Washington University in St. Louis

SCHOOL OF MEDICINE

MIR Mallinckrodt Institute of Radiology

RCIF Applied HPC Seminar Series

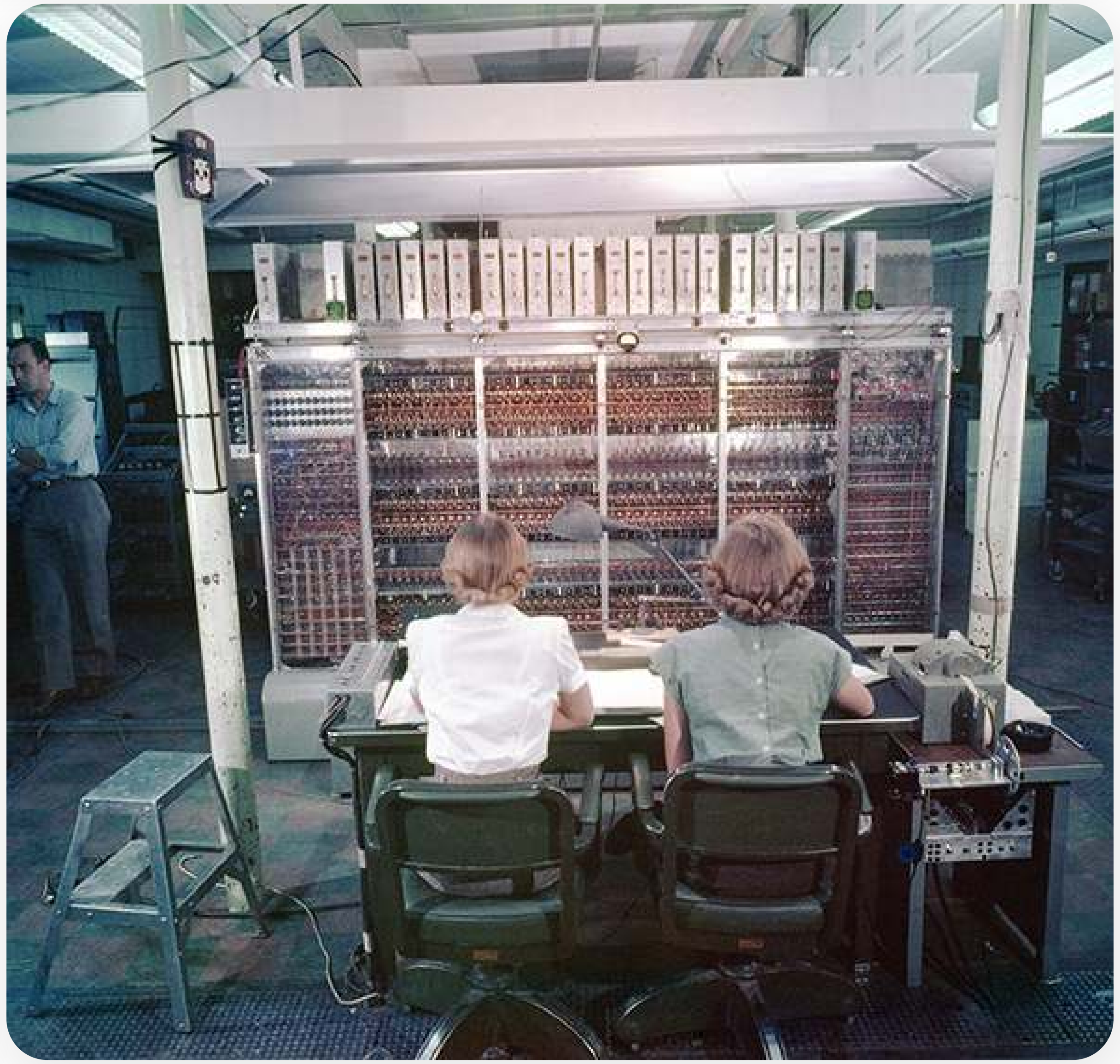
Introduction to High Performance Computing

Introduction to High Performance Computing

- High Performance Computing: Relies on utilizing parallel computing or computing at scale to solve complex computational problems
- 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 HPC

- 1940s-50s: Early electronic computers
  - ENIAC (1945): First general-purpose electronic computer
  - UNIVAC I (1951): First commercial computer in the US
- 1960s: Dawn of supercomputers
  - CDC 6600 (1964): Considered the first supercomputer
  - IBM System/360 (1964): Introduced concept of computer families
- 1970s: Vector processing era
  - Cray-1 (1976): Revolutionary vector processor design
  - ILLIAC IV (1976): One of the first parallel computing systems
- 1980s: Massively parallel systems
  - Connection Machine (1985): Introduced massively parallel processing
  - Intel iPSC/1 (1985): Early commercial hypercube computer



Part Deux

- 1990s: Rise of cluster computing
  - Beowulf clusters (1994): Low-cost supercomputing with commodity hardware
  - ASCI Red (1997): First teraflop supercomputer
- 2000s: Petascale and GPU acceleration
  - IBM Blue Gene/L (2004): Breakthrough in power-efficient supercomputing
  - NVIDIA Tesla GPUs (2007): General-purpose GPU computing
- 2010s: Exascale initiatives
  - Tianhe-2 (2013): First to exceed 30 petaflops
  - Summit (2018): First to exceed 100 petaflops
- Present and near future:
  - Fugaku (2020): First arm-based supercomputer to top TOP500
  - Frontier (2022): First exascale supercomputer
  - Quantum and neuromorphic computing research intensifies

What is the RCIF CHPC?

Hint: We're not RIS...

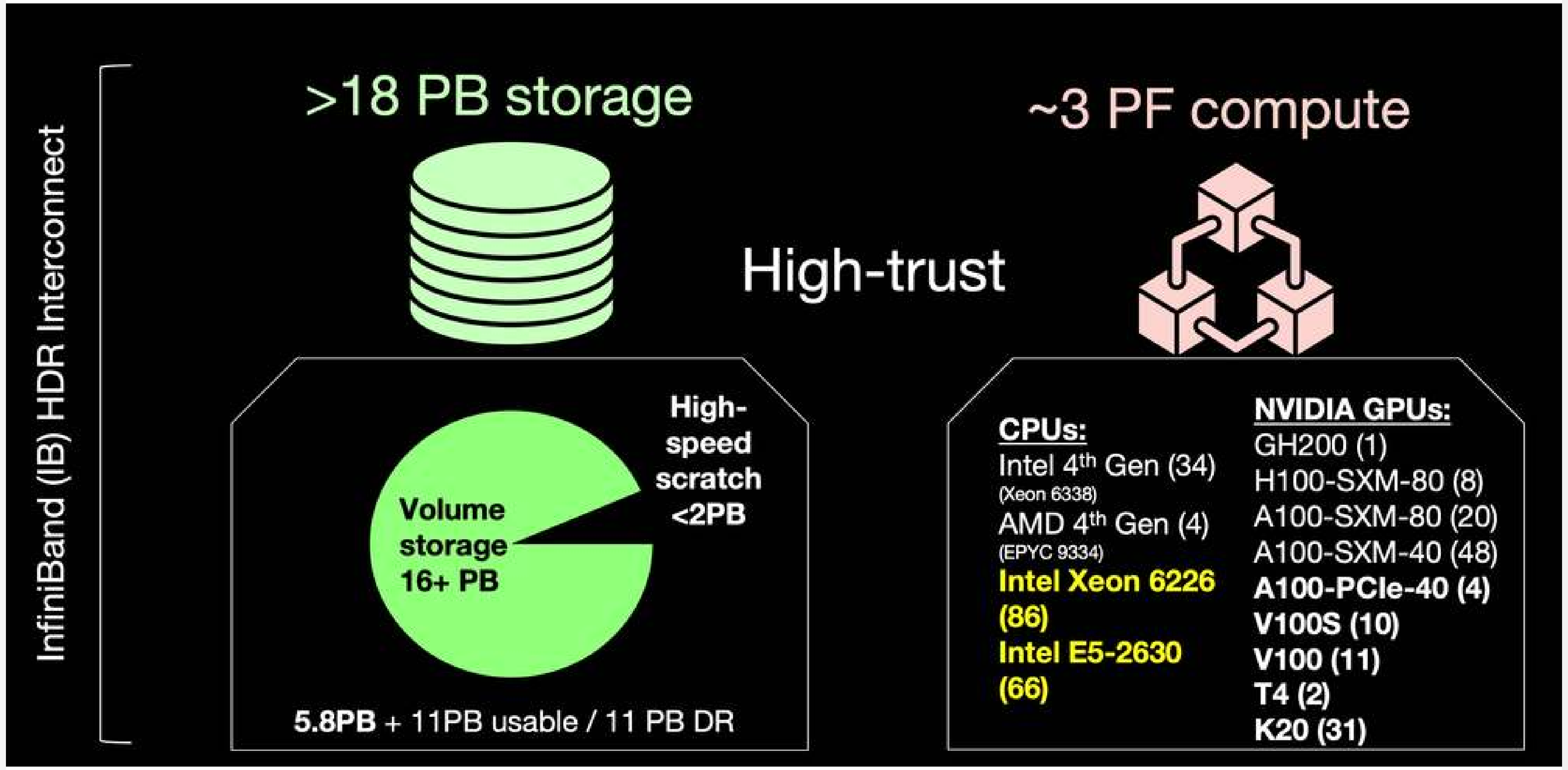
White Glove Support

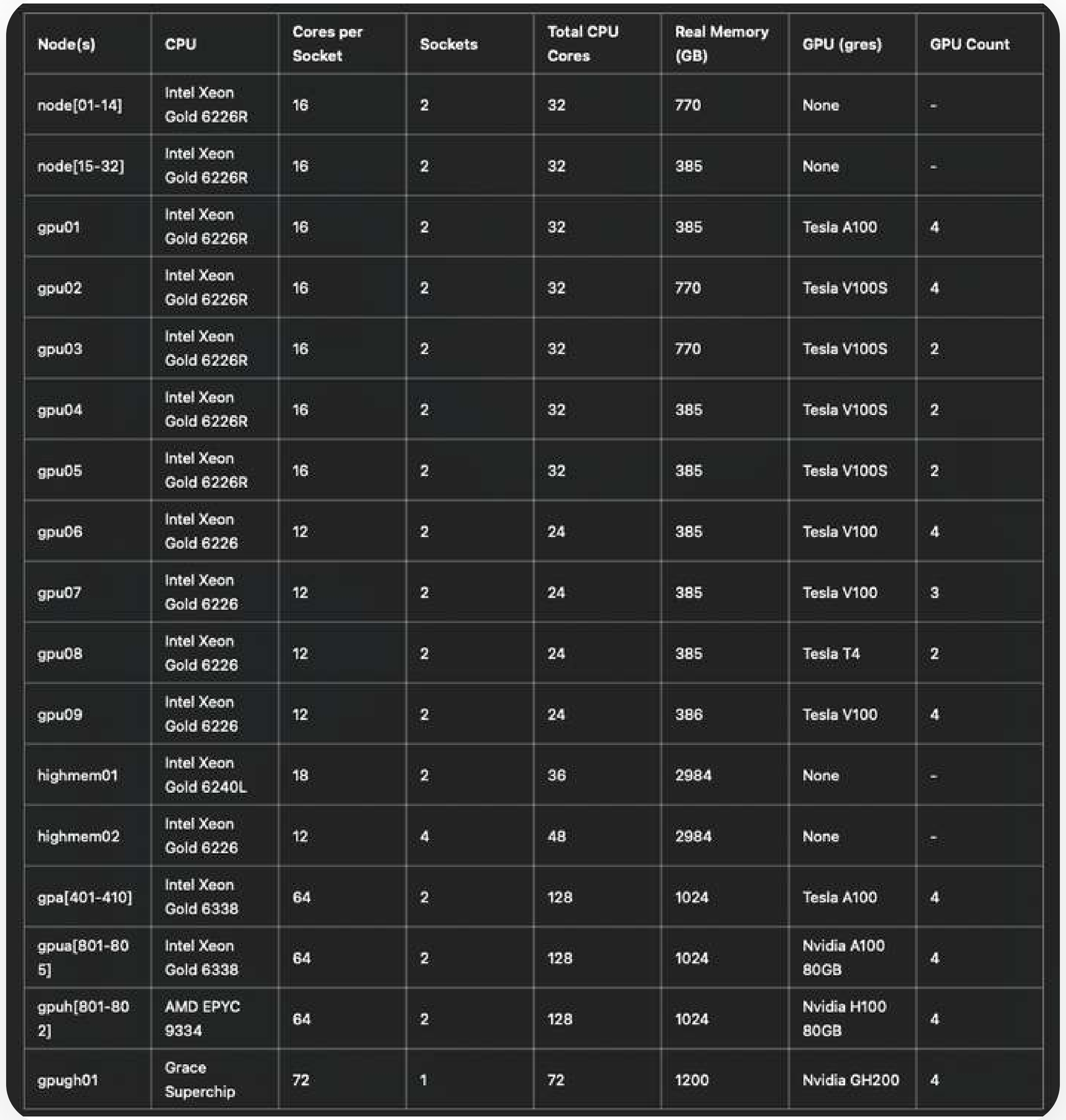
- Dedicated support staff available for consultation
- Assistance with applications, software, and workflows
- On-demand support via Slack, Zoom, and email via support@rcif.wustl.edu
- Let us manage your hardware (co-locate your servers in the CHPC)
- We are focused on helping users take their first steps to go from 0 to 1 on their projects

Shared Datasets

- HCP Young Adult: Data from 1200 healthy young adults, including MR imaging and behavioral data, accessible via ConnectomeDB.
- OASIS Series: Multiple studies on aging and Alzheimer's disease, providing cross-sectional and longitudinal MRI data, available for access through the OASIS website.
- UK Biobank: Includes a diverse array of data from 500,000 UK adults, covering brain, heart, and abdominal MRI, alongside genetic and phenotypic data.
- TCIA: Focuses on cancer imaging with datasets like Breast-Cancer-Screening-DBT, Duke-Breast-Cancer-MRI, and others, detailing patient outcomes, treatment, and genomics linked to imaging data.
- BJC Bulk Clinical Data Sets: Clinical datasets have been retrieved from the BJC Picture archiving and communication systems (PACS) for analysis.
- Connectomes Related to Human Disease (CRHD) Projects: CRHD projects apply HCP-style data collection protocols toward subject cohorts at risk for, or suffering from, diseases or disorders affecting the brain, with a goal of providing comparable data to healthy HCP subjects across the lifespan.
- CT-RATE: Textual data paired with imagery data comprised of chest CT volumes paired with corresponding radiology text reports, multi-abnormality labels, and metadata.
- New Datasets added frequently…
- Requesting Data: Data access requires registration and agreement to data use terms through respective portals for each dataset series.

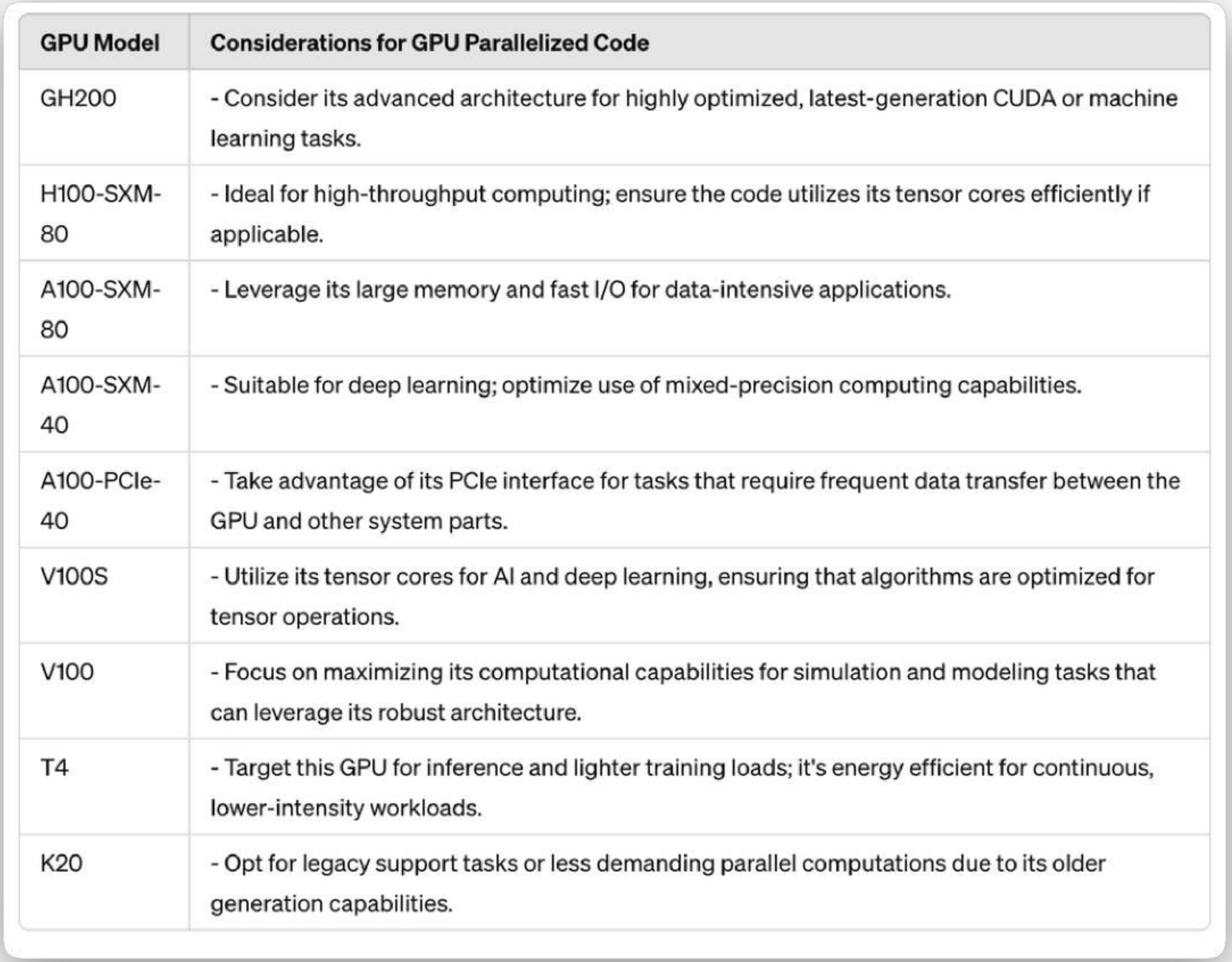
Our Hardware





Node(s)	CPU	Cores per Socket	Sockets	Total CPU Cores	Real Memory (GB)	GPU (gres)	GPU Count
node[01-14]	Intel Xeon Gold 6226R	16	2	32	770	None	-
node[15-32]	Intel Xeon Gold 6226R	16	2	32	385	None	-
gpu01	Intel Xeon Gold 6226R	16	2	32	385	Tesla A100	4
gpu02	Intel Xeon Gold 6226R	16	2	32	770	Tesla V100S	4
gpu03	Intel Xeon Gold 6226R	16	2	32	770	Tesla V100S	2
gpu04	Intel Xeon Gold 6226R	16	2	32	385	Tesla V100S	2
gpu05	Intel Xeon Gold 6226R	16	2	32	385	Tesla V100S	2
gpu06	Intel Xeon Gold 6226	12	2	24	385	Tesla V100	4
gpu07	Intel Xeon Gold 6226	12	2	24	385	Tesla V100	3
gpu08	Intel Xeon Gold 6226	12	2	24	385	Tesla T4	2
gpu09	Intel Xeon Gold 6226	12	2	24	386	Tesla V100	4
highmem01	Intel Xeon Gold 6240L	18	2	36	2984	None	-
highmem02	Intel Xeon Gold 6226	12	4	48	2984	None	-
gpa[401-410]	Intel Xeon Gold 6338	64	2	128	1024	Tesla A100	4
gpua[801-805]	Intel Xeon Gold 6338	64	2	128	1024	Nvidia A100 80GB	4
gpuh[801-802]	AMD EPYC 9334	64	2	128	1024	Nvidia H100 80GB	4
gpugh01	Grace Superchip	72	1	72	1200	Nvidia GH200	4

The Best Hardware for the Job



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.

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

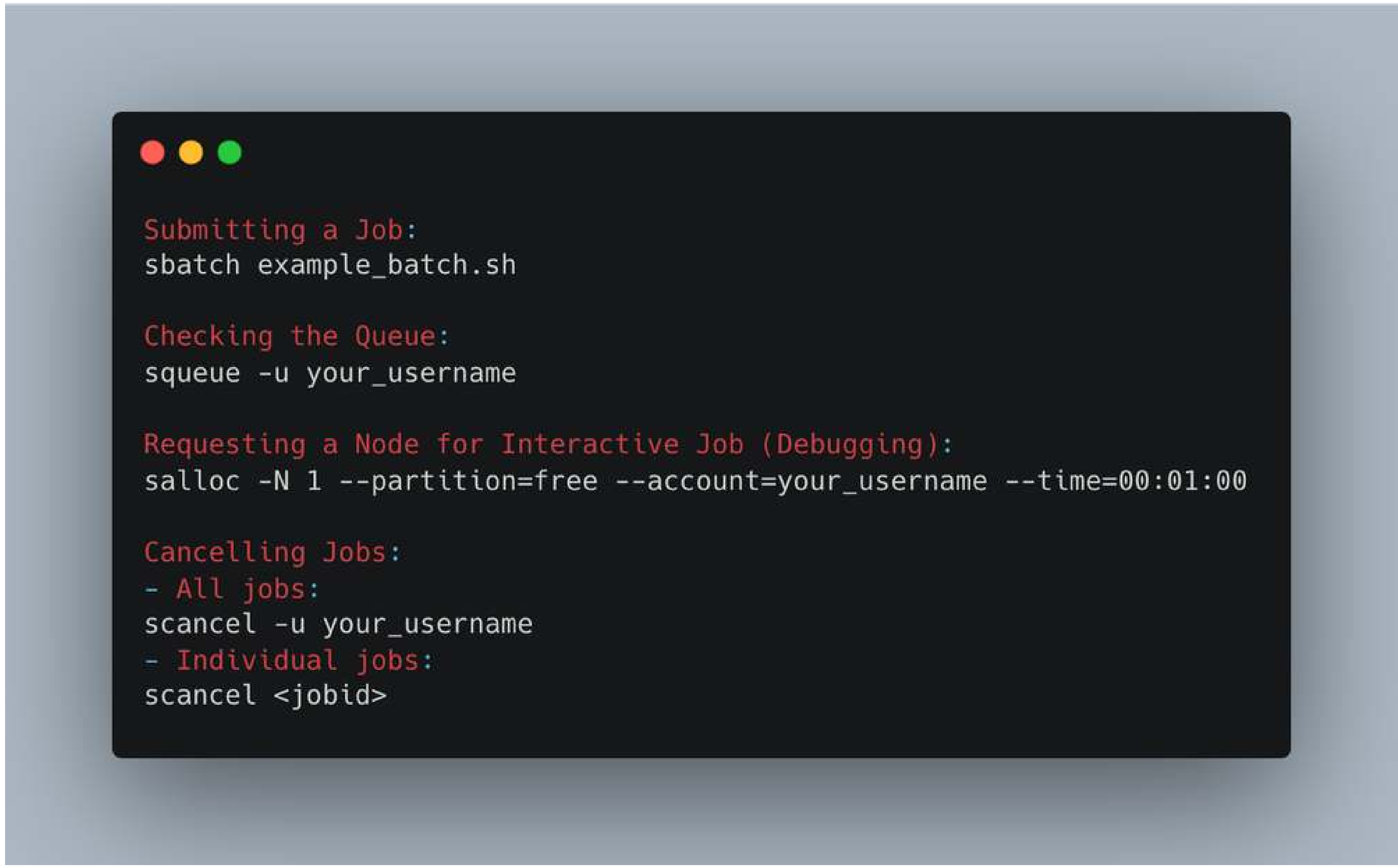
- Massive Parallelism: GPUs contain hundreds to thousands of smaller, efficient cores designed for executing multiple parallel tasks simultaneously, ideal for operations that can be broken down into small, independent tasks.
- Specialized Hardware: GPUs are optimized for compute-intensive, highly parallel 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.

How Can You Use These Amazing Systems???

Request an Account!

I Have An Account, Now What?

- Migrate your code/env
- Submit a job
- Analyze the data

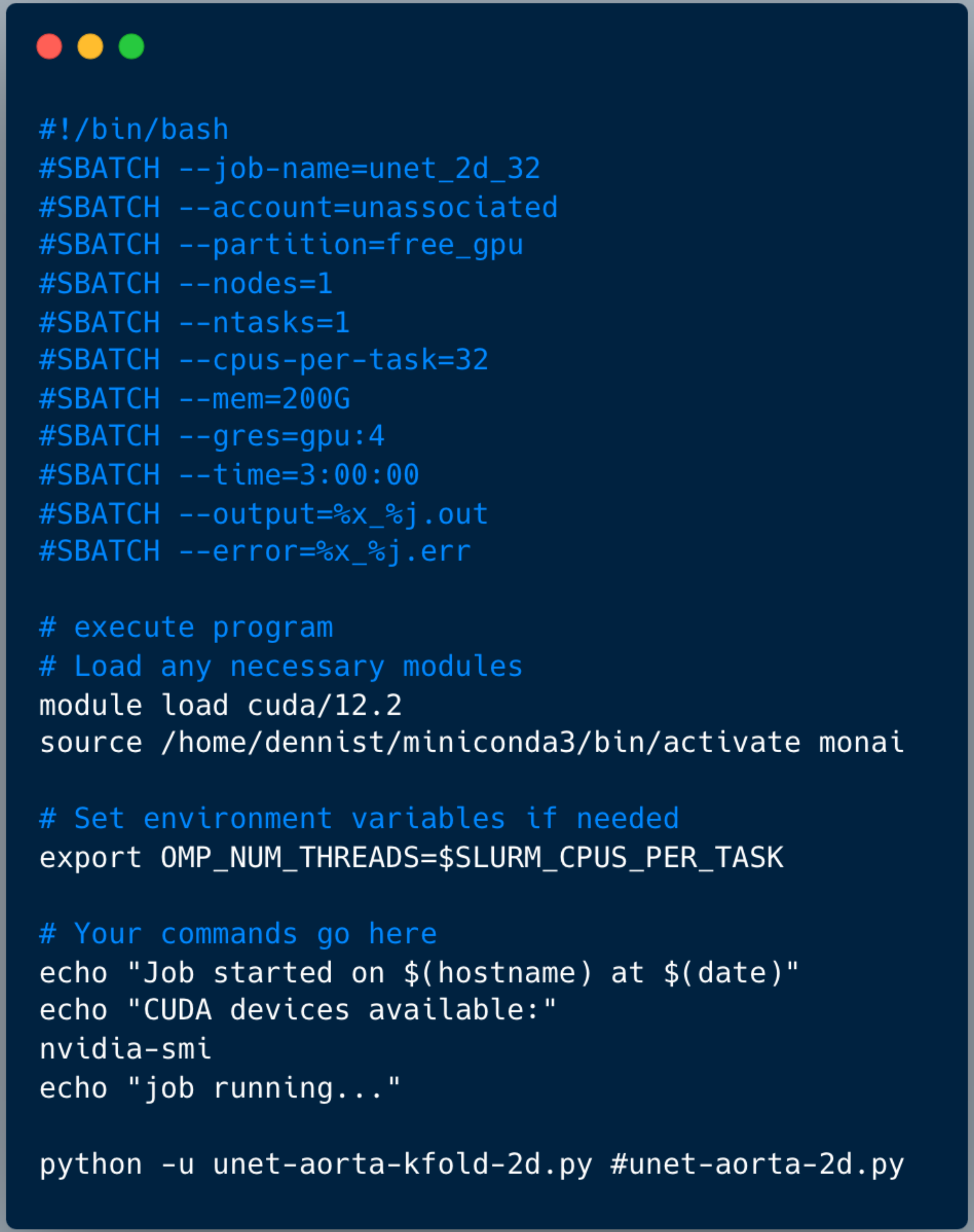


```
Submitting a Job:
sbatch example_batch.sh

Checking the Queue:
squeue -u your_username

Requesting a Node for Interactive Job (Debugging):
salloc -N 1 --partition=free --account=your_username --time=00:01:00

Cancelling Jobs:
- All jobs:
scancel -u your_username
- Individual jobs:
scancel <jobid>
```

Check out the Docs

Need More Help?

Let's Submit a Job…

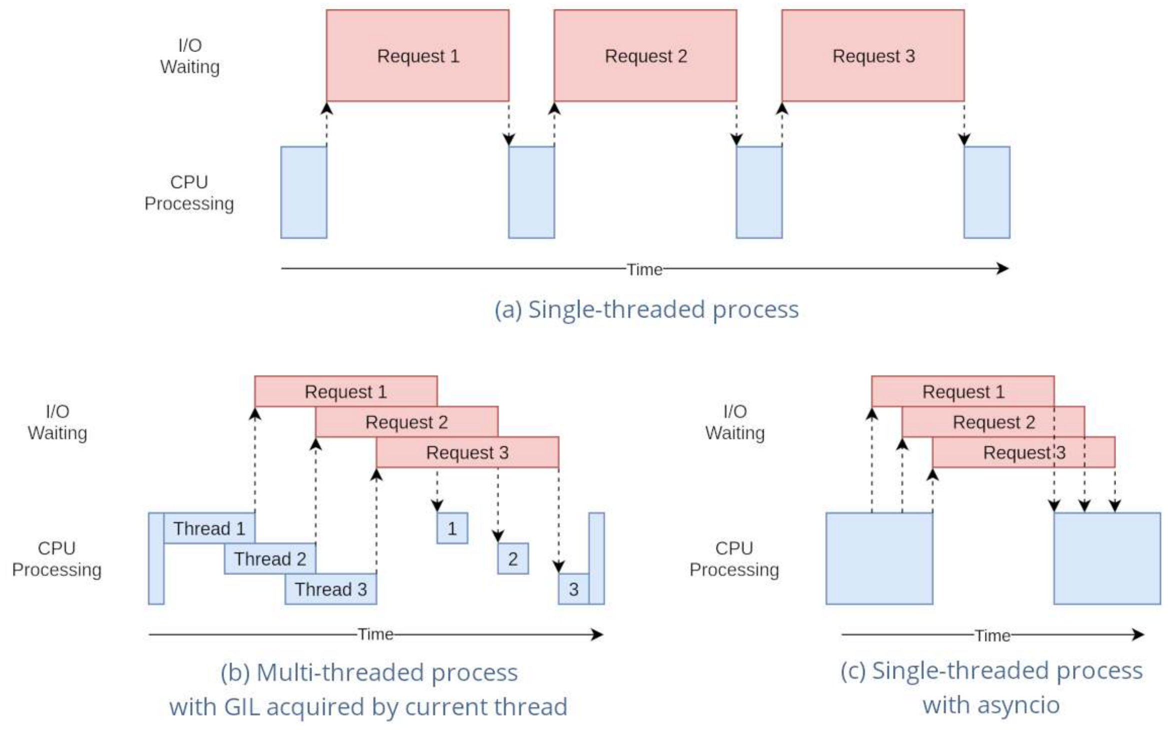


```bash
#!/bin/bash
#SBATCH --job-name=unet_2d_32
#SBATCH --account=unassociated
#SBATCH --partition=free_gpu
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=32
#SBATCH --mem=200G
#SBATCH --gres=gpu:4
#SBATCH --time=3:00:00
#SBATCH --output=%x_%j.out
#SBATCH --error=%x_%j.err

# execute program
# Load any necessary modules
module load cuda/12.2
source /home/dennist/miniconda3/bin/activate monai

# Set environment variables if needed
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

# Your commands go here
echo "Job started on $(hostname) at $(date)"
echo "CUDA devices available:"
nvidia-smi
echo "job running..."

python -u unet-aorta-kfold-2d.py #unet-aorta-2d.py
```

Parallel Computing Techniques

Multithreading Techniques

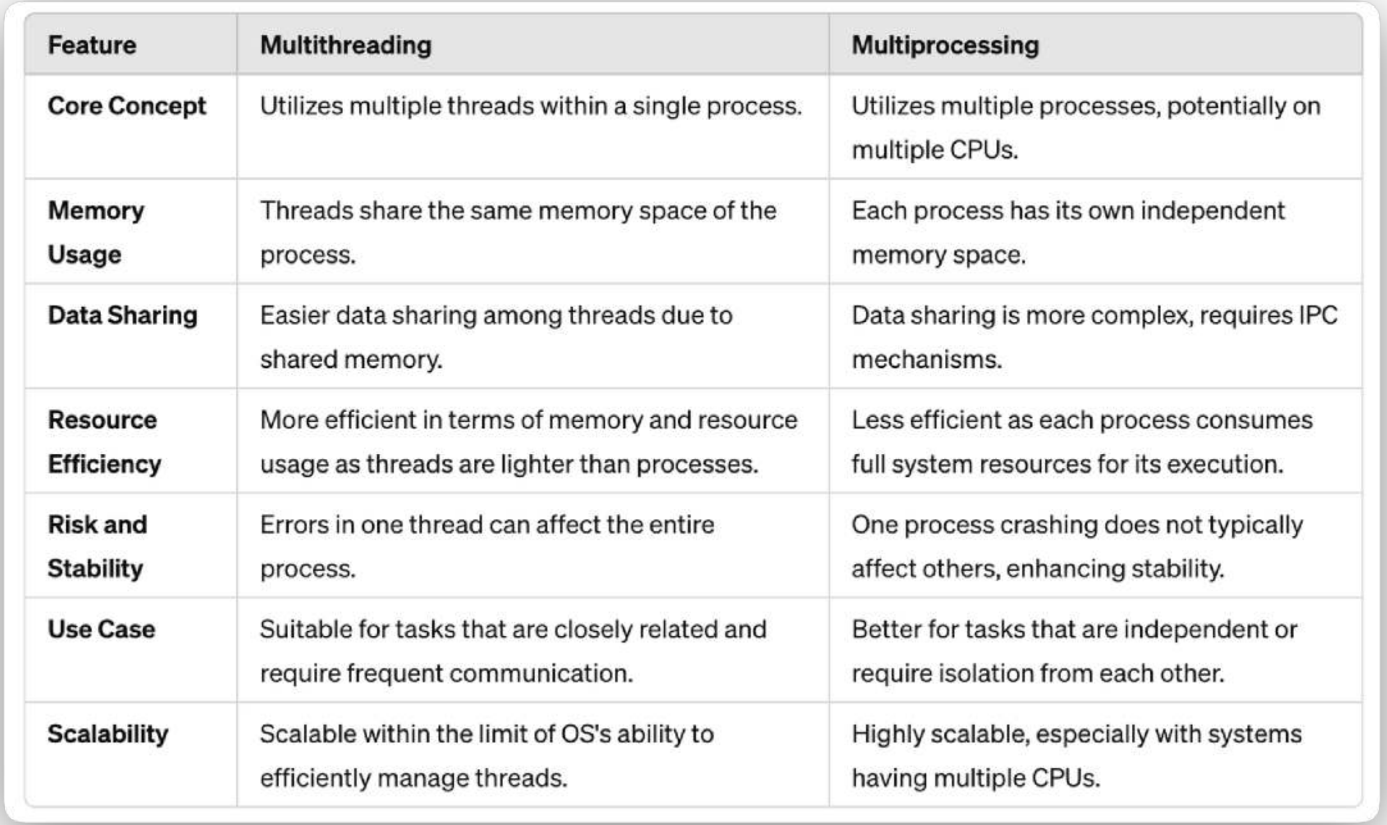
- Concurrent Execution: Multithreading allows multiple threads to run concurrently within a single process, sharing the process's resources but operating independently to enhance execution efficiency and responsiveness.
- Resource Sharing: Threads within the same process share memory and resources, facilitating 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

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.

Multithreading vs Multiprocessing on CPU

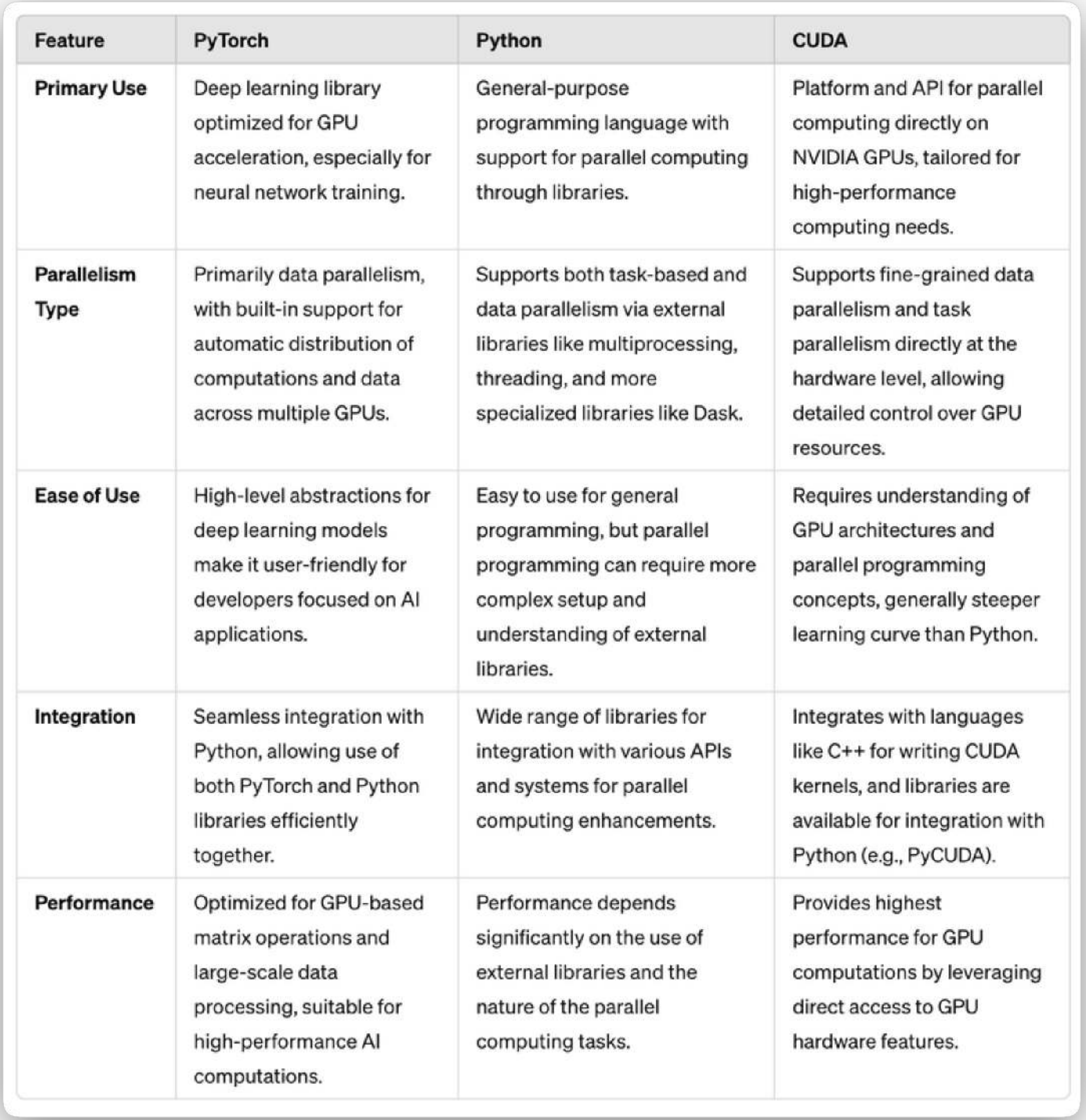


Feature	Multithreading	Multiprocessing
Core Concept	Utilizes multiple threads within a single process.	Utilizes multiple processes, potentially on multiple CPUs.
Memory Usage	Threads share the same memory space of the process.	Each process has its own independent memory space.
Data Sharing	Easier data 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.

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.

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

- Load Balancing: Ensures even distribution of workload across all available processing units (CPUs or GPUs) to 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 utilization, memory access patterns, and communication overhead, guiding targeted optimizations.

Sign Up for the next Seminar