RCIF Applied HPC Seminar Series

Best Practices for Running Parallel Processes on CPU/GPU

Washington University in St. Louis

School of Medicine

MIR Mallinckrodt Institute of Radiology

Introduction to Parallel Processing

- Parallel processing involves the simultaneous execution of multiple computing tasks across various processors, designed to speed up computations and increase efficiency.
- Architecture: Utilizes multiple CPU cores or GPUs, each handling parts of a task concurrently, often using architectures like multi-core processors, distributed systems, or cloud computing platforms.
- Divide and Conquer: Tasks are divided into smaller sub-tasks, distributed among processors, and computed in parallel, reducing overall processing time compared to sequential execution.
- Communication: Requires efficient communication between processors to synchronize tasks and compile results, using mechanisms such as shared memory, message passing, or data distribution.

A Brief History of Parallel Computing

- 1950s-1960s: Early parallel computing concepts by John von Neumann; IBM 7030 Stretch implemented pipelining and parallelism.
- 1970s: Cray-1, designed by Seymour Cray, used vector processing and became the fastest computer.
- 1980s: Massively parallel processors (MPPs) developed by companies like Connection Machine.
- 1990s: Distributed and cluster computing emerged, with systems like the Beowulf cluster.
- 2000s: GPUs began being used for general-purpose computing (GPGPU), enhancing parallel processing.
- 2000s-Present: Multi-core processors became standard, enabling hardware-level parallelism.
- 2010s-Present: HPC systems like IBM’s Blue Gene and China’s Sunway TaihuLight achieved petaflop and exaflop performance.
- Present: Quantum computing explores new parallel computing frontiers using quantum mechanics.
- 2010s-Present: Cloud computing platforms and big data frameworks democratized scalable, parallel data processing.

CPU-based Parallel Processing

- Core Utilization: CPU-based parallel processing leverages multiple CPU cores, allowing each core to execute different threads or processes simultaneously, increasing computational speed and efficiency.
- Thread Management: Efficient management of threads within and across CPU cores is essential, often facilitated by advanced operating systems and specialized software that allocate tasks based on CPU availability and workload.
- Synchronization: Ensures that multiple tasks operating in parallel do not interfere with each other by coordinating access to shared resources using locks, semaphores, or barriers.
- Memory Architecture: Utilizes shared or distributed memory models where processors either access a common memory space or have individual memory, influencing how data is accessed and managed during parallel execution.
- Scalability: CPU parallel processing can scale horizontally by adding more processors or vertically by upgrading to CPUs with more cores, though effectiveness is influenced by the software's ability to decompose tasks and manage increased complexity.

“If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens?”

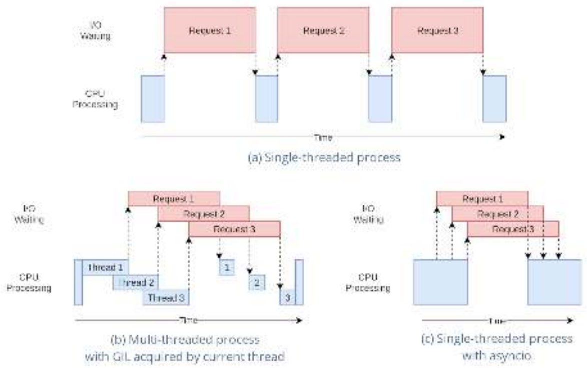
-Seymour Cray

GPU-based Parallel Processing

- Massive Parallelism: GPUs contain hundreds to thousands of smaller, efficient cores designed for executing multiple parallel tasks simultaneously, ideal for operations that can be broken down into small, independent tasks.
- Specialized Hardware: GPUs are optimized for compute-intensive, highly parallel computations and are more efficient than CPUs for algorithms where processing can be performed in parallel, such as matrix operations or pixel rendering.
- Memory Bandwidth: High memory bandwidth in GPUs facilitates faster data transfer rates, which is crucial for handling large datasets and textures quickly, significantly speeding up processing tasks.
- Programming Models: Utilizes specific programming frameworks and languages like CUDA (for NVIDIA GPUs) or OpenCL that allow developers to directly harness GPU's parallel processing capabilities for general-purpose computing.
- Application-Specific: Especially effective in fields such as deep learning, graphics rendering, and scientific simulations, where parallel processing can dramatically reduce computation times and increase performance.

Multithreading Techniques

- Concurrent Execution: Multithreading allows multiple threads to run concurrently within a single process, sharing the process's resources but operating independently to enhance execution efficiency and responsiveness.
- Resource Sharing: Threads within the same process share memory and resources, facilitating communication and data exchange between threads without the overhead of inter-process communication mechanisms.
- Synchronization Tools: Implements synchronization techniques such as mutexes, locks, and semaphores to manage access to shared resources and prevent data inconsistencies or race conditions.
- Task Segmentation: Efficiently breaks down complex tasks into smaller, manageable sub-tasks that can be executed across multiple threads, optimizing resource use and reducing processing time.
- Scalability and Responsiveness: Improves the scalability of applications by distributing workload across available CPU cores, enhancing system responsiveness by allowing intensive tasks to be processed in the background without freezing the user interface.


I/O Waiting
CPU Processing
Request 1
Request 2
Request 3
Time
(a) Single-threaded process
Thread 1
Thread 2
Thread 3
1
2
3
(b) Multi-threaded process with GIL acquired by current thread
(c) Single-threaded process with asyncio

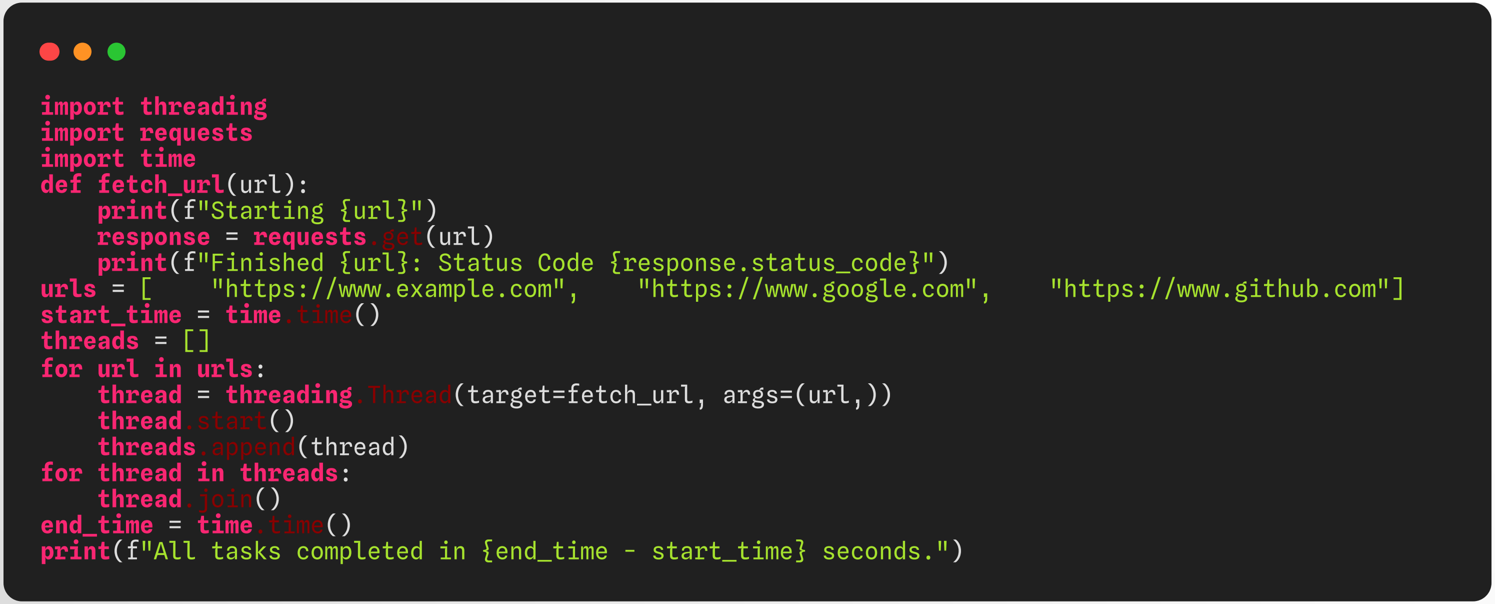
Python and the GIL

- Multithreading in Python is constrained by the Global Interpreter Lock (GIL), which allows only one thread to execute Python bytecode at a time, this can significantly limit the performance of multithreaded programs, especially for CPU-bound tasks.
- Multithreading Limitations: Suitable for I/O-bound tasks where the GIL's impact is minimal, but inefficient for CPU-bound operations as threads compete for the single execution slot allowed by the GIL, leading to potential performance bottlenecks.
- Multiprocessing Benefits: Bypasses the GIL by using separate processes, each with its own Python interpreter and memory space, enabling true parallel execution on multiple CPU cores, ideal for CPU-intensive tasks.
- Resource Overhead: Multiprocessing increases resource usage due to the need for more memory and potential overhead from inter-process communication, compared to multithreading which shares memory within the same process.

Multiprocessing Techniques

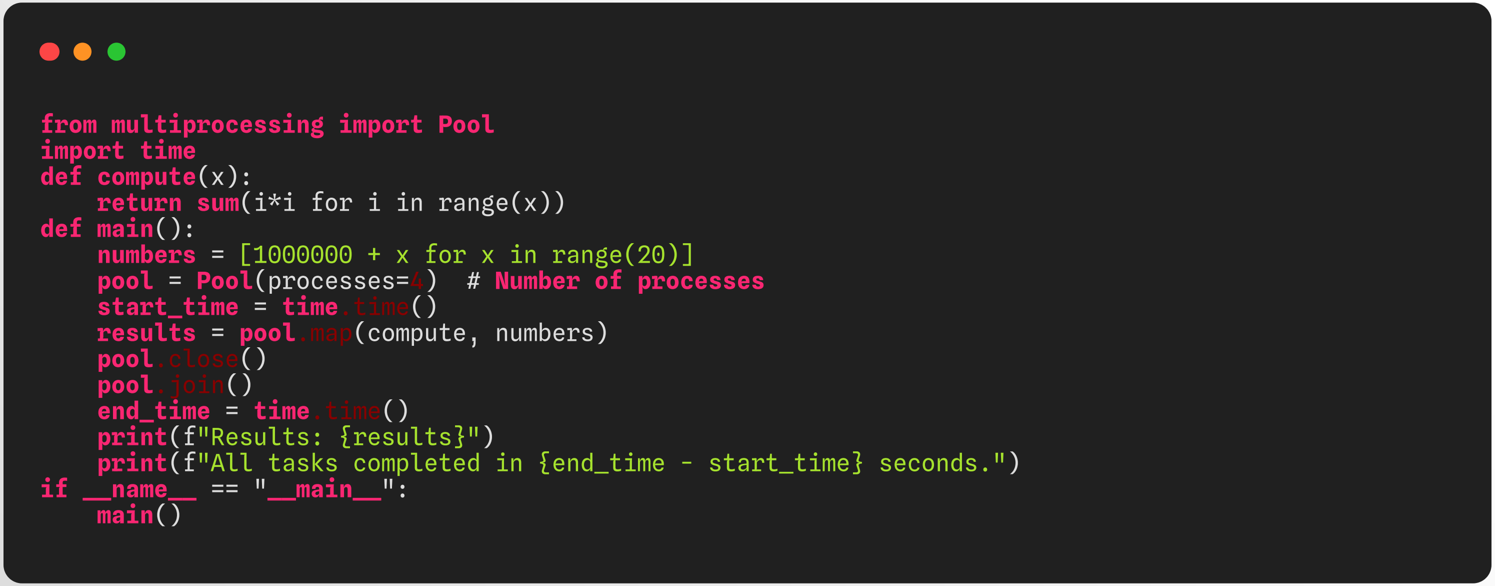
- Multiple Processors: Multiprocessing involves using two or more CPUs within a single computer system, allowing each processor to execute tasks independently, which enhances parallel execution and improves performance.
- Process Isolation: Each process operates in its own memory space, ensuring that processes do not interfere with each other, which enhances security and stability by preventing one process from crashing others.
- Load Balancing: Efficiently distributes tasks across multiple processors to optimize resource utilization and reduce processing time, often managed by the operating system or dedicated scheduling algorithms.
- Inter-process Communication (IPC): Utilizes mechanisms such as pipes, message queues, shared memory, and sockets to enable processes to communicate and synchronize their operations, crucial for tasks that require coordination.
- Scalability: Can scale effectively by adding more processors to the system, which can directly increase the capacity to handle more processes or tasks simultaneously, suitable for high-performance and real-time computing applications.

I/O Bound Tasks: Threading



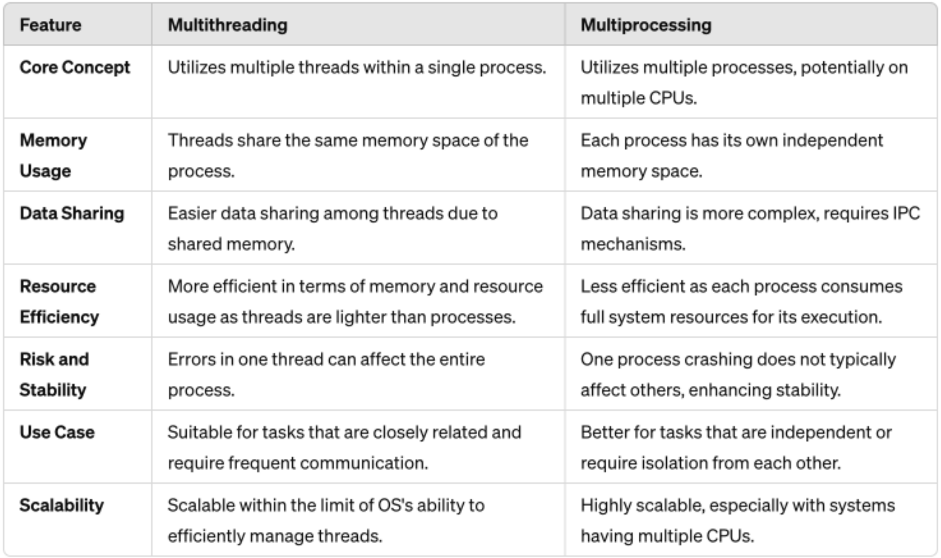
```python
import threading
import requests
import time
def fetch_url(url):
    print(f"Starting {url}")
    response = requests.get(url)
    print(f"Finished {url}: Status Code {response.status_code}")
urls = [    "https://www.example.com",    "https://www.google.com",    "https://www.github.com"]
start_time = time.time()
threads = []
for url in urls:
    thread = threading.Thread(target=fetch_url, args=(url,))
    thread.start()
    threads.append(thread)
for thread in threads:
    thread.join()
end_time = time.time()
print(f"All tasks completed in {end_time - start_time} seconds.")
```

CPU Bound Tasks: Multiprocessing (sum of squares)



```python
from multiprocessing import Pool
import time
def compute(x):
    return sum(i*i for i in range(x))
def main():
    numbers = [1000000 + x for x in range(20)]
    pool = Pool(processes=4)  # Number of processes
    start_time = time.time()
    results = pool.map(compute, numbers)
    pool.close()
    pool.join()
    end_time = time.time()
    print(f"Results: {results}")
    print(f"All tasks completed in {end_time - start_time} seconds.")
if __name__ == "__main__":
    main()
```

Multithreading vs Multiprocessing on CPU



Feature	Multithreading	Multiprocessing
Core Concept	Utilizes multiple threads within a single process.	Utilizes multiple processes, potentially on multiple CPUs.
Memory Usage	Threads share the same memory space of the process.	Each process has its own independent memory space.
Data Sharing	Easier data sharing among threads due to shared memory.	Data sharing is more complex, requires IPC mechanisms.
Resource Efficiency	More efficient in terms of memory and resource usage as threads are lighter than processes.	Less efficient as each process consumes full system resources for its execution.
Risk and Stability	Errors in one thread can affect the entire process.	One process crashing does not typically affect others, enhancing stability.
Use Case	Suitable for tasks that are closely related and require frequent communication.	Better for tasks that are independent or require isolation from each other.
Scalability	Scalable within the limit of OS's ability to efficiently manage threads.	Highly scalable, especially with systems having multiple CPUs.

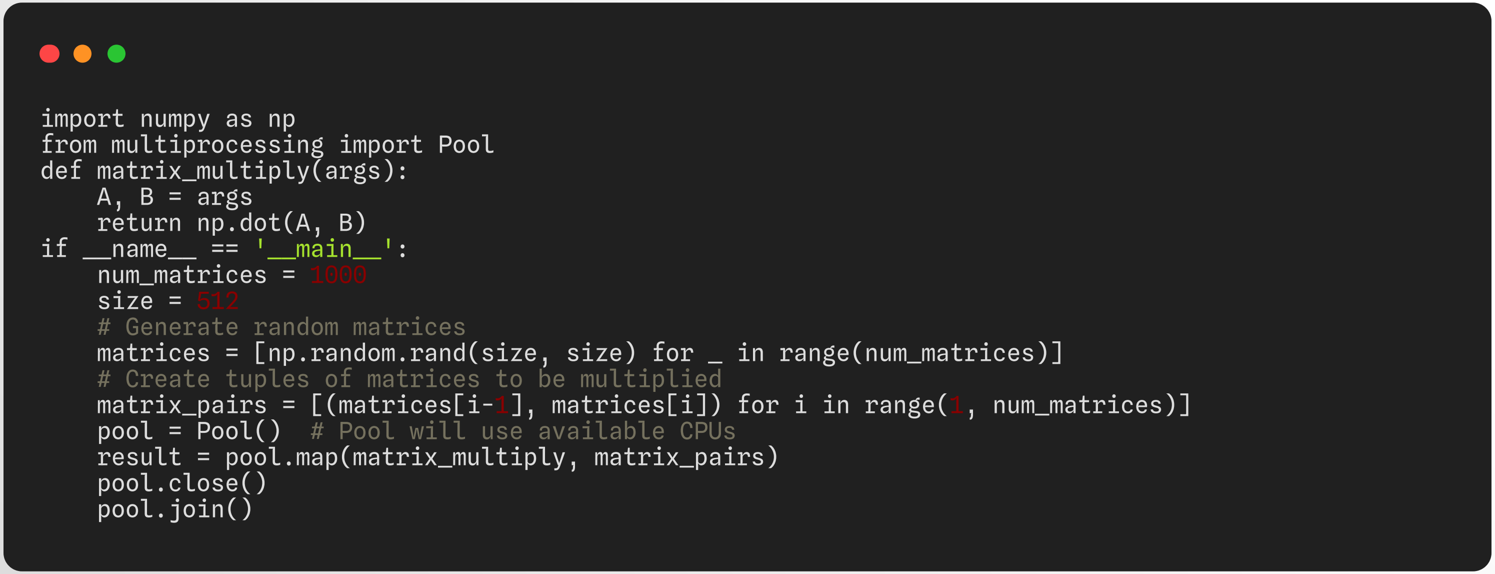
Hybrid Parallel Programming Models

- Combination of Models: Hybrid parallel programming combines both shared memory (multithreading) and distributed memory (multiprocessing) models to leverage the strengths of each approach, optimizing performance and scalability.
- Efficiency on Heterogeneous Systems: Particularly effective on heterogeneous systems with multi-core CPUs and multiple nodes, allowing fine-grained parallelism on individual nodes and coarse-grained across nodes.
- Reduced Overhead: Minimizes the communication overhead by using multithreading locally within each node to exploit shared memory, and multiprocessing across nodes to handle inter-node communication.
- Flexible and Scalable: Offers flexibility in balancing load and data distribution, adapting to different hardware architectures and scales efficiently from small clusters to large supercomputers.
- Common Tools and Languages: Utilizes programming frameworks and languages like MPI (Message Passing Interface) for inter-node communication combined with OpenMP or POSIX threads for intra-node thread management, enhancing the programmer's control over parallel execution.

GPU Programming Models

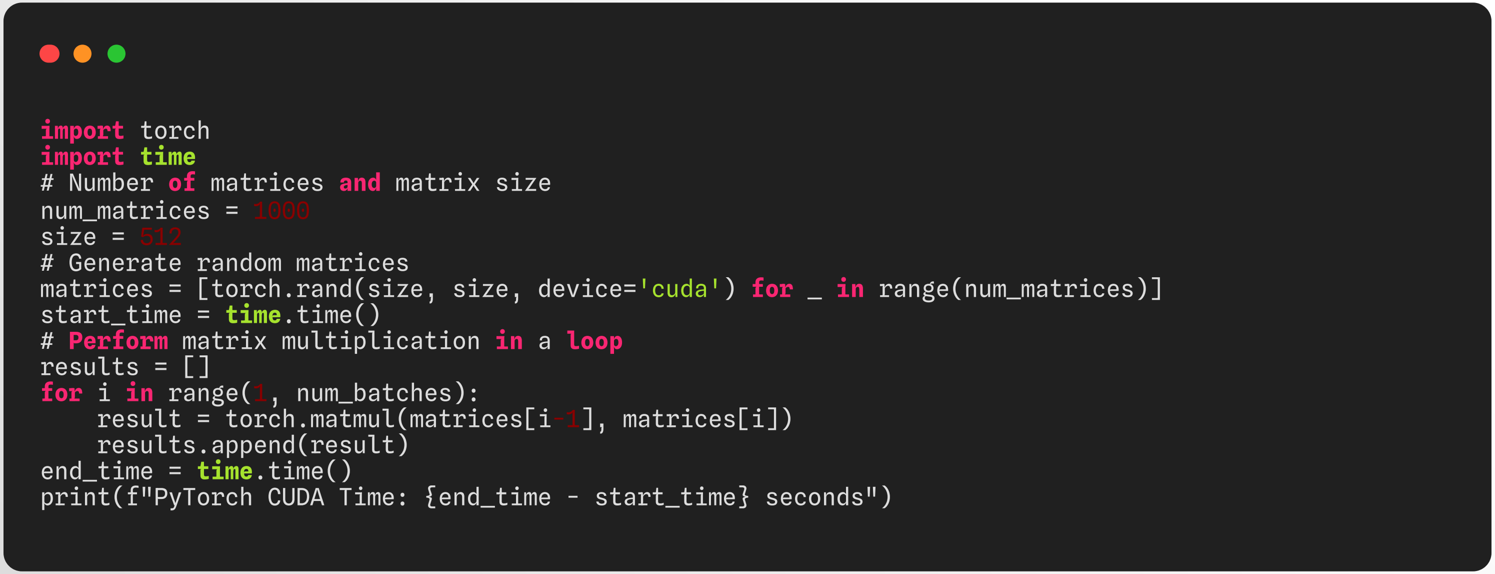
- CUDA (Compute Unified Device Architecture): A parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). It allows developers to use C/C++ along with CUDA-specific extensions to directly build software that can execute on GPUs, offering fine-grained control over GPU operations and memory management.
- Python with CUDA: Python does not natively support GPU programming, but libraries like Numba and PyCUDA allow Python code to interface with CUDA APIs. These libraries enable Python programmers to define functions that execute on the GPU, leveraging CUDA's power without writing C/C++ code.
- PyTorch: A high-level deep learning library that provides extensive GPU acceleration. PyTorch integrates seamlessly with CUDA to optimize and execute deep learning models. It simplifies GPU usage through automatic differentiation and native support for tensor operations on GPUs, allowing developers to focus more on designing models rather than managing GPU workflows.

Python + Numpy



```python
import numpy as np
from multiprocessing import Pool
def matrix_multiply(args):
    A, B = args
    return np.dot(A, B)
if __name__ == '__main__':
    num_matrices = 1000
    size = 512
    # Generate random matrices
    matrices = [np.random.rand(size, size) for _ in range(num_matrices)]
    # Create tuples of matrices to be multiplied
    matrix_pairs = [(matrices[i-1], matrices[i]) for i in range(1, num_matrices)]
    pool = Pool()  # Pool will use available CPUs
    result = pool.map(matrix_multiply, matrix_pairs)
    pool.close()
    pool.join()
```

Pytorch

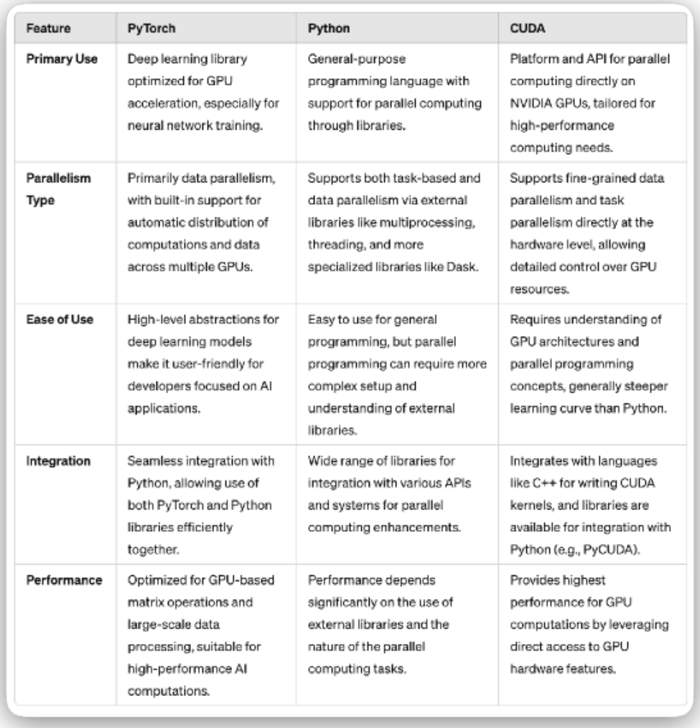


```python
import torch
import time
# Number of matrices and matrix size
num_matrices = 1000
size = 512
# Generate random matrices
matrices = [torch.rand(size, size, device='cuda') for _ in range(num_matrices)]
start_time = time.time()
# Perform matrix multiplication in a loop
results = []
for i in range(1, num_batches):
    result = torch.matmul(matrices[i-1], matrices[i])
    results.append(result)
end_time = time.time()
print(f"PyTorch CUDA Time: {end_time - start_time} seconds")
```

Cuda TLDR: It's more complicated

Code Example

Comparing Models



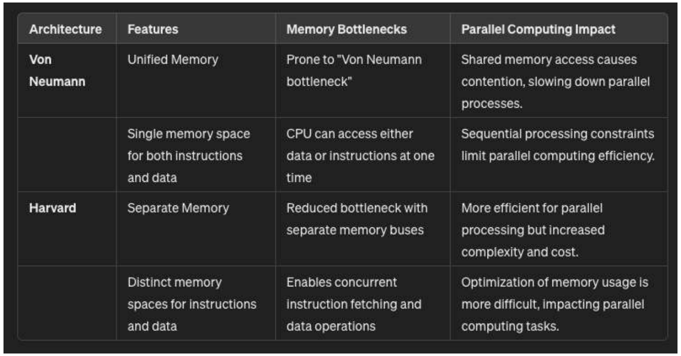
Feature	PyTorch	Python	CUDA
Primary Use	Deep learning library optimized for GPU acceleration, especially for neural network training.	General-purpose programming language with support for parallel computing through libraries.	Platform and API for parallel computing directly on NVIDIA GPUs, tailored for high-performance computing needs.
Parallelism Type	Primarily data parallelism, with built-in support for automatic distribution of computations and data across multiple GPUs.	Supports both task-based and data parallelism via external libraries like multiprocessing, threading, and more specialized libraries like Dask.	Supports fine-grained data parallelism and task parallelism directly at the hardware level, allowing detailed control over GPU resources.
Ease of Use	High-level abstractions for deep learning models make it user-friendly for developers focused on AI applications.	Easy to use for general programming, but parallel programming can require more complex setup and understanding of external libraries.	Requires understanding of GPU architectures and parallel programming concepts, generally steeper learning curve than Python.
Integration	Seamless integration with Python, allowing use of both PyTorch and Python libraries efficiently together.	Wide range of libraries for integration with various APIs and systems for parallel computing enhancements.	Integrates with languages like C++ for writing CUDA kernels, and libraries are available for integration with Python (e.g., PyCUDA).
Performance	Optimized for GPU-based matrix operations and large-scale data processing, suitable for high-performance AI computations.	Performance depends significantly on the use of external libraries and the nature of the parallel computing tasks.	Provides highest performance for GPU computations by leveraging direct access to GPU hardware features.

Performance Optimization

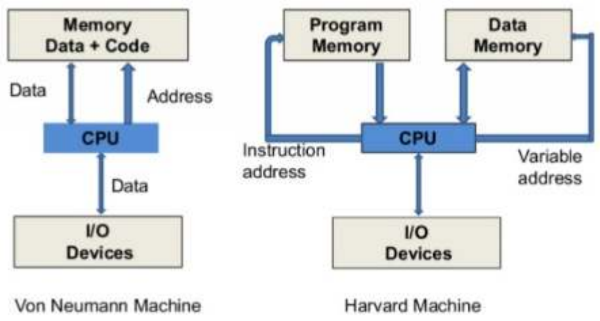
maximize resource utilization and minimize idle time, crucial for maintaining optimal performance in parallel computing environments.

- Minimizing Communication Overhead: Reduces the frequency and volume of data exchange between processes or threads, since communication can be a significant bottleneck, particularly in distributed systems. Techniques include using local data wherever possible and optimizing communication protocols.
- Optimal Use of Memory Hierarchy: Involves arranging data and computations to take advantage of various levels of memory (registers, cache, RAM) efficiently. This includes techniques like tiling and cache-friendly algorithms that minimize cache misses and reduce memory access times.
- Fine-Tuning Task Granularity: Adjusts the size of tasks assigned to each thread or process to achieve a balance between overhead and computational efficiency. Smaller tasks might reduce computation time but increase scheduling and communication overhead, while larger tasks might underutilize system resources.
- Profiling and Analysis Tools: Utilizes specialized tools to identify bottlenecks and inefficiencies in parallel applications. Profiling tools help in understanding the behavior of an application regarding CPU and GPU

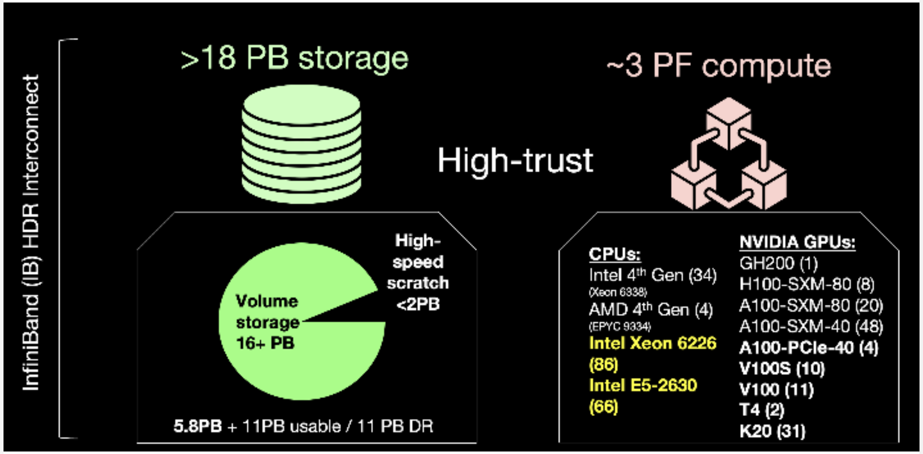
Architectural Limitations (Harvard vs Neumann)

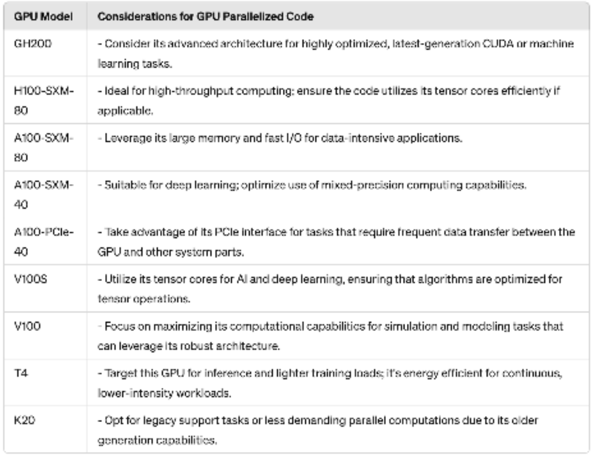


Architecture	Features	Memory Bottlenecks	Parallel Computing Impact
Von Neumann	Unified Memory	Prone to "Von Neumann bottleneck"	Shared memory access causes contention, slowing down parallel processes.
	Single memory space for both instructions and data	CPU can access either data or instructions at one time	Sequential processing constraints limit parallel computing efficiency.
Harvard	Separate Memory	Reduced bottleneck with separate memory buses	More efficient for parallel processing but increased complexity and cost.
	Distinct memory spaces for instructions and data	Enables concurrent instruction fetching and data operations	Optimization of memory usage is more difficult, impacting parallel computing tasks.



Our Hardware





GPU Model	Considerations for GPU Parallelized Code
GH200	- Consider its advanced architecture for highly optimized, latest-generation CUDA or machine learning tasks.
H100-SXM-80	- Ideal for high-throughput computing; ensure the code utilizes its tensor cores efficiently if applicable.
A100-SXM-80	- Leverage its large memory and fast I/O for data-intensive applications.
A100-SXM-40	- Suitable for deep learning; optimize use of mixed-precision computing capabilities.
A100-PCIe-40	- Take advantage of its PCIe interface for tasks that require frequent data transfer between the GPU and other system parts.
V100S	- Utilize its tensor cores for AI and deep learning, ensuring that algorithms are optimized for tensor operations.
V100	- Focus on maximizing its computational capabilities for simulation and modeling tasks that can leverage its robust architecture.
T4	- Target this GPU for inference and lighter training loads; it's energy efficient for continuous, lower-intensity workloads.
K20	- Opt for legacy support tasks or less demanding parallel computations due to its older generation capabilities.

Parallel Programming Challenges

- Complexity of Design and Debugging: Parallel programs are inherently more complex than sequential ones, making them more challenging to design, debug, and test. Race conditions, deadlocks, and data inconsistencies require sophisticated debugging tools and techniques.
- Scalability Issues: Efficiently scaling a parallel program across different architectures and larger datasets can be challenging due to issues like synchronization overhead, communication delays, and uneven workload distribution.
- Dependency Management: Managing data dependencies and ensuring correct execution order in parallel tasks can be difficult, particularly when tasks are interdependent. This requires careful planning and synchronization, which can introduce performance overhead.
- Optimal Resource Utilization: Maximizing the use of available hardware resources (CPU cores, memory bandwidth, network bandwidth) requires fine-tuning and often bespoke solutions for different hardware platforms, complicating the development and deployment processes.

Sign Up for the next Seminar