Now, we will explore the contents of the container by first pulling the appropriate image version 13.10. Then, after allocating either H100 or A100 GPU using `salloc`, we will run the container, navigate to the workspace directory, and list its contents, as shown in Figure 4.2. This directory includes several NVIDIA benchmark files and executables, among which we will focus on the HPL and STREAM benchmarks to evaluate computational performance and memory bandwidth.

```
$ singularity pull docker://nvcr.io/nvidia/hpc-benchmarks:23.10
$ singularity run --nv hpc-benchmarks_23.10.sif
$ cd /workspace/
$ ls
NVIDIA_Deep_Learning_Container_License.pdf  hpcg-linux-x86_64  hpcg.sh  hpl-linux-x86_64  hpl-mxp-linux-
x86_64  hpl-mxp.sh  hpl.sh  stream-gpu-linux-x86_64  stream-gpu-test.sh  third_party.txt
```

Figure 4.2. *Bechnmarks' Container Content*

`Important Note: All the benchmarks were run on a single GPU node.`

4.4 *HPL Benchmark*

HPL is a fundamental benchmark in HPC that evaluates a system's floating-point computational performance. It is widely used to assess the efficiency of both CPUs and GPUs when solving large dense linear systems. HPL not only measures raw performance in terms of gigaflops or teraflops, but it also helps to understand how well the system handles parallel computation, memory access patterns, and communication overhead. This makes it an essential tool for performance tuning and system comparison in HPC environments.

4.4.1 Characteristics of the HPL Benchmark

High-Performance Linpack (HPL) is a benchmark designed to measure the computational speed of systems when solving a dense system of linear equations. It is one of the most widely used benchmarks for evaluating the performance of high-performance computing (HPC) systems and serves as a key metric for ranking supercomputers in the TOP500 list.

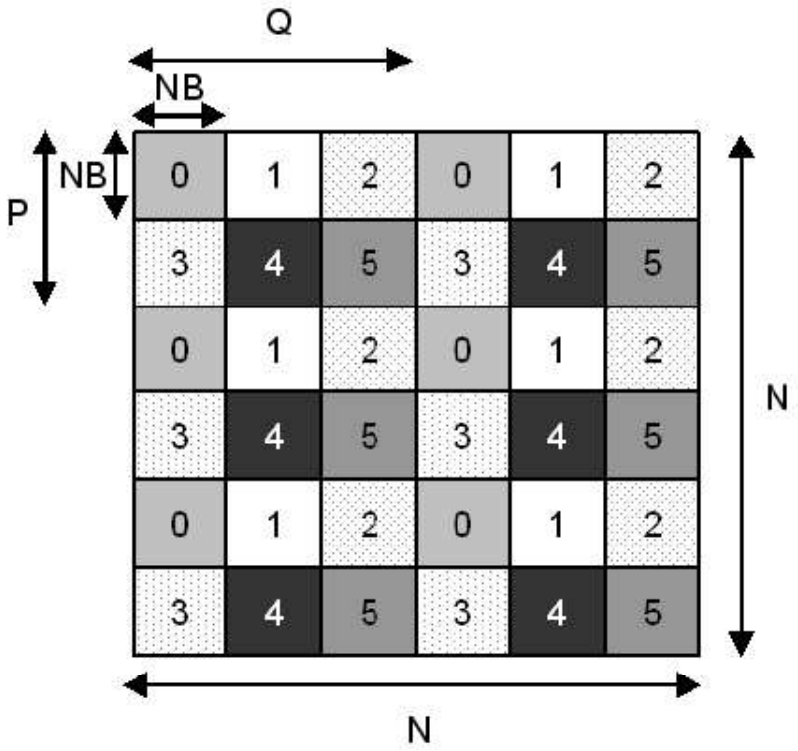

Figure 4.3. *HPL Parameters [20]*

Figure 4.3 displays the parameters of the HPL benchmark. `N` represents the global matrix size, and `NB` denotes the block size. These blocks form the basic unit of computation and are used to decompose the matrix into smaller tiles. The distribution is done in a 1D block-cyclic manner, which ensures a balanced workload among processes and minimizes communication overhead. The parameters `P` and `Q` define the dimensions of the process grid, where each MPI process is assigned multiple blocks based on its coordinates in the grid. For optimal performance, the product `P` $\times$ `Q` should match the total number of MPI processes, and `P` and `Q` should be chosen to be as balanced as possible.

4.4.2 HPL Benchmark Input File

The High-Performance Linpack (HPL) benchmark requires a configuration file, typically named `HPL.dat`, which specifies execution parameters such as matrix sizes, block sizes, process grids, and algorithmic options. This plain-text file has the extension `.dat` and controls the benchmark execution. An example is shown in Figure 4.4.

```
1  HPLinpack benchmark input file
2  Innovative Computing Laboratory, University of Tennessee and Frankfurt Institute for Advanced Studies
3  HPL.out      output file name (if any)
4  6            device out (6=stdout,7=stderr,file)
5  5        # of problems sizes (N)
6  20000 40000 60000 80000 100000      Ns
7  1            # of NBs
8  576           NBs
9  1            PMAP process mapping (0=Row-,1=Column-major)
10 1            # of process grids (P x Q)
11 2            Ps
12 1            Qs
13 0.1          threshold
14 1            # of panel fact
15 1            PFACTs (0=left, 1=Crout, 2=Right)
16 1            # of recursive stopping criterium
17 64           NBMINs (>= 1)
18 1            # of panels in recursion
19 2            NDIVs
20 1            # of recursive panel fact.
21 0            RFACTs (0=left, 1=Crout, 2=Right)
22 1            # of broadcast
23 6            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM,6=MPI)
24 1            # of lookahead options
25 3            LOOKAHEADs (enable = 1)
26 8            memory alignment in double (> 0)
27 100           Seed for the matrix generation
```

Figure 4.4. *HPL.dat [21]*

The main parameters of `HPL.dat` are:

- **Header (Lines 1 to 2):** Informational text, usually left as default.
- **Output (Lines 3 to 4):** Defines output file (e.g., `HPL.out`) and destination (stdout, stderr, or file).
- **Problem Sizes (Lines 5 to 6):** Number and values of matrix sizes to test (e.g., 3000, 6000, 10000).
- **Block Sizes (Lines 7 to 8):** Number and values of block sizes (e.g., 80, 100, 160).
- **Process Grid (Lines 9 to 11):** Defines row/column process mapping and grid dimensions ($P \times Q$).
- **Threshold (Line 13):** Residual check tolerance, typically set to 16.0 or negative to bypass.

- **Algorithmic Parameters (Lines 14+):** Options for panel factorizations (PFACT, RFACT), recursion depth (NDIV), stopping criteria (NBMIN), and broadcast methods.
- **Lookahead (Line 15):** Enables overlapping of computation and communication.
- **Memory Alignment (Line 16):** Boundary alignment in doubles (e.g., 8 = 64 bytes).
- **Random Seed (Line 17):** Ensures reproducibility of generated matrices.

In practice, HPL explores all combinations of these parameters to identify the optimal setup for a given system.

4.4.3 Execution Steps

To run the HPL (High Performance Linpack) benchmark, follow these steps: First, navigate to your desired working directory (replace `/path/to/working/directory` with the actual location). Create a new directory named `data` to store benchmark-related files, and move into this newly created directory. Then, create an empty `HPL.dat` file, which will store the configuration parameters for the benchmark, and open it using a text editor (e.g., Vim) to modify and define the necessary parameters.

```
$ cd /path/to/working/directory
$ mkdir data
$ cd data/
$ touch HPL.dat
$ vim HPL.dat
```

Figure 4.5. *HPL Step 1 Execution*

After setting up the `HPL.dat` file, we proceed with executing the benchmark using the appropriate configurations. The `HPL.dat` file contains key parameters that define the execution settings of the benchmark, including problem sizes, block sizes, process grid dimensions, and algorithmic parameters. These configurations must be carefully selected based on the available hardware resources to optimize performance.

Next, bind the local directory on the host machine to a directory inside the Singularity container. This ensures that files like `HPL.dat`, which reside on the host system, are accessible within the container. The `pwd` command prints the current working directory, which is assumed to contain the necessary data file (`HPL.dat`). Use the `singularity exec` command with the `-bind` flag to bind the local directory to the container's `/mnt` directory. You can then display the contents of `/mnt/HPL.dat` inside the container to make sure of the successfull binding.

```
$ singularity exec --bind $(pwd):/mnt ./hpc-benchmarks_23.10.sif cat /mnt/HPL.dat
```

Figure 4.6. *HPL Step 2 Execution*

If the session is closed, the bind must be re-executed each time; to avoid this, you can add the bind command to your `.bashrc` file, which runs automatically for new terminal sessions using the environment variable `$SINGULARITY_BINDPATH`.

```
$ echo export SINGULARITY_BINDPATH=/home/karim.bezine-ext/lustre/sw_stack-
373lcd9r8io/users/karim.bezine-ext/src/data:/mnt >> ~/.bashrc
$ source ~/.bashrc
```

Figure 4.7. *HPL Step 3 Execution*

Before running any GPU-accelerated benchmarks, allocate GPU resources. This ensures that the system reserves a node with a GPU available for your benchmark. You can verify the GPU allocation by checking the NVIDIA driver status.

Finally, navigate to the directory where the HPL benchmark files are located inside the container after the successful binding. Run the benchmark using MPI, specifying the number of processes (e.g., `-np 1` for a single process) and the location of the `HPL.dat` configuration file. Once executed, the benchmark begins its HPL test, performing matrix multiplications to measure the system's floating-point performance.

```
$ singularity run --nv hpc-benchmarks_23.10.sif
$ cd /workspace/hpl-linux-x86_64/
$ mpirun -np 1 hpl.sh --dat /mnt/HPL.dat --gpu-affinity 2 --no-multinode --cuda-compat
```

Figure 4.8. *HPL Step 4 Execution*

- **mpirun -np 1**: Runs an MPI process with a single instance.
- **hpl.sh**: The script that launches the HPL-NVIDIA benchmark.
- **–dat /mnt/HPL.dat**: Specifies the path to the HPL.dat configuration file.
- **–gpu-affinity 0**: Ensures execution runs on GPU 0.
- **–no-multinode**: Disables multi-node execution for a single-node setup.
- **–cuda-compat**: Enables CUDA forward compatibility for running binaries compiled with older CUDA versions.

4.4.4 A100 GPU Partition

a) Single GPU, Single MPI Process

To run the HPL benchmark on a single GPU, we allocate a GPU node with one task using the following `salloc` command:

```
salloc -t 4:00:00 -n 1 --ntasks=1 -A sw_stack-373lcd9r8io-default-gpu
-p gpu --gres=gpu:1
```

Each part of the command has a specific role:

- `salloc`: Allocates resources for an interactive Slurm job.
- `-t 4:00:00`: Sets the maximum job runtime to 4 hours.
- `-n 1`: Requests 1 node for the job.

- `-ntasks=1`: Requests 1 MPI task, ensuring a single process.
- `-A sw_stack-373lcd9r8io-default-gpu`: Specifies the account/project to charge the job to.
- `-p gpu`: Requests the job to run in the gpu partition (group of A100 GPU nodes).
- `-gres=gpu:1`: Requests 1 GPU on the allocated node.

This ensures that the benchmark runs on a single A100 GPU on one node with one MPI process. Below is a table summarizing the results from five different test cases with varying problem sizes ('N'). The results include the time taken for each test and the achieved computation speed in Gflops (Billions of Floating-Point Operations Per Second):

This table demonstrates how the time taken and the computation speed change as the problem size increases. It is important to note that larger problem sizes (higher 'N' values) typically result in higher computation speeds, though they also take longer to process.

The HPL benchmark is optimized to use FP64 Tensor Cores by default when running on A100-SXM4-80GB GPUs. These specialized cores significantly boost double-precision floating-point performance, enabling higher efficiency in large-scale numerical computations.

For the NVIDIA A100 GPU, the theoretical peak for FP64 operations is calculated as follows:

$$\text{Theoretical Peak (GFLOPS)} = \text{Number of CUDA Cores} \times 2 \times \text{Boost Clock (GHz)}$$

- Number of CUDA cores: 6912 - Boost clock: 1.41 GHz

$$\text{Theoretical Peak} = 6912 \times 2 \times 1.41 \approx 19,487 \text{ GFLOPS} \approx 19.5 \text{ TFLOPS}$$

The efficiency of a benchmark run can be calculated as:

$$\text{Efficiency (\%)} = \frac{\text{Measured Gflops}}{\text{Theoretical Peak Gflops}} \times 100$$

Using the result for the largest problem size (‘N=100000‘) with a measured computation speed of 17,860 Gflops:

$$\text{Efficiency} = \frac{17,860}{19,487} \times 100 \approx 91.7\%$$

This confirms that the HPL benchmark is achieving a high fraction of the theoretical peak performance of the A100 GPU, indicating that the system is well-optimized for double-precision operations.

Table 4.1. *HPL Results on one A100 GPU*

Problem Size, N	Time Taken (s)	Computation Speed (Gflops)
20000	0.51	9,731
40000	1.63	15,890
60000	8.36	17,150
80000	19.01	17,600
100000	36.94	17,860

b) Two GPUs, Two MPI Processes, One MPI Process Each

To utilize two A100 GPUs efficiently, we allocate a GPU node with two GPUs and launch one MPI process per GPU. This setup allows the problem to be distributed evenly across both devices, maximizing throughput.

The Slurm command for this configuration is:

```
salloc -t 4:00:00 -n 1 --ntasks-per-gpu=1 --gres=gpu:2 \
-A sw_stack-373lcd9r8io-default-gpu -p gpu
```

Each MPI process is bound to a separate GPU, ensuring that both devices operate independently on their respective data partitions in VRAM. The performance measurements from the HPL benchmark in this configuration are shown in Table 4.2.

Table 4.2. *HPL results on two A100 GPUs*

Problem Size, N	Time Taken (s)	Computation Speed (Gflops)
20000	0.55	9,106
40000	1.66	25,230
60000	4.56	31,430
80000	9.98	33,560
100000	19.00	34,720

The combined theoretical FP64 peak for two A100 GPUs is simply twice the single-GPU peak:

$$\text{Theoretical Peak}_{\text{2xA100}}(\text{GFLOPS}) = 2 \times \text{CUDA Cores} \times 2 \times \text{Boost Clock (GHz)}.$$

Using the same device characteristics as before (6912 CUDA cores, 1.41 GHz boost):

$$\text{Theoretical Peak}_{\text{1xA100}} \approx 6912 \times 2 \times 1.41 \approx 19{,}487 \text{ GFLOPS},$$

$$\text{Theoretical Peak}_{\text{2xA100}} \approx 2 \times 19{,}487 = 38{,}974 \text{ GFLOPS } (\approx 39.0 \text{ TFLOPS}).$$

The efficiency for the largest problem size ($N = 100{,}000$) is then:

$$\text{Efficiency } (\%) = \frac{\text{Measured GFLOPS}}{\text{Theoretical Peak GFLOPS}} \times 100 = \frac{34{,}720}{38{,}974} \times 100 \approx 89.1\%.$$

For completeness, the speedup relative to the single-GPU case at $N = 100{,}000$ is:

$$\text{Speedup} = \frac{34{,}720}{17{,}860} \approx 1.95\times,$$

which corresponds to a parallel efficiency of:

$$\text{Parallel Efficiency} = \frac{1.95}{2} \times 100 \approx 97.3\%,$$

indicating excellent scaling with minimal inter-GPU overhead at this problem size. Compared to the single GPU configuration (Table 4.1), the two-GPU setup shows a significant increase in computation speed, particularly for larger problem sizes. For instance, at $N = 100{,}000$, performance increases from 17,860 Gflops on a single GPU to 34,720 Gflops on two GPUs — achieving approximately $1.94\times$ the throughput. This near-linear scaling demonstrates excellent parallel efficiency, indicating that the workload is well-balanced and that inter-GPU communication overhead is minimal in this configuration, as illustrated in the graphical result in Figure 4.9.

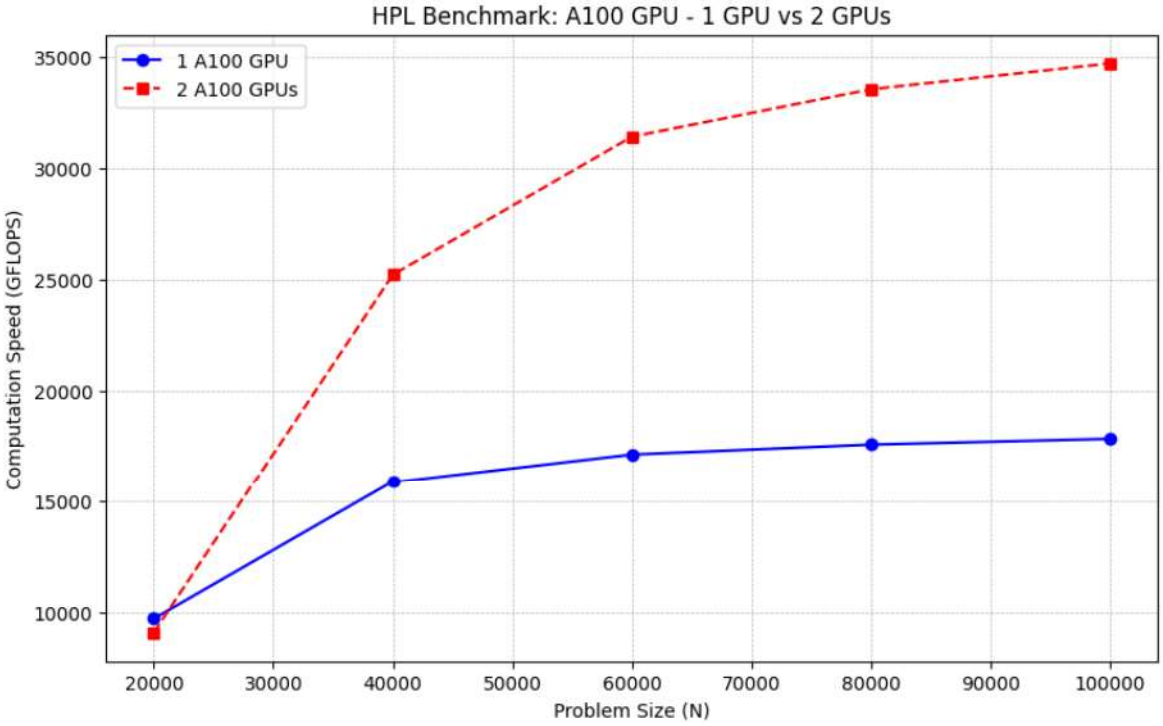

Figure 4.9. *HPL Results on A100*

It is observed that for the smallest problem size (N = 20,000), the single-GPU run performs slightly better than the two-GPU run. This occurs because the workload is too small to fully utilize multiple GPUs efficiently. When using two GPUs, additional time is required for data distribution and inter-GPU communication, which introduces overhead. For larger problem sizes, the computational workload dominates the communication cost, and the two-GPU configuration achieves significantly higher performance, demonstrating effective scaling.

4.4.5 H100 GPU Partition

a) Single GPU, Single MPI Process

To run the HPL benchmark on a single H100 GPU, we allocate a GPU node with one task using the following Slurm command:

```
salloc -t 4:00:00 -n 1 --ntasks=1 -A sw_stack-373lcd9r8io-default-gpu
-p gpu_h100 --gres=gpu:1
```

Each part of the command has a specific role:

- `salloc`: Allocates resources for an interactive Slurm job.
- `-t 4:00:00`: Sets the maximum job runtime to 4 hours.
- `-n 1`: Requests 1 total task for the job.
- `-ntasks=1`: Requests 1 MPI task, ensuring a single process.
- `-A sw_stack-373lcd9r8io-default-gpu`: Specifies the account/project to charge the job to.
- `-p gpu_h100`: Requests the job to run in the `gpu_h100` partition (group of H100 GPU nodes).
- `-gres=gpu:1`: Requests 1 GPU on the allocated node.

Below is a table summarizing the results from five different test cases with varying problem sizes (‘N‘). The results include the time taken for each test and the achieved computation speed in Gflops, as illustrated in Table 4.3:

Table 4.3. *HPL Results on one H100 GPU*

Problem Size, N	Time Taken (s)	Computation Speed (Gflops)
20000	0.31	16,130
40000	1.17	35,760
60000	3.43	41,760
80000	7.57	44,130
100000	11.43	45,110

The theoretical peak for FP64 operations on an NVIDIA H100 GPU is:

$$\text{Theoretical Peak (GFLOPS)} = \text{Number of CUDA Cores} \times 2 \times \text{Boost Clock (GHz)}$$

- Number of CUDA cores: 16,896 - Boost clock: 1.6 GHz

$$\text{Theoretical Peak} = 16,896 \times 2 \times 1.6 \approx 54,067 \text{ GFLOPS} \approx 54 \text{ TFLOPS}$$

Efficiency for the largest problem size (‘N=100000‘) is:

$$\text{Efficiency (\%)} = \frac{\text{Measured Gflops}}{\text{Theoretical Peak Gflops}} \times 100 = \frac{45,110}{54} \times 100 \approx 83.5\%$$

b) Two GPUs, Two MPI Processes, One MPI Process Each

To run the HPL benchmark on two H100 GPUs (one MPI process per GPU), we use the same command but request two GPUs and two tasks (value change of ntaks and gpu):

```
salloc -t 4:00:00 -n 1 --ntasks=2 -A sw_stack-373lcd9r8io-default-gpu
-p gpu_h100 --gres=gpu:2
```

All other options remain the same. The results for two GPUs are summarized below, as show in Table 4.4:

Table 4.4. *HPL Results on two H100 GPUs*

Problem Size, N	Time Taken (s)	Computation Speed (Gflops)
20000	0.40	11,510
40000	1.01	41,460
60000	1.11	64,700
80000	4.36	76,730
100000	7.95	81,970

The combined theoretical peak for two H100 GPUs is:

$$\text{Combined Peak} = 2 \times 54,067 \approx 108,134 \text{ GFLOPS}$$

Efficiency for the largest problem size (‘N=100000‘) is:

$$\text{Efficiency (\%)} = \frac{81,970}{108,134} \times 100 \approx 75.8\%$$

With two GPUs, performance reaches 81,970 GFLOPS, which is about 75.8% of the combined theoretical peak. Compared to the single-GPU case, this represents a speedup factor of 1.82×, resulting in a parallel efficiency of approximately 91%, which indicates good scaling with low inter-GPU communication overhead, as illustrated in the graphical result in Figure 4.10.

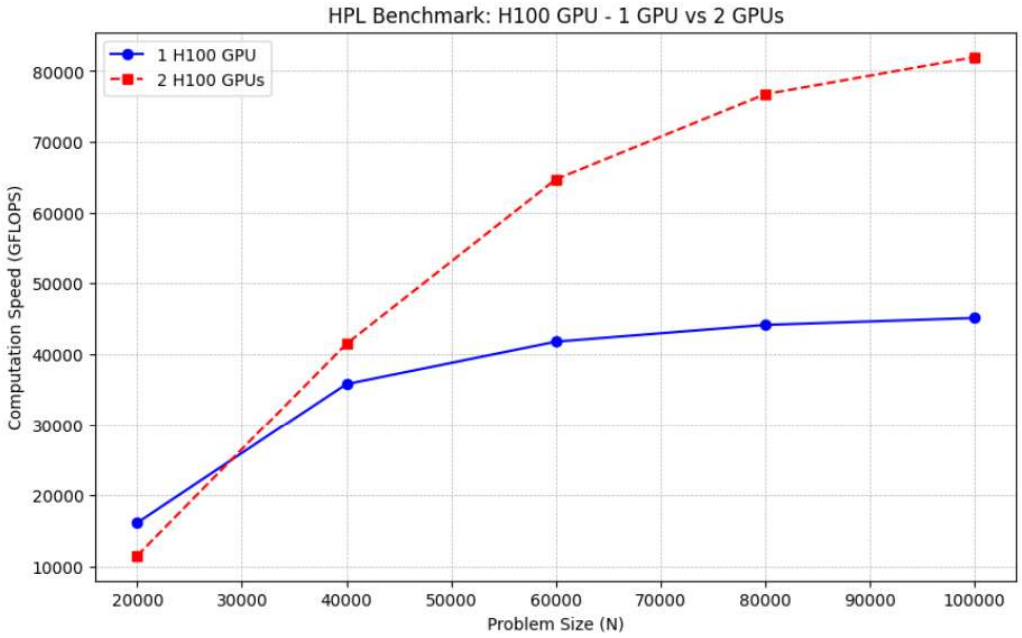

Figure 4.10. *HPL Results on H100*

4.4.6 Performance Analysis: A100 vs H100

Table 4.5. *HPL Results : A100 vs. H100*

Problem Size, N	A100 Computation Speed (Gflops)	H100 Computation Speed (Gflops)
20000	9,731	16,130
40000	15,890	35,760
60000	17,150	41,760
80000	17,600	44,130
100000	17,860	45,110

The results in Table 4.5 from the HPL benchmark clearly demonstrate the performance superiority of the NVIDIA H100 GPU compared to the A100 across all tested problem sizes.

- **Higher Gflops Throughput:** The H100 consistently delivers significantly higher Gflops than the A100, ranging from a 1.7× improvement at $N = 20{,}000$ to a 2.53× improvement at $N = 100{,}000$. This indicates that the H100 executes floating-point operations more efficiently with higher throughput.

- **Better Scalability:** The H100 shows better scalability with increasing problem size. While the A100's performance saturates around 17,800 Gflops, the H100 continues to scale linearly, achieving over 45,000 Gflops. This suggests that the H100 handles large-scale linear algebra computations with greater efficiency, making it ideal for HPC workloads.

- **Improved Architecture:** The performance improvements can be attributed to architectural advancements in the Hopper-based H100 over the Ampere-based A100:

  - *Tensor Core Enhancements:* Fourth-generation Tensor Cores in the H100 support a broader range of data types (including FP8) and offer higher throughput per SM compared to the A100's third-generation Tensor Cores.

  - *Increased SM Count and Clock Speeds:* More Streaming Multiprocessors (SMs) and higher clock frequencies increase parallelism and compute density.

  - *Transformer Engine:* Although not directly exploited in HPL, the Transformer Engine accelerates matrix-heavy operations, contributing to more efficient execution of large matrix solves.

  - *Memory Bandwidth and Latency Improvements:* Higher memory bandwidth reduces bottlenecks for large matrix data movement, critical in memory-bound operations like those in HPL.

- **Reduced Execution Time:** Execution times on the H100 remain low even for large problem sizes. For example, solving $N = 100{,}000$ takes only 11.43 seconds on a single GPU, significantly faster than the A100.

- **Energy Efficiency:** Although not explicitly measured, the H100 is expected to provide higher performance-per-watt compared to the A100, which is important for large-scale deployments where power efficiency is crucial.

- **Future-Proofing and Software Stack Optimization:** The H100 benefits from the latest NVIDIA software stack improvements, including CUDA 12+, optimized

cuBLAS libraries, and Hopper-specific tuning in frameworks like HPC-Bench. These enhancements further boost performance compared to legacy support for the A100.

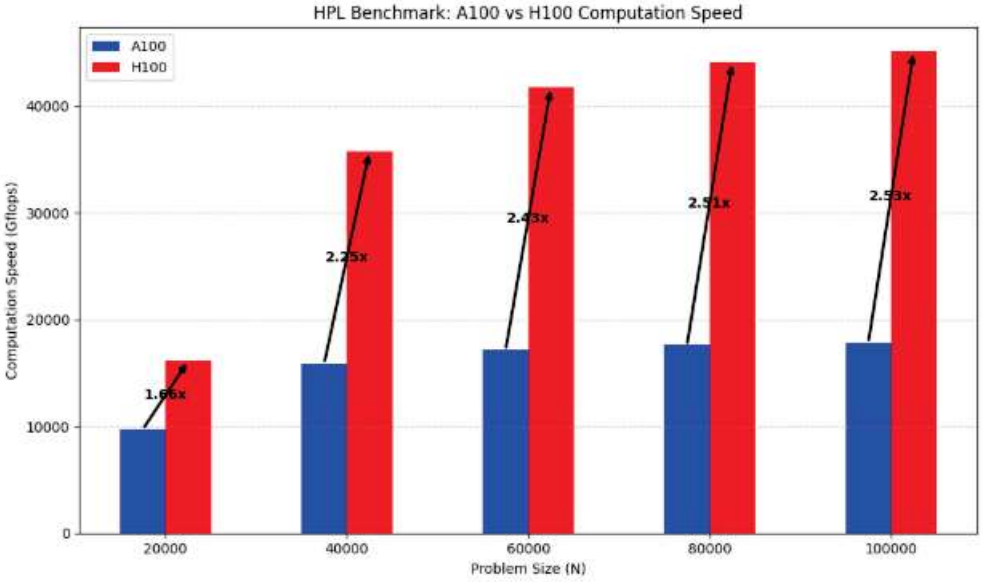

Figure 4.11. *HPL Results : A100 vs. H100*

4.5 *STREAM Benchmark*

The STREAM benchmark is widely used in HPC to evaluate the memory subsystem performance of a computing node. Unlike compute-intensive benchmarks such as HPL, STREAM focuses on the sustainable memory bandwidth and efficiency of basic vector operations. By measuring how quickly data can be read from and written to memory, STREAM provides insight into potential memory bottlenecks and helps in tuning applications that are memory-bound. This makes it a valuable tool for assessing the impact of memory hierarchy and optimizing system performance.

4.5.1 Characteristics of the STREAM Benchmark

The STREAM benchmark is a straightforward synthetic benchmarking tool designed to assess sustainable memory bandwidth in megabytes per second (MB/s) on a single node. It evaluates how effectively memory can be accessed and utilized under various workloads.