FEBRUARY 9

SPRING 2025

SUPERCOMPUTING @ GEORGIA TECH

Attendance

Club Linktree

WORKSHOP 1

FALL 2024

Club Linktree

SUPERCOMPUTING
@ GEORGIA TECH

GPU & CUDA FUNDAMENTALS: WORKSHOP 1

Our mission is to increase accessibility and awareness to supercomputing (aka **HPC,** high-performance computing)**.**

We’re here to..

Expose GT expertise

GT has a very strong presence in the supercomputing research community and industry!

Increase accessibility

We want to leverage these resources and foster a welcoming community for you to learn more!

Exposure

Supercomputing @ GT
Faculty Panel

Industry & National Lab
Guest Speakers

Discussions of Papers
or News Articles

Understand

~~Project track~~ WORKSHOPS!

Tech talks

Contextual talks

How to stay involved?

Linktree

No dues.

Just attend meetings & events regularly.

Plug in

Fill out the Interest Form & stay in touch with our platforms: Discord, Instagram, Email Newsletter, LinkedIn Page.

Overview

- **GPU Overview** Page 11
- **Setup** Page 18
- **CUDA Programming** Page 32
- **Exercise 1: Vector Add** Page 42
- **Exercise 2: Matrix Multiply** Page 50
- **Exercise 3: Complex Function Visualization** Page 58

GPU Programming:

What? Why? How?

In this section…

- What are GPUs, and what is CUDA?
- Why should I care?
- Technical Requirements

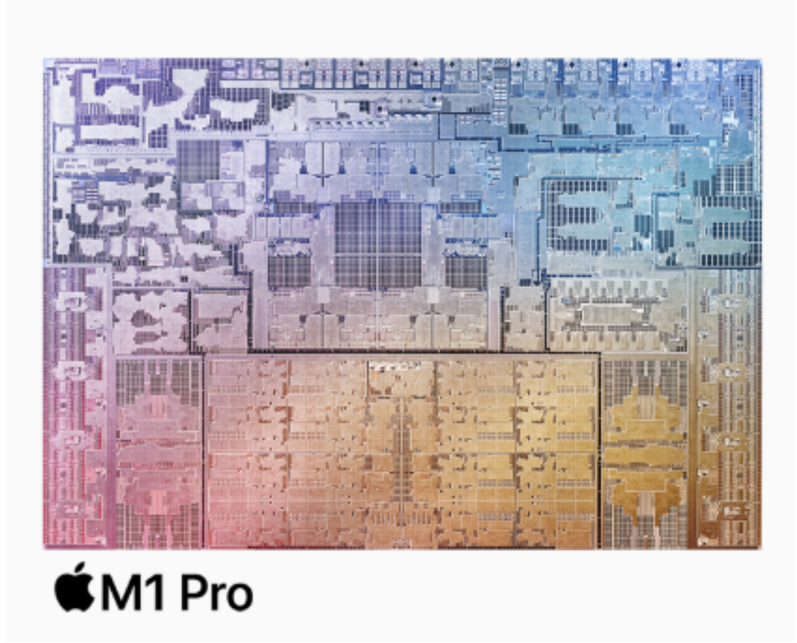
Please feel free to:

- Ask questions
- Try out commands and anything in the repository
- Grab some food!

What are GPUs?

Consumer CPUs

- What kind of tasks do our personal CPUs perform?
- What does a CPU look like?
  - 10 cores
    - ALUs
    - Control Units
  - Many layers of memory
    - Cache
    - DRAM



Apple M1 Pro - my own computer's CPU!

CPUs are good!

- CPUs are **pretty good** at **general-purpose** computing!
- They are…
  - **Fast***
  - Relatively **efficient***
  - Generally **affordable***
- Key: **General-Purpose**
- Is this always enough?

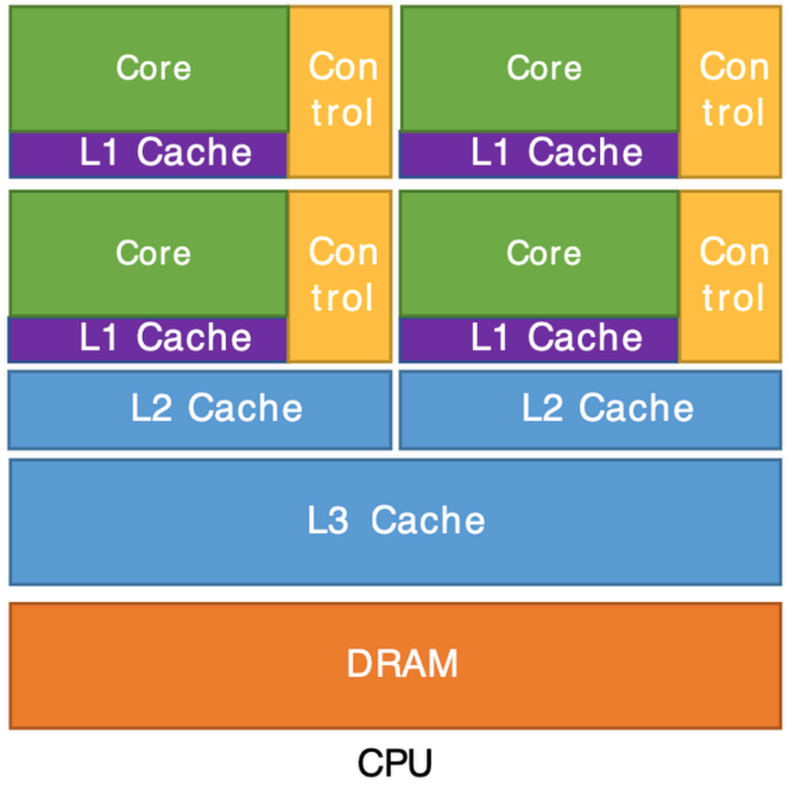


Diagram of Basic CPU Architecture

When do CPUs fail?

- Some examples include…
  - AI/LLM Training
  - Climate Modeling
  - Large-scale star simulations
- Why can’t we use CPUs?
  - **Serial workloads** are optimized on CPUs
  - **Parallel Workloads…?**

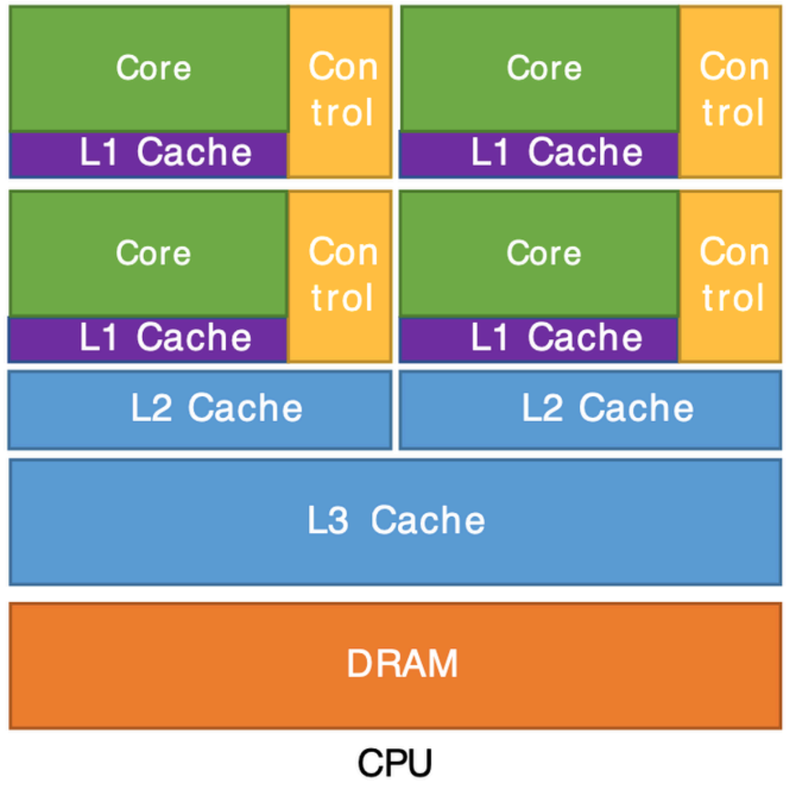


Diagram of Basic CPU Architecture

Enter Graphics Processing Units!

- Pioneered to accelerate computer graphics & image processing
- Found to be extremely useful for any **parallel workloads** of (mostly) repeated computations
- **Single Instruction Multiple Data**
  - **SIMD,** type of parallel arch.
  - Widely used in computing

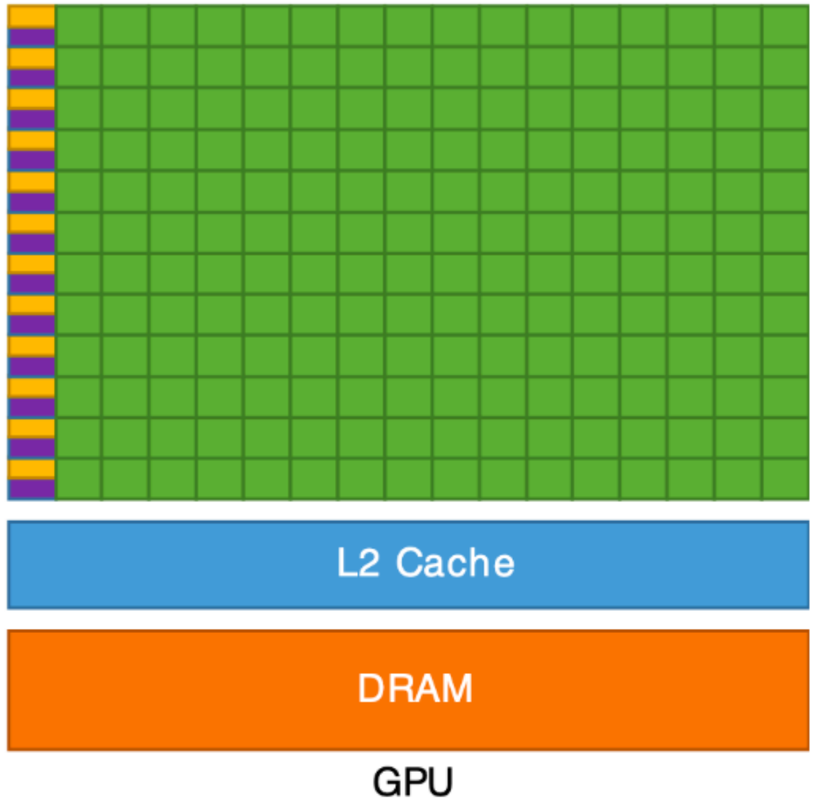
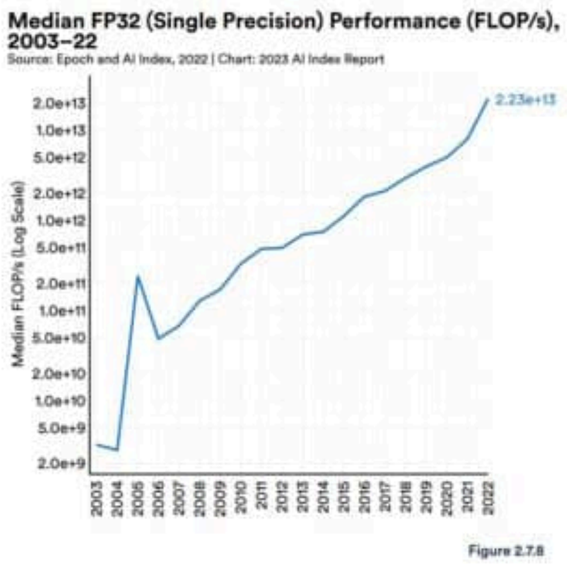
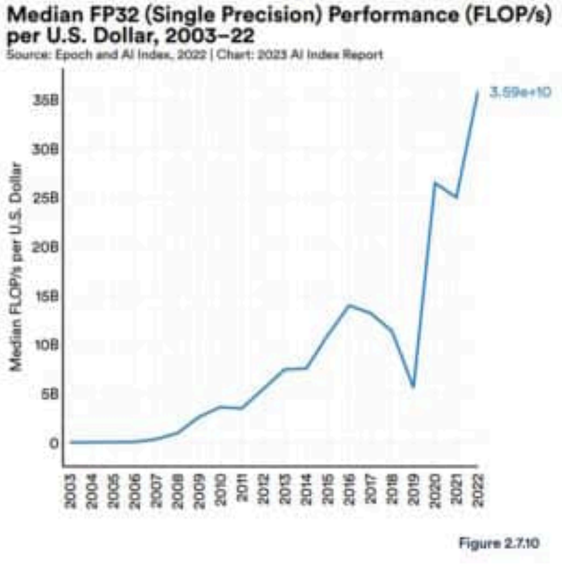


Diagram of Basic GPU Architecture

How *much* better are GPUs?

- It’s not even ***close.*** Why?
- **Extremely optimized** for math-heavy tasks
  - Matrices & vectors
- **Designed** for parallel computing loads
  - CPU architecture is designed for **sequential** loads





GPUs have drastically increased in both performance & value over the past ~15 years

How can we interact with GPUs?

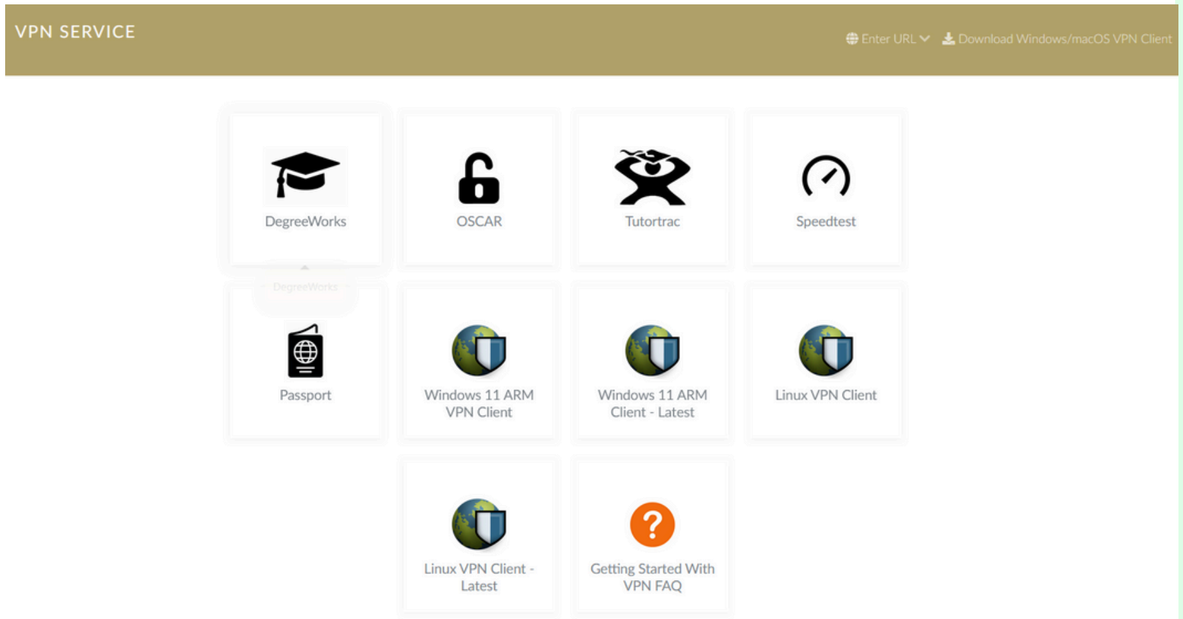
NVIDIA H100s in a computing node

- Take advantage of parallelism!
  - **Parallelize** GPU workloads
  - Use **CUDA** to write fast parallel GPU code
- What is **CUDA?**
  - Parallel computing API
  - Specify **thread-level** parallelism for GPUs
  - **Taught today : D**

Technical Requirements

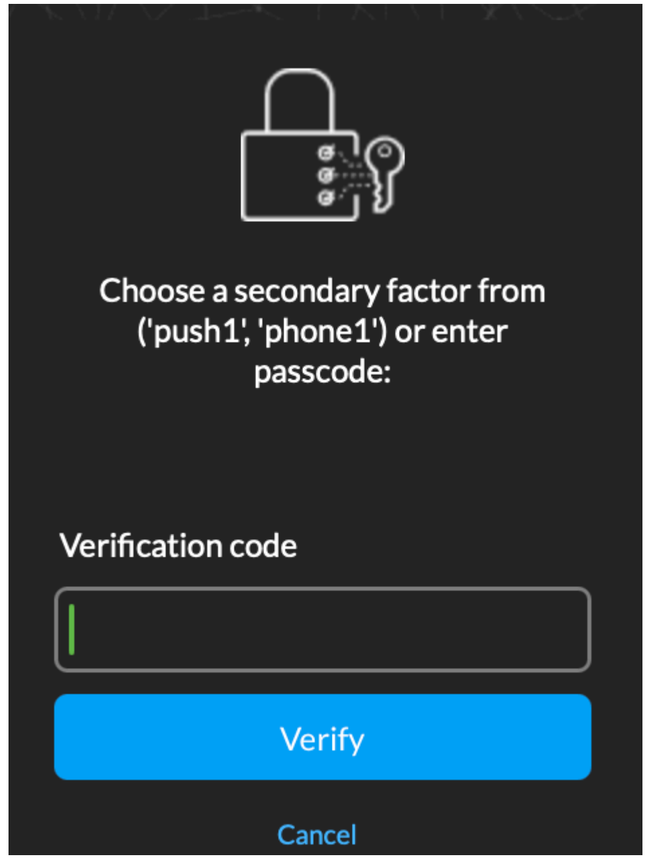
GT VPN Service

- Visit **vpn.gatech.edu**.
- Sign-in using your Georgia Tech credentials.
- Download the VPN client for your system.
- Make sure that you're connected to GT Wifi!



GT VPN Service

1. Once installed, open the GlobalProtect client. You will be prompted for the Portal. Type: vpn.gatech.edu
2. You will be prompted for a username and password. Enter your GT username (your GT email **without the @gatech.edu part**) and your password.
3. Once you press Connect, you will be prompted to verify with a secondary factor. Enter push1 into the Verification code box and open your Duo Mobile app to verify your login :)



IDEs

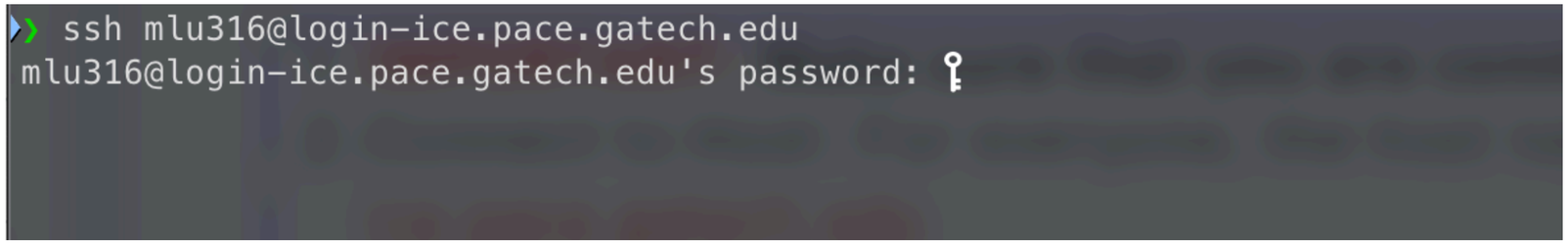
https://code.visualstudio.com

https://www.jetbrains.com/products/compare/?product=pycharm&product=pycharm-ce

Honorable mentions for the freaks :)
We might not be able to help if you are using these
(only use these if u know how to save and close ur files)

SSH Instructions

1. **IMPORTANT: Make sure that you are connected to the GT VPN!**
2. Connect to Host: For everyone, the host name is <gt-username>@login-ice.pace.gatech.edu
3. Type in your GT password upon seeing the prompt. **DO NOT BE ALARMED IF NOTHING SHOWS UP AS YOU TYPE** (its for safety etc.) Just finish typing your password and press <Enter>.



https://gatech.service-now.com/home?id=kb_article_view&sysparm_article=KB0042100

Remote-SSH w/ Terminal

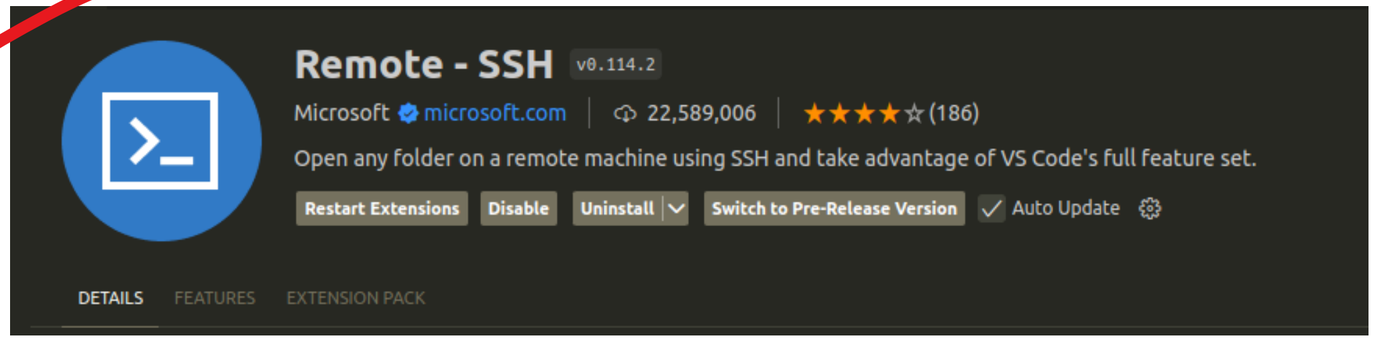
- Please check that you have access to an **SSH-Capable Terminal!**
  - **Windows Powershell**
  - **MacOS Terminal**
  - **Linux Terminal**
- We'll use **SSH** to log onto the cluster you'll be using today.
- Follow the previous instructions to login to the PACE cluster. If you've successfully logged in, you'll see a prompt like the following:

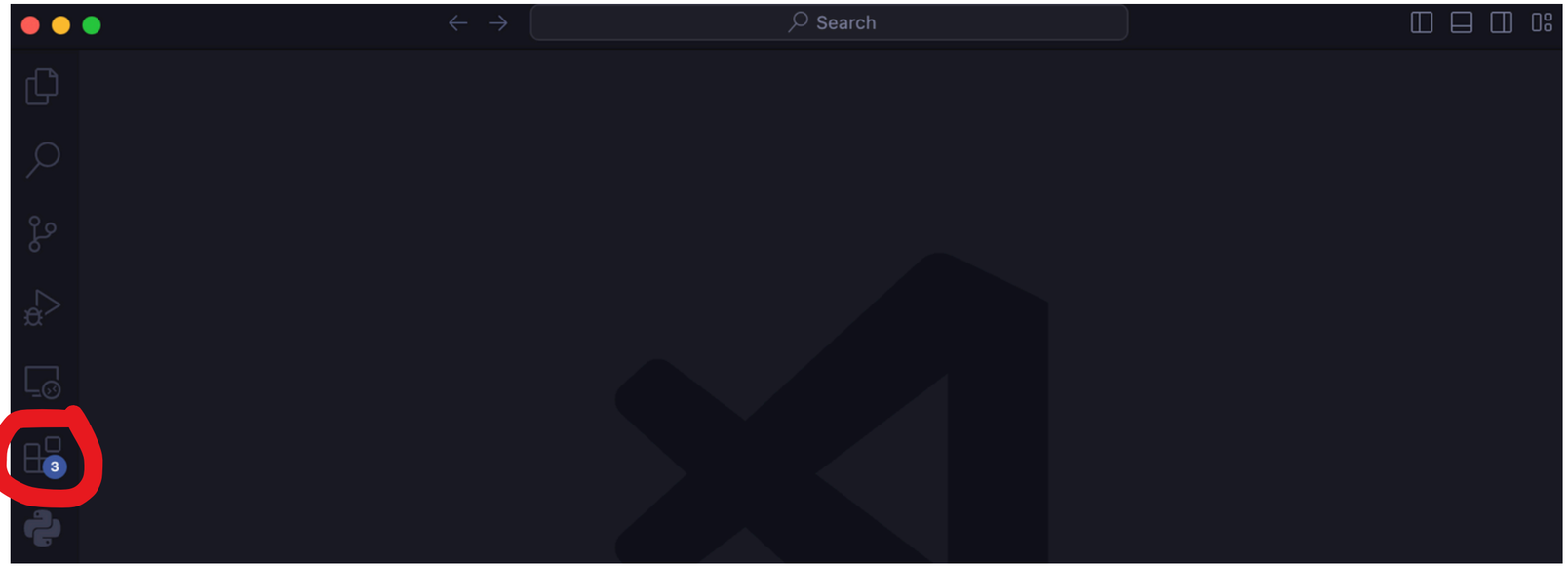
```
[asaha92@login-ice-1 ~]$
```

- You are now in a **login node** of the cluster, which you can use to access files, write code, submit jobs, etc.

Remote-SSH w/ VSCode

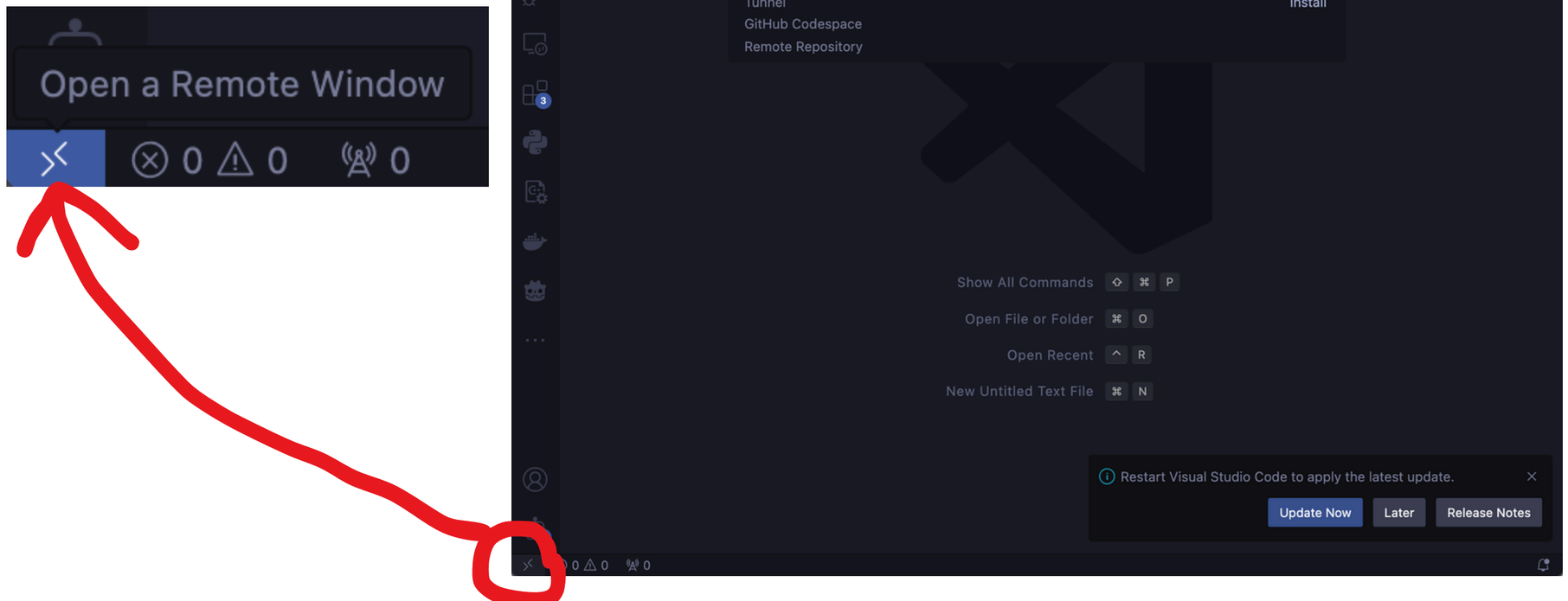
1. Install the extension from the VSCode extension store.





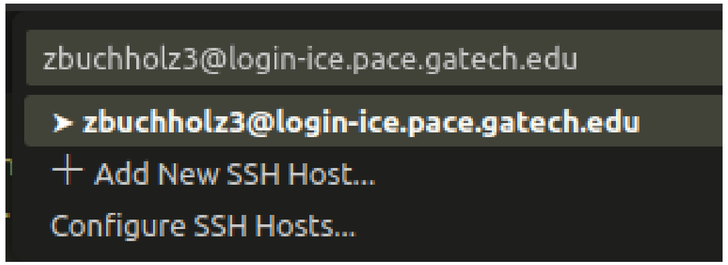
Remote-SSH w/ VSCode

2. Notice that at the bottom left of the application, you will now see a button which appears to show a broken link. Click on it.

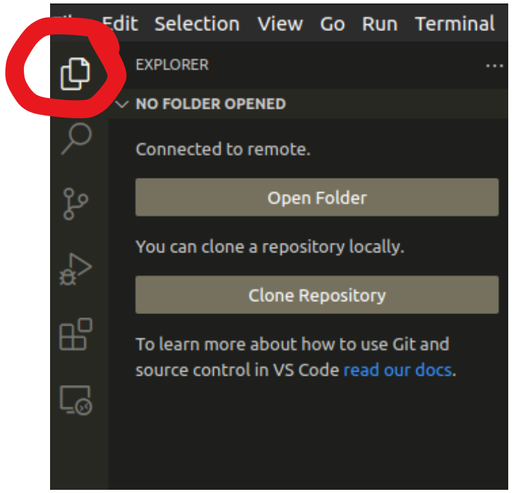


Remote-SSH w/ VSCode

3. Select **Connect-to-Host** (should be the first option) and enter the hostname and password as shown previously.



4. You should now be able to open the folders icon to see the contents of the remote.



Accessing GPU through the Cluster

Try this command to **make sure** that you can use a **GPU** node,

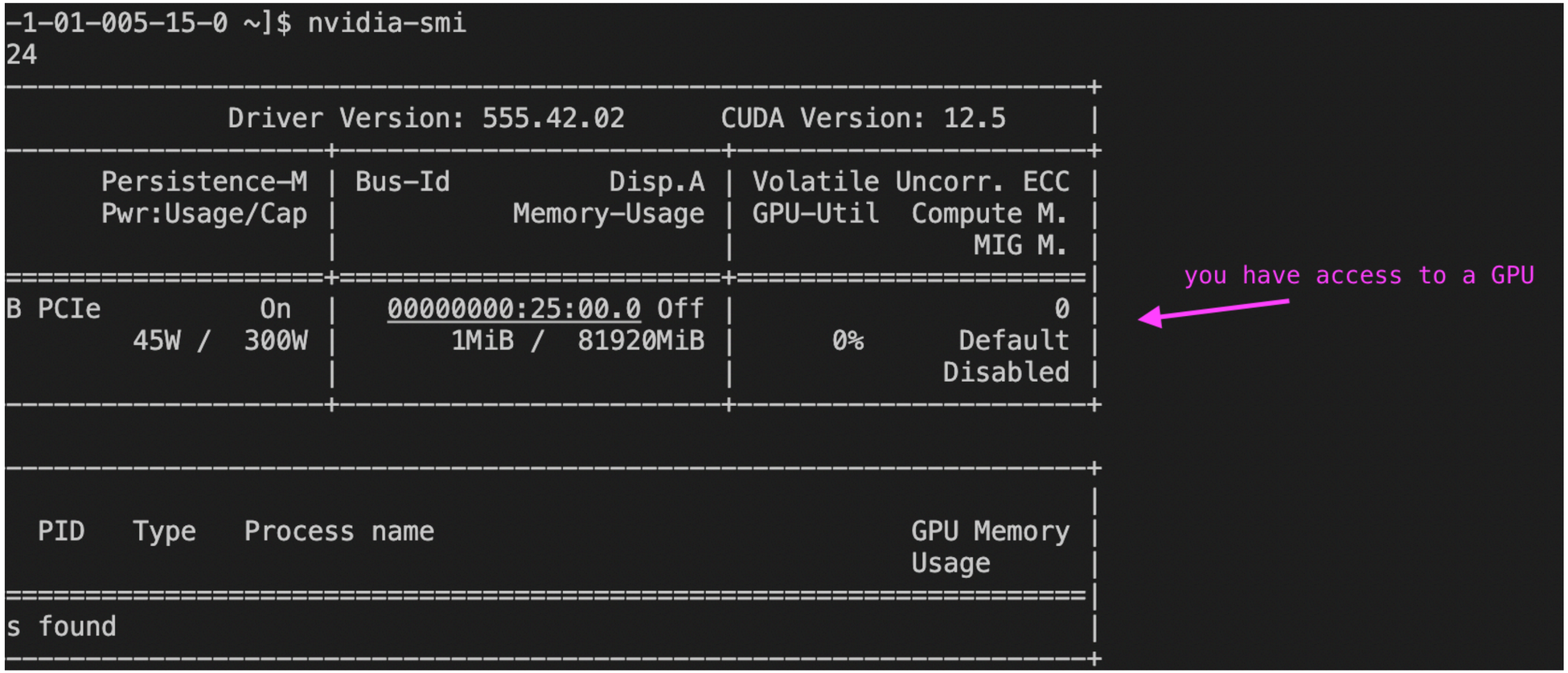
```
salloc --nodes=1 --ntasks-per-node=1 -G1 --time=0:15:00
```

If you're successful, the following should print.

```
salloc: Pending job allocation 570744
salloc: job 570744 queued and waiting for resources
salloc: job 570744 has been allocated resources
salloc: Granted job allocation 570744
salloc: Waiting for resource configuration
salloc: Nodes atl1-1-02-009-32-0 are ready for job
----------------------------------------
Begin Slurm Prolog: Apr-07-2024 11:23:21
Job ID:    570744
User ID:   asaha92
Account:   coc
Job name:  interactive
Partition: pace-gpu,ice-gpu
----------------------------------------
```

- Make sure to run the **exit** command after you finish to end your GPU node session

Accessing GPU through the Cluster



Lunch Break!

Make sure that you can access the cluster and that you can install dependencies.

Access Repository for Exercises

- **Email:** the link to the exercise repository can be found in the confirmation email.
- **Discord:** alternatively, the link can be found in the **“#sp25-workshop1”** channel

GPU Programming:

A gentle introduction with CUDA

Background

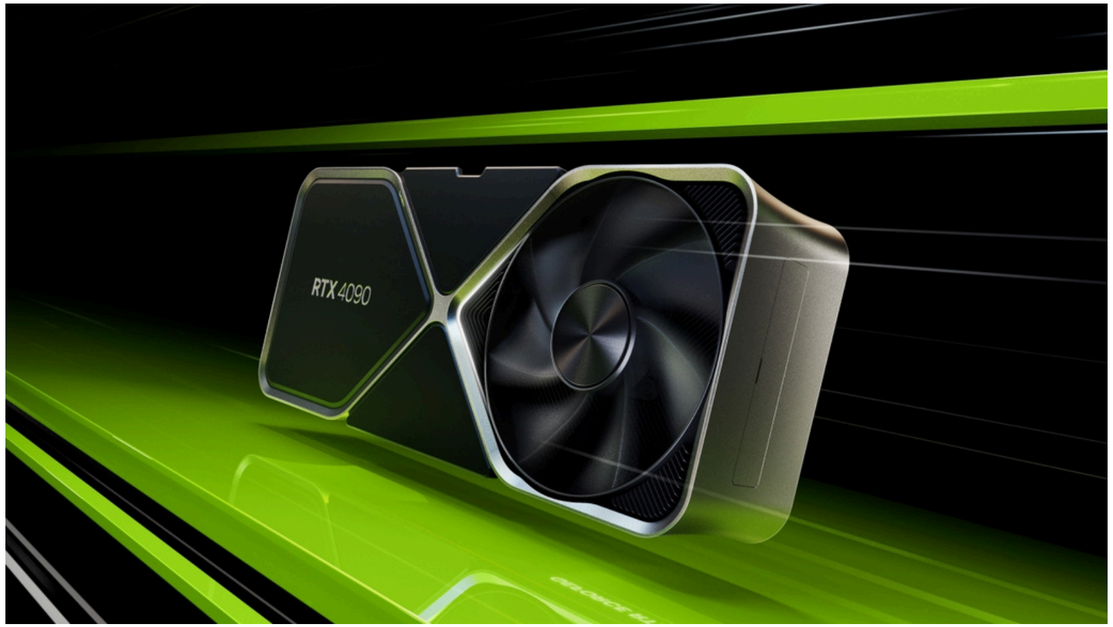
- Who are you?
  - You’ve heard of CUDA, maybe haven’t used it before
  - Experience programming in Python
  - Experience using basic Linux commands
  - **How can you leverage GPUs to accelerate your computing?**

Flynn’s Taxonomy

NUMBER OF INSTRUCTION STREAMS
VS NUMBER OF DATA STREAMS

SISD

MIMD



SIMD

https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/

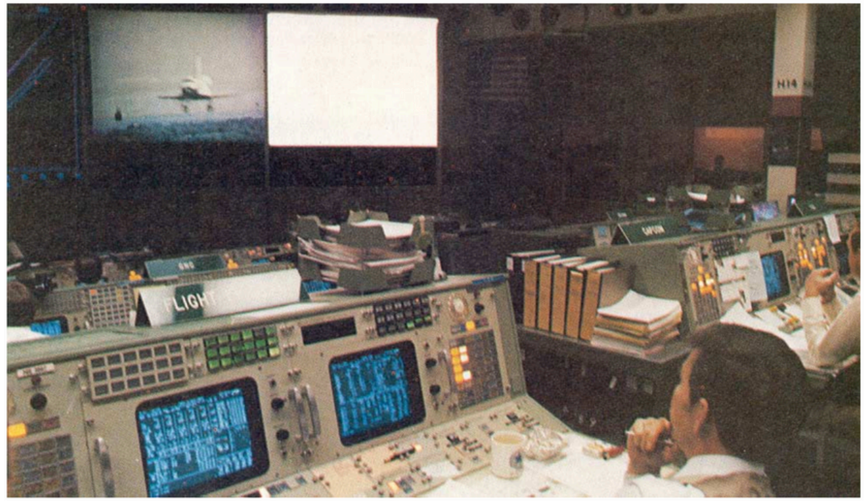


FIGURE 1. The Control Room at Mission Control in Houston.

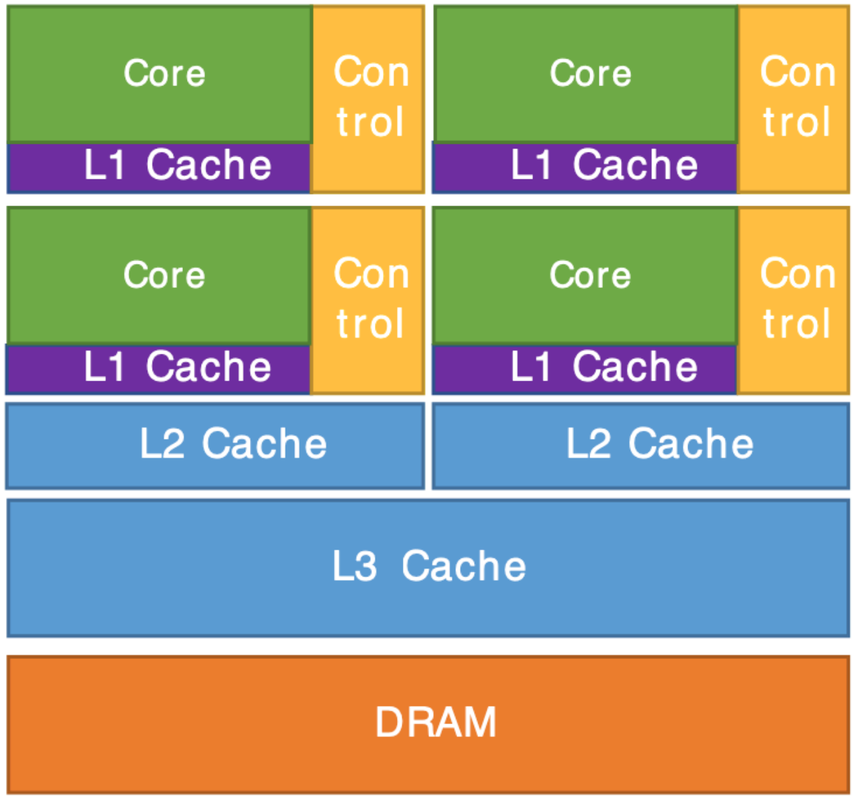
MISD

https://dl.acm.org/doi/10.1145/358234.358246

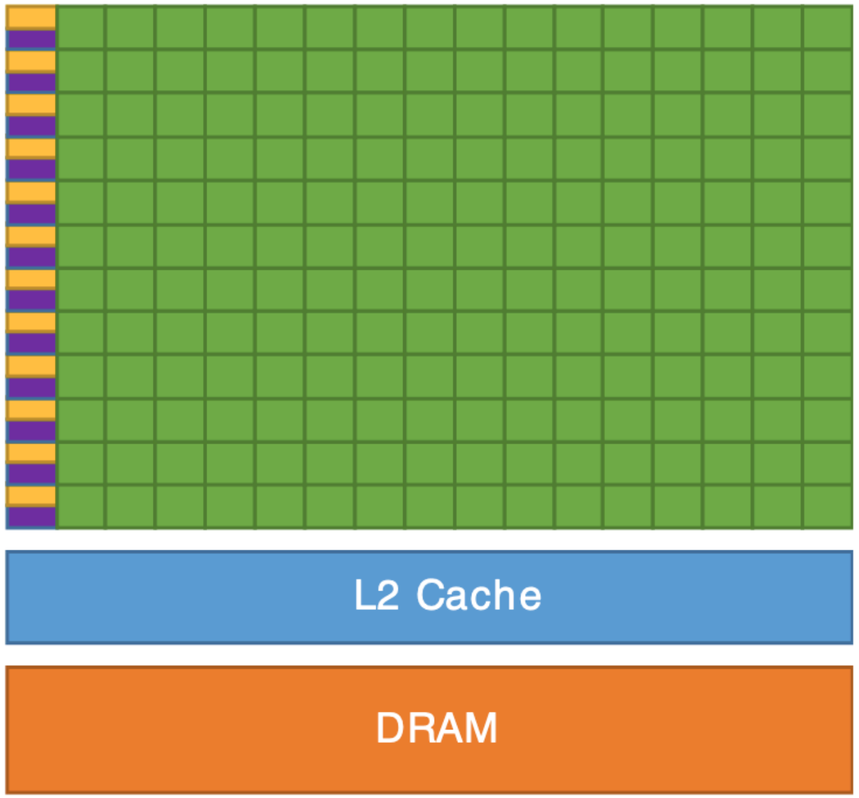
Single Instruction Multiple Data

- Known as **SIMD** model of parallel processing
- Essentially, **multiple** data streams are given a **single instruction**
  - Can be performed **separately**
- **What is an instruction?**

GPU vs. CPU

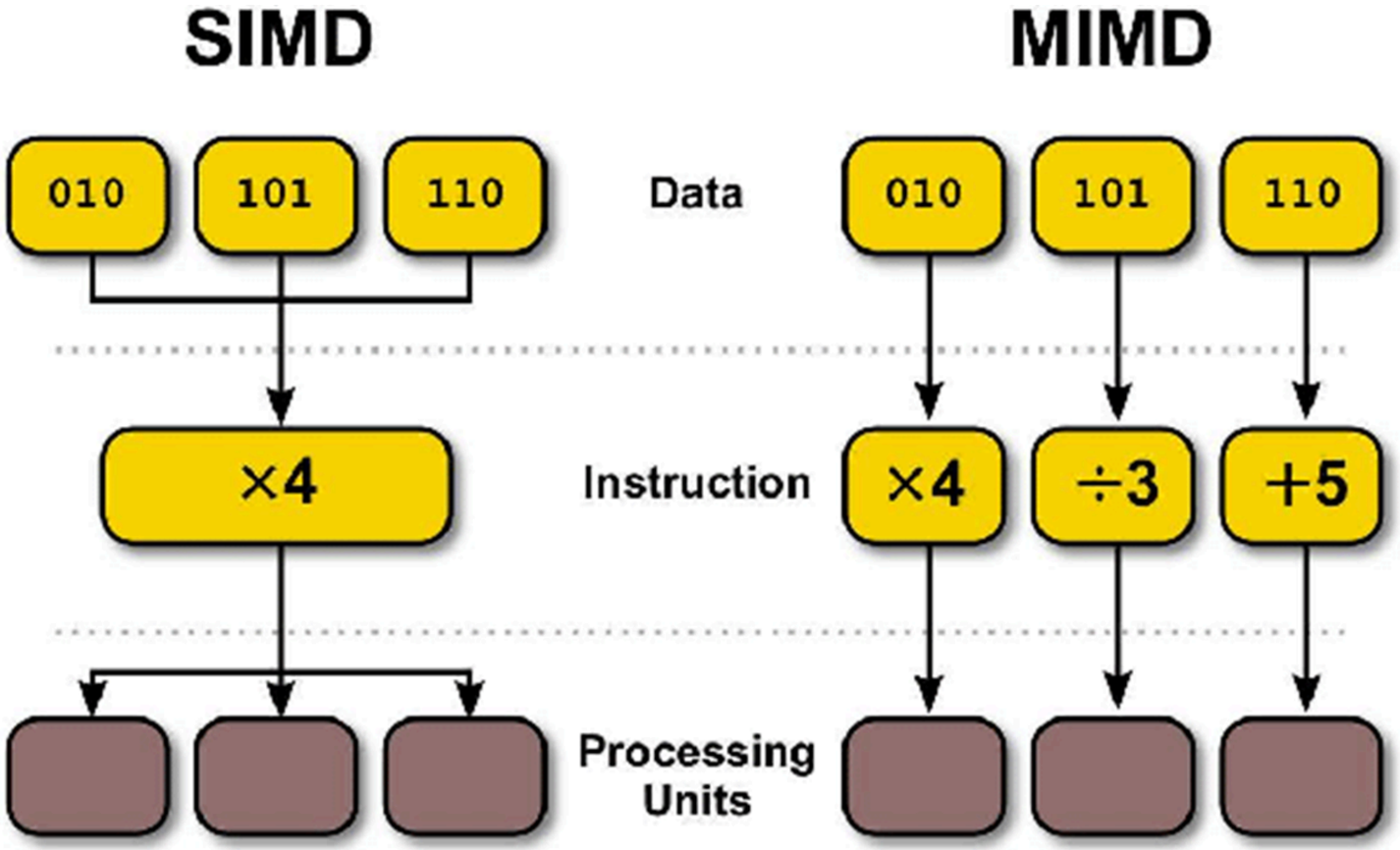


CPU

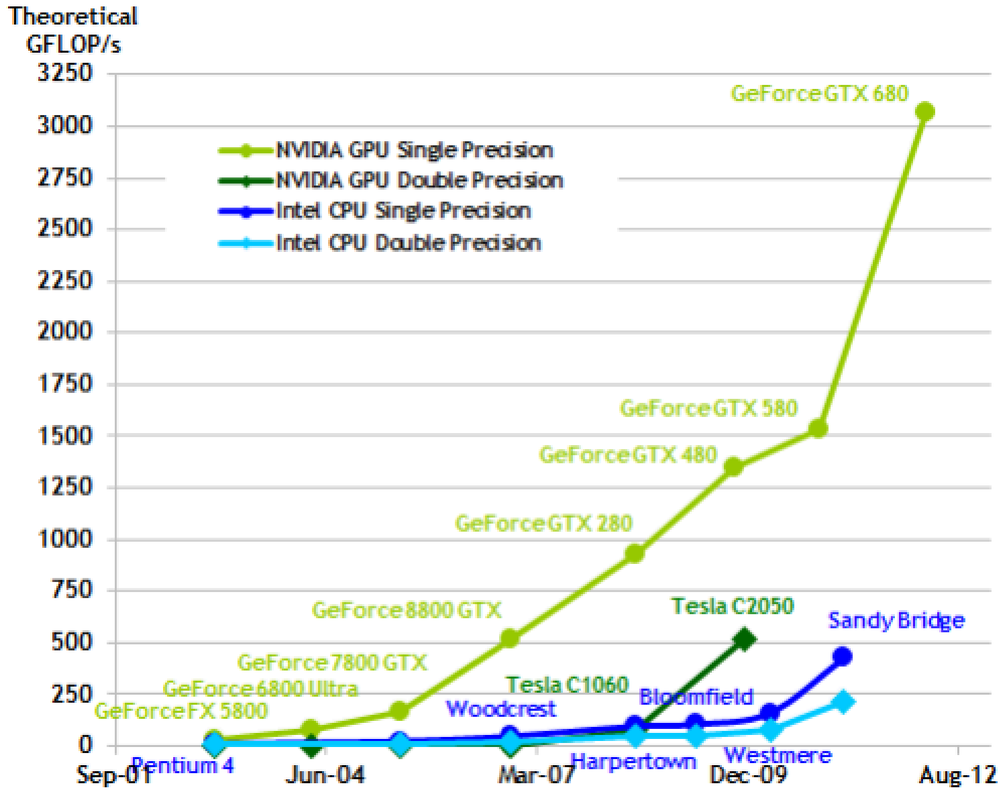


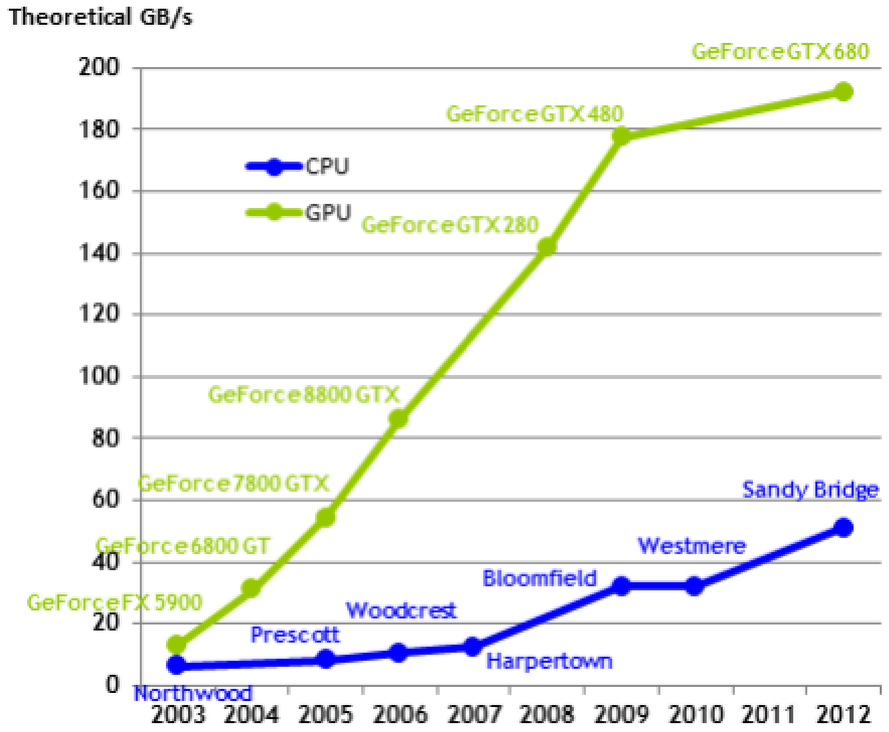
GPU

Parallelism in SIMD



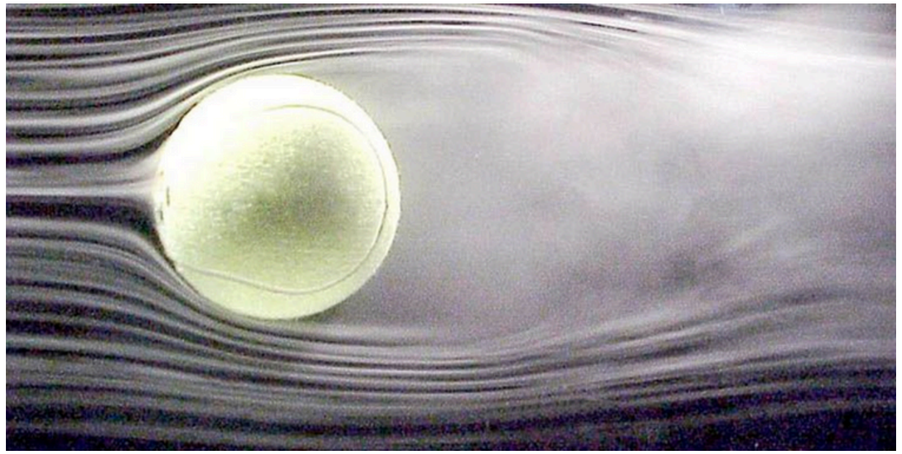
History of GPU to GPGPU



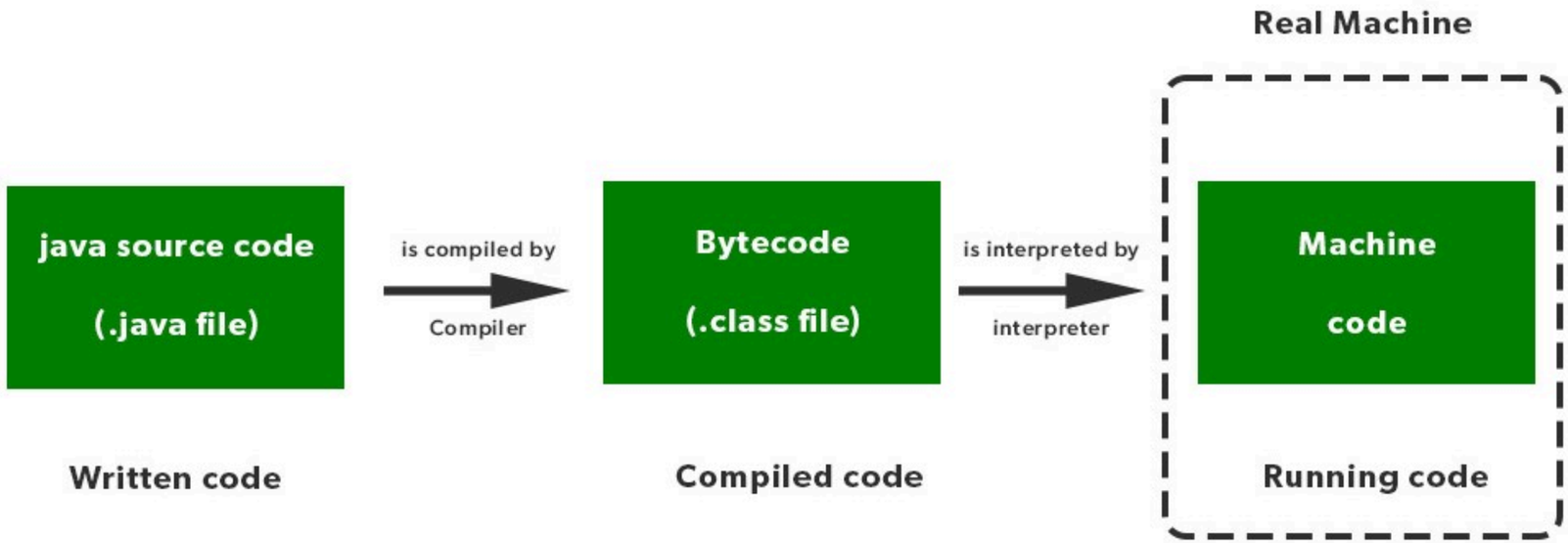


What is CUDA?

- An Application Programming Interface (API) for GPUs
- **Allows users to leverage GPUs for wide variety of computing tasks**, not just computer graphics

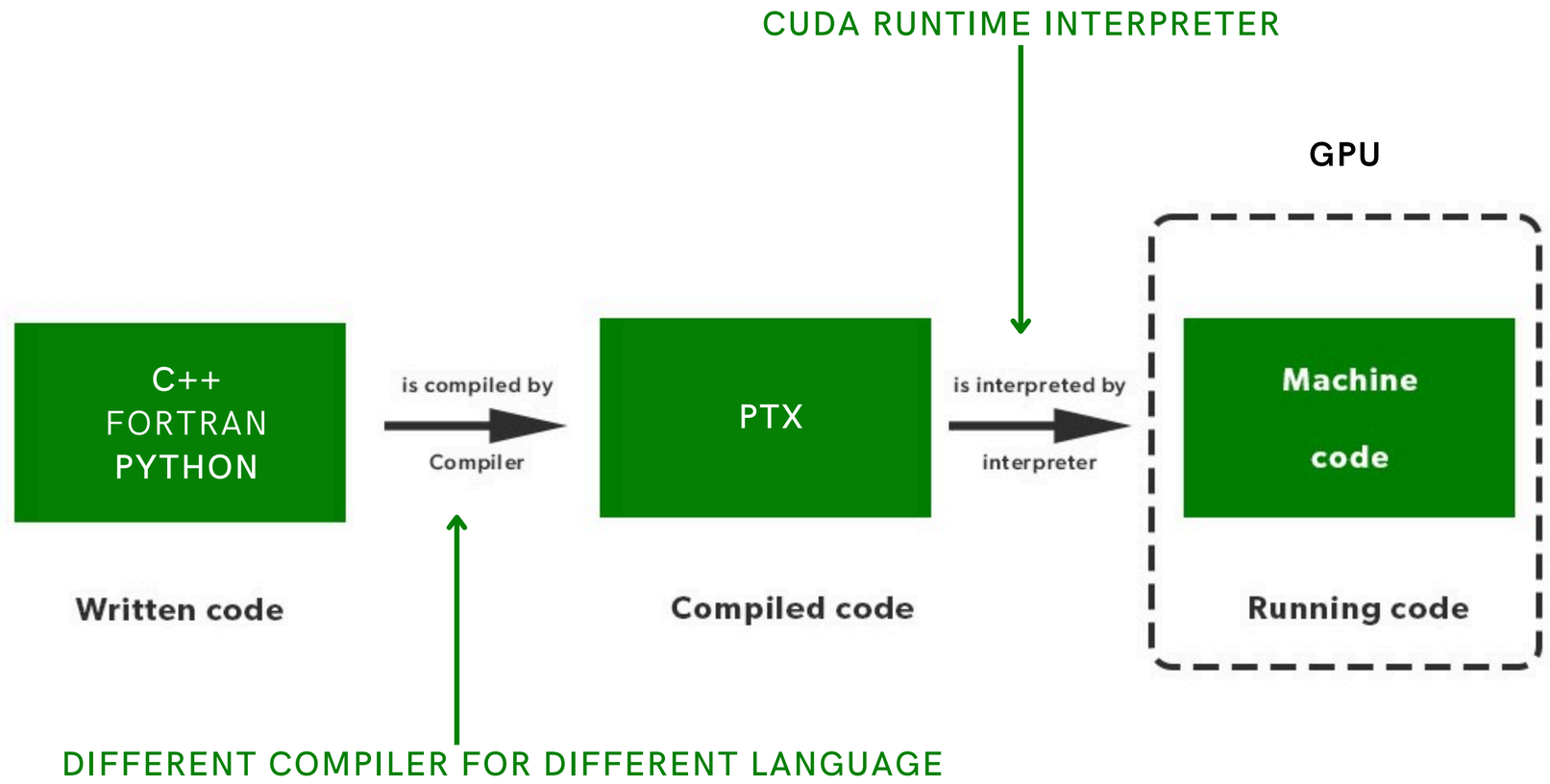


The language the GPU uses



Remember learning how Java is compiled in 1331?

The language the GPU uses



CUDA is analogous to this!

Part I

Vector Add

Access the Part I Exercise Folder:

- Navigate to the **Part I: Vector Add** folder in the workshop's Git page

- Folder organization:
  - **problems.md:** Description of the tasks
  - **setup.md:** Reminder of setup instructions
  - **Question folders:**
    - q[n]_file.py: The file you edit
    - q[n]_sol.py: Solution file
    - lib.py: Our tester (do not edit)

Part I: Low-Level Python (3 min.)

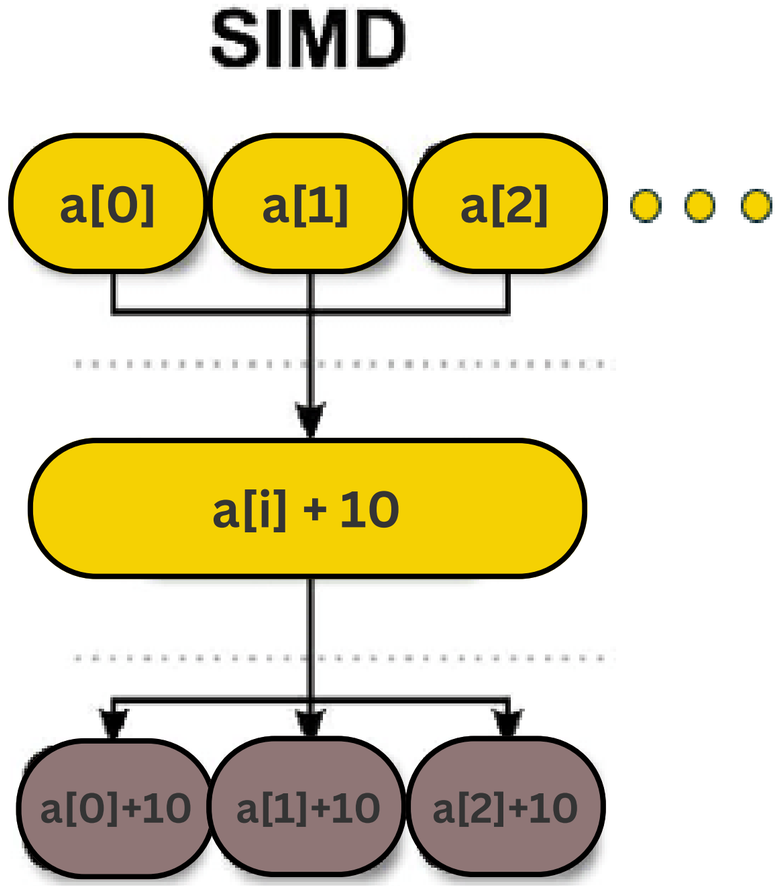
- Work on questions **Q1 & Q2**
  - You don’t have to worry about **threads;**
  - **N(threads) = N(positions)**
- **Q1:** Implement a kernel that adds 10 to each position of vector **a** and stores it in vector **out**. You have 1 thread per position.
- **Q2:** Implement a kernel that adds together each position of **a** and **b** and stores it in **out**. You have 1 thread per position.

Question 1

- Given a vector **a**, write a gpu program that will add 10 to each element in parallel.

[5, 2, 11, 15, 6]

[15, 12, 21, 25, 16]



Part 1: Solutions

- Write out inner logic **without** using high-level implementations

```python
# CUDA kernel for vector addition
@cuda.jit
def gpu_vector_add(a, b, c):
    idx = cuda.grid(1)
    if idx < a.size:
        c[idx] = a[idx] + 10
```

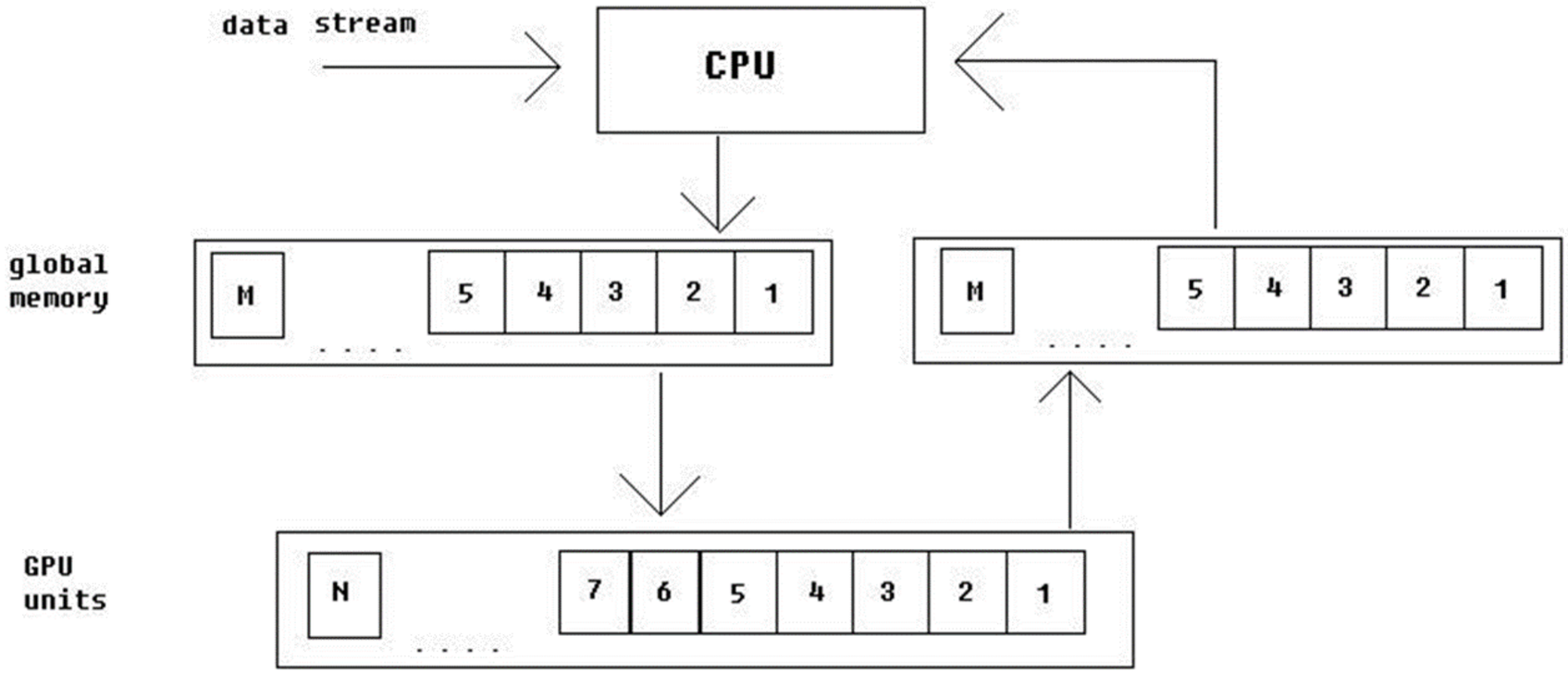
```python
# CUDA kernel for vector addition
@cuda.jit
def gpu_vector_add(a, b, c):
    idx = cuda.grid(1)
    if idx < a.size:
        c[idx] = a[idx] + b[idx]
```

Part 1: Solutions

- Kernel Launch

```
# Allocate memory on the GPU
a_gpu = cuda.to_device(a)
b_gpu = cuda.to_device(b)
c_gpu = cuda.device_array(N)

# Time the GPU vector addition
threads_per_block = 1024
blocks_per_grid = (N + (threads_per_block - 1)) // threads_per_block

start = time.time()
gpu_vector_add[blocks_per_grid, threads_per_block](a_gpu, b_gpu, c_gpu)
cuda.synchronize()
gpu_time = time.time() - start

# Copy result back to the host
c_result = c_gpu.copy_to_host()
```

Device <-> Host Data Transfers



Part 1: Solutions

- Device host data transfer

```
# Allocate memory on the GPU
a_gpu = cuda.to_device(a)
b_gpu = cuda.to_device(b)
c_gpu = cuda.device_array(N)
```

```
# Time the GPU vector addition
threads_per_block = 1024
blocks_per_grid = (N + (threads_per_block - 1)) // threads_per_block

start = time.time()
gpu_vector_add[blocks_per_grid, threads_per_block](a_gpu, b_gpu, c_gpu)
cuda.synchronize()
gpu_time = time.time() - start

# Copy result back to the host
c_result = c_gpu.copy_to_host()
```

Part II

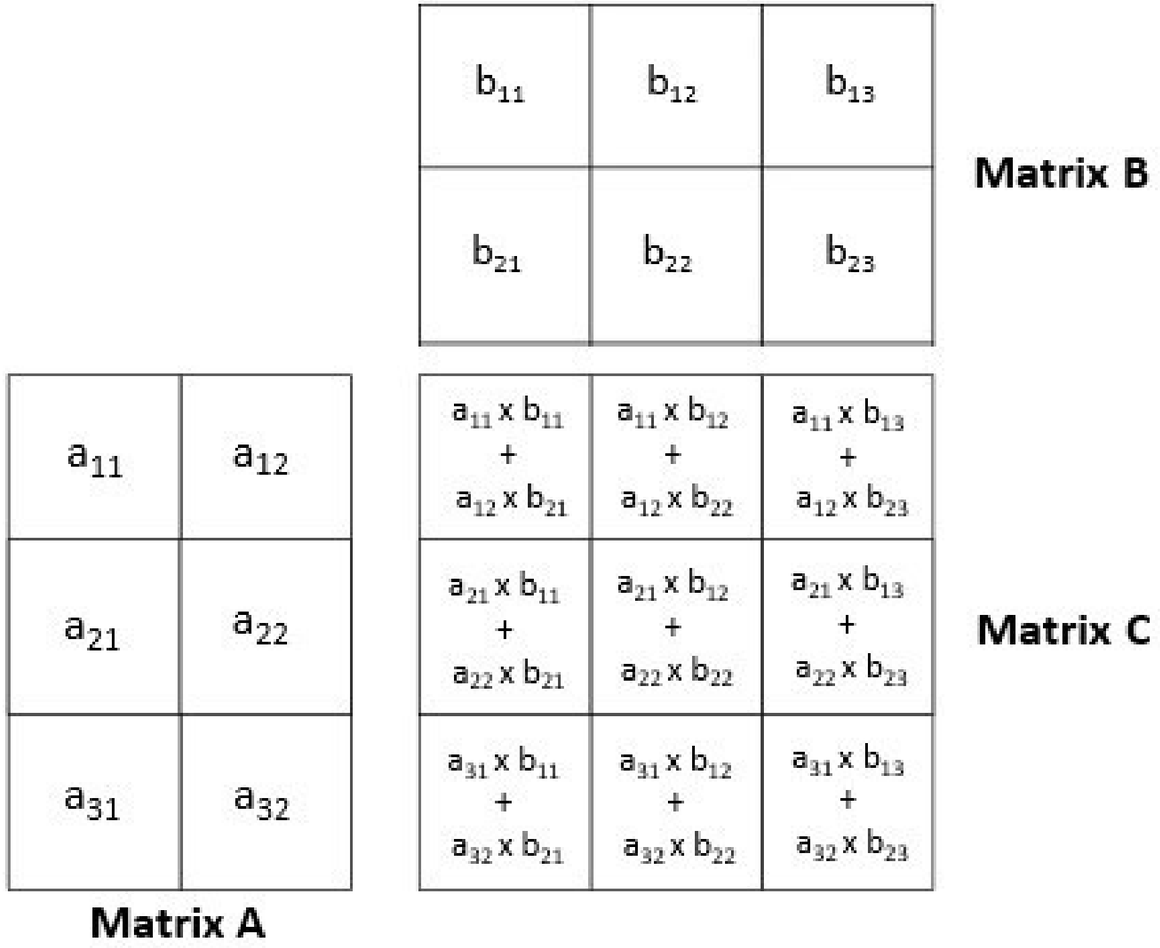
Matrix Multiply Shared Memory

Access the Part 2 Examples Folder:

- Navigate to the **Part II: Matrix Multiply** folder in the workshop's Git page

- Examples contained in this folder

Matrix Multiplication



Tips

Consider:

- What parts can be run in parallel?

- For a thread, how do we select the correct elements of A/B to multiply?

- How do we accumulate the results?

Part 2: Solutions

- Have each thread calculate one element of C

```
@cuda.jit
def matmul(A, B, C):
    """Perform square matrix multiplication of C = A * B
    """
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

Part 2: Solutions

- Create arrays in shared memory for faster access

```
sA = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
sB = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
```

- Get information on dimensions

```
x, y = cuda.grid(2)

tx = cuda.threadIdx.x
ty = cuda.threadIdx.y
bpg = cuda.gridDim.x    # blocks per grid

if x >= C.shape[0] and y >= C.shape[1]:
    # Quit if (x, y) is outside of valid C boundary
    return
```

Part 2: Solutions

- Load data into memory
  - In each iteration, a row from A multiplies with a column from B.
- Wait for all threads to preload

```
# Each thread computes one element in the result matrix.
# The dot product is chunked into dot products of TPB-long vectors.
tmp = 0.
for i in range(bpg):
    # Preload data into shared memory
    sA[tx, ty] = A[x, ty + i * TPB]
    sB[tx, ty] = B[tx + i * TPB, y]

    # Wait until all threads finish preloading
    cuda.syncthreads()
```

Part 2: Solutions

- Multiply using the shared memory

```
# Computes partial product on the shared memory
for j in range(TPB):
    tmp += sA[tx, j] * sB[j, ty]

# Wait until all threads finish computing
cuda.syncthreads()
```

- Place the result from this thread into the solution

```
C[x, y] = tmp
```

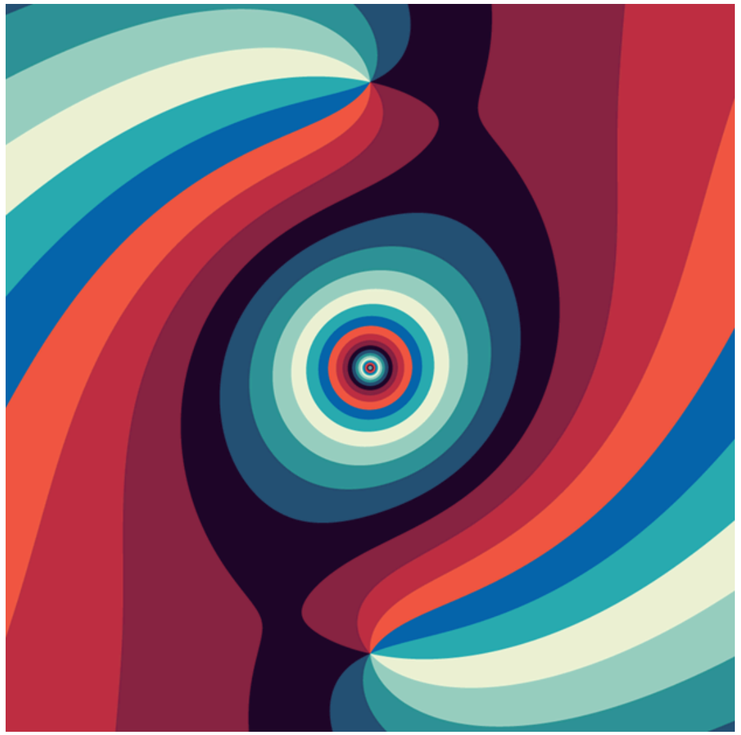
Part III

Complex Function Visualization

Part 3: Graphing in the XY Plane

- Navigate to **imagegen_example.py** file
- How can we compute **a function's values** using **Numba?**

$$f(z) = z \cdot (c_0)^i \cdot z^{(c_1)i}$$



Part 3: Graphing in the XY Plane

- Navigate to **1_example.py** file
- How can we compute **a function's values** using **Numba?**

```
def generate_image(output, color_map):
    """
    Populates the output array with an image representing the graph of a complex function
    """
    for x in range(128):
        for y in range (128):
            # Color the pixel with respect to the value of the complex function at the pixels coordinate
            z = complex(float(x) / output.shape[0] * 2.0 - 1.0, float(y) / output.shape[1] * 2.0 - 1.0)
            f_z = z ** 1.5 ** (complex(-1,-1)) if z != 0 else 0
            magnitude = abs(f_z)
            color_index = int(magnitude * 10 % len(color_map))
            output[x, y] = color_map[color_index]
```

Part 3: Graphing in the XY Plane

- Navigate to _ _ _ _ _ file
- How can we compute **a function's values** using **Numba?**

MAIN FUNCTION:

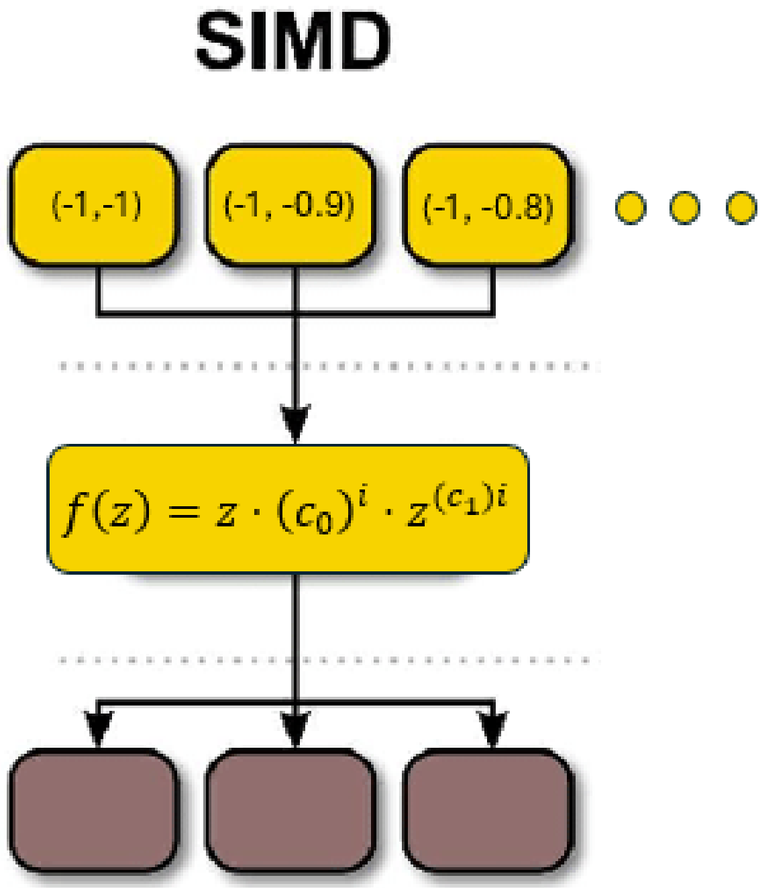
```
shape=(128, 128, 4)
image = np.zeros(shape, dtype=np.float32)

# Call the function
generate_image(image, colors)
```

```
output[x, y] = color_map[color_index]
```

Parallelism in Part 3

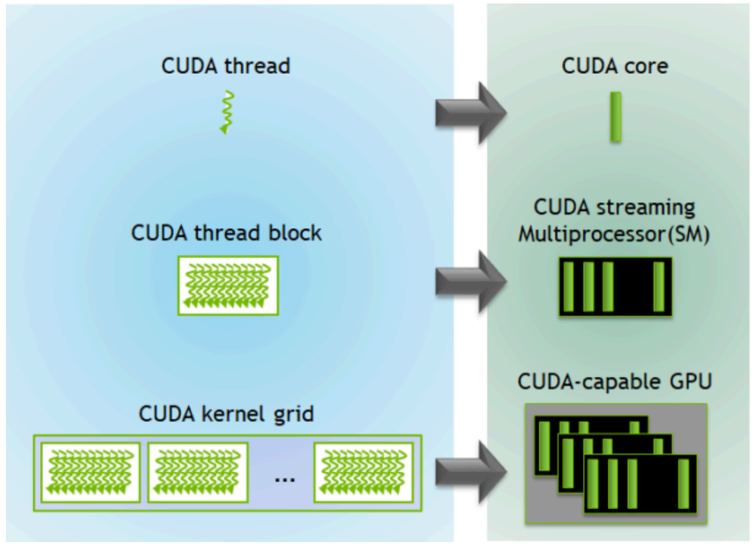
- How does **SIMD** help accelerate computing in **Part III?**

- Multiple **data streams** are given the same **instruction**
  - **Instruction** is given by ***f(z)***

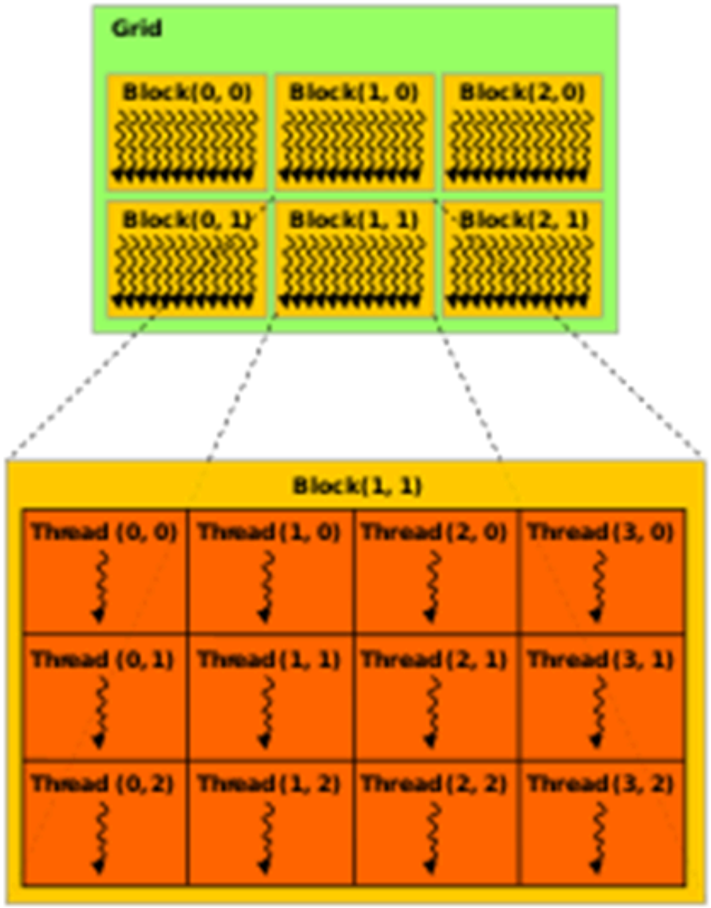


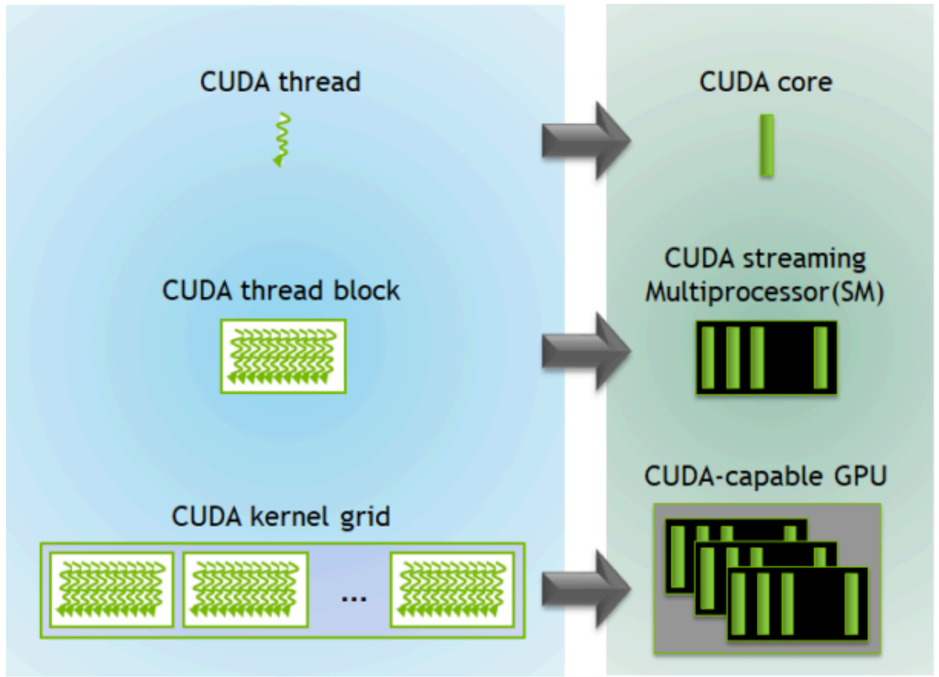
GPUs and CUDA: How?

- **CUDA** interacts with GPUs: **How?**
- **Thread**
  - A single process on a single CUDA core within the GPU
- **Thread Block**
  - We organize threads logically into groups called blocks
- **Kernel**
  - Functions executed **N** times in parallel by **N** different threads
  - *"Single Instruction"* on multiple datastreams



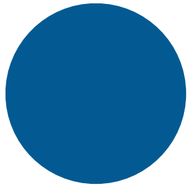
Grid Block Architecture





Concrete Example of SIMD

- Let’s introduce a concrete example of a program using SIMD!

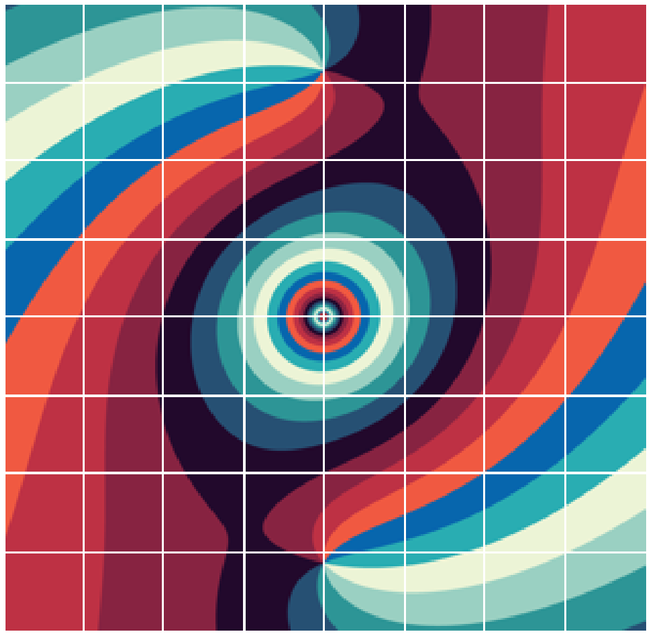


Goal: Manually write a parallel program to generate a (128 x 128) image using GPU-friendly code

- In doing so, we’ll achieve a speedup!
  - **13.0s -> 2.5s (CPU -> GPU)**

What does this look like?

- To do this, we'll break our image into a series of **blocks.**



We want to perform **computations** on each of the (16 x 16) tiles in the image **in parallel** to generate the full image

- In doing so, we'll achieve a speedup!
  - **13.0s -> 2.5s (CPU -> GPU)**

Generating a single tile

- Let’s write code to generate a **single (16 x 16) tile.**

```
def generate one tile(output, color_map):
```

- This is the **kernel function,** in terms of our previous architectural description for SIMD

Generating a single tile

- Let's write code to generate a **single (16 x 16) tile.**

MAIN FUNCTION:

8 x 8 instances of this program will be launched and run in parallel on the GPU!

```
shape=(128, 128, 4) # = 8*16, 8*16, 4
image_gpu = cuda.device_array(shape,dtype=np.float32)

blocks_per_grid = (1, 1)
threads_per_block = (8, 8)
generate_one_tile[blocks_per_grid, threads_per_block](image_gpu, colors_gpu)
```

- This is the **kernel function,** in terms of our previous architectural description for SIMD

Generating a single tile

```
def generate one tile(output, color_map):
```

- How can we **guarantee** that a **launched instance** knows **which part of the image (e.g. tile) it is generating?**

Generating a single tile

```
def generate one tile(output, color_map):
```

- How can we **guarantee** that a **launched instance** knows **which part of the image (e.g. tile) it is generating?**

```
tile_x, tile_y = cuda.grid(2)
```

- **We can use this to generate the appropriate pixel!**

```
for xi in range(16):
        for yi in range(16):
            x,y = tile_x*16 + xi, tile_y*16 + yi

            #code to generate the pixel
```

Generating a single tile

```
def generate one tile(output, color_map):
```

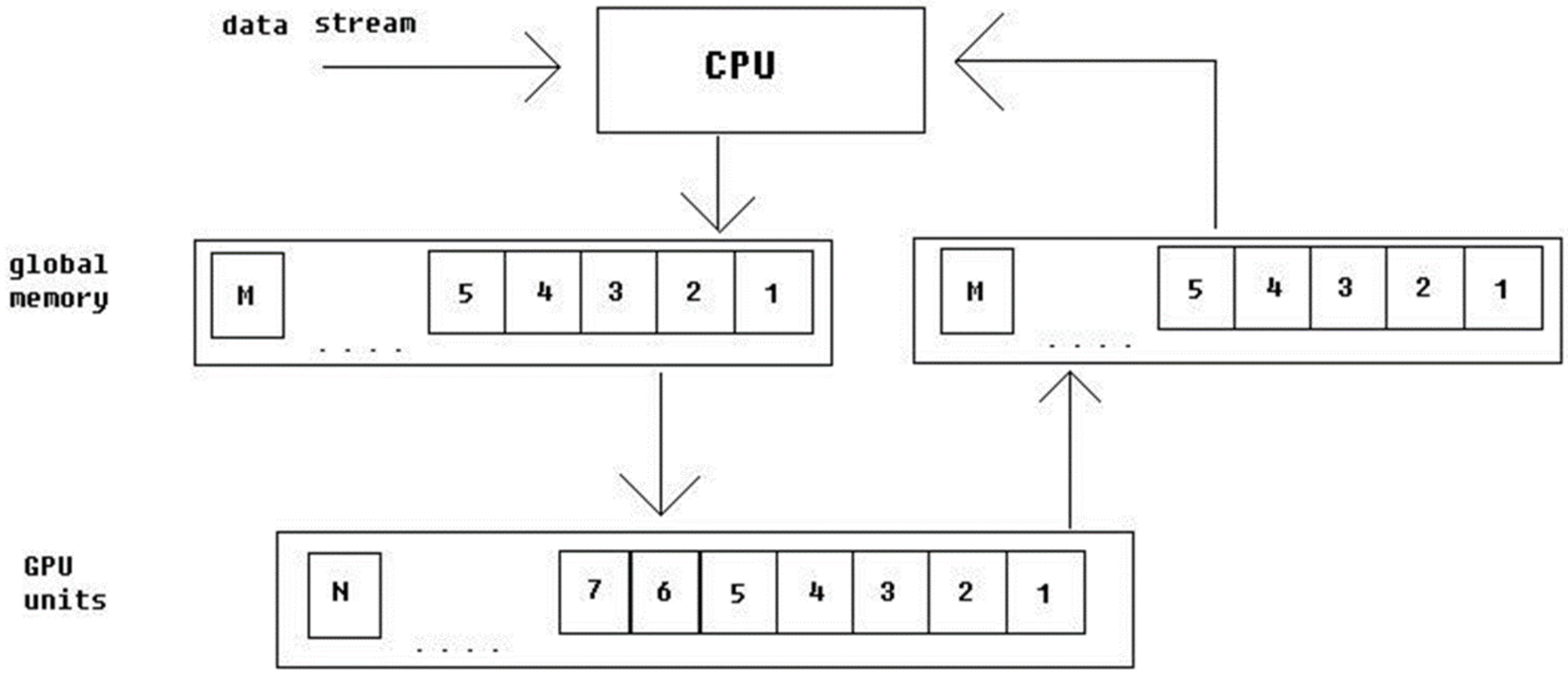
MAIN FUNCTION:

We can scale this by a factor of 32 x 32. This is 32 x 32 x 8 x 8 = 65536 processes in parallel!

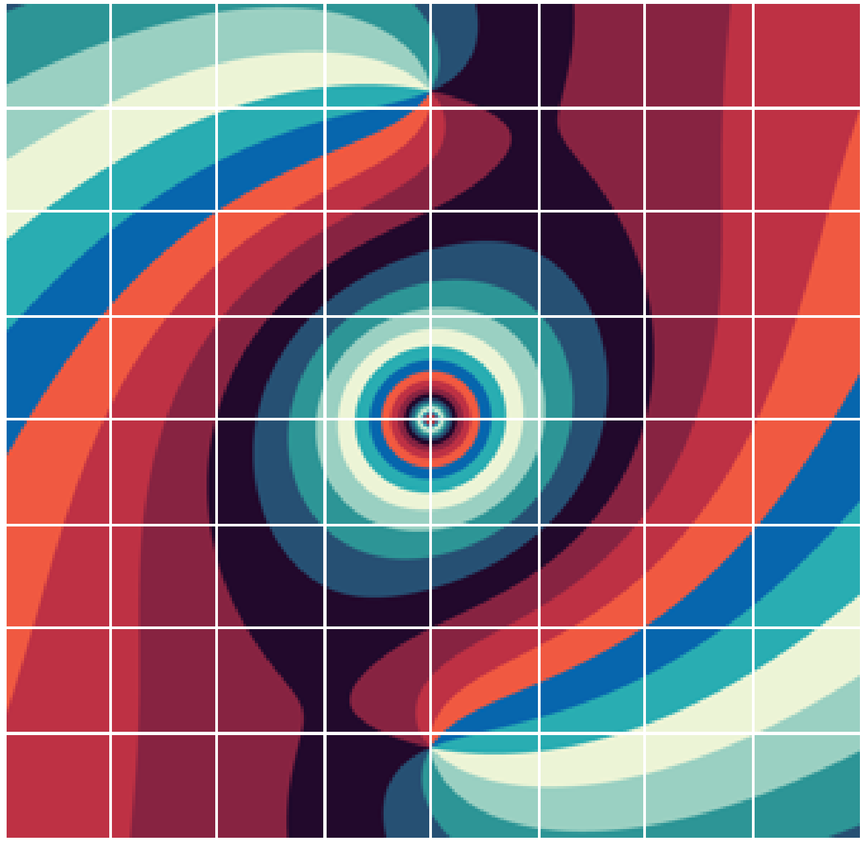
```
shape=(4096, 4096, 4) # = 8*16, 8*16, 4
image_gpu = cuda.device_array(shape,dtype=np.float32)

blocks_per_grid = (32, 32)
threads_per_block = (8, 8)
generate_one_tile[blocks_per_grid, threads_per_block](image_gpu, colors_gpu)
```

```
x,y = tile_x*16 + xi, tile_y*16 + yi

#code to generate the pixel
```

Device <-> Host Data Transfers



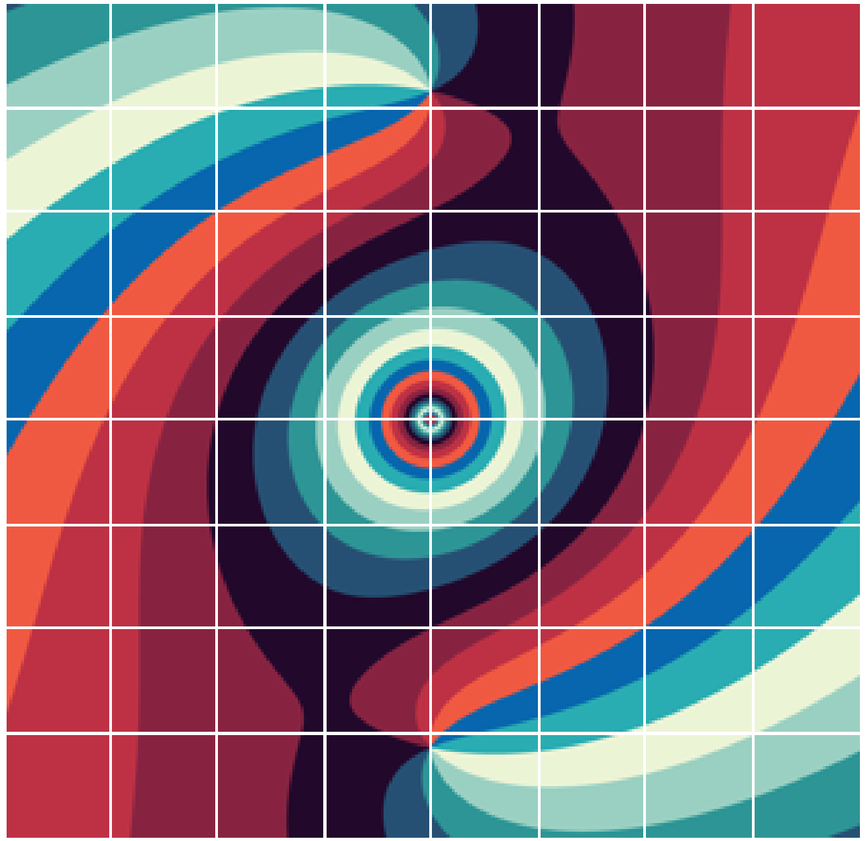
Device <-> Host Data Transfers



Data copied from CPU (host) to GPU (device)

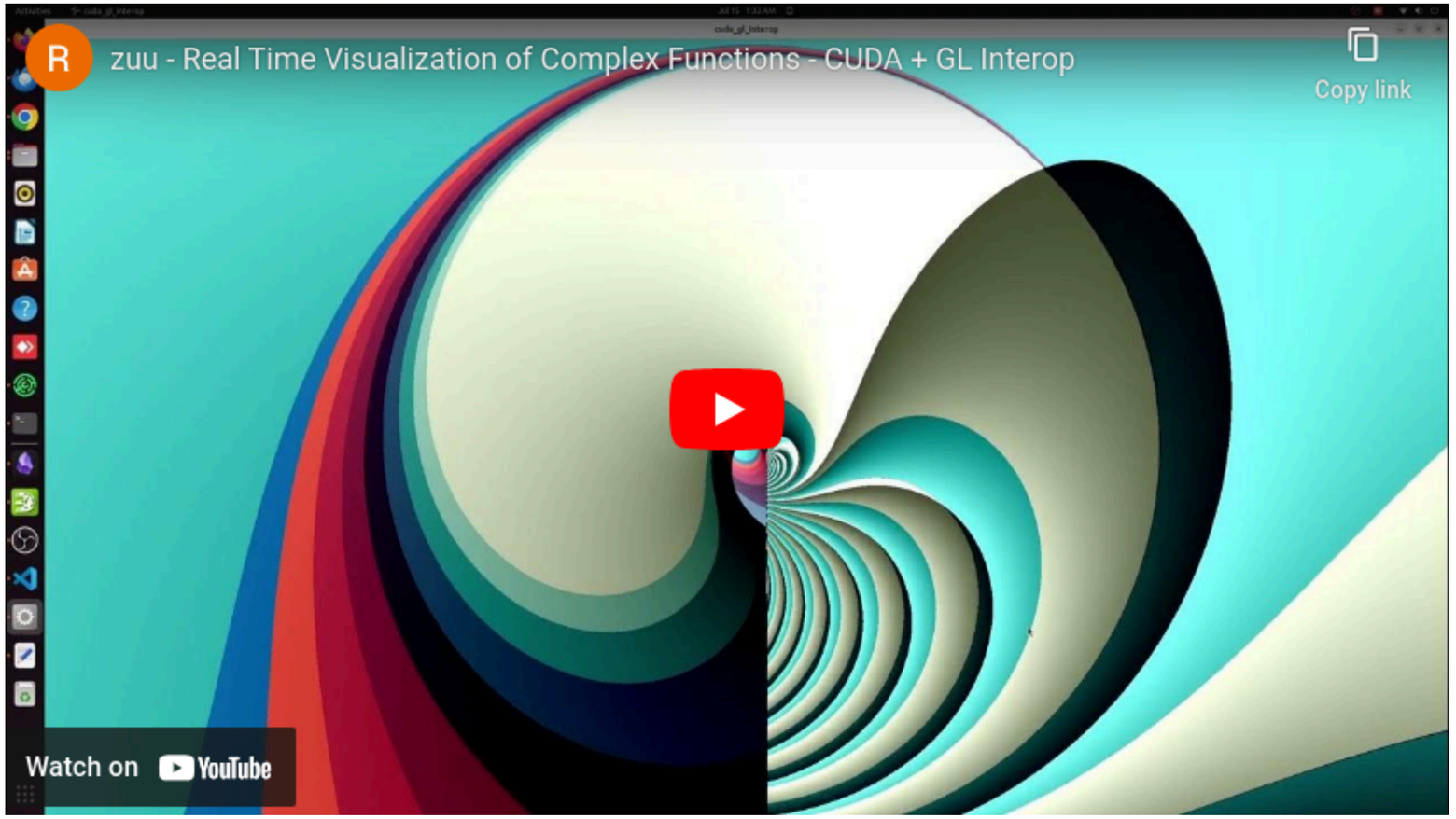
```
colors_gpu = cuda.to_device(colors)

shape=(4096, 4096, 4) # = 32*8*16, 32*8*16, 4
image_gpu = cuda.device_array(shape,dtype=np.float32)

# Grid and block dimensions
blocks_per_grid = (32, 32)
threads_per_block = (8, 8)

# Launch the kernel
generate_one_tile[blocks_per_grid, threads_per_block](image_gpu, colors_gpu)

cpu_array = image_gpu.copy_to_host()
```

Device <-> Host Data Transfers



```
colors_gpu = cuda.to_device(colors)

shape=(4096, 4096, 4) # = 32*8*16, 32*8*16, 4
image_gpu = cuda.device_array(shape,dtype=np.float32)

# Grid and block dimensions
blocks_per_grid = (32, 32)
threads_per_block = (8, 8)

# Launch the kernel
generate_one_tile[blocks_per_grid, threads_per_block](image_gpu, colors_gpu)

cpu_array = image_gpu.copy_to_host()
```

Data copied from GPU (device) to CPU (host)


R
zuu - Real Time Visualization of Complex Functions - CUDA + GL Interop
Copy link
Watch on YouTube

COMPETITION TIME!!!

https://play.blooket.com

Thank you guys for attending : D

We’re excited to see you next time!

WEDNESDAY

SPRING 2024

SUPERCOMPUTING @ GEORGIA TECH

Workshop Attendance

Club Linktree