$ with 1329x1321 pixels
  - Simulation is a stellar merger, and we are viewing it from top-down some time after the plunge-in phase