



SUPERCOMPUTING

@ GEORGIA TECH

Attendance



Club Linktree





WORKSHOP 2:

Matrix Multiplication Optimization



Slides by: Panya B., Min L., Saatvik A.



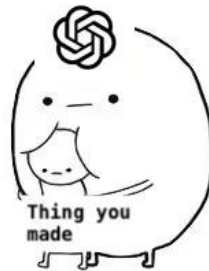
Agenda

1. Introduction on Matrix Multiplication
2. Tools Overview
3. More on HPC
4. The Message-Passing Paradigm (MPI)
- 5. The Matrix Multiplication Algorithm**
6. Solution 1: Brute Force
7. Optimization 0: Divide-and-conquer
8. Time/Space Complexity Analysis
9. Caching
10. Optimization 1: Tiling
11. Optimization 2: Cannon's Algorithm
12. Optimization 3: Sparse Matrix Multiplication
- 13. Implementation Time!**
14. Applications
15. Key Takeaways



Why matrix multiplication?

- Mathematical concept that most of you guys already know.
- Most classic and straightforward problem to emphasise the importance of HPC.
- Well-studied and foundational algorithms
- Able to be implemented using a variety of HPC paradigms relatively straightforwardly.
- Broad applications in many areas.





Tools Overview

- Python 3.11
- mpi4py
- PACE Instructional Cluster (ICE)
- Your brains!



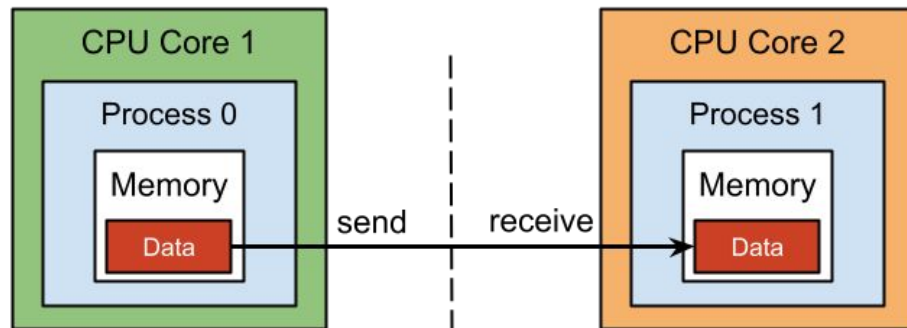
A little more on HPC...

- Despite what NVIDIA may want you to think, CUDA is not the only software stack that exists in the HPC world!
- There are many programming paradigms in the HPC world, with pros and cons for each.
 - Data-Parallel,
 - Shared-Memory,
 - Message-Passing,
 - ...
- In this workshop we will be focused on message-passing using the MPI protocol!
- MPI allows communication between different machines with their own distributed memory.



The Message-Passing Paradigm

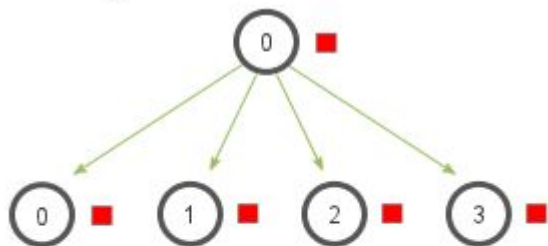
- Group of nodes = “world”
- Each node has a “rank”
- Send and receive messages between different ranks
- Common functions:
 - (Prefix) Scan
 - Broadcast
 - Scatter
 - Reduce / AllReduce
 - Gather / AllGather
 - All-to-all / Many-to-many



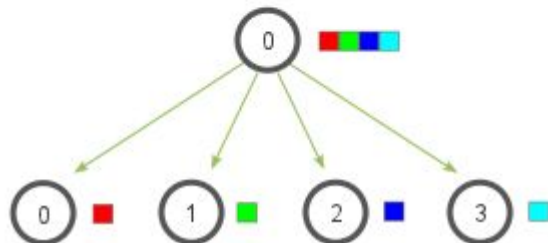


MPI Functions

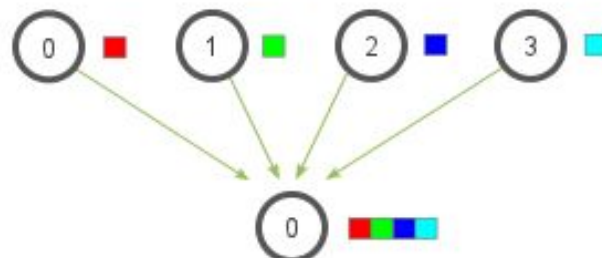
MPI_Bcast



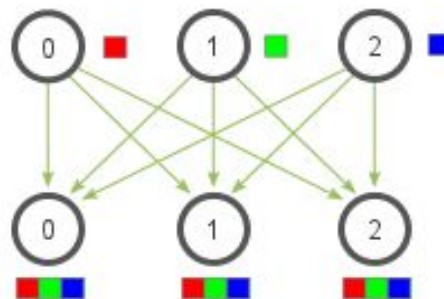
MPI_Scatter



MPI_Gather

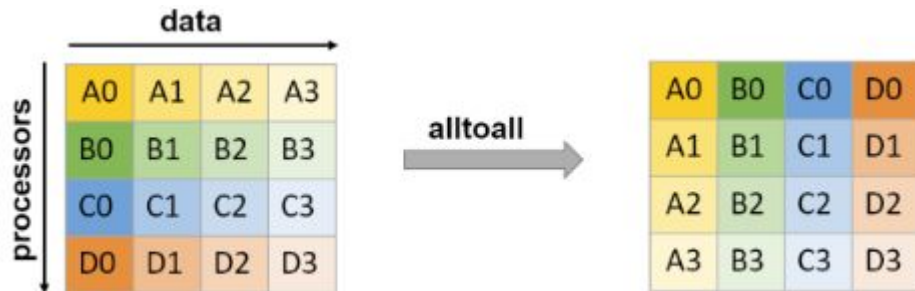


MPI_Allgather

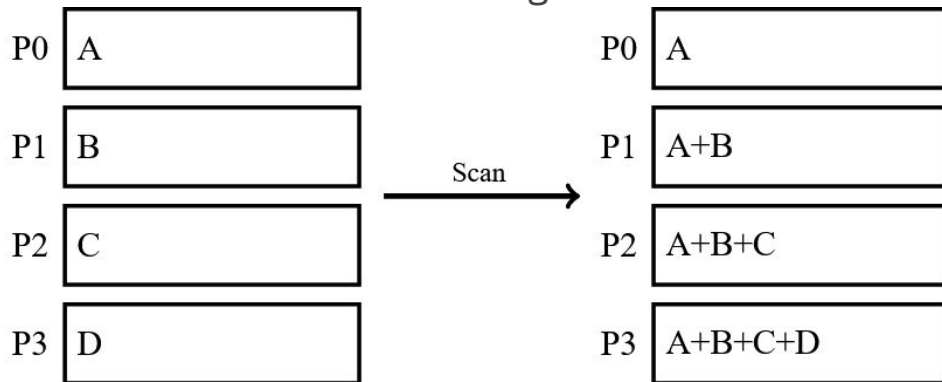




MPI Functions



- Many-to-many is similar, but with variable data lengths





mpi4py

Rank, Size, COMM: `comm.Get_rank()`, `comm.Get_size()`, `MPI.COMM_WORLD`

Send: `comm.send(sendbuf, dest, tag)`

Receive: `comm.recv(source, tag)`

Broadcast: `comm.bcast(sendbuf, root)`

Reduce: `comm.reduce(int, root, operation)`

AllReduce: `comm.allreduce(int, operation)`

Scatter: `comm.scatter(sendbuf, recvbuf, root=root)`

Gather: `comm.gather(sendbuf, recvbuf, root=root)`

AllGather: `comm.allgather(sendbuf, recvbuf)`



Set Up

Getting Started with PACE



SSH Instructions

- **IMPORTANT:** Make sure that you are connected to the GT VPN!
- Connect to Host: For everyone, the host name is `<gt-username>@login-ice.pace.gatech.edu`
- 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>.

```
(base) ipsec-10-2-128-166:~ panyabhinder$ ssh pbhinder3@login-ice.pace.gatech.edu
pbhinder3@login-ice.pace.gatech.edu's password: ?
```

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

Remote SSH w/ Terminal

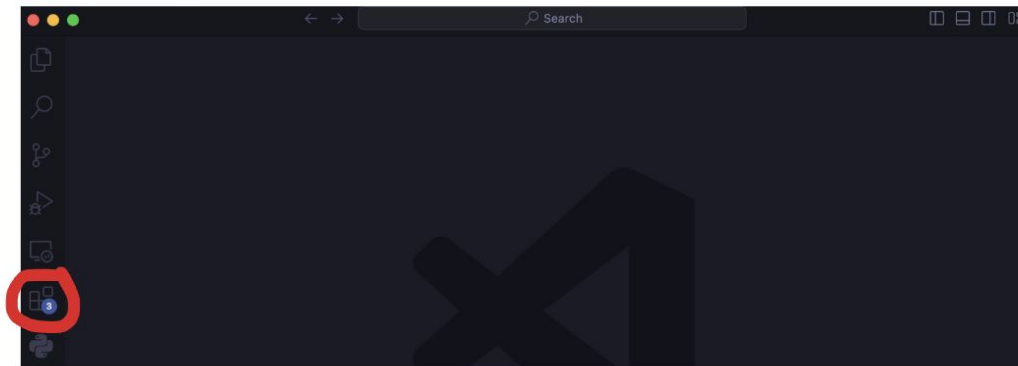
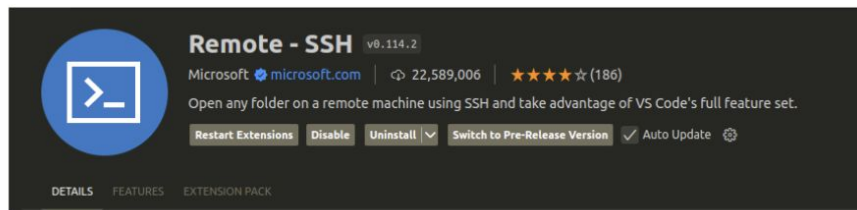
- Please check that you have access to an **SSH-Capable Terminal!**
 - Windows Powershell
 - MacOS Terminal
 - Linux Terminal
- We'll use **SSH** to log onto the cluster you'll be using today.
- Follow the previous instructions to login to the PACE cluster. If you've successfully logged in, you'll see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

```
[pbhinder3@login-ice-3 ~]$
```

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

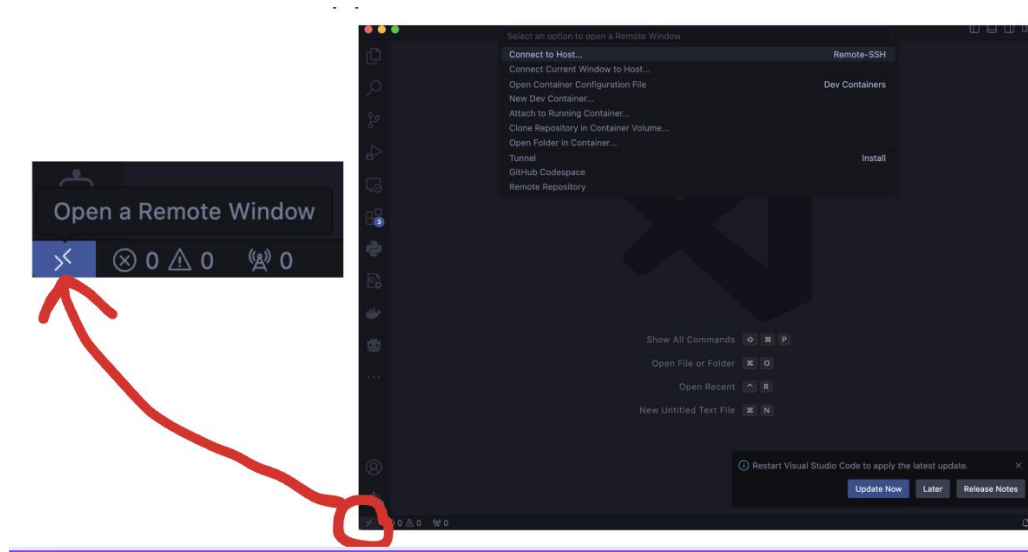
Remote SSH w/ VSCode

Install the extension from the VSCode extension store.



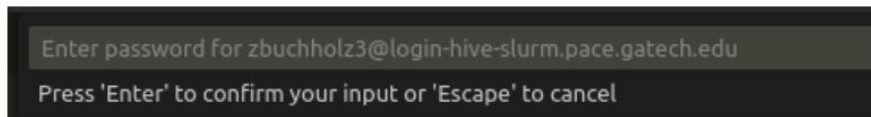
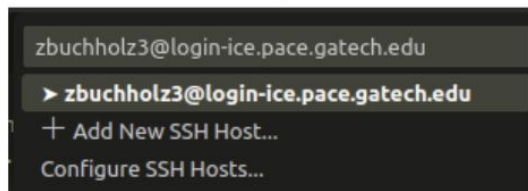
Remote SSH w/ VSCode

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.

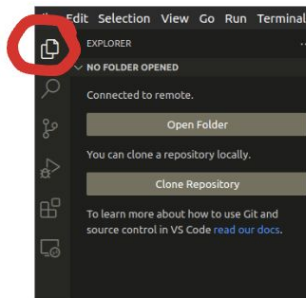


Remote SSH w/ VSCode

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.





Set up repository

In your terminal, clone the workshop repository:

```
git clone https://github.com/suco-gt/Spring25-Workshop2-Optimization.git
```

Follow the setup instructions in the README:

- Run `chmod +x *.sh` to give executable permissions to the shell scripts that are going to be used for this workshop.
- Run `./get_nodes.sh [num_nodes]` to allocate nodes on the cluster for 1 hour. Default num_nodes is 4.
- `module load anaconda3`
- Install conda environment: `conda create --name sp25_suco2 -c conda-forge python=3.11 numpy openmpi mpi4py -y`
- `conda activate sp25_suco2`
- Test if mpi4py is working for you successfully by running `./run.sh test`.
- You will run your code with the given bash file "run.sh":
- For example: To run the intro exercises- `./run.sh intro`



Agenda

1. Introduction on Matrix Multiplication
2. Tools Overview
3. More on HPC
4. The Message-Passing Paradigm (MPI)
- 5. The Matrix Multiplication Algorithm**
6. Solution 1: Brute Force
7. Time/Space Complexity Analysis
8. Optimization 0: Divide-and-conquer
9. Caching
10. Optimization 1: Tiling
11. Optimization 2: Cannon's Algorithm
12. Optimization 3: Sparse Matrix Multiplication
- 13. Implementation Time!**
14. Applications
15. Key Takeaways



Inner Matrix Multiplication Algorithm

Problem Statement: Given matrices A and B of size $(m \times n)$ and $(n \times p)$ respectively, where, $m \neq n \neq p$ and $m, n, p > 0$, calculate their inner product AB .

$$\begin{array}{c} \text{row } i \\ \left[\begin{array}{c} \text{blue box} \end{array} \right] \\ A \\ m \times n \end{array} \times \begin{array}{c} \text{col } j \\ \left[\begin{array}{c} \text{red box} \end{array} \right] \\ B \\ n \times p \end{array} = \begin{array}{c} \left[\begin{array}{c} \text{white box} \end{array} \right] \\ C \\ m \times p \end{array}$$

$c_{ij} = \text{row } i \times \text{col } j = \sum_{k=1}^n a_{ik} b_{kj}$



Solution 1: Brute Force Approach

Naive Approach: Using Nested Loops – $O(n^3)$ Time and $O(n^2)$ Space

$$\begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} \times \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} = \begin{bmatrix} a_1a_2 + b_1c_2 & a_1b_2 + b_1d_2 \\ c_1a_2 + d_1c_2 & c_1b_2 + d_1d_2 \end{bmatrix}$$

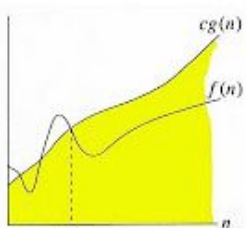


How do we make
this faster?

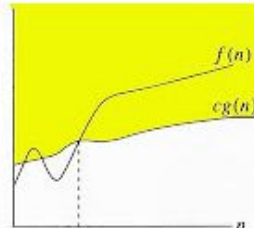


Time/Space Complexity Analysis of Parallel Programs

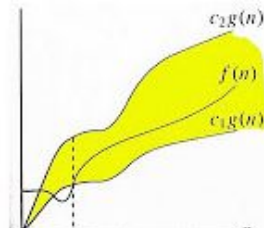
- With serial algorithms, it is much easier to analyse the time and space complexity– the program executes as a single thread, so the number of instructions = the time complexity.
- Meanwhile, with HPC we may split an instruction over multiple threads instead, meaning the number of instructions \neq the time complexity, i.e., the amount of work done by the program \neq the time it takes for the program to actual finish!



Big Oh



Omega



Theta



Optimization 0: Divide and Conquer

- Theorem:

Suppose that we partition each of A , B , and C into four $n/2 \times n/2$ matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \quad (4.9)$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \quad (4.10)$$

Equation (4.10) corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \quad (4.11)$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \quad (4.12)$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \quad (4.13)$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \quad (4.14)$$

Cormen et al.

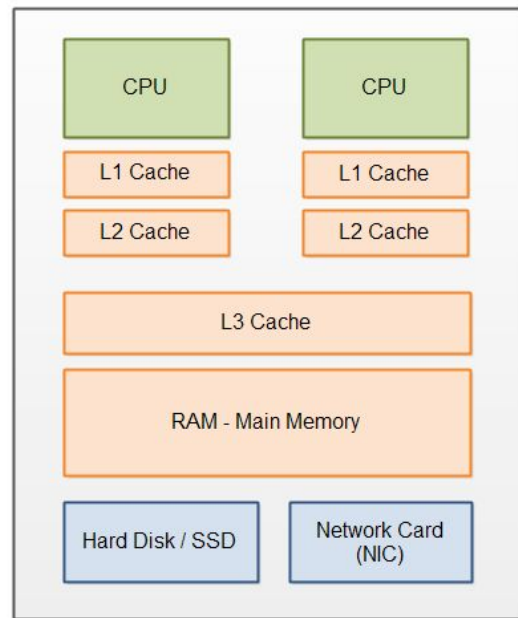
- This can be done on a single processor using the divide-and-conquer paradigm, or alternatively, we can split the workload across multiple processors!



Caching

- Memory access is expensive
- **Caches** are fast, smaller memory near the CPU that stores recently or frequently accessed data
- **Spatial locality:** processor likely to access memory locations near each other
- **Temporal locality:** memory accessed at a specific time is likely to be accessed again
- **Problem with naive matmul:** poor spatial and temporal locality → lots of cache misses

What if we broke the matrices into smaller blocks stored in the cache ...





Optimization 1: Blocked Multiplication

Goal: Increase parallelism

How?

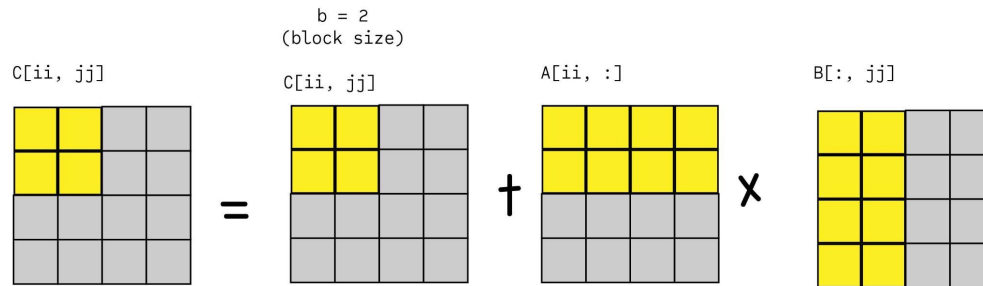
- Divide the matrix into evenly sized blocks, 1 per processor (2D partitioning of both matrices).
- Send all the blocks in a row to all processors in a row, and send all the blocks in a column to all processors in a column.
- Each processor performs matrix multiplication for its block.
- The root collects all of the results.

Pros and Cons?

- Computation is divided among the processors
- However, total cache usage is suboptimal still as blocks are present multiple times.



Optimization 1: Blocked Multiplication



ii, jj, kk denote block indices while i, j, k denote element indices

```
// C = C + A * B
// b = n / N (where b is the block size)
for ii = 1 to N:
  for jj = 1 to N:
    for kk = 1 to N:
      C[ii, jj] += A[ii, kk] * B[kk, jj]
```



Optimization 2: Cannon's Algorithm

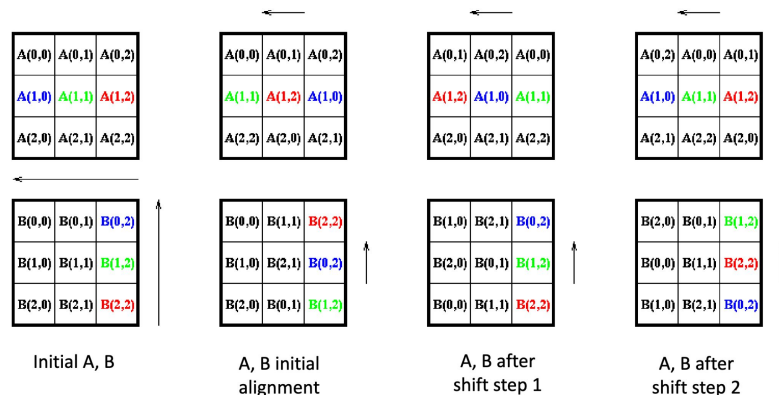
Goal: Improve memory efficiency

How?

- Multiplies 2 square matrices using processors in a 2D grid
- Distributes submatrices (blocks) of A and B to each processor
- Minimizes communication by reusing data blocks over multiple computation steps

Key Steps

- **Align data:** Alignment: Shift A blocks left by row index i . Shift B blocks up by column index j
- **Main Loop** (repeat p times): Each processor multiplies local $A_{jj} \times B_{ij}$ accumulates in C_{ij} Then:
 - Shift A left by 1 (wraparound)
 - Shift B up by 1 (wraparound)



What improvements does this have?

- Avoids global all-to-all broadcast (reduces computation time)
- Computation dominates for larger matrices
- Memory needed does not increase as processors increase



Optimization 3: Sparse Matrix Mult.

- Sparse Matrix: Matrix with primarily zeros.
- Using dense matrix multiplication algorithms wastes time and memory.
- Much of the matrix multiplication done in HPC is SpMM.
- How to design a SpMM algorithm?
 1. Use better data structures to represent sparse matrices.
 2. For matrix multiplication, we can say there are 4 kinds of multiplications: 0 and 0, nonzero and nonzero, 0 and nonzero, nonzero and 0.
 3. In the case of SpMM, notice that (nonzero x nonzero) will occur much lesser in frequency than other kinds of multiplication. We optimize the algorithm by making sure we only do (nonzero x nonzero) multiplications!
 4. Matrix multiplication can be rewritten using a different formula:

$$c_i = \sum_{a_{iv} \neq 0} a_{iv} b_v. \quad \text{for } 1 \leq i \leq p.$$

“The i th row of resulting matrix C is the linear combination of the v rows of B for which $a_{iv} \neq 0$.”

– Gustavson (September, 1978)



Designing A SpMM Algorithm

1. Designing a better data structure:

→ Solution: The **Coordinate Representation (COO)**

	0	1	2	3
0	12		26	
1				
2		19		
3		14		7

(0, 0, 12)
(0, 2, 26)
(2, 1, 19)
(3, 1, 14)
(3, 3, 7)

row + column index +
weight per nonzero
(easy to build / modify)



Designing A SpMM Algorithm

(2-4). Minimize the number of multiplications we are doing with 0 elements of both matrices by using the dot product formula instead

RowWise_SpGEMM(C, A, B)

```
1  // set matrix C to  $\emptyset$ 
2  for  $a_{i*}$  in matrix A in parallel
3      do for  $a_{ik}$  in row  $a_{i*}$ 
4          do for  $b_{kj}$  in row  $b_{k*}$ 
5              do  $value \leftarrow a_{ik}b_{kj}$ 
6                  if  $c_{ij} \notin c_{i*}$ 
7                      then insert ( $c_{ij} \leftarrow value$ )
8                      else  $c_{ij} \leftarrow c_{ij} + value$ 
```



But wait... Limitations of COO

- Revisit step 1: Notice that the only way to implement COO data structure results in **low cache locality!**
- Solution: A new data structure called **Compressed Sparse Row (CSR) representation.**

	0	1	2	3
0	12		26	
1				
2		19		
3		14		7

ptr	0	2	2	3	5	(row starts in CSR)
	↓		↘			
ind	0	2	1	1	3	(column ids in CSR)
val	12	26	19	14	7	(numerical values in CSR)



Implementation Time!



Your Job

Race to implement each optimization! (Feel free to work in groups)

Repo setup:

- Your code will be implemented in the “problems” folder
- Solutions are provided for your benefit– please attempt the problems first

Your tasks:

1. Implement the brute force matmul implementation in `bruteforce.py`
2. Implement the blocked matrix multiplication optimization in `blocked.py`
3. Implement Cannon's Algorithm in `cannon.py`
4. Implement COO SpGEMM in `spgemm.py`

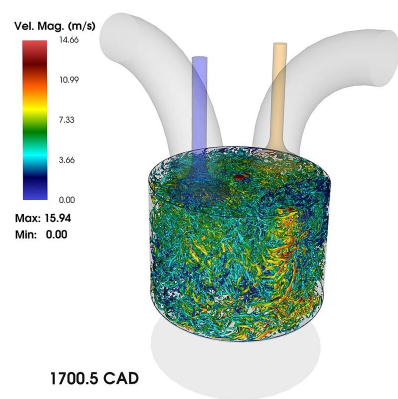
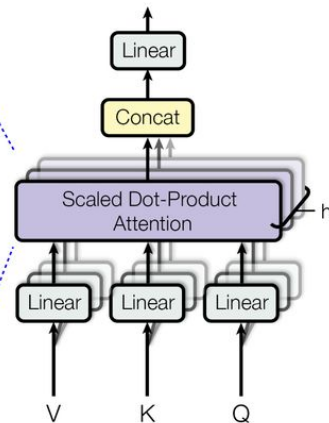
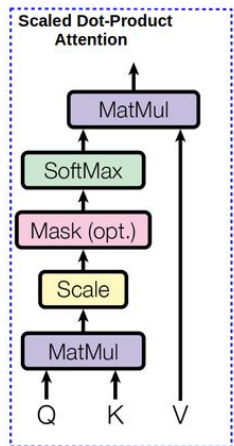
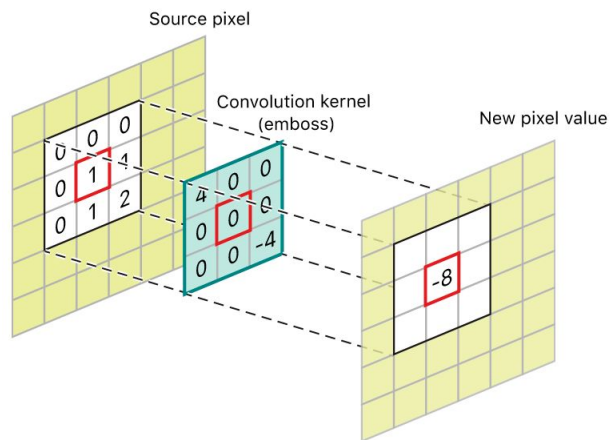


Agenda

1. Introduction on Matrix Multiplication
2. Tools Overview
3. More on HPC
4. The Message-Passing Paradigm (MPI)
- 5. The Matrix Multiplication Algorithm**
6. Solution 1: Brute Force
7. Time/Space Complexity Analysis
8. Optimization 0: Divide-and-conquer
9. Caching
10. Optimization 1: Tiling
11. Optimization 2: Cannon's Algorithm
12. Optimization 3: Sparse Matrix Multiplication
- 13. Implementation Time!**
14. Applications
15. Key Takeaways



Applications





Key Takeaways

- Parallelism and scaling
 - Breaking up computation into smaller tasks speed up execution
 - Difference between shared memory (OpenMP) and distributed memory parallelism (MPI)
- Memory hierarchy and data locality - hardware impacts efficiency
 - Tiling and blocking improves cache efficiency
 - Impact of sparsity on performance
- Algorithmic trade-off
 - Communication vs computation bottlenecks
 - Choosing right optimization based on problem size and hardware constraints