

First and foremost, thank you for taking part in Workshop 1 - GPU Programming. This doc serves as a guide in case you feel like you are stuck on certain parts or you want to move quicker through the slides.

Why do we care about GPUs?

Graphics Processing Units (GPUs) have evolved far beyond their original purpose of rendering graphics. Today, they are a cornerstone of high-performance computing (HPC), machine learning, scientific simulations, and data-intensive workloads. But why should we care about GPUs in programming?

In modern supercomputing, GPUs are indispensable for accelerating computationally intensive workloads that would take far longer on CPUs alone. Unlike CPUs, which are optimized for low-latency, sequential processing, GPUs excel in high-throughput parallelism, executing thousands of threads simultaneously. This makes them far superior for workloads that involve massive data parallelism, such as scientific simulations, AI/ML training, molecular dynamics, and climate modeling.

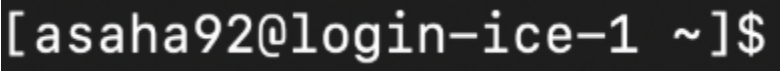
For these workloads, GPUs deliver orders of magnitude speedup over CPUs by leveraging SIMD (Single Instruction, Multiple Data) execution, optimized memory hierarchies, and high-bandwidth interconnects. Many of the world's top supercomputers, including Frontier and Fugaku, integrate GPUs to achieve exascale performance.

How Do We Interact with GPUs?

To harness GPU power, we need to take advantage of parallelism by parallelizing workloads efficiently. This is where CUDA comes in—a parallel computing API that allows us to write fast, parallel GPU code. Unlike traditional CPU programming, CUDA enables us to explicitly specify thread-level parallelism, mapping computations across thousands of lightweight GPU threads. Today, we'll dive into CUDA programming, learning how to structure parallel workloads, optimize memory access, and maximize performance for supercomputing applications.

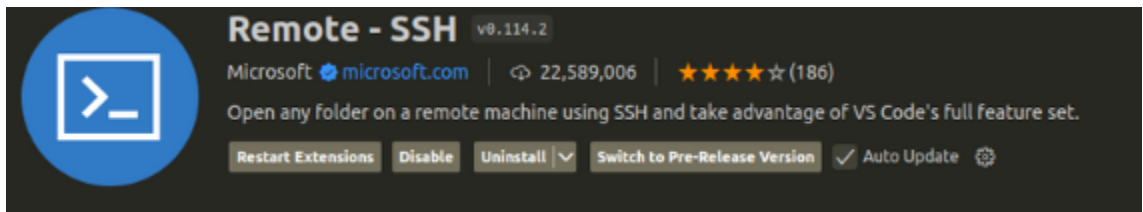
Now that we have covered why we decided to have this workshop in the first place, Let's delve into the Technical Requirements!

Technical Requirements

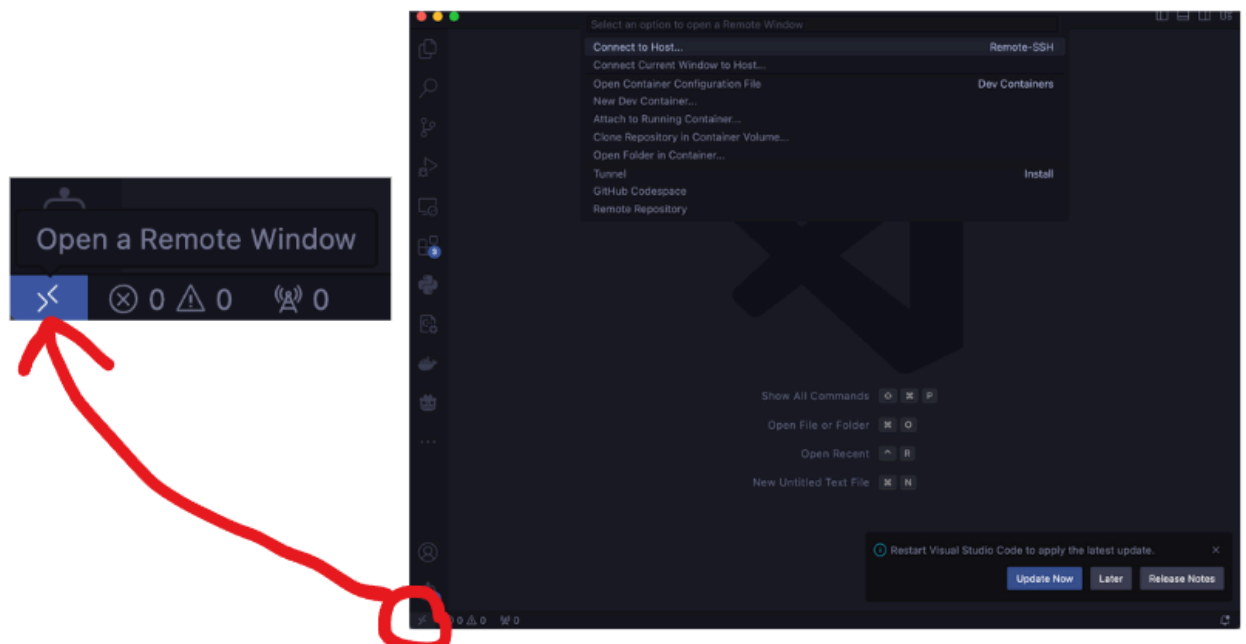
- First, we want to make sure we can access the computing cluster PACE (GT's supercomputing org) has provided.
- This means downloading the GT VPN at the link:
https://gatech.service-now.com/home?id=kb_article_view&sysparm_article=KB0026837
- The link above takes you to a webpage which has various installation guides based on what OS you have.
- Once you have installed GT VPN service (Global Protect). Follow the steps below:
 - 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 :)
- Next is your **IDE**, it **doesn't really matter what you are running as long as you know how** to use it. For the purposes of this guide we will be using **VS Code**.
- We can now move on to **SSHing into the cluster** which is just a fancy way of saying **lets login to the cluster using a secure key**. Follow the instructions below:
 - 1) Please check that you have access to an **SSH-Capable Terminal!**
 - a) **Windows Powershell**
 - b) **MacOS Terminal**
 - c) **Linux Terminal**
 - 2) **IMPORTANT: Make sure that you are connected to the GT VPN!**
 - 3) Connect to Host: For everyone, the command is **ssh**
<gt-username>@login-ice.pace.gatech.edu
 - 4) 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>**.
 - 5) You can also view this guide:
https://gatech.service-now.com/home?id=kb_article_view&sysparm_article=KB0042100
 - 6) Follow the **previous instructions to login** to the PACE cluster. If you've successfully logged in, you'll see a prompt like the following:

 - 7)
 - 8) You are now in a **login node** of the cluster, which you can use to access files, write code, submit jobs, etc.
 - 9) **BELOW IS THE GUIDE FOR VS CODE specifically.**

SSHing with VS CODE:

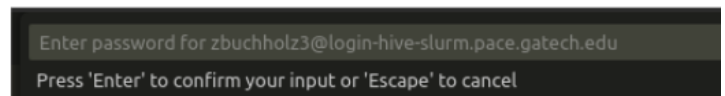
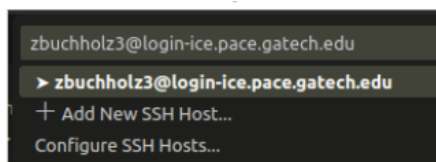
- Install the extension from the VSCode extension store.



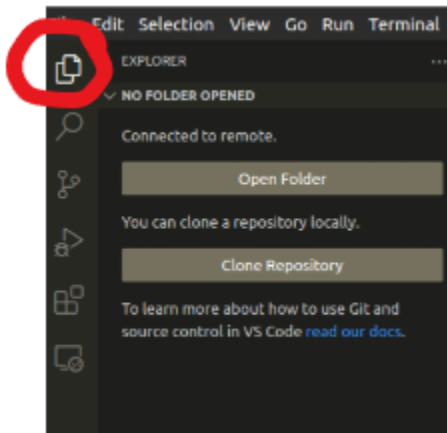
- Notice that at the bottom left of the application, you will now see a button which appears to show a broken link. Click on it.



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



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




Accessing the GPUs on the cluster:

- Try this command to **make sure** that you can use a **GPU** node,
- **salloc --nodes=1 --ntasks-per-node=1 --gres=gpu:[GPU_name]:1 --time=0:15:00**
- **where you replace [GPU_name] with one of H100, A100, V100, A40.** 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
```

-
- Note: Do not put in “[]” around the GPU name.
- Make sure to run the exit command after you finish to end your GPU node session

Driver Version: 555.42.02			CUDA Version: 12.5	
Persistence-M Pwr:Usage/Cap	Bus-Id	Disp.A Memory-Usage	Volatile GPU-Util	Uncorr. ECC Compute M. MIG M.
B PCIe	On	00000000:25:00:0 Off		0
45W / 300W		1MiB / 81920MiB	0%	Default Disabled
PID	Type	Process name	GPU Memory Usage	
s found				



SIMD (Single Instruction, Multiple Data)

Example: Adding two arrays of numbers—SIMD adds multiple pairs simultaneously.

GPU programming leverages the massively parallel architecture of **Graphics Processing Units (GPUs)** to accelerate computation. Unlike CPUs, which are optimized for sequential tasks, GPUs excel at executing thousands of threads concurrently.

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel computing platform that provides developers with direct access to the GPU's processing power. With CUDA, you can write parallel code in C/C++ and execute it on NVIDIA GPUs for significant speedups in data-parallel workloads.

Parallelism in SIMD and GPUs

SIMD is the foundation of **SIMT (Single Instruction, Multiple Threads)**, used by GPUs to execute **the same instruction across thousands of threads** in parallel. This makes GPUs perfect for tasks with high data parallelism, such as deep learning, simulations, and video processing.

Feature	CPU	GPU
Architecture	Few cores, high clock speed	Thousands of small cores
Parallelism	Low (dozens of threads)	Massive (thousands of threads)
Optimized for	Latency-sensitive tasks	Throughput-intensive tasks
Workload Examples	Real-time data processing, databases	Deep learning, simulations, image processing
Performance	High single-thread performance	High throughput for parallel tasks

Why GPUs for HPC?

- 1. **Massive Parallelism** – Thousands of cores handle data simultaneously.
- 2. **Higher Throughput** – Ideal for compute-heavy, data-parallel tasks.
- 3. **CUDA Ecosystem** – Mature libraries and tools for fast development.

Exercise 1: Vector Add

Problem Background

For each of the following problems, you'll be writing the core logic of the GPU Kernel function and decide how data streams & threads will be organized. This sounds complicated, but is not too hard (as far as these examples go!).

You'll be writing Python code that closely parallels the algorithmic approach that you would use with low-level C code written for CUDA. Some of the later examples contain material that we'll discuss later on in the problem session, so don't worry too much if you don't already know the relevant methods.

Warning This code looks like Python but it is really CUDA! You cannot use standard python tools like list comprehensions or ask for Numpy properties like shape or size (if you need the size, it is given as an argument). The puzzles only require doing simple operations, basically +, *, simple array indexing, for loops, and if statements. You are allowed to use local variables. If you get an error it is probably because you did something fancy.

Tip: Think of the function call as being run 1 time for each thread. The only difference is that `cuda.threadIdx.x` changes each time.

Although this is not relevant until later, recall that a global read occurs when you read an array from memory, and similarly a global write occurs when you write an array to memory. Ideally, you should reduce the extent to which you perform global operations, so that your code will have minimal overhead. Additionally, our grader (may) throw an error if you do something not nice!

Credits to Sasha Rush for the problems. Similar problems are available at this link:

<https://github.com/srush/GPU-Puzzles/tree/main>

Grading

For each problem, you can complete the code directly in nano or a code editor of your choice. Afterwards, once you've logged onto a node that contains a NVIDIA GPU that is CUDA-capable, you may run your code for the nth question as follows.

```
python q[n]_file.py
```

Make sure to not remove the lib.py file, as this contains some of the driver code for our grader!

Problems!

1. Implement a kernel that adds 10 to each position of vector a and stores it in vector out. You have 1 thread per position.

2. Implement a kernel that adds together each position of a and b and stores it in out. You have 1 thread per position.

Exercise 2: Matrix Multiplication

Recall that for matrix multiplication $A \times B = C$, the calculation is performed such that entry c_{ij} of the product is calculated by multiplying element-by-element the i th row of A with the j th column of B, and summing the products.

A serial implementation might iterate over every single row of A, multiply with every single row of B, and sum the result to calculate every product in C. This algorithm is $O(n^3)$. However, there are many ways to optimize, and in this exercise we will speed up the multiplication by parallelizing.

Navigate to the 'student_code' directory in 'Part2_Matrix_Multiply'.

There you will see two files:

- matmul_student.py
- matmul_shared_mem_student.py

The first file is a naive, unoptimized implementation of matrix multiply. After completing this file, try working on 'matmul_shared_mem_student.py'. Compare runtime results using the 'compare_results.py' file. Observe the performance difference between the two implementations and the speedup achieved by using shared memory. Reflect on why the second file runs faster.

		Matrix B		
		b_{11}	b_{12}	b_{13}
		b_{21}	b_{22}	b_{23}
Matrix A	a_{11}	$a_{11} \times b_{11}$ +	$a_{11} \times b_{12}$ +	$a_{11} \times b_{13}$ +
	a_{12}	$a_{12} \times b_{21}$	$a_{12} \times b_{22}$	$a_{12} \times b_{23}$
	a_{21}	$a_{21} \times b_{11}$ +	$a_{21} \times b_{12}$ +	$a_{21} \times b_{13}$ +
	a_{22}	$a_{22} \times b_{21}$	$a_{22} \times b_{22}$	$a_{22} \times b_{23}$
	a_{31}	$a_{31} \times b_{11}$ +	$a_{31} \times b_{12}$ +	$a_{31} \times b_{13}$ +
	a_{32}	$a_{32} \times b_{21}$	$a_{32} \times b_{22}$	$a_{32} \times b_{23}$
		Matrix C		

Exercise 3: Complex Function Visualization

In this section, you'll be walking through an application of GPU Programming/CUDA. In this application, you will be creating a method that generates a visualization of complex functions. Image processing is a natural extension of parallelism and GPU programming. Image processing tasks inherently involve operations on millions of pixels, where each pixel can be processed independently or in groups. This makes it an ideal candidate for GPU acceleration, leveraging thousands of threads running concurrently to dramatically reduce computation time.

Navigate to the student_code directory to fill-in exercises to complete the application. If you are stuck, you can find solutions in the solutions directory.

