Southern University of Science and Technology

ASC24 Student Supercomputing Challenge

Preliminary Round Contest Proposal

Authors:
ASC Team 628
Team Immerse In Clusters

- Yicheng Xiao
- Haibin Lai
- Haoyang Qin
- Hemu Liu
- Junjie Qiu

Advisors:
Jing Fan

Mentors:

- Zhuozhao Li
- Jiahua Zhao

January 21, 2024

Acknowledgement

Virtue, Truth, Advance.

This is the motto of our school. We conduct our work under the guidance of such spirits. Our work is supported by the Center for Computational Science and Engineering (CCSE) at Southern University of Science and Technology (SUSTech). All team members are students currently enrolled in SUSTech.

Our work distributions are as follows:

- Introduction to SUSTech CCSE and Team Immerse In Clusters: Hemu Liu, Yicheng Xiao
- HPC System Design: Haoyang Qin
- HPL Benchmark Test: Haibin Lai
- HPCG Benchmark Test: Yicheng Xiao
- Optimization for LLM inferences: Junjie Qiu
- OpenCAEporo: Haoyang Qin, Hemu Liu

Contents

List of Figures

List of Tables

1 Introduction to SUSTech CCSE and Team Immerse In Clusters

1.1 Brief intro to SUSTech and CCSE

Southern University of Science and Technology (SUSTech) is officially established in February 2, 2011. It is a public university founded in the Shenzhen Special Economic Zone of China. It is intended to be a top-tier international university that excels in interdisciplinary research, nurturing innovative talents and delivering new knowledge to the world. In order to train students with high-performance computing technology and boost compute intensive scientific research, SUSTech established the Supercomputing Center (now known as the Center for Computational Science and Engineering, CCSE)(Figure 1.1) in 2015, and SUSTech signed a strategic cooperation agreement with Inspur and Lenovo to build the HPC platforms together.

1.2 Supercomputing-related hardware and software platform

The high performance computing platform of the CCSE of Southern University of Science and Technology includes two parts: local computing platform and cloud platform.The local computing platform is composed of two sets of clusters, "Taiyi" and "Qiming" .

The total number of CPU cores has reached 43,136, and the number of GPU cards has reached 38 (including 10*V100 and 28*A100). The total dual-precision computing capability of the platform exceeds 3.5 PFlops, and the total raw storage capacity is 6.6 PB.

科学与工程计算中心
Center for Computational Science and Engineering

Figure 1.1: CCSE

1.2.1 Taiyi

Taiyi contains 815 dual-blade nodes, two large-memory nodes, and four GPU nodes. It uses the Intel Omni-Path high-speed (100 Gbps OPA) computing network, has a raw storage capacity of 5.5 PB, and the measured write I/O bandwidth exceeds 45 GB/s. The measured read I/O bandwidth exceeds 60 GB/s.

Later, in December 2018, TaiYi (shown in Figure 1.2), the second HPC platform at SUSTech, began to operate. TaiYi has about 820 computing nodes, 40,000 cores, and 4.0 PB of storage space. The measured floating-point computing performance is 1.6 Pflops. It is ranked 127th on the Top 500 list (November 2018) and 39th on the Chinese Supercomputer Top 100 list (October 2018).

Figure 1.2: Taiyi

Operating System	Rocky Linux 8.6 (Green Obsidian)
Cluster monitoring software	TSCE
Cluster job scheduling system	Torque + maui
Cluster parallel environment	Intel MPI-2015, Open MPI-1.10.1
Cluster development environment	Intel Parallel Studio XE 2015
	PGI-16.1 Compiler
	GNU Compilers for C/C++
	CUDA Toolkit 9.0
	Intel MKL mathematical core library

Table 1.1: Software Configuration of Taiyi

Role	**SKU**	**Specification**	**Quantity**
Management and Login	ThinkSystem SR650	CPU:Intel Xeon Gold 6140, 2.3GHz 18 cores	2
		Memory: 32G, DDR4 2666Mhz RDIMM	12
		Hard disk: 1.2TB, SAS 10K rpm	4
		Network: Intel OPA 100 1 Series Single-port PCIe 3.0 x16 HFA	1
CPU-only Computing	ThinkSystem SR530	CPU:Intel Xeon Gold 6148, 2.4GHz 20 cores	2
		Memory: 16G, DDR4 2666Mhz RDIMM	12
		Hard disk: 1.92TB SSD	1
GPU-Enabled Heterogenous Computing	ThinkSystem SR650	CPU:Intel Xeon Gold 6148, 2.4GHz 20 cores	2
		Memory: 32G, DDR4 2666Mhz RDIMM	12
		Hard disk: 1.92TB SSD	3
		GPU: Nvidia Tesla V100 16GB PCIe Passive GPU	2
Large-memory Fat Node	ThinkSystem SR950	CPU:Intel Xeon Platinum 8160 2.1GHz, 24 cores	8
		Memory: 64G, DDR4 2666Mhz RDIMM	96
		Hard disk: Intel SSD DC S4500, 3.84TB	2

Table 1.2: Hardware Configuration of Taiyi

1.2.2 Qiming 2.0

In 2022, after an upgrade, the Qiming cluster is now increased to 269 blade nodes, 12 large-memory compute nodes and 7 GPU nodes, using Mellanox EDR high-speed (100 Gbps IB) computing network, storage raw capacity of 1.1 PB, and measured write I/O bandwidth of more than 45 GB/s. The measured read I/O bandwidth exceeds 90 GB/s.

With the rapid development of the scientific research, the requirement for resource allocation and equipment management grows. In June 2016, SUSTech acquired its first-phase HPC cluster known as QiMing. This platform (Shown in Figure 1.3) was built by Inspur. The system has a total measured peak more than 200 Tflops, among the top universities in China. The system consists of blade node, fat node and GPU node, including four login nodes and two management nodes. The storage system adopts IEEL parallel file system, and the calculation network adopts Mellanox EDR 100 Gbps InfiniBand. With these equipments, the huge demand for different kinds of programs from various majors are meet. It is worth mentioning that QiMing was the first supercomputer in China to utilize Mellanox EDR 100 Gbps InfiniBand in that year.

Figure 1.3: Qiming

Operating System	Rocky Linux 8.6 (Green Obsidian)
Cluster parallel environment	Intel MPI-2021.7.0, Open MPI-4.1.2
Cluster development environment	Intel Parallel Studio XE 2021
	NVHPC/22.11
	GNU Compilers for C/C++
	CUDA Toolkit 11.8
	Intel MKI mathematical core library

Table 1.3: Software Configuration of Qiming 2.0

During the competition, most of our applications are run on queue 38, queue 1t75c and an independent small cluster for GPU tuning. The detailed configurations of the nodes are as follows:

Table 1.4: Queues Configuration on Qiming 2.0

Role	Nodes	CPU & GPU Specification	Memory	Quantity
Management and login	1	2 * Intel Xeon Gold 4314 (2.4GHz, 16 cores)	128GB	1
	2	2 * Intel Xeon E5-2690v3 (2.6GHz, 12 cores)	128GB	2
CPU-only computing	38	2 * Intel Xeon Gold 6338 (2.0GHz, 32 cores)	512GB	36
	amd73x	2 * AMD EPYC 7773X (2.2GHz, 64 cores)	512GB	1
	amd63	2 * AMD EPYC 7763 (2.45GHz, 64 cores)	512GB	1
	v3-64	2 * Intel Xeon E5-2690v3 (2.6GHz, 12 cores)	64GB	225
	v3-128	2 * Intel Xeon E5-2690v3 (2.6GHz, 12 cores)	128GB	6
	1t75c	2 * Intel(R) Xeon(R) Platinum 8375C CPU (2.90GHz, 32 cores)	-	10
GPU-Enabled Heterogenous Computing	hgxa100	2 * AMD EPYC 7763 (2.45GHz, 64 cores)	1TB	2
	4a100-80	2 * AMD EPYC 7763 (2.45GHz, 64 cores)	512GB	4
	4a100-40	2 * Intel Xeon Gold 6258R (2.7GHz, 28 cores)	384GB	4
	2a100-80	2 * Intel Xeon Gold 6230 (2.1GHz, 20 cores)	192GB	2
	2a100-40	2 * Intel Xeon Gold 6230 (2.1GHz, 20 cores)	192GB	2
	2v100	2 * Intel Xeon Gold 6230 (2.1GHz, 20 cores)	192GB	1
Large memory Fat Node	ot38	2 * Intel Xeon Gold 6338 (2.0GHz, 32 cores)	512GB + 2TB Optane	1
	2t-50c	2 * Intel Xeon Platinum 8350c (2.6GHz, 32 cores)	2TB	4
	v3-6t	8 * Intel Xeon E7-8880v3 (2.3GHz, 18 cores)	6TB	7

Queue	38
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
CPU(s)	64
On-line CPU(s) list	0-63
Vendor ID	GenuineIntel
Model name	Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz
CPU family	6
Thread(s) per core	1
Core(s) per socket	32
Socket(s):	2
CPU max MHz	3200
CPU min MHz	800
BogoMIPS	4000.0
L1d cache:	48K
L1i cache:	32K
L2 cache:	1280K
L3 cache:	49152K
NUMA node0 CPU(s):	0-31
NUMA node1 CPU(s):	32-63

Table 1.5: Hardware Description on queue 38

Queue	v3-64
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
CPU(s)	64
On-line CPU(s) list	0-63
Vendor ID	GenuineIntel
Model name	Intel(R) Xeon(R) Platinum 8375C CPU @ 2.90GHz
CPU family	6
Thread(s) per core	1
Core(s) per socket	32
Socket(s):	2
CPU max MHz	3500
CPU min MHz	800
BogoMIPS	5800.00
L1d cache:	48K
L1i cache:	32K
L2 cache:	1280K
L3 cache:	55296K
NUMA node0 CPU(s):	0-31
NUMA node1 CPU(s):	32-63

Table 1.6: Hardware Description on queue 1t75c

Queue	GPU-cluster
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
CPU(s)	128
On-line CPU(s) list	0-127
Vendor ID	AuthenticAMD
Model name	AMD EPYC 7773X 64-Core Processor
CPU family	25
Thread(s) per core	1
Core(s) per socket	64
Socket(s):	2
CPU max MHz	3527.7339
CPU min MHz	1500.0000
BogoMIPS	4391.64
L1d cache:	4 MiB (128 instances)
L1i cache:	4 MiB (128 instances)
L2 cache:	64 MiB (128 instances)
L3 cache:	1.5 GiB (16 instances)
NUMA node0 CPU(s):	0-63 (Connected with GPU 0)
NUMA node1 CPU(s):	64-127 (Connected with GPU 1,2)
GPU model name	NVIDIA A100-SXM4-80GB
FP64	9.7 TFLOPS
FP64 Tensor Cores	19.5 TFLOPS
Total core(s)	6912 CUDA cores

Table 1.7: Hardware Description on GPU-cluster

1.3 Supercomputing-related courses, trainings, and interest groups

Apart from providing hardware platforms, CCSE also signed a strategic cooperation agreement with Inspur to build the HPC cloud platform together. They would set up a scholarship to encourage achievements by students in HPC. CCSE usually invites sponsors' engineers to train students in high-performance computing. For example, a training lecture for all the users of our clusters (Figure 1.4). About 200 persons attend this training lecture. CCSE has also made an dedicated outreach to both undergraduate and graduate students. The SUSTech HPC club is set up

Figure 1.4: HPCuserstraining

for student engagement in late 2020 under the support and supervision of CCSE. Including the early stage of HPC club before official kick-off, we already hosted and sent member student to all three sessions of NVIDIA Deep Learning Institute accelerated computing courses, during which students are certificated for CUDA-accelerated programming. Meanwhile, selected members from club have also given a series of HPC beginner's seminar and workshop. Participants are well-trained for working with HPC and make simple optimizations on their code.

SUSTech provides many courses related to HPC and AI for students (especially undergraduates) with the help of CCSE. For example, Parallel Computing which taught by Prof. Georgios Theodoropoulos, provides students with the skills they will need to design and maintain software for HPC; and Deep Learning which offered by Prof. Luca Rossi, provides an introduction to the field of deep learning, covering the main deep learning techniques from both a theoretical and practical point of view. Of course, there are other computing courses offered by other departments, such as Computational Chemistry, Computational Physics and Computational Biology. The availability of these courses has improved students' skills in High Performance Computing.

Figure 1.5: competition

Besides training, CCSE also organized interesting competition to give all students who have interests in HPC a platform to compete and communicate.(Figure 1.5)

1.4 Supercomputing-related Research and Applications

SUSTech and CCSE promote the use of high performance computing in scientific research. The clusters has now established more than one hundred user accounts. These accounts were mainly used by research teams from biology, chemistry, physic, aerospace engineering and some other departments.

The purpose of the HPC varies: in the chemistry part, density functional theory calculation can be performed in the platform to predicted the thermodynamic and dynamic properties of the compounds and condensed matter systems. On a bioinfomatic note, intensive loads of sequence processing like alignment and assembly of Next-Generation Sequencing (NGS) reads are performed high efficiently on HPC, which is turning work of weeks to couple hours. Nevertheless, for fluid dynamics, HPC is turning those impossible simulations a decade ago into reality, consequently the algorithms are proven and results can serve the most cutting edge aerospace design and researches.

Specially for structural biology, by virtue of on-site cryo-EM, terabytes of EM micrographs are generated within a day. Dedicated HPC is running over day and night to reconstruct 3- dimensional structure of particles basing on the micrographs and making new breakthroughs in understanding life. All above HPC practice is opening new horizons of scientific research for researchers on campus.

1.5 Key Achievements on Supercomputing

The competition achievements mainly come from the students supercomputing team of SUSTech. The team was formally established in 2018. With the support of the Center for Computational Science and Engineering of SUSTech.

The key achievements of recent years are as follows:

- ASC-19 First Prize (Rank 7th)
- ASC-19 Best Popular Team
- ASC-19 Application Innovation Award (ShengBTE)
- APAC-20 HPC-AI Competition The Third Place (Rank 3rd)[1]
- SC-20(VSCC) miniVite (Rank 2nd) & HPL (Rank 3rd)
- ASC-20 21 First Prize (Rank 5th)
- APAC-21 HPC-AI Competition Champion (1st) & AI Special Award
- SC-21 The Third Place & Highest Linpack Benchmaark Winner (Figure 1.6)
- ISC-22 The Third place
- APAC-22 HPC-AI Competition The Second Place
- IndySCC-22 The Second Place & Hero HPL Challenge(2nd)
- ASC-22 23 The First Prize & Super Team Award (Figure 1.6).
- Kunpeng Application Innovation Competition-23 First Prize
- APAC-23 HPC-AI Competition The Second Place

Figure 1.6: ASC-22 23

1: In 2020, the winners of 2020 APAC HPC-AI Competition were announced in the SC20 held online. SUSTech students supercomputing team, the only winning university team in mainland China, won third place in the competition (Shown in Figure 1.9).

Figure 1.7: SC21

Figure 1.8: Kunpeng Application Innovation Competition-23 First Prize

1.6 Brief intro to Team Immerse In Clusters

1.6.1 Introduction to team supervisors and members

Our team is a big family. At present, the core members are:

- **Dr. Jing Fan (Advisor)** He is the chief engineer of the Center for Computational Science and Engineering of SUSTech, whose main research direction is computational chemistry and high performance scientific computing.
- **Zhuozhao Li (Mentor)** He is an assistant professor in the Department of Computer Science and Engineering at Southern University of Science and Technology. His research interests include High Performance Computing, Distributed Systems, Cloud/Edge Computing, and Internet of Things.
- **Jiahua Zhao (Mentor)** He is a PhD student in the Marie Sklodowska-Curie Actions (ENGAGE programme), studying at the Delft University of Technology and the Cyprus Institute, focusing on HPC and AI applications in exploration geophysics. During his BSc and MSc studies, he worked on HPC and AI applications in scientific computing at Columbia University, Harvard University, Edinburgh Center for Parallel Computing, and King Abdullah University of Science and Technology. He is also the builder and advisor of the student supercomputing team of Southern University of Science and Technology, and has guided students in various international supercomputing competitions, achieving excellent results.

Figure 1.9: Group Photo: Starting from left are Haoyang Qin, Haibin Lai, Yicheng Xiao, Hemu Liu, Junjie Qiu

- **Yicheng Xiao (Captain, 2022-Present):** Sophomore majoring in Computer Science, participant of APAC HPC-AI Competition and IndySCC 2023. He has experience in cluster designs, high performance computing tuning and software optimization. Current research insterests are Computer Graphics and Simulation Technology.
- **Haibin Lai: (member, 2022-Present):** Sophomore majoring in Computer Science, participant of lndyscc2023. He has experience in High performance computing tuning, C++ programming and DCU programming. Current research interest is High performance computing software optimization.
- **Haoyang Qin (member, 2023-Present):** Qin is a sophomore majoring in Intelligent Medical Engineering. He has observed ASC22 as an observer and also participated in APAC as a team member. He has a slight understanding of machine learning algorithms and has used scientific computing software such as LAMMPS and MPAS in previous competitions.
- **Hemu Liu (member, 2022-Present):** Sophomore majoring in Computer Science, participant of APAC HPC-AI Competition 2023. Having experience on high performance computing tuning and software optimization. Current research insterests are Computer Vision and Image Processing.
- **Junjie Qiu (member, 2023-Present):** Junior majoring in data science and big data technology at SUSTech. Currently work in SUSTech Machine Learning Laboratory and take the major responsibility on an independent research project. Mainly focus on member inference attack on deep learning models, especially under strict settings which challenge previous SOTA methods. He is experienced in arranging deep learning experiments and model tuning.

1.6.2 Team Motto

Debug for excellence, Refactor for future.

2 HPC System Design

In the contemporary landscape of technological advancement, particularly in areas such as artificial intelligence, big data analytics, and complex data processing, the limitations of conventional computing platforms have become increasingly evident. This realization has spurred interest in more robust computing architectures, with a significant focus on heterogeneous systems that integrate Central Processing Units (CPUs) with Graphics Processing Units (GPUs). These CPU+GPU configurations have emerged as potent solutions, demonstrating substantial efficacy, especially as GPUs surpass CPUs in areas of parallel data processing and memory bandwidth management.

The progress in the domain of General-Purpose computing on Graphics Processing Units (GPGPU) has notably accentuated the prominence of CPU+GPU heterogeneous parallelism. This technological strategy harnesses the formidable computational strength of GPUs in conjunction with the logical processing capabilities of CPUs. Its application extends across various computationally intensive fields, offering marked enhancements in processing speeds and efficiency, particularly in disciplines such as image processing, petroleum exploration, and deep learning.

In our endeavor to leverage this advanced computational paradigm, we have meticulously crafted a solution for the establishment of a CPU+GPU heterogeneous computing platform. And we carefully considered the characteristics of each competition question, so that the cluster can well meet the needs of these questions.

2.1 The analysis for the problem and its application scenario

For this competition, the task is to construct a computing cluster that does not exceed a peak power usage of 3000 watts, with the goal of maximizing computing performance. However, it's crucial to consider more than just performance. The design must also be tailored to meet the specific needs of intended applications. This overview outlines the key criteria for the contest, emphasizing the need for a balance between high performance and practical application.

- **HPL and HPCG**

  HPL is the most common benchmark for verifying the performance of high-performance computers, it is a comparative standard for high-performance computers TOP500. HPL measures the floating-point computing capability of a high-performance computer by solving the linear dense equation system. With the continuous expansion of high-performance technology applications,there exists a deal of difference between the traditional high-performance computers' computing model and the current major model. The system which has a better performance when tested using HPL does not always performance well in practical applications,because the largest high-performance calculation command is to solve partial differential equations nowadays(such as typical atmospheric ocean numerical prediction models), these applications require higher band width and lower latency than HPL, and a lot of irregular access to memory is needed. In fact,it's more likely to lead the increment of the number of redundant parts or system complexity with little improvement in actual application performance if we simply aiming at improving the performance of high-performance computers when building a high performance computer system. In simple terms,Linpack tests the theoretical performance of super-calculated processors,while HPCG put more emphasis on actual performance,which has higher requirements on memory system and network latency. With the field of high-performance applications expanding, HPL's test patterns and conclusions have failed to fully reflect the practical performance of HPC. Therefore,we mainly focus on the optimization of the HPCG. So the first thing we need to pay attention to is CPU computing power, memory size and network configuration.

- **OpenCAEPoro**

  OpenCAEPoro, a cutting-edge permeation numerical simulation software. Its primary goal is to advance human understanding of efficient exploitation of deep-earth energy resources and the safe development and utilization of deep-earth space. Permeation mechanics, the foundation of OpenCAEPoro, has wide applications in scientific research across various fields such as environmental protection, earthquake prediction, and biomedical science. It is also instrumental in engineering technologies including the prevention and treatment of ground subsidence or seawater intrusion, construction of large-scale hydroelectric projects, agricultural and forestry engineering, and permafrost engineering. The study of multiphase fluid permeation, one of the most challenging aspects of permeation theory, has become a key discipline driving the advancement of permeation mechanics. The simulation conducted by OpenCAEPoro is a fine-grained infiltration numerical simulation. It is crucial in many industrial applications but are computationally intensive. For instance, in oil and gas development, the resolution of oilfield networks in the horizontal direction usually ranges from ten to hundreds of meters. In order to accurately describe reservoir characteristics, a significant increase in computational load is re-

quired, and there is a high demand for computational complexity. In order to speed up computation, multi node parallelism is often used, but the communication overhead it brings is also enormous.

- **LLM inference**

  Large language model (LLM) has become a hot keyword in AI research after the technological explosion brought by OpenAI. It has the potential to revolutionize the way we interact with technology and each other by enabling more natural and intuitive communication. Meanwhile, deploying LLM inference at low cost, efficiently and cost-effectively can bring huge business value. LLaMA 2 is a second-generation open-source LLM developed by Meta. It can be used to build chatbots like ChatGPT or Google Bard, and has been trained on a vast amount of data to generate coherent and natural-sounding outputs. In ASC24, we need to optimize inference with LLaMA2-70B model on a dataset with 10 thousand samples to low latency response. The LLM inference task is basically a series of matrix computation, which is float-computing-intensive and can be naturally performed on GPU. In addition, models like LLaMA2-70B can not be well fit in the memory of a single GPU[1], and thus parallelism methods like model parallelism and tensor parallelism are considered to split model into multiple cards[2]. However, every parallelism need an amount of communication overhead. Based on the above reasons, we need to take GPU resources and communication overhead into account to design our own cluster. For example, we need at least 2 A100-80G cards to load model and perform inference and try everything we can to decrease the card-to-card communication overhead. In traditional view, LLM tasks are not CPU-intensive but there still exists CPU overhead which decay the overall response to request. Therefore, a inference engine that suppress the above issues is needed to optimize the computation process.

1: It takes about 140 GB to load LLaMA2-70B into memory, which most types of GPU won't be able to do with single card as of January 2024.

2: Data parallelism is not our main consideration, because it is only by stacking computation resources in exchange for higher inference speed, although the technical difficulty is not high, but is does not inherently reduce the inference cost.

2.2 Design scheme of HPC System

2.2.1 Hardware Configuration

We have not found any relevant information about IEIT NF5280M7, so we decided to use the IEIT NF5280 G7 as the server to achieve better performance in the cluster which can satisfy our all requirements.

We read the IEIT NF5280G7 product technical white paper and learned that the NF5280G7 has excellent performance in terms of scalability, usability, manageability, etc. Figure 2.1 shows the display of NF5280G7.

Figure 2.1: NF5280 G7

The IEIT NF5280G7 is a high-performance server designed to meet the demands of modern computing environments. It is built on the Intel® Xeon® 4th and 5th Generation Scalable Processors, offering a single CPU with up to 64 cores and 128 threads. This server supports CPUs with a maximum Thermal Design Power (TDP) of 350W and boasts top turbo frequencies of up to 4.2GHz, ensuring swift and efficient processing.

Additionally, it features four sets of 20GT/s Ultra Path Interconnect (UPI) links, providing robust and fast communication between components.

Memory-wise, the NF5280G7 excels with support for up to 32 slots of 5600MT/s DDR5 ECC memory, utilizing RDIMM modules to deliver exceptional speed and high availability.

In terms of storage and expandability, the NF5280G7 offers versatile options to suit a wide range of needs. It can accommodate up to 45x 2.5-inch drives or 20x 3.5-inch drives for extensive storage capacity. Additionally, it includes optional rear-mounted M.2/E1.S SSD modules for further storage customization. The server's expansiveness continues with support for PCIe 5.0/4.0, allowing up to 20 PCIe expansion slots.

Networking capabilities are another highlight of the NF5280G7, with support for 2 or 3 optional hot-pluggable OCP 3.0 modules. These modules provide a variety of network interface options, ranging from 1G to 400G, ensuring high-speed and flexible connectivity for diverse network requirements.

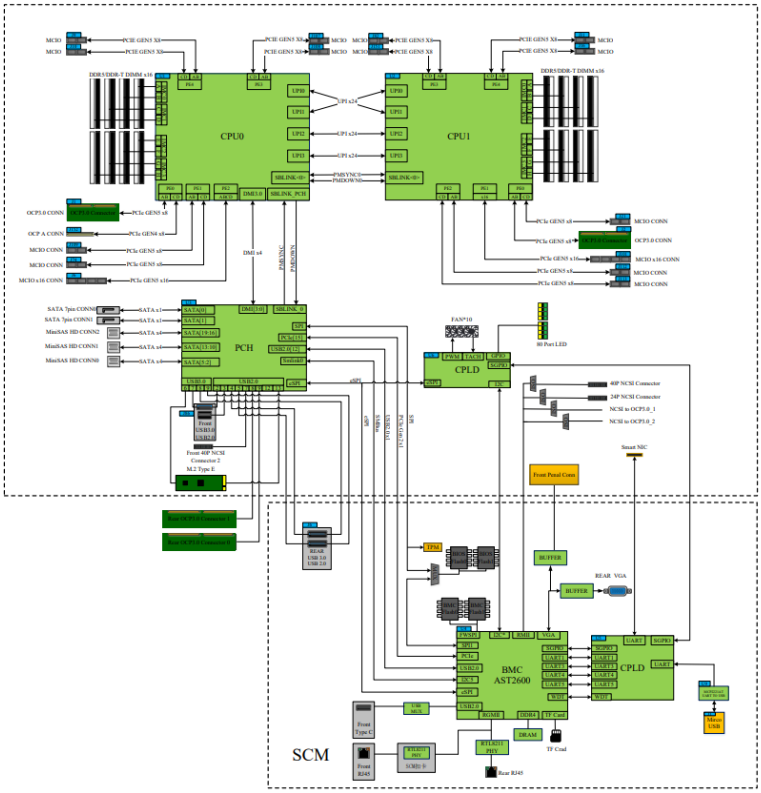


Figure 2.2: The logical block diagram of IEIT NF5280G7

We can see more detailed information from the logical block diagram of IEIT NF5280G7.

In terms of computing roles, the scheme divides the nodes into three parts: integrated management (login storage) node (hereinafter referred to as a "management" node), general purpose computing node, and heterogeneous computing node. The first is the integrated management node, which is the control node of various management measures of the cluster system. It is responsible for monitoring the running status of each node and network in the cluster. It requires only low computational

performance, but due to the limitation of 3000W power consumption, the management node is also used as both a login node and a storage node. In extreme cases, idle cores on this node will also be evoked for computational use. The number of management nodes required for our solution is one. The computing node is the core of the entire cluster, and its function is to perform parallel computing tasks. Currently, utilizing heterogeneous systems for general scientific and engineering calculations has become a popular trend. Using GPUs for parallel computing, especially in the field of artificial intelligence, can greatly improve computational efficiency. Therefore, we have decided to design four heterogeneous nodes. In terms of network (switching) equipment, we use GbE Switch and HDR-IB Switch. Gigabit Ethernet and GbE Switch carry TCP data and IPMI management data. InfiniBand network and HDR-IB Switch will meet the requirements of network performance and intelligence for generalized network transmission and storage I/O. Therefore, we design HPC system related information. As shown in Table 2.1:

Table 2.1: The hardware information of the HPC system

Item	Name	Configuration	Quantity	Power consumption
Manage login and Storage Server * 1	IEIT NF5280 G	CPU: Intel Xeon Gold 6430, 2.1GHz, 32 cores	2	540w
		Memory: 32G, DDR5, 2933Mhz	16	10w
		Hard disk: 1.5T	4	40w
		HCA card: HDR	1	9w
CPU/GPU Sever * 4	IEIT NF5280 G7	CPU: Intel Xeon Gold 6430, 2.1GHz, 32 cores	2	540w
		Memory: 32G, DDR5, 2933Mhz	16	10w
		Hard disk: 1.5T	1	10w
		GPU: NVIDIA A100 PCIe	2	500w
		HCA card: HDR	1	9w
Switch	GbE Switch	Ethernet switch: 10/100/1000 Mb/s, 24 ports	1	30w
	HDR-IB Switch	QM8700	1	274w

Based on the information in the table, we can calculate the theoretical power consumption of the HPC system:

$$P_M = 540W + 10W + 40W + 9W = 599W \tag{2.1}$$

$$P_{CG} = (540W + 10W + 10w + 500W + 9W) \times 4 = 4276W \tag{2.2}$$

$$P_S = 30W + 274W = 304W \tag{2.3}$$

$$P_{All} = P_M + P_{CG} + P_S = 599W + 4276W + 304W = 5179W \tag{2.4}$$

Where P_M is the theoretical power consumption of the management node, P_{CG} is the theoretical power consumption of all compute nodes, P_S is the power consumption of the switching device, and P_{All} is the theoretical total power consumption of the HPC system.

In addition to evaluating power consumption, we also need to evaluate the computing performance of the HPC System. For the CPU, its theoretical

floating point computing powerP_{peak}^{CPU}

$$P_{peak}^{CPU} = N_{proc} \times F_{clock} \times FLOPs/clock \tag{2.5}$$

N_{proc} is the number of processors available, F_{clock} is the frequency of a processor, and $FLOPs/clock$ is the floating-point operation per clock cycle. The theoretical floating-point computing power of the GPU P_{peak}^{GPU} can be obtained in the NVIDIA GPU product white paper. For an A100 (PCIe), its double-precision floating-point (FP64) is 9.7 TFlops (FP64 Tensor Core is 19.5TFlops). Therefore, the theoretical total floating-point computing power P_{All} of the HPC system we designed is (without the management node)

$$P_{peak}^{CPU} = (32 \times 8) \times 2.1GHz \times 32FLOPs/clock = 17.20TFlops \tag{2.6}$$

$$P_{peak}^{GPU} = 9.7TFlops \times 8 = 77.6TFlops \tag{2.7}$$

$$P_{peak}^{All} = P_{peak}^{CPU} + P_{peak}^{GPU} = 17.20TFlops + 77.6TFlops = 94.80TFlops \tag{2.8}$$

Finally, the simple logical architecture diagram of our HPC system is shown in Figure 2.3.

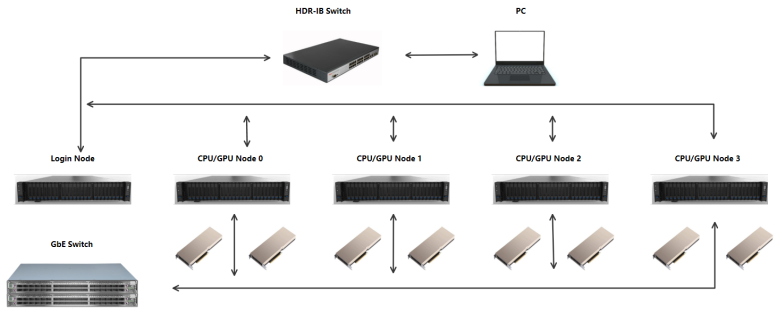


Figure 2.3: HPC system logical architecture diagram

2.2.2 Software Configuration

Apart from the hardware design, the configuration of the software environment is also important. By combined the Linux operation and maintenance experience with the software requirements of the content of this competition, we finally decided to use the software in Table 2.2 as the basic configuration.

Classification	Description	Version
OS	GNU/Linux Red Hat 8.5.0-10	Linux version 4.18.0
Math Library	Intel MKL	2022.2
MPI	Intel MPI	2021.6.0
Analysis Tool	Intel Vtune Profiler	2024.0.0
Compiler	icc	2022.2.0
	gcc	11.2.0
	cmake	3.25.2
Container	Docker	24.0.7
Cuda Compiler	Nvida Cuda	12.1

Table 2.2: HPC system Software Environment Configuration

2.3 Analysis and discussion

2.3.1 Actual power consumption

From the second part, we can calculate that the theoretical total power consumption of the designed cluster is 5179w. However, HPL and HPCG are divided into CPU and GPU versions. When using the CPU to run HPL or HPCG, we will set the GPU to sleep mode, so that the power consumption of the 8 A100s can be neglected. Similarly, when using the GPU to run HPL and HPCG again, the CPU is almost not working. For OpenCAEporo, it runs on the CPU and does not use the GPU. On the other hand, for LLM conferences, inference mainly runs on the GPU, with very little CPU workload. It is precisely because the calculations for each competition are only concentrated on the CPU or GPU that we only need to ensure that the power consumption does not exceed 3000W when using only the CPU or GPU.

After subtracting the CPU power consumption of the computing nodes, the total power consumption of the cluster is only 3019w. However, if the GPU power consumption of the computing nodes is subtracted, the total power consumption of the cluster is only 3179w. At the same time, since the CPU of the login management node hardly runs at full power consumption. At the same time, we will also set the clock frequency of the CPU of each compute node to reduce the final total power consumption. So we can easily control the total power consumption to be below 3000w

2.3.2 Architecture Advantages and disadvantages

Advantages

For the HPC system we designed, there are four major advantages:

1. Fully meet power consumption requirements
The cluster we designed, whether facing high CPU requirements or GPU requirements, can fully reach 3000W under power consumption limitations.

2. Powerful computing and communication capabilities
The powerful computing and communication capabilities given by Intel

Xeon Glod 6554s and InfiniBand network is suitable for computing-intensive software like OpenCAEporo, and will perform well in high-level software migration and operation

3. More adaptable to competition questions
In our testing, we found that OpenCAEporo has good parallelism, and more cores can provide faster running speeds. The Intel Xeon Gold 6430 has 32 cores, making it ideal for computationally intensive software like OpenCAEporo. Meanwhile, we found that in multi node scenarios, the post-processing time of OpenCAEporo is much longer, and high-speed and high bandwidth InfiniBand networks can minimize communication congestion as much as possible.

4. Massively Parallel Computing and Processing Core
Multi-core CPU just uses a few threads simultaneously, while GPU can execute thousands of threads at the same time, which allows the entire system to handle more information flows.

Disadvantages

However, the HPC system we designed is also flawed:

1. Fewer GPUs on a node
For large language models with many parameters, two GPUs on one node are not sufficient for fast training and inference. Increasing the number of nodes helps to disperse training, but an increase in the number of nodes leads to a corresponding increase in communication overhead.

2. Limitations of hardware devices
Due to power consumption limitations, we are unable to run all CPUs and GPUs simultaneously, resulting in some devices being limited. Additionally, having too many idle devices can increase power consumption, ultimately affecting overall performance.

3 HPL Benchmark Test

3.1 HPL Algorithm Introduction and its Principle

3.1.1 HPL Algorithm Introduction

Linpack is the most popular benchmark for testing floating-point performance of highperformance computer systems. The test which is used in this competition is called HPL (High Performance Linpack), is a parallel version of Linpack Benchmark on distributed storage computers. It uses the block algorithm and is often used to test the actual maximum performance that high-performance computers can achieve. And its now widely used to performing the Top500 supercomputers ranking test.

HPL using a Gaussian elimination method to solve a system of linear equations. When the problem size is N, the number of floating-point operations is

$$\frac{2}{3}N^3 - 2N^2 \quad (1)$$

This value is a certainty, so as long as the problem size N is given and the system computing time T is measured, the performance parameters of the machine can be calculated quantitatively by **Rpeak floating pointoperations/sec**

$$R_{peak} = \frac{\frac{2}{3}N^3 - 2N^2}{T} \quad (2)$$

On the basis of HPL hotspot function analysis and source code algorithm analysis, our team weighs the key adjustment parameters and formulates a set of evaluation system based on the ranking of parameter influence factors, which is used to find the optimal path for HPL parameter tuning.

3.1.2 HPL Algorithm Principle

The Linpack benchmark is an implementation of the LU decomposition to solve a dense N × N system of linear equations:

$$Ax = b \, (3)$$

With A ∈ $R_{n\times n}$ and x, b ∈ R_n.The solution is obtained by Gaussian elimination with partial pivoting, resulting in a floating point workload of . The basic idea of the Gaussian elimination method is to first perform

LU decomposition on the matrix A, where L is the unit lower triangular matrix and U is the non-singular upper triangular matrix . Let Ux = y, Ly = b, solve Ux = y. The decomposition of matrix A is expressed as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{13} & A_{14} \end{pmatrix} \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} = LU \quad (4)$$

In the formula, Aij and Lij are known, while Uij is to be found (i, j = 1, 2). It can be obtained from formula 1.4 that

$$A_{11} = L_{11}U_{11} \quad (5)$$

In formula 1.5, L11 and U11 are computed by decomposing the principal column elements, so L21 and U12 can then be obtained by

$$L_{21} = A_{21}U_{11} \quad (6)$$

$$U_{12} = A_{12}L_{11}^{-1} \quad (7)$$

In addition, formula 1.7 gives us the conclusion that

$$A_{22} - L_{21}U_{12} = L_{22}U_{22} \quad (8)$$

The benchmark is run for different matrix sizes N in an effort to find the size Nmax for which the maximum performance Rmax is obtained.

As figure shows, the grey area represents the portion of the matrix already factored. The red area is the current block being factorized. Once this factorization is ready, it is applied to the sub-matrix on the right. The final step is to update the trailing sub-matrix in yellow. There are several variants of the blocked LU decomposition. Most of the computational work for these variants is contained in three routines: the matrix-matrix multiply (DGEMM), the triangular solve with multiple righthand sides (DTRSM) and the unblocked LU factorization for operation within a block column. The right-looking variant of the LU factorization is shown in Figure 1. Most of the Linpack runtime is spent in matrix-matrix multiplies for the update of trailing matrices (the yellow area in Figure). The bigger the problem size N is, the more time is spent in this routine [1].

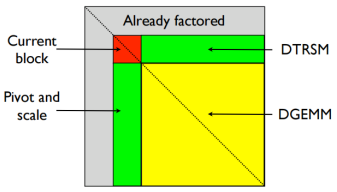


Figure 3.1: HPL Factorization

In HPL CPU version, the dgemm algorithm is implemented by following code if we set TRANSB and TRANSA into NoTRAN form.

```
1 static void HPL_dgemmNN( M, N, K, ALPHA, A, LDA, B, LDB,
      BETA, C, LDC )
2    const int                 K, LDA, LDB, LDC, M, N;
3    const double              ALPHA, BETA;
4    const double              * A, * B;
5    double                    * C;
6 {
7    register double           t0;
8    int                       i, iail, iblj, icij, j, jal,
         jbj, jcj, l;
9
```

```
10    for( j = 0, jbj = 0, jcj  = 0; j < N; j++, jbj += LDB,
          jcj += LDC )
11    {
12       HPL_dscal( M, BETA, C+jcj, 1 );
13       for( l = 0, jal = 0, iblj = jbj; l < K; l++, jal +=
             LDA, iblj += 1 )
14       {
15          t0 = ALPHA * B[iblj];
16          for( i = 0, iail = jal, icij = jcj; i < M; i++,
                iail += 1, icij += 1 )
17          { C[icij] += A[iail] * t0; }
18       }
19    }
20 }
```

We can easily find that the algorithm runs 3 for-loop and has a time complexity of $O(M \times N \times K)$. Since it is a matrix operation and it is independent from other operation, the HPL GPU version move the DGEMM algorithm into GPU platform to exert the power of GPU as the following figure shows [1].

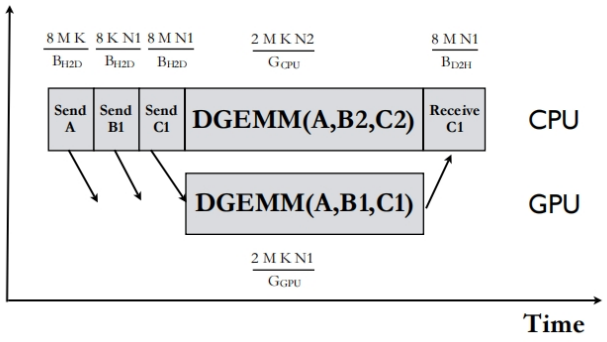


Figure 3.2: HPL DGEMM implemented by GPU

3.2 Platform Configuration and HPL Installation

In this section we describe the hardware and explain the software stack used in this work. Our research is based on the latest available version of hpl v2.3 and NVIDIA HPC-Benchmark 23.10. Real performance data was collected from two different platforms, one of which was used to test the CPU performance and one of which was used to test the GPU performance.

3.2.1 HPL CPU Platform A description

We chose the QiMing 2.0 cluster of Center for Computational Science and Engineering at SUSTech queue 1t75c as the HPL CPU test platform. This queue contains CPU of Intel(R) Xeon(R) Platinum 8375C CPU @ 2.90GHz. The details of the cluster can be seen in 1.6 .

The software description on platform A can be seen as follows.

Classification	Description	Installation Path	Version
OS	GNU/Linux Red Hat 8.5.0-10	-	Linux version 4.18.0
Math Library	Intel MKL	/work/intel/oneapi/mkl	2022.2
MPI	Intel MPI	/work/intel/oneapi/mpi	2021.6.0
Analysis Tool	Intel Vtune Profiler	/work/intel/oneapi/vtune	2024.0.0
Compiler	icc	/work/intel/oneapi/icc	2022.2.0
HPL	HPL CPU	-	2.3

3.2.2 HPL GPU Platform B description

We chose another smaller cluster with AMD EPYC 7773X 64-Core CPUs and three NVIDIA A100-SXM4-80GB GPUs to run HPL GPU version. The detail of the cluster can be seen in 1.7 .

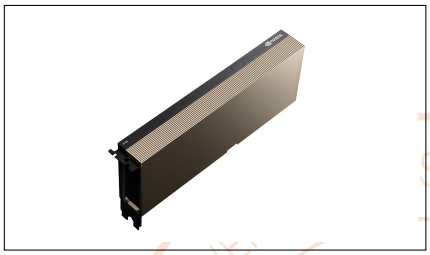


Figure 3.3: nvidiaA100

The software description on platform B can be seen as follows.

Classification	Description	Installation Path	Version
OS	GNU/Linux Red Hat 8.5.0-10	-	Linux version 4.18.0
Docker	Container	-	24.0.7
NVIDIA NVSHMEM	NVIDIA HPC-Benchmarks	/opt/nvidia	2.10.1
NVIDIA NCCL	NVIDIA HPC-Benchmarks	/opt/nvidia	2.16.5
CUDA Compiler	Nvidia CUDA	/usr/local/cuda	12.1
Math Library	Intel MKL	/home/asc648/intel/oneapi/mkl	2022.2
MPI	Intel MPI	/home/asc648/intel/oneapi/mpi	2021.6.0
Analysis Tool	Nvidia Nsight Compute	/usr/local/cuda/	2023.2.2
HPL	HPL GPU	-	23.10.0

3.2.3 Install HPL CPU version

First, we download the latest 2.3 version of HPL software from Innovative Computing Laboratory on https://netlib.org/benchmark/hpl .Unzip the file, and get a directory called hpl-2.3. To compile HPL, create a build folder first and then modify it based on the file setup/Make.Linux_Intel64. The content which should be modified is as follows.

```
tar -xvf hpl-2.3.tar.gz
cp hpl-2.3/setup/Make.Linux_Intel64 hpl-2.3/
cd hpl-2.3
vim Make.Linux_Intel64
```

TOP directory setups

```
TOPdir=/work/ssc-iscc/hpl-2.3
INCdir=$(TOPdir)/include
BINdir=$(TOPdir)/bin/$(ARCH)
LIBdir=$(TOPdir)/lib/$(ARCH)
HPLlib=$(LIBdir)/libhpl.a
```

MPI setups

```
MPdir=/work/share/intel/oneapi/mpi/latest
MPinc=-I$(MPdir)/include
MPlib=$(MPdir)/lib/libmpicxx.a
```

In the test we use Intel MPI. The Intel MPI Library is a multi-fabric message passing library that implements the Message Passing Interface.

There are several reasons why we chose Intel MPI.

First of all, it has high performance optimization. Intel MPI is highly optimized for Intel processors, leveraging various features of Intel architecture to achieve superior performance. This includes instruction set optimizations and performance tuning tailored for Intel processors, often resulting in better communication performance on Intel hardware.

Secondly, it contains good Integration with Intel Libraries and Compilers. Intel MPI integrates well with Intel MKL, harnessing optimizations to improve application performance.

Thirdly, when using Intel MPI, we benefited from other Intel tools such as Intel@Vtune Performance Analyzer, which can determine the HPL hotspot function for later program improvement and optimization.

Last but not least, Intel MPI supports large-scale parallel computing, making it suitable for systems with thousands of processors. It incorporates optimizations and techniques to ensure good scalability on large clusters, which will surely benefit our later HPL testing.

By using the Intel MPI library, we could compile and run HPL on multiple cluster interconnects.

MKL setups

```
LAdir   =/work/share/intel/oneapi/mkl/latest
ifndef  LAinc
LAinc       =$(LAdir)/include
endif
ifndef  LAlib
LAlib= -L$(LAdir)/lib/intel64\
        -Wl,--start-group\
        $(LAdir)/lib/intel64/libmkl_intel_lp64.a \
        $(LAdir)/lib/intel64/libmkl_intel_thread.a \
        $(LAdir)/lib/intel64/libmkl_core.a \
        -Wl,--end-group -lpthread -ldl
endif
```

In the test we chose the Intel MKL on the platform as the BLAS we use. The Intel MKL contains BLAS level1,2,3 and LAPACK and many other Math Libraries, which support linear algebra operations, including linear equation solving, eigenvalue problems, matrix factorization. It will definitely support for computing the HPL test. Also, according to the testing we did before, the Intel MKL did much benefit to the performance of system floating-point operations than other BLAS like OpenBLAS. The integration and Multi-thread support let it became our choice.

GNU Compiler setups

```
CC=/work/share/intel/oneapi/mpi/latest/bin/mpiicc
CCNOOPT=$(HPL_DEFS)
OMP_DEFS=-qopenmp
CCFLAGS=$(HPL_DEFS) -O3 -w -ansi-alias -i-static -z
    noexecstack -z relro -z now -nocompchk -Wall
```

According to the Intel Compiler installation path on the test platform, we modified the CC compiler path information to the path to mpiicc. We also add -qopenmp to help compile the Compiler.

Start Compiling

After modifying the Make.Linux_Intel64 file, we compiled HPL. The command is as following:

```
make arch=Linux_Intel64
```

The compilation ended correctly, and the file 'HPL.dat' and 'xhpl' will be generated in the bin/Linux_Intel64 directory. 'HPL.dat' is the configuration file and 'xhpl' is the test program which should be executed. At this point, we finished the compilation of the HPL and related software. Then we can run the hpl cpu version as following code.

Testing

```
mpirun -n 24 ./xhpl >> answer.txt &
```

If the answer.txt has the result, then we successfully install the HPL CPU version.

3.2.4 Install HPL GPU version from NVIDIA

The NVIDIA HPC-Benchmarks collection provides HPL benchmark for performance on NVIDIA accelerated HPC systems.

NVIDIA's HPL benchmarks provide software packages to solve a (random) dense linear system in double precision (64 bits) arithmetic and in mixed precision arithmetic using Tensor Cores, respectively, on distributed-memory computers equipped with NVIDIA GPUs, based on the netlib HPL benchmark. To get it, we need to use docker to pull the Nvidia HPC Benchmarks image as the following codes.

```
docker pull nvcr.io/nvidia/hpc-benchmarks:23.10
```

This command will fetch the hpc-benchmarks:23.10 image from the Nvidia Container Registry (nvcr.io). Once finished, we upload HPL.dat file into the image.

```
FROM nvcr.io/nvidia/hpc-benchmarks:23.10
COPY HPL.dat /workspace
```

Then we can use the following command to run HPL GPU version.

```
docker build -t hpl:N .
sudo docker run --gpus all --ipc=host --ulimit memlock=-1
    --ulimit stack=67108864 hpl:N mpirun -n 1
    /workspace/hpl.sh --dat $dat  --exec-name
    /workspace/hpl-linux-x86_64/xhpl >> Ns_log.txt
```

3.3 HPL CPU HPC Performance Characterization Analysis and Optimization

3.3.1 HPL CPU HPC Performance Characterization Analysis

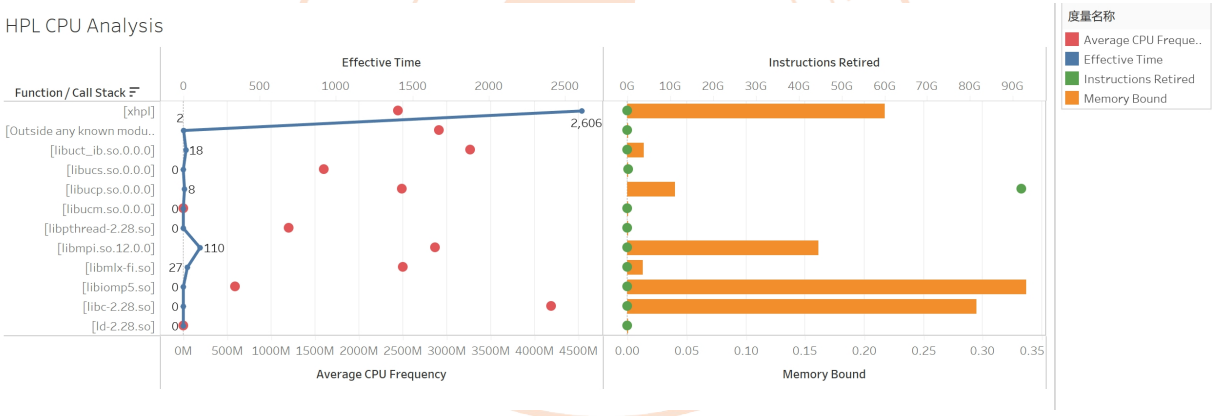


Figure 3.4: HPL HPC Performance Characterization

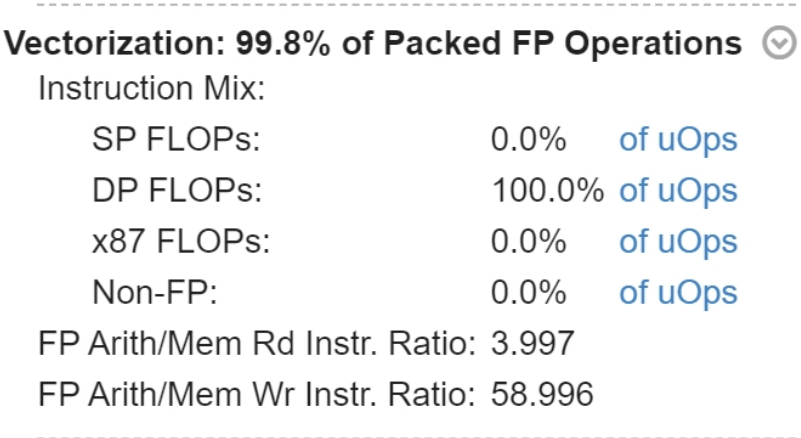


Figure 3.5: HPL Vectorization

Figure above shows the details of HPL HPC Performance Characterization during HPL execution, which was measured by the Intel@Vtune Performance Analyzer.

We find that most of the performance in LU factorization needs FP operations.Floating-point operations involve mathematical computations with real numbers, which are represented in a floating-point format to handle a wide range of values. The DP Gflops is 49.592, which is mixed in vectorization in large scale. We analyze the communication between nodes and cores. We can find that libmpi use 110s time to communicate between nodes, which can be saved if we take optimization. Also, The average frequency of xhpl is 2.45GHz, which is lower than our CPU maximum frequency.If we can higher our frequency, we may get xhpl run faster. The memory bounding tells us the memory bounded by the library or program. Since its maximum is still lower than our node's maximum memory, we can try to scaling up the N parameter and get full use of our memory.

3.3.2 HPL CPU Optimization

Before our optimization start, we first compute the

Theoretical Peak (in FLOPs) = Number of Cores × Frequency (in GHz)

×FLOPs per Cycle × 10^9 × Number of CPUs(3.9)

In our case using `Intel(R) Xeon(R) Platinum 8375C CPU @ 2.90GHz`,which has the average frequency of 2.90GHz and maximum frequency of 3.50GHz. we can calculated by the maximum frequency that

$$R_{peak} = 24 \times 3.50 \times 32 \times 10^9 \times 1 = 2688\text{GFlops} \qquad (3.10)$$

Selection of the suitable size of the problem

We use the computation from HPL Calculator Estimate Your Supercomputer's Rank Among the TOP500 - Mohamad Sindi - 2009 (sourceforge.net) to boost our tuning. We take our node information as input:

Number of Nodes	1
Cores Per Node	24
Speed Per Core (GHz)	2.6
Memory Per Node(GB)	128
Instructions Per Cycle	32

Table 3.1: Node input

Then the Calculator returns some suitable size of testing features. After that we design the Tuning according to our recent year experience in HPL. We first make the decision of the following parameter:

P,Q	4,6
PMAP	Column-major
PFACTs	left
threshold	16.0
NBMINs	2
NDIVs	2
RFACTs	left
BCASTs	BelongM
DEPTHs	0
SWAP	2

Table 3.2: HPL.dat unchange input

Here we make the decision by the following year's experience. The following paragraphs is the explanation of each parameter.

P × Q represents the number of parallel processes. This is the HPL specification. For the cluster nodes we used for testing, P × Q = 24. In theory, P ≤ Q will have a better floating-point performance, because column-oriented traffic is much costing than horizontal communication. Also, P,Q should be near to each other for factoring the square matrix. After all possible combinations of P × Q = 24 are tested by former testing, tests shows that for P=4,Q=6 shows best result for performance.

PMAP is used to select whether the processor array is arranged by column or by row. The default choice in HPL is by column.

The elimination algorithm parameter combination includes PFACT, NBMIN, NDIV, and RFACT. The LU decomposition of the Panel is based on matrix multiplication and recursion, decomposing the Panel into NDIV sub-panels. The Panel decomposition algorithm here is specified by RFACT. When the current number of Panel columns is not greater than NBMIN, recursion stop. At this time, HPL uses a decomposition operation based on matrix vector multiplication. The decomposition algorithm is specified by PFACT. In LU decomposition, there are many specific decomposition algorithms. When HPL eliminates each small matrix, three algorithms are used: Left(L), Right(R), and Crout(C). The ranges of PFACT and RFACT are 0, 1,and 2, indicating L decomposition, R decomposition, and C decomposition, respectively. The NBMIN and NDIV values should be 2 or 4.

In the main loop of the algorithm, the current Panel column is broadcast along the virtual loop topology. HPL provides six broadcast modes: 1ring, 1ringM, 2ring, 2ringM, Blong, and BlongM. The corresponding BCASTs value range is 0-5.We take BlongM as the broadcast choice.

Since L's decomposition process is a time-consuming process, in order to improve performance, HPL uses the method which first performs a partial decomposition and then broadcasts the results of this part. DEPTH's value indicates the times L is broadcast. The DEPTHs value ranges from 7 to 0. 0 means that only after the current Panel update is fully completed, the next Panel can start decompose; while 1 means that the next Panel begins to update at the same time. Here we take 0 as it can reach the highest performance according to formal testing.

To test whether our former experience is working. We first take different factor including PMAP,PFACTs,RFACTs and so on. And we take 16 size of N from 35896 to 87926, NBs=192 to test experience and our formula.

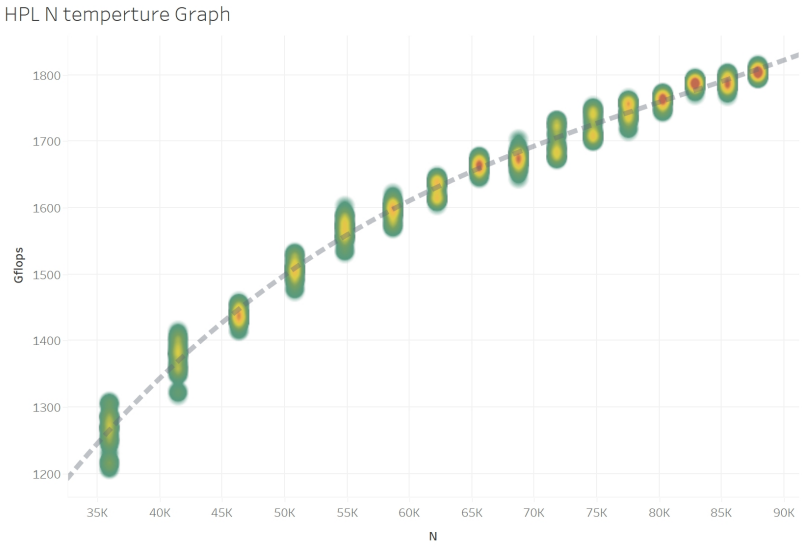


Figure 3.6: HPL N tuning Heat map graph

And we test it for 576 times, each N for 36 times to different factor. Finally we generate a **Heat map graph 3.6** . In the heat map, the greener the graph is, less data is on that location. And we can find that as N goes up, the data become dense and has come to a major computing point.

The graph reveal several discovery. Firstly, as problem size N goes up, the performance goes up in cubic as the regression line shows, which R square is 0.9892. Secondly, As the problem size increases, the performance peak of HPL appears to be more concentrated, with its performance becoming less influenced by other factors. Also, its performance center is closer to the regression line as problem size rises up. Thirdly, we analyze the peak when N is same. We found that when the N rise to 50764, most of the highest performance is exactly the input we take in experience. That shows out that our experience still have strong impact on tuning.

Then we start to tuning HPL with different parameter.We can make the decision of the following parameter: N, NB.

Selection of different N

N is the dimension of the solved linear system Ax = b. Theoretically, the value of N needs to consider the ratio of node memory capacity and the operation access. Larger matrix scales will increase the ratio of computing and communication for better performance. Based on the empirical formula:

$$N \approx \sqrt{memory\,(GB) \times 1024^3 \times \alpha / 8} \tag{3.11}$$

Where α less than 90% is suitable. Then we choose the parameter of N when α from 80% to 94%, and the difference of α is 2%

We first take these parameter in the table into the HPL.dat file and test the performance of HPL. The result is shown in the figure 3.7.

N	104832 107328 110016 112704 115200 117888 125076
NBs	192
P,Q	4,6

Table 3.3: HPL.dat N parameter tuning

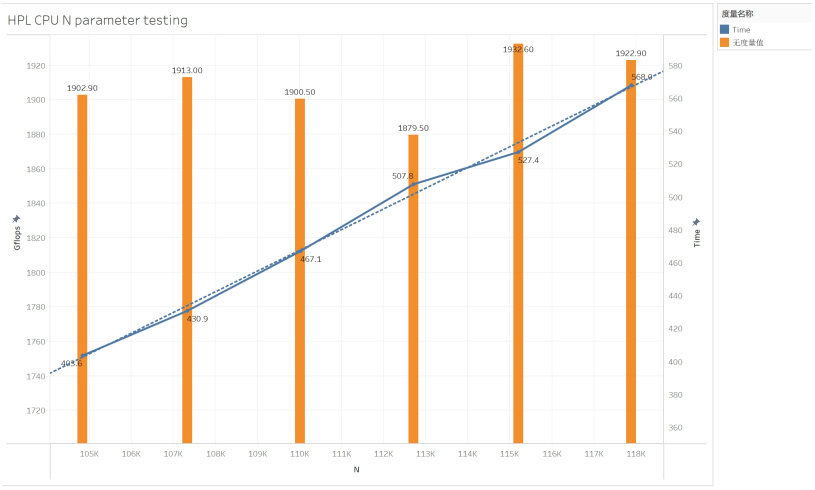


Figure 3.7: HPL N tuning

From the graph we know that the maximum Gflops happens in N=115200, the Gflops goes to R = 1932.6. The average Gflops is 1909.7. If we take CPU frequency as 2.5GHz as VTune analysis shows, we compute the R_{peak1} as:

$$R_{peak1} = 24 \times 2.50 \times 32 \times 10^9 \times 1 = 1920 \text{ GFlops} \tag{3.12}$$

Since all of the result are between 1850 to 1950, we consider that most of test has the CPU frequency near 2.5GHz.

The efficiency of the benchmark is

$$\eta = \frac{R}{R_{peak}} = \frac{1932.6}{2688} = 71.875\% \tag{3.13}$$

We can find that our performance efficiency is 71.875%.

We also see that the time cost of the HPL program CPU version is linear related to the problem size N. This is because the program factorize the problem in NBs blocks. For each same size blocks, the program takes the same time to run the algorithm. As the problem size scale up, the number of the blocks need to be factorized linearly goes up, which let the spending time goes up.

Selection of different NB

In HPL program, the matrix is split into the block sized NB to cyclically allocated to each process, where NB refers to the block granularity (size). The choice of NBs is related to many hardware and software factors. The current experience thinks that, from the perspective of data distribution, NBs can optimize the load balance between processes and make the use of CPU concentrated on computing instead of waiting or communicating when take a small number. However, think from the computational point of view, smaller NBs increase communication overhead and lead to overall performance degradation. So the size of the NBs should be as close as possible to the size of the Cache line, which can not only exert the Catch performance but also reduce the cache conflict.

Now we try to find out the NBs selection effect on the performance. We take the following table parameter as HPL.dat input.If for better

performance, We can slightly change the size of N in order to make NBs can divide N, so that the program can do computation in full blocks without lossing performance. In this testing section, we take N as same merely to test NBs effect on performance.

N	104832				
NBs	128	160	192	224	256
P,Q	4,6	4,6	4,6	4,6	4,6

Table 3.4: HPL.dat NB parameter tuning

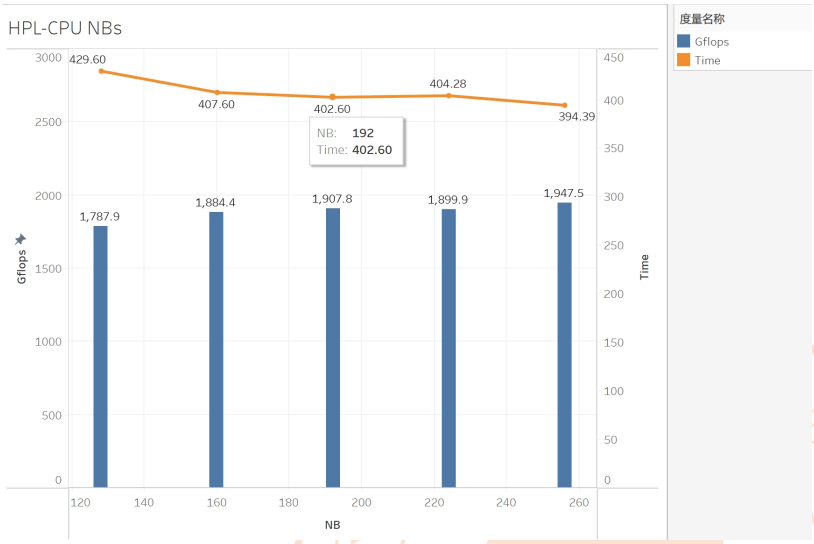


Figure 3.8: HPL NBs tuning

And here we get the result as 3.8 shows.

From the result we know that as the NBs goes up, the performance has a trend of rising up. When NBs = 256, we find that the performance goes to 1947.5 Gflops. Also we find that when NBs = 192, we have a local maximum of performance, which is the previous best choice when we compute HPL on v3-64 queue though it is not the best choice on 1t75c.

Compilers and Optimization Flags

We now consider to optimize from compiler since the DGEMM algorithm may be compiled into machine codes in different ways.We change the compilation code as following:

```
1 #Original
2 #CCFLAGS  = $(HPL_DEFS) -O3 -w -ansi-alias -i-static -z
      noexecstack -z relro -z now -nocompchk -Wall
3 #Now
4 CCFLAGS  = $(HPL_DEFS) -O1 -w -ansi-alias -i-static -z
      noexecstack -z relro -z now -nocompchk -Wall
```

Then we run N=115200,NBs=192 to see whether they have difference. The result can be seen in figure 3.9.

As figure shows, it can be seen that, with different compilation options, the performance trend of system floating-point operations is going down, the peak value of -O3 floating-point operations is higher than

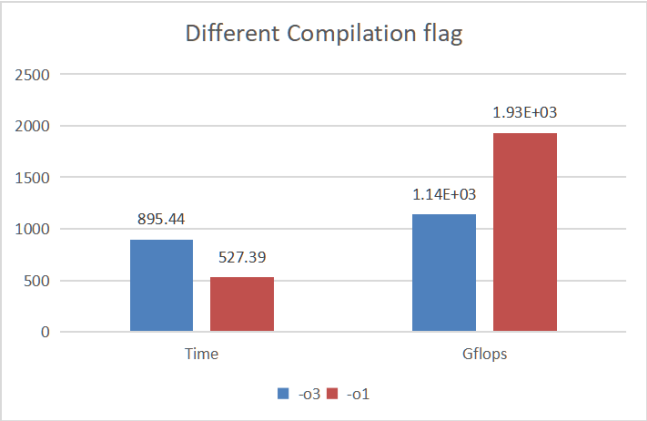


Figure 3.9: HPL CPU Compile running Result

-o1 options.Also -o3 uses nearly just two third of the time that -o1 uses. optimizes have little effect to floating-point peaks.

3.3.3 HPL CPU Benchmark result

For 24 core, we find a series of parameter of the following table to get high performance of the program. The total peak is 1947.5 Gflops.

Table 3.5: HPL best

N 256 P,Q	115200 NBs 4,6
PMAP	Column-major
PFACTs	left
threshold	16.0
NBMINs	2
NDIVs	2
RFACTs	left
BCASTs	BelongM
DEPTHs	0
SWAP	2

With the theoretical performance, we calculate our HPL efficiency, which is:

$$\eta = \frac{R}{R_{peak}} = \frac{1947.5}{2688} = 72.45\% \tag{3.14}$$

That is the end of our HPL CPU benchmark.

3.4 HPL GPU HPC Performance Characterization Analysis and Optimization

In this section, we are going to run HPL program on GPU platform B. We test serval factor of HPL including N,NBs. We also try to run multiple GPUs to test what performance will change during multiple GPUs. What's more, since our cluster has a power limit of 3000W, we try to limit the GPU power to test what performance will change during

the power changing. And since the platform B is AMD structure, intel VTune Profiler can't run at the platform.

3.4.1 HPL GPU Performance Analysis

We try to use NVIDIA Nsight Systems to run HPL GPU Analysis. NVIDIA Nsight Systems is a low-overhead performance analysis tool designed to provide developers with insights necessary for optimizing software. Unbiased activity data is visualized in the tool, helping users investigate bottlenecks, avoid inference misreports, and increase the likelihood of achieving higher performance. Users will be able to identify issues such as GPU idle time, unnecessary GPU synchronization, insufficient CPU parallelization, and even unexpectedly expensive algorithms on their target platform. It is designed to scale across various NVIDIA platforms, including large Tesla multi-GPU x86 servers, Quadro workstations, Optimus-supported laptops, DRIVE devices with Tegra+dGPU running multiple operating systems, and Jetson. NVIDIA Nsight Systems can also provide valuable insights into the behavior and workload of deep learning frameworks like PyTorch and TensorFlow, allowing users to fine-tune models and parameters to improve overall utilization of single or multiple GPUs.

However, the HPL provided by NVIDIA needs to run in the docker container which has complex environment then the normal cuda and nvhpc environment. It requires library likes nccl.so.2,which is easy to locate. But it also needs nvshmem_host.so.2 library, which is not include in our cuda-12.2 since we only have nvshmem_host.so . We have tried to make a soft connect to the library, however, the program runs into fault maybe the library version is different.

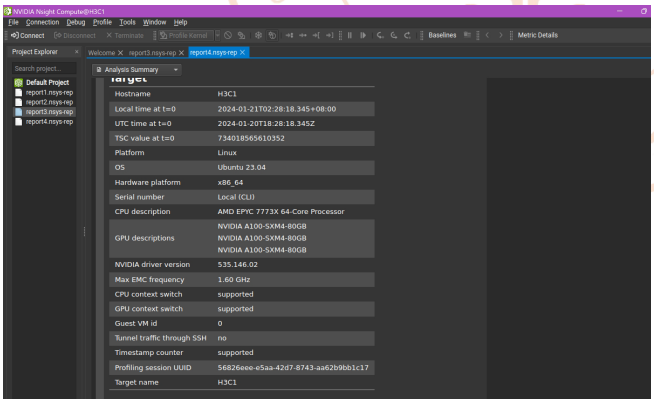


Figure 3.10: HPL Nsight running

So in the end we try to optimize the HPL GPU benchmark using our previous experience in CPU.

3.4.2 HPL GPU Optimization

Selection of the suitable size of the problem

HPL GPU N tuning Now we are going to choose the right size of HPL. We first decide the size of N as it may be the most essential figure in

HPL GPU program. We use table 3.6 to test. The test result can be seen in figure 3.15.

N	10240 20480 30720 40960 51200 61440 71680 81920 92160 102400
NBs	192
P,Q	1,1

Table 3.6: Testing GPU N data

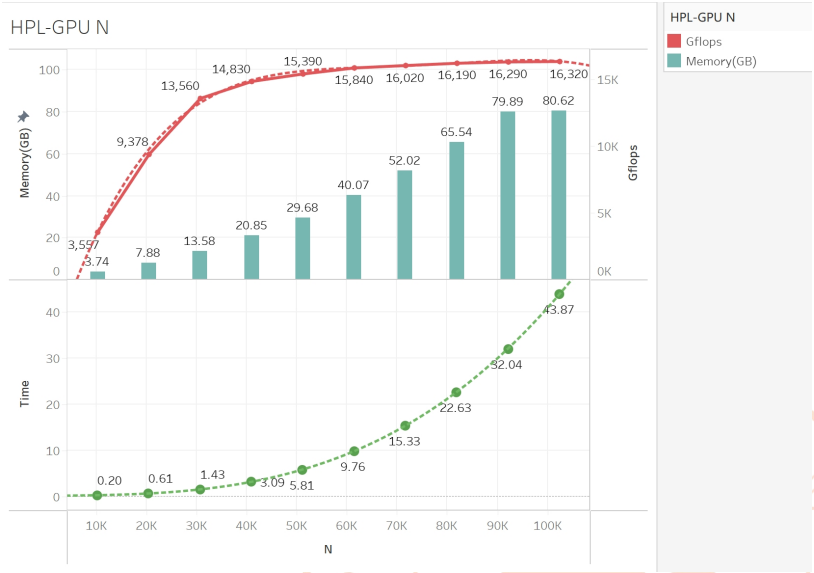


Figure 3.11: HPL GPU running N

We can find that as N rises up, the performance rises in N^4 as the regression line contains R square of 0.9981. Also we can find that when N=92160,the rises becomes growing slowly, and when N=102400 the performance nearly don't change. We find that at that time our memory becomes 80.62GB, which is nearly our maximum memory of GPU. We use our empirical formula and use $\alpha = 0.8$ to calculate the problem size:

$$N \approx \sqrt{80 \times 1024^3 \times 80\%/8} = 92681.9$$

Which is exactly our expectation of N since N=92160 has nearly comes to the limitation. So the formula also exists when running the HPL GPU version. However, since the GPU can do communication during the factorization algorithm, so actually when we scale up the problem size, the GPU first will balance the algorithm and communication so that its performance still rises. So we consider that our limitation on GPU may be caused by the limitation of GPU memory and result in extra communication between CPU and GPU. This communication happens using PCIe, which has a communication frequency limitation and cause time and efficiency down.

What's more, we analyse that our spending time of HPL GPU is N^3 to the problem size with regression line's R square of 0.999997. That may means that the algorithm runs in $O(N^3)$. We consider the most cost time algorithm: DGEMM algorithm. We know before in CPU we calculate the DGEMM in $O(M \times N \times K)$ [1] , however in NVIDIA GPU, we have multiple SM(Streaming Multiprocessor) helps us to calculate DGEMM[2]. On A100 GPU, it has 108 SM to help calculate with each SM have 64

FP/32 core and 32 FP/64 core, which can do 256 FP's FMA operation according to NVIDIA Ampere Architecture Whitepaper.[3] So it may runs time in $O\left(\frac{N^3}{n_{sm}}\right)$, which lower much of the calculating time.

Using this table, we can calculate the peak computational capability of the GPU with the following formula:

$$\text{Peak FLOPS} = F_{\text{clk}} \times N_{\text{SM}} \times T_{\text{ins}} \times 2 \tag{3.15}$$

Here, F_{clk} represents the operating frequency of the GPU cores, N_{SM} is the number of GPU Streaming Multiprocessors, T_{ins} is the instruction throughput for a specific data type, and the multiplication by 2 is because a fused multiply-add (FMA) is considered as two floating-point operations.

For example, considering the A100 FP64 CUDA Cores with an instruction throughput of $T_{\text{ins}} = 64$, a core operating frequency of $F_{\text{clk}} = 1.41$ GHz, and a total number of streaming multiprocessors $N_{\text{SM}} = 108$, the calculation would be:

$$\text{Peak FLOPS} = 1.41 \times 108 \times 64 \times 2 = 19,491\,\text{GFLOPS} \tag{3.16}$$

Comparing this result to the FP64 peak computational capability mentioned in the NVIDIA Ampere whitepaper [3] of 19.5 TFLOPS, the values are essentially consistent.

GPU Features	NVIDIA Tesla P100	NVIDIA Tesla V100	NVIDIA A100
GPU Codename	GP100	GV100	GA100
GPU Architecture	NVIDIA Pascal	NVIDIA Volta	NVIDIA Ampere
GPU Board Form Factor	SXM	SXM2	SXM4
SMs	56	80	108
TPCs	28	40	54
FP32 Cores / SM	64	64	64
FP32 Cores / GPU	3584	5120	6912
FP64 Cores / SM (excl. Tensor)	32	32	32
FP64 Cores / GPU (excl. Tensor)	1792	2560	3456
INT32 Cores / SM	NA	64	64
INT32 Cores / GPU	NA	5120	6912
Tensor Cores / SM	NA	8	4[2]
Tensor Cores / GPU	NA	640	432
GPU Boost Clock	1480 MHz	1530 MHz	1410 MHz
Peak FP16 Tensor TFLOPS with FP16 Accumulate[1]	NA	125	312/624[3]
Peak FP16 Tensor TFLOPS with FP32 Accumulate[1]	NA	125	312/624[3]
Peak BF16 Tensor TFLOPS with FP32 Accumulate[1]	NA	NA	312/624[3]

Figure 3.13: NVIDIA Ampere Architecture parameter

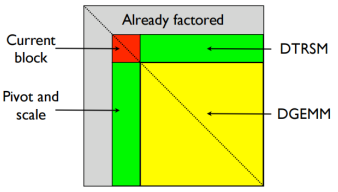


Figure 3.12: HPL Factorization

```
// Copy A from CPU memory to GPU memory devA
   status = cublasSetMatrix (m, k    , sizeof(A[0]), A, lda, devA, m_gpu);
// Copy B1  from CPU memory to GPU memory devB
   status = cublasSetMatrix (k ,n_gpu, sizeof(B[0]), B, ldb, devB, k_gpu);
// Copy C1  from CPU memory to GPU memory devC
   status = cublasSetMatrix (m, n_gpu, sizeof(C[0]), C, ldc, devC, m_gpu);

// Perform  DGEMM(devA,devB,devC) on GPU
// Control immediately return to CPU
   cublasDgemm('n', 'n', m, n_gpu, k, alpha, devA, m,devB, k, beta, devC, m);

// Perform  DGEMM(A,B2,C2) on CPU
   dgemm_cpu('n','n',m,n_cpu,k, alpha, A, lda,B+ldb*n_gpu, ldb, beta,C+ldc*n_gpu, ldc);

// Copy devC  from GPU memory to CPU memory C1
   status = cublasGetMatrix (m, n, sizeof(C[0]), devC, m, C, *ldc);
```

Figure 3.14: HPL factorization on DGEMM

HPL GPU NBs tuning We also try to find if NBs solution can affect the performance on GPU program.

N	102400				
NBs	512	768	1024	1280	1408

Table 3.7: HPL GPU NBs parameter tuning

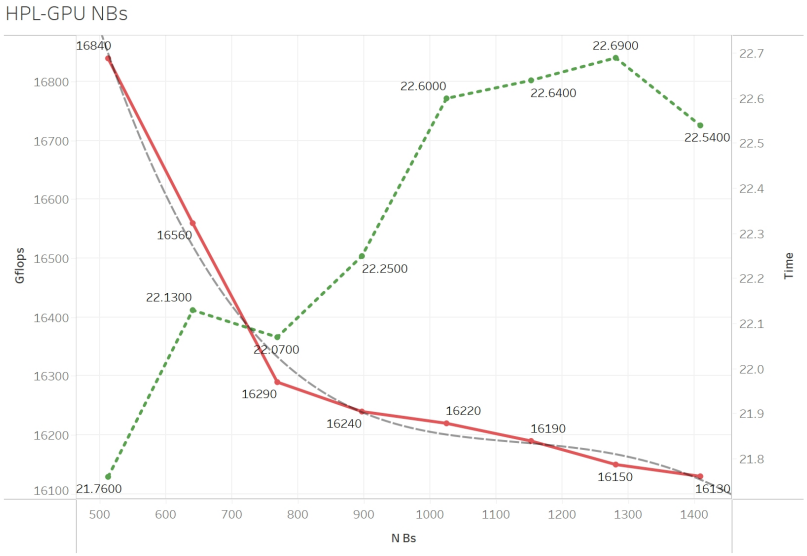


Figure 3.15: HPL GPU NBs running result

What we find is that NBs seems don't have too much effect on performance. However, lowering the NBs may help the performance goes higher. With less NBs blocks, the efficiency goes higher and time cost goes lower.

h!

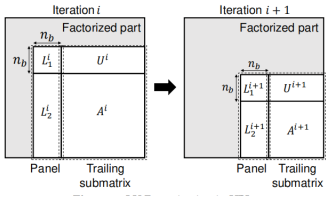
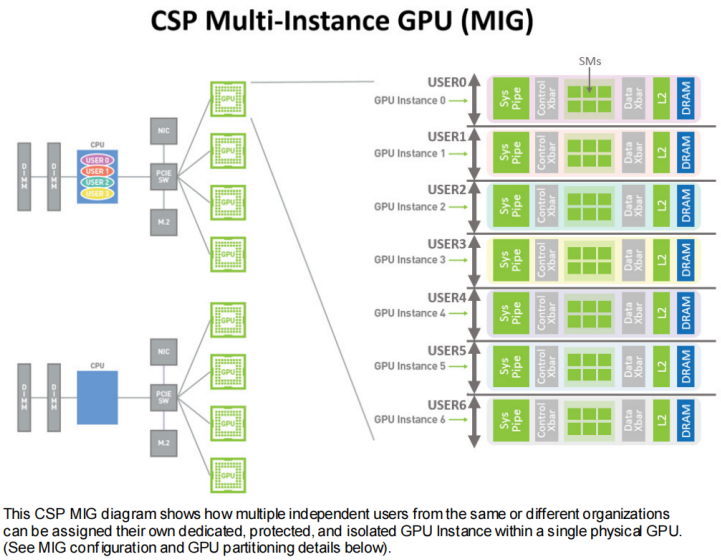


Figure 3.16: HPL factorization2



This CSP MIG diagram shows how multiple independent users from the same or different organizations can be assigned their own dedicated, protected, and isolated GPU Instance within a single physical GPU. (See MIG configuration and GPU partitioning details below).

Figure 3.17: A100 CSP Multi-Instance GPU

We consider the reason of SM. In each of the Factorization cycle, SM will get a blocks of matrix and do FP operations. However, with higher NBs, each SM needs to handle a bigger problem size of matrix, inside of SM the four tensor core will communicate with each other by L1 Instructure Cache, between each of the SM they will communicate with each other by L2 Cache. When they try to communicate, they will use L2 Cache as figure 3.17 shows. So the total performance may be down since the communication cost rises up.

Also, we may learn that we should use the the number of multiple of SMs like 108,216,532.

Figure 3.18: A100 SM Architecture

Multiple GPU solutions

We now test multiple GPU on a single node to run a HPL program for N=115200 and NBs=1024, and see whether the performance will be like. And the result can be seen on figure 3.19 .

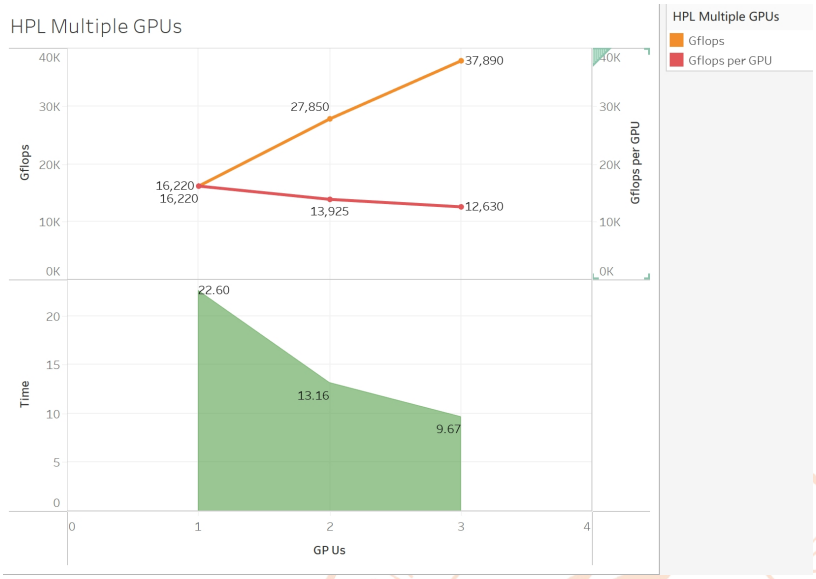


Figure 3.19: HPL Multiple GPU running

We find that when GPU Number scales up, the time cost lower down and the performance rises. As the number of GPU rises up, the performance per GPU drops. It is easy to understand the situation since more GPU brings more Calculations and more communication costs.

Power control effect on HPL performance

During our competition, we faced the challenge of configuring a computing cluster to operate under a stringent total power constraint of 3000W. This necessitated a series of meticulous experiments to understand how various power caps could impact the performance of our HPCG benchmarks.

We have used `nvidia-smi -pl <target-limit>` to control the limits of the power. And we test the performance when GPU power limit is 400w, 350w, 300w, 250w, and 200w. The results are shown in Figure. 3.20.

From the result we can see that as we lower the power, the performance drop quickly starting from power=350w. And when the power is about 200w, the performance is bad and is only about one third of the power when power is 400w.

We use the performance Gflops to divide the power to get average Gflops per power. And we find that when power = 350w, the GPU can extend its max power usage to get performance. Which means that if we can find a maximum of Gflops per power, we may get a nice tuning on HPL GPU that GPU can do the best with the prioritized resources.

And the time costing rises up for the lower power with the relation of P^-3. This is because as the power is down, with whole work unchanged, it will seem like the problem size for power limited GPU scales up in

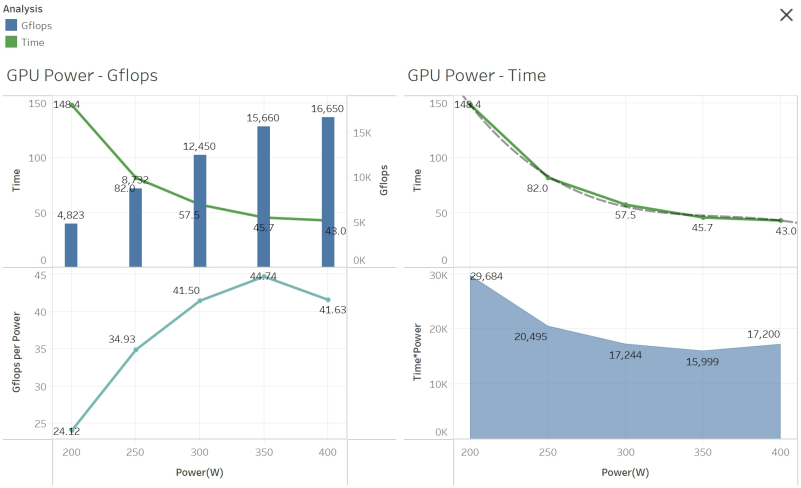


Figure 3.20: Results of the HPL performance with different power limits

$O(N^3)$. If we multiple power and time to get the whole work done by the GPU, we can get a rough total works done by GPU. We can see that when power is 350w, GPU needs to do less work in total, which can be a good feature to measure the efficiency of GPU running.

3.4.3 HPL GPU Benchmark result

During our benchmark, we find that when we take N=102676, NBs=532 on a single GPU, we can find a maximum efficiency of HPL GPU version is 1.689e+04 Gflops on a single GPU. We have tried NBs=512 and we find that we can't reach the maximum when NBs=532. Theoretically it is also the right result for choosing the num suitable for SM. And for N we use empirical formula to have a num that benefit both on communication and memory.

From the NVIDIA Ampere Architecture White-paper we know that the Peak FP64 TFlops(non-Tensor) is 19.5, which is 19500Gflops. And our maximum efficiency is that:

$$\eta = \frac{R}{R_{peak}} = \frac{16890}{19500} = 86.62\% \tag{3.17}$$

We reach a 86.62% efficiency in HPL GPU test with 16890 Gflops on a single GPUs.

4 HPCG Benchmark Test

4.1 HPCG Algorithm Introduction

HPCG is a benchmark program that solves a sparse linear system arising in solving a three-dimensional heat diffusion problem.

HPCG was proposed because High Performance Linpack(HPL) were no longer so strongly correlated to real application performance, especially for the broad set of HPC applications governed by differential equations, which tend to have much stronger needs for high bandwidth and low latency, and tend to access data using irregular patterns.[4]

The main calculations pattern in HPL algorithms are dense matrix-matrix multiplication and related kernels. Another computation pattern, which has much lower computation-to-data-access-ratios, irregular memory access and fine-grain recursive computations, is crucial for reflecting real system capability.

As explained in the technical specification[5], HPCG intends to solve the linear system generated from a model problem, a single degree of freedom heat diffusion model problem with zero Dirichlet boundary conditions. Such linear system is synthetic, discretized and sparse.

The sparse linear system solved in HPCG is $Ax = b$. Here A is a sparse matrix obtained from the 27-point discretization of the Poisson equation, b is the right-hand-side vector generated by using an exact solution with all one's. In HPCG, sparse matrices and vectors are stored distributively among MPI processes corresponding to the decomposed subdomains. It is allowed to modify the data formats of both sparse matrices and vectors in HPCG to achieve better performance, but it is prohibited to take advantage of the specific structure of the sparse matrix (such as the diagonal structure, the specific values of nonzero entries, etc), nor is it permitted to replace sparse matrices with stencil computations.

In HPCG, the linear system $Ax = b$ is solved by using a Preconditioned Conjugate Gradient (PCG) algorithm as follows, where M is the multigrid preconditioner to be elaborated later. In the main loop of the algorithm, the major cost consists of:

- One sparese matrix vector multiplication (SpMV) in line 12
- One preconditioner using Symmetric Gauss-Seidel algorithm (SYMGS) in line 3
- Three vector dot-products (DDOT) in line 6, line 12 and line 15
- Three vector updates (WAXPBY) in line 10, line 13 and line 14

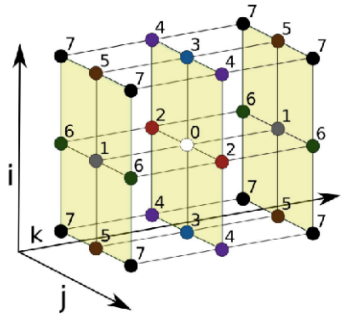


Figure 4.1: 27-point stencil operator

```
1  p_0 ← x_0, r_0 ← b − Ap_0
2  for i = 1,2,..., to max_iterations do
3    z_i1 ← M^{-1} r_{i-1}
4    if i = 1 then
5      p_i ← z_i
6      α_i ← dot_prod(r_{i-1}, z_i)
7    else
8      α_i ← dot_prod(r_{i-1}, z_i)
9      β_i ← α_i / α_{i-1}
10     p_i ← β_i p_{i-1} + z_i
11   end if
12   α_i ← dot_prod(r_{i-1}, z_i)/ dot_prod(p_i, Ap_i)
13   x_{i+1} ← x_i + α_i p_i
14   r_i ← r_{i-1} − α_i Ap_i
15   if ||r_i||_2 < tolerance then
16     STOP
17   end if
18 end for
```

Listing 4.1: Preconditioned Conjugate Gradient

If we want to look at the algorithm in an more abstract way we can get the Pseudocode of the HPCG main loop and the multi-grid routine. [6]

```
1  for i = 1, 2,..., to 50 and normr > error do
2      MG (A, r, z);
3      DDOT (r, t, rtz);
4      Allreduce (rtz); //MPI communication
5      if(i > 2) then
6          beta = rtz / rtzold;
7          WAXPBY (z, beta, p);
8      end if
9      ExchangeHalos (A, p);
10     SpMV (A, p, Ap);
11     DDOT (p, Ap, pAp);
12     Allreduce (pAp); //MPI communication
13     alpha = rtz / pAp;
14     WAXPBY (x, alpha, p);
15     WAXPBY (r, -alpha, Ap);
16     DDOT (r, r, normr);
17     Allreduce (normr); //MPI communication
18     normr = sqrt (normr);
19 end for
```

Listing 4.2: Main loop of the HPCG

```
1  if(depth < 3) then
2      ExchangeHalos ();
3      SYMGS ();
4      ExchangeHalos ();
5      SpMV ();
6      MG (depth++);
7      ExchangeHalos ();
8      SYMGS ();
9  else
```

```
    ExchangeHalos();
    SYMGS();
end if
```

Listing 4.3: The Multi-Grid routine

From the abstraction of the pseudocode, we can better understand how the functions and procedures are called during the test.

In the HPCG benchmark, the preconditioner M^{-1} is based on a four-level V-cycle within a geometric multigrid approach, as illustrated in a referenced figure and the above pseudocode. This multigrid cycle, which includes the finest grid as one of its levels, utilizes the symmetric Gauss-Seidel method for smoothing procedures—both before and after the main iteration—and also as the solver at the coarser grids. At every level of the cycle, only a single iteration of the symmetric Gauss-Seidel method is performed. For movement between different grid levels, straightforward injection and its corresponding transpose operation are employed for the grid transfer processes, which means for restricting and prolonging the grid data, respectively.

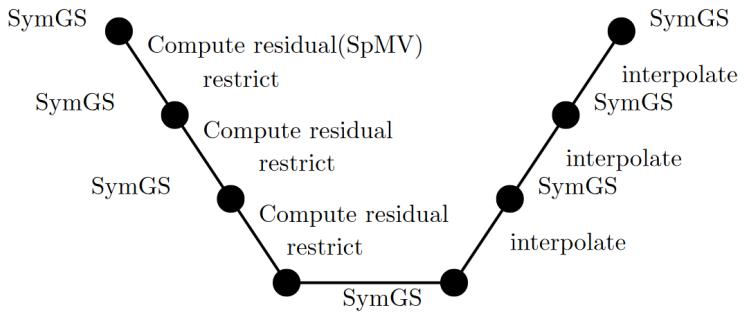


Figure 4.2: SymGS

4.2 Platform

In this section we describe the hardware and explain the software stack used in this work. Our research is based on the latest available version of HPCG v3.1 and NVIDIA HPC-Benchmark 23.10. Real performance data was collected from two different platforms, one of which was used to test the CPU performance and one of which was used to test the GPU performance.

Platform A hardware configuration

We chose the QiMing 2.0 cluster of Center for Computational Science and Engineering at SUSTech queue 38 as the HPCG CPU test platform. This queue contains 36 nodes with `Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz` Processors. Detailed information please refer to table 1.5.

Figure 4.3: Intel(R) Xeon(R) Gold 6338

Platform A software configuration

On platform A, we have used the Intel Base Toolkit to help run the task. The software configuration is:

Classification	Description	Installation Path	Version
Math Library	Intel MKL	`/share/intel/oneapi/mkl`	2022.2
MPI	Intel MPI	`/share/intel/oneapi/mpi`	2021.7.0
Analysis Tool	Intel Vtune Profiler	`/share/intel/oneapi/vtune`	2024.0.0
Compiler	icc	`/share/intel/oneapi/icc`	2022.2.0
CMake	cmake	-	3.25.2

Table 4.1: Software configurations on Platform A

Platform B hardware configuration

We chose another smaller cluster with `AMD EPYC 7773X 64-Core` CPUs and three `NVIDIA A100-SXM4-80GB` GPUs. Detailed information please refer to table 1.7.

Figure 4.4: NVIDIA A100-SXM4-80GB

Platform B software configuration

On platform B, we have used Docker to run the NVIDIA HPC Benchmark. The software configuration is:

Classification	Description	Installation Path	Version
Container	Docker	-	24.0.7
Cuda Compiler	Nvidia Cuda	`/usr/local/cuda`	12.1
Benchmark	NVIDIA HPC-Benchmark	˜	23.10
MPI	OpenMPI	`Inside the image`	4.1.5

Table 4.2: Software configurations on Platform B

4.3 HPCG Compilation

We first download the latest version of HPCG test package (hpcg-3.1.tar.gz) from hpcg official github page, unzip the file, and get a directory called hpcg_3.1. To compile the HPCG, create a build folder first and then modify it based on the file `setup/Make.MPI_ICPC`. The content which should be modified is as follows.

4.3.1 Message Passing Interface

```
MPdir = /share/intel/oneapi-2023.1/mpi/2021.9.0/
MPinc = -I$(MPdir)/include
MPlib = $(MPdir)/lib/release/libmpi_ilp64.a
```

We use Intel MPI by default, and modify the MPdir mapping according to the Intel MPI installation path on the test platform. To use the self-installed MPI, modify the MPdir to the correct path information.

4.3.2 Compile Time Option

```
HPCG_OPTS = -DHPCG_NO_OPENMP
```

The main modification here is the option we used when compiling. HPCG turns on OpenMP by default, But we closed the OpenMP option during the initial test.

4.3.3 Compilers / linkers - Optimization flags

```
CXX = /share/intel/oneapi-2023.1/mpi/2021.9.0/bin/mpiicpc
CXXFLAGS = $(HPCG_DEFS) -O3
LINKER = $(CXX)
LINKFLAGS = $(CXXFLAGS)
```

We modified the compiler path information for `CXX` and used the default `CXXFLAGS`. Also, we modified `Make.MPI_ICPC`, and then compile HPCG, the order is as follows:

```
cd ../build
../configure MPI_ICPC
make
```

If the compilation ended correctly, hpcg.dat and xhpcg will be generated in the directory build/bin, where hpcg.dat is the configuration file and xhpcg is the test program which should be executed. At this point, the compilation of HPCG is over.

4.4 Benchmark Result

In order to compare the subsequent optimization results and speedups, here we need to define the standard results. Since the official recommended that the HPCG can give an accurate result only when the runtime is longer than 3600s. We modified the `hpcg.dat` file in the build/bin, change the 60s to 3600s, but keep the original problem scale unchanged. Then we submit the test operations, the results are shown in Table 4.4.

Table 4.3: HPCG Benchmark Result CPU

Node Number	Core Number	n_x	n_y	n_z	Time/s	GFlops
1	64	104	104	104	3600	18.1204
1	64	104	104	104	60	18.3745

We use this test result as a benchmark, and its acceleration ratio $S_p = 1.00$. The theoretical floating-point peak value for a single dual blade node is known to be calculated by using the following equations.

$$\text{Theoretical Peak (in FLOPs)} = \text{Number of Cores} \times \text{Frequency (in GHz)} \times \text{FLOPs per Cycle} \times 10^9 \times \text{Number of CPUs} \tag{4.18}$$

In our case using `Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz`, we can calculated that

$$R_{peak} = 32 \times 2.00 \times 32 \times 10^9 \times 2 = 4096\text{GFlops} \tag{4.19}$$

The efficiency of the benchmark is

$$\eta = \frac{R}{R_{peak}} = \frac{18.1204}{4096} = 0.44\% \tag{4.20}$$

From the benchmark we can also know that the result get from 60s and 3600s will not differ significantly, the bulk tendency should be the same using either runtime. We will be using 60s runtime to perform our experiments further and achieve our final results with runtime 3600s to ensure accuracy.

4.5 Hotspot Analysis

Before the optimization work started, we use the Intel Vtune Performance Analyzer to determing the hot-spot function of the HPCG, and so we can identified the direction for subsequent optimization work.

Figure 4.5 shows the detail of the run time distribution of the sub-functions during the execution of the HPCG which was measured by the Intel Vtune Performance Analyzer. As can be seen, about 87% of the total CPU time is occupied by two main sub-functions, which are `ComputeSPMV_ref()` and `ComputeSYMGS_ref()`.

`ComputeSPMV_ref()` performs the sparse matrix and vector multiplication (SPMV) and `ComputeSYMGS_ref()` performs the symmetric Gauss-Seidel operations to make the multigrid preconditioner smooth and relaxed. Here "_ref" stands for "reference function" and refers to an

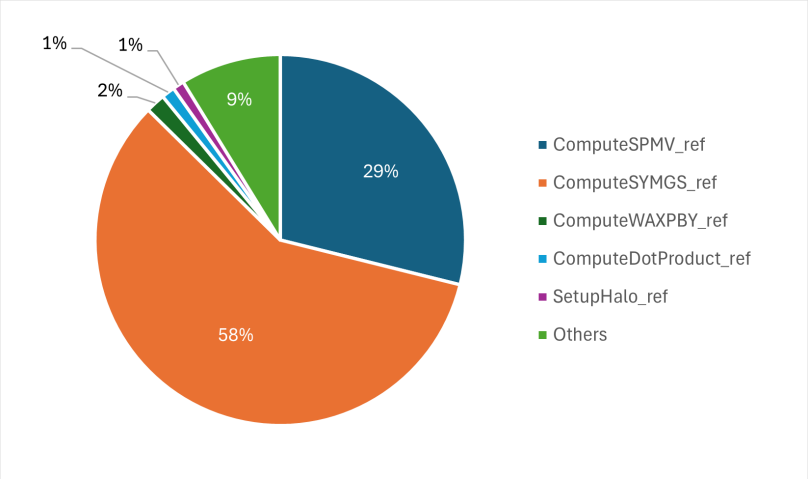


Figure 4.5: Breakdown of CPU time distribution of major functions in HPCG

original version — the first version without any modification. Although the figure above only represent the case of one node, all cases show the same trend in fact. Therefore, these two sub-functions are the focus of our subsequent improvement. The results are compatible with our analysis of the main loop of HPCG algorithm (Listing 4.2) and the Multi-grid routine (Listing 4.3)

4.6 HPCG Optimization on CPU Platform A

4.6.1 Compilers

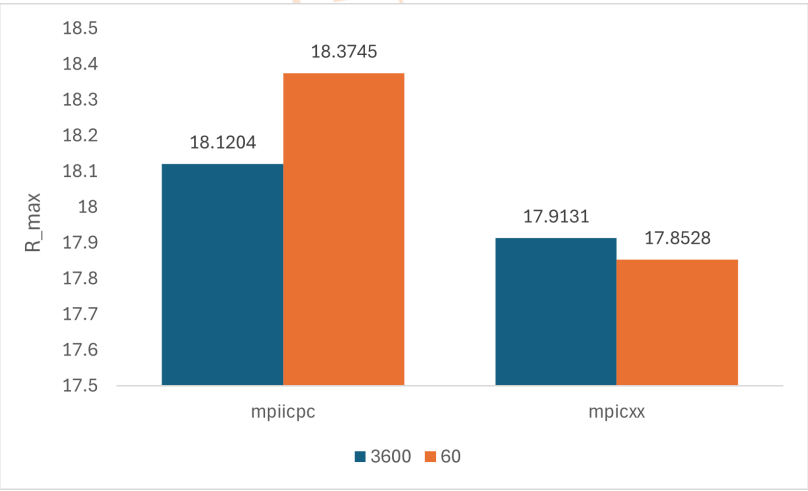


Figure 4.6: Different outcome using different Compilers

Since we have little understanding of HPCG source code and algorithms to the beginning of the optimization, we started with the simplest way, compiler selection is our first research direction. We have chosen two C++ compilers here: Intel's compiler `mpiicpc` and GNU's compiler `mpicxx`, then we tested the two compilers respectively. The result is shown in Figure 4.6. It is obvious that the The peak of the floating-point operation of the HPCG compiled by two compilers is basically the same. Therefore, In the case of $n_x = n_y = n_z = 104$, the two compiler compiled HPCG, whose floating-point computing peak is basically the same, but we use the Intel's compiler `mpiicpc` for subsequent tests.

4.6.2 Selection of the suitable size of the problem

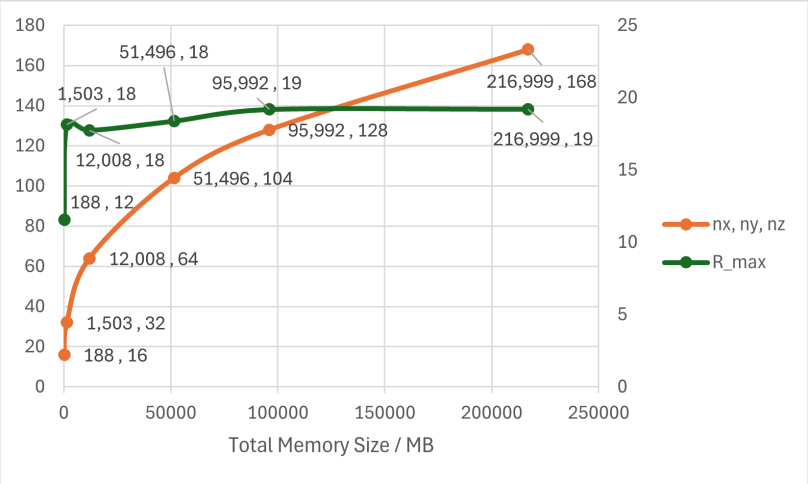


Figure 4.7: Different Performance over different problem size

Different from HPL, the benchmark test of HPCG offers fewer test parameters which can be adjusted. There are only two parameters in the hpcg.dat file: Problem size N and run time t. Above all, we test the problem size N. The number of startup processes is 64 and each processor starts 1 thread; the scope of the local domain problem is in the range of $n_x = n_y = n_z \in [16, 168]$. And the dimension for each processor is always $n_{p_x} = 4, n_{p_y} = 4, n_{p_z} = 4$. Therefore, the range of the total scale of the corresponding problems is $n_x \in [64, 672]$, $n_y \in [64, 672]$, $nz \in [64, 672]$. The test results are shown in figure 4.7.

As mentioned in various essays [6][7], a relatively high performance should be achieved in problem size $n_x = n_y = n_z = 16$. This is because the problem size does not exceed the L3 Cache limits. At this time, the performance of the system's floating-point operations is higher. When the data size exceeds the cache limit, the floating-point computing capability of the system is greatly reduced due to the speed of data transmission. In addition, we have expanded the problem size test and found that when the scale of the problem is $n_x = n_y = n_z = 176$, HPCG would not calculate the final result, that is, it could not be able to generate the .yaml. We also find that when $n_x = n_y = n_z \leq 16$, HPCG will set the problem size as $n_x = n_y = n_z = 16$. Therefore, we think that the scale of the problem will seriously impact the system's floating-point performance, and even will lead to the inability to calculate the results.

For our hardware, the L3 Cache Limits are 50 MB and the smallest problem size for 64 processors is 188MB, which has already exceeded the cache limits. Therefore, in our system, we fail to observe the phenomenon of having a relatively high R_{max} output.

However, in order to ensure that the tested performance is the real performance of the system, there must be some limitations when optimizing, among which the limitation of memory usage is the key, because the size of memory usage is also the limitation of the scale of the problem. On the one hand, we need to increase the scale of the problem to ensure that the run time is long enough; on the other hand, if the scale of the problem is too small, all data can be held in the caches, so that HPCG data access irregularity and memory overhead can not be reflected. Considering

the above two reasons, we believe that the memory size of the problem must be larger than $\frac{1}{4}$ of the total memory of the system. For example, the memory of CPU nodes used by us is 125 GiB, while the memory required for HPCG testing is at least 31.25 GiB. Considering the limitation of memory usage and floating-point computing capability, we finally choose the default problem size $n_x = n_y = n_z = 104$ as the best choice, and its floating-point computing capability is $R_{real} = 18.1204$ GFlops.

In a word, the scale of the problem is properly reduced in the appropriate range, which will do good to improve the system's floating-point computing capability.

4.6.3 Optimization Flags

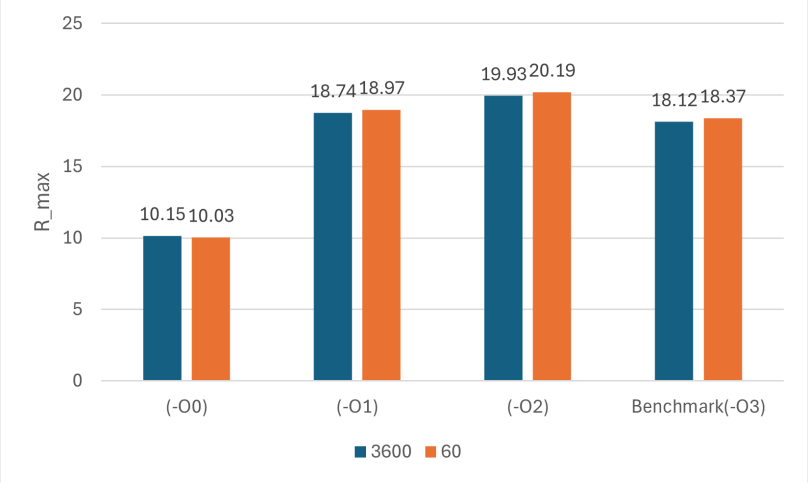


Figure 4.8: Different outcome in different optimization flags

As shown in figure 4.8, when the Intel's optimized compilation options was not open, which means `-O0`, the system has the worst floating-point performance, $R_{min} = 10.15$GFlops. When turning on default optimization compilation options: `-O2`, the system has the largest floating-point performance, $R_{max} = 19.93$GFlops, increased to 1.96 times comparing to `-O0`. For the file size optimization `-O1` and the aggressive optimization option `-O3` has little influence with floating-point performance compared to `-O2`, so we used -O3 as the optimization compile option in subsequent tests.

4.6.4 Vectorization

When we analyze the profiler, we find that in the benchmark when the algorithm conducts floating point computations, the vector instruction used is `SSE2` without `-mavx` compiling flags. The percentage of double precision floating point operations from all operations executed by the applications is 16.9% and the percentage of all packed 128-bit floating point operations with a double precision is 72.6% and scalar floating point operations account for the remainging 27.4%. Therefore, we try to see whether the performance will be improved using `-xHost`, `-xCORE-AVX512`, `-xCORE-AVX2`, `-xSKYLAKE-AVX512`.

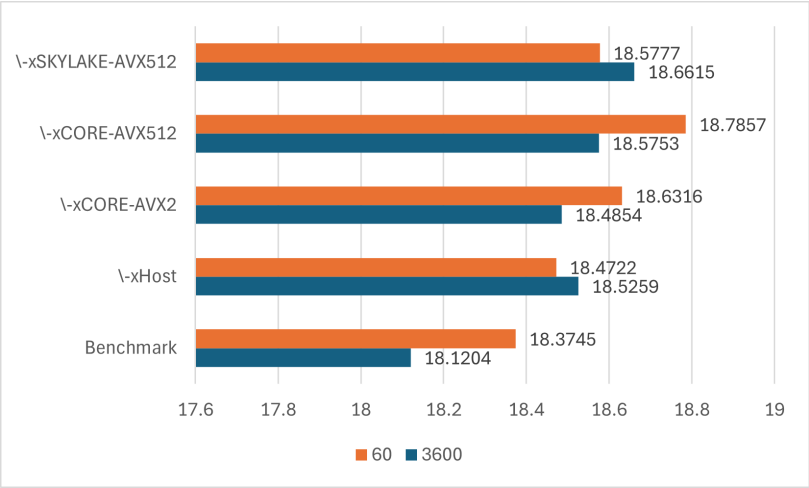


Figure 4.9: Different outcome in different vectorization flags

As we can see from the results, after changing to avx instruction set, we can improve the performance. However, the profiler still shows that the width of the packed floating point operations with a double precision is still 128-bit, which means HPCG does not support 512-bit packed floating point operations. The final results will be using `AVX` instruction sets.

4.6.5 Parallelization of Multiple Nodes

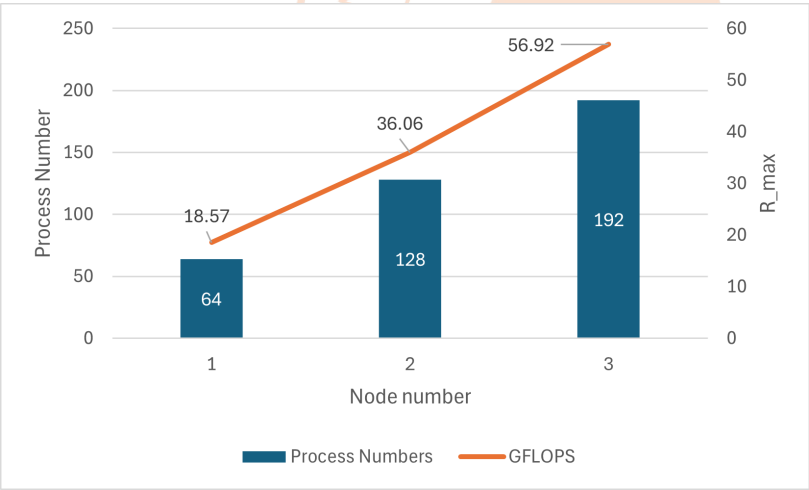


Figure 4.10: Different outcome with different node number

The performance of the outcome of HPCG greatly relys on how many resources it run on. As shown in the figure 4.10, the R_{max} is positively proportionate to the usage of node numbers. And when running on single node or multiple nodes, the average physical core utilization is around 90%. We have conducted osu benchmark testing on the latency and bandwidth inside the nodes and between the nodes. The results shown that the bandwidth and latency on one side or between different nodes are relatively the same, which lead to such results.

4.6.6 Parallelization of OpenMP

We have not used `-qopenmp`(Intel) up to now, but when we looked through the source code and found that HPCG uses the MPI + OpenMP

to optimize itself, we decide to compile HPCG with the `-qopenmp` option. We compared the HPCG with `-qopenmp` to the one which do not use, and set up `OMP_NUM_THREADS=2` and `OMP_NUM_THREADS=4`, the result is shown in Figure 4.11.

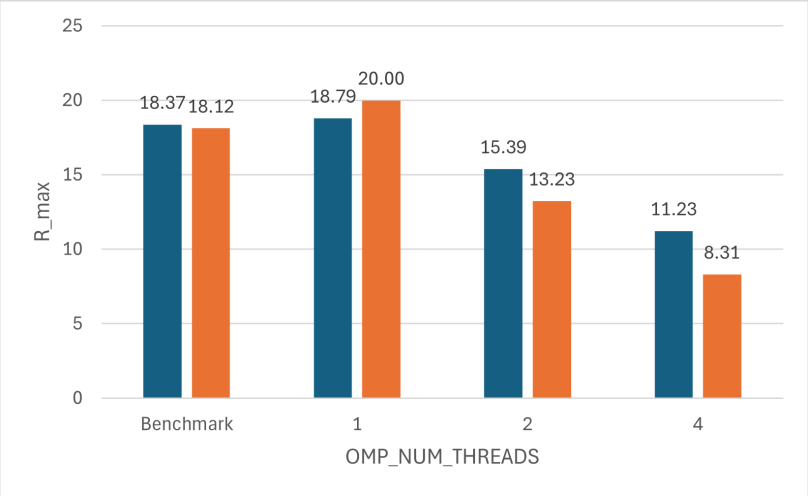


Figure 4.11: Default with OpenMP

The reasons for the results will be explained later in the report.

During the process of in-depth analysis of the source code, we found that not all computationally intensive areas optimized using OpenMP, especially `ComputeSYMGS_ref()`. Take the example of forward scan.

```
for (local_int_t i=0; i< nrow; i++) {
    const double * const currentValues = A.matrixValues[i];
    const local_int_t * const currentColIndices =
        A.mtxIndL[i];
    const int currentNumberOfNonzeros = A.nonzerosInRow[i];
    const double  currentDiagonal = matrixDiagonal[i][0];
        // Current diagonal value
    double sum = rv[i]; // RHS value

    for (int j=0; j< currentNumberOfNonzeros; j++) {
      local_int_t curCol = currentColIndices[j];
      sum -= currentValues[j] * xv[curCol];
    }
    sum += xv[i]*currentDiagonal; // Remove diagonal
        contribution from previous loop
    xv[i] = sum/currentDiagonal;
}
```

The forward scan here is the compute-intensive part in `ComputeSYMGS_ref()`, however, it is not optimized using OpenMP, so we tried to use OpenMP for optimization here. It is practical because every iteration in the loop processes every row in the matrix, which will not affect other iterations. Therefore, there does not exist data dependence between each iteration.

Listing 4.4: ComputeSYMGS_ref forward code

```
#pragma omp parallel for
for (local_int_t i=0; i< nrow; i++) {
    const double * const currentValues = A.matrixValues[i];
```

```
    const local_int_t * const currentColIndices =
        A.mtxIndL[i];
    const int currentNumberOfNonzeros = A.nonzerosInRow[i];
    const double  currentDiagonal = matrixDiagonal[i][0];
        // Current diagonal value
    double sum = rv[i]; // RHS value

    for (int j=0; j< currentNumberOfNonzeros; j++) {
      local_int_t curCol = currentColIndices[j];
      sum -= currentValues[j] * xv[curCol];
    }
    sum += xv[i]*currentDiagonal; // Remove diagonal
        contribution from previous loop
    xv[i] = sum/currentDiagonal;
}
```

Listing 4.5: ComputeSYMGS_ref forward code

Similarly, we also add `#pragma omp parallel for` at the outer loop for the backward loop part, and then we compile and test. The result is shown in Figure 4.12.

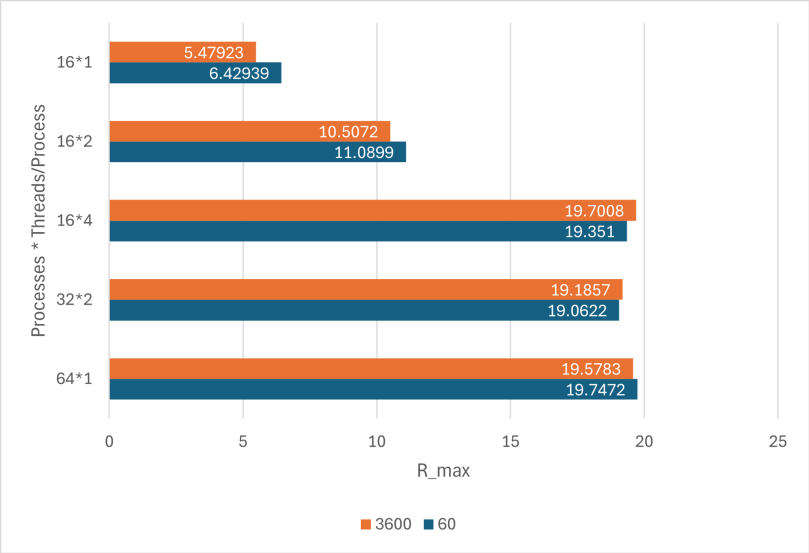


Figure 4.12: Results after first modification of OpenMP

In the result we can see that, if we choose to have the same number of processes, then applying OpenMP will greatly improve the performance. And the relation between the `OMP_NUM_THREADS` and the performance is nearly linear. And if we choose to have the same number of threads with different processes. The problem size will be relatively smaller compared if the number of processes is less. With larger `OMP_NUM_THREADS`, the iteration steps will be more. There are small improvement in the result. The reason for this strange outcome is we are only testing on one node.

A significant issue on a single-node system is that memory is shared within the same NUMA (Non-Uniform Memory Access) node. Therefore, even if MPI (Message Passing Interface) is used, the communication overhead won't be very high. As a result, there is not a significant difference in performance between running the program between only MPI and MPI-OpenMP hybrid.

However, when we try to test on multiple nodes, the modification we made cannot support multiple nodes because in the modified codes, `ExchangeHalo` method will update the matrix it needs to access on several nodes at the same time. Using OpenMP will redundantly calculate the same matrix over and over and lead to errors when trying to combining the data. Under this circumstances, the modification on SYMGS will produce invalid results.

The best result we get from our modifications is 19.7472 GFLOPS. And the acceleration ratio is 1.09. The efficiency is around 0.479%.

4.6.7 Data access ordering improvement for SYMGS

While we were doing OpenMP optimization, we were gradually familiar with the algorithm of `ComputeSYMGS_ref()`. We found that in the original code, the direction of the inner loop in the backward sweep of SYMGS is reverse to the outer loop.

```
1  for (local_int_t i=nrow-1; i>=0; i--) {
2      const double * const currentValues = A.matrixValues[i];
3      const local_int_t * const currentColIndices =
           A.mtxIndL[i];
4      const int currentNumberOfNonzeros = A.nonzerosInRow[i];
5      const double  currentDiagonal = matrixDiagonal[i][0];
           // Current diagonal value
6      double sum = rv[i]; // RHS value
7
8      for (int j = 0; j< currentNumberOfNonzeros; j++) {
9        local_int_t curCol = currentColIndices[j];
10       sum -= currentValues[j] * xv[curCol];
11     }
12     sum += xv[i]*currentDiagonal; // Remove diagonal
           contribution from previous loo
13     xv[i] = sum/currentDiagonal;
14   }
```

Listing 4.6: ComputeSYMGS_ref backward code

Here, the outer loop `i` proceeds in the backward direction. However, the inner loop `j` proceeds in the forward direction. In the source code, the array `matrixDiagonal` stores the pointer to the diagonal value of each row. Using this implementation, after outer loop `i` switches to the next iteration, the memory address used by the first inner loop `j` iteration will not be in cache because the inner loop `j` proceeds in the reverse direction to the outer loop `i` as in figure 4.13.

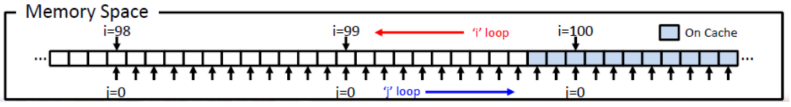


Figure 4.13: Memory access pattern analysis

In figure 4.13, the outer loop `i` is switched to `i = 99` from `i = 100` and the inner loop `j` starts at `j = 0`. The address of `j = 0`, is discontinuous to the last referenced position at `i = 100`, and it is not in cache. To avoid this, we modified the direction of the inner loop `j` to match the direction of the

outer loop `i` because the direction of the inner loop `j` is not constrained to the forward direction.

We have modified the code to the following:

```
for (local_int_t i=nrow-1; i>=0; i--) {
    const double * const currentValues = A.matrixValues[i];
    const local_int_t * const currentColIndices =
        A.mtxIndL[i];
    const int currentNumberOfNonzeros = A.nonzerosInRow[i];
    const double  currentDiagonal = matrixDiagonal[i][0];
        // Current diagonal value
    double sum = rv[i]; // RHS value

    for (int j = currentNumberOfNonzeros; j >= 0; j--) {
      local_int_t curCol = currentColIndices[j];
      sum -= currentValues[j] * xv[curCol];
    }
    sum += xv[i]*currentDiagonal; // Remove diagonal
        contribution from previous loo
    xv[i] = sum/currentDiagonal;
  }
```

Listing 4.7: ComputeSYMGS_ref backward code

In this modification, both outer loop `i` and inner loop `j` proceed in the backward direction. Thus, after the outer loop `i` switches to the next iteration, the memory address used by the first iteration of the inner loop `j` will be in cache because the inner loop `j` proceeds in the same direction as the outer loop `i`. Furthermore, we expect that a pre-fetch mechanism can work effectively because the memory access pattern used by this modification will be simple.

For the SYMGS backward sweep, the Data Access Ordering Improvement improves the memory throughput and cache hit rate. The best result we get from our modifications is 19.824 GFLOPS. And the acceleration ratio is 1.10. The efficiency is around 0.482%. This result is produced on one node with 64 cores and using 1 `OMP_NUM_THREAD`. The results are still invalid when running on multiple nodes.

4.6.8 Optimization Results

During our benchmark, we find that when we take problem size $104 \times 104 \times 104$, runtime 3600 on a single GPU, we can find a maximum efficiency of HPL GPU version is 19.998 Gflops.

$$\eta = \frac{R}{R_{peak}} = \frac{19.998}{4096} = 0.488\% \tag{4.21}$$

4.7 HPCG Optimization on GPU Platform B

After doing the HPCG optimization on the CPU platform A, we tried to optimize the HPCG on the GPU platform. We download NVIDIA HPC

Benchmark 23.10 from https://catalog.ngc.nvidia.com/orgs/nvidia/containers/hpc-benchmarks, and tried to run on the GPU platform using Docker. Inside the Docker, the MPI implementation used is Open MPI 4.1.5. The benchmark results for the GPU platform are as follows:

Table 4.4: HPCG Benchmark Result CPU

Node Number	GPU	n_x	n_y	n_z	Time/s	GFlops
1	1	256	256	256	3600	325.434
1	1	256	256	256	60	338.609

As we can see from the benchmark, the results differ largely when having different run time. To ensure the validity of the outcome, we choose 1810s in the following experiments according to the official suggestions.

4.7.1 Multiple GPU solutions

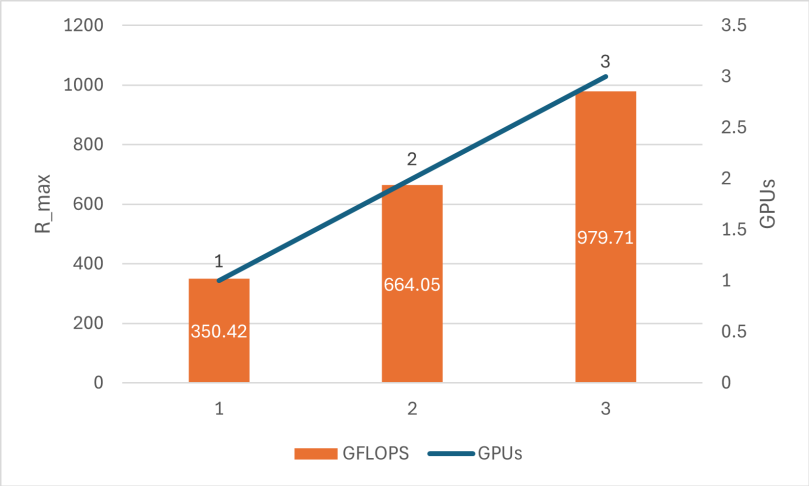


Figure 4.14: Different outcome with different numbers of GPU

Analyzing the data in figure 4.14, we observe a positive correlation between the number of CPU cores and the GFLOPS output. With a single CPU core, the system delivers approximately 350.42 GFLOPS. This performance nearly doubles to 664.05 GFLOPS with two CPU cores. With the addition of a third core, the performance reaches 979.71 GFLOPS, suggesting a near-linear improvement in computational speed with the addition of CPU cores.

The outcome reflects that the HPCG algorithm scales well with the number of CPU cores and GPUs, as evidenced by the increase in R_{max}. This indicates that the algorithm can effectively leverage additional hardware to enhance performance, which is characteristic of well-optimized parallel processing algorithms in high-performance computing environments.

Because of the scalability of HPCG problem, the following section is tested on single node.

4.7.2 Selection of the suitable size of the problem

To choose the proper size for GPU, we have started small with $16 \times 16 \times 16$. However, the result is even worse than the performance on CPU platform. When testing the default problem size $104 \times 104 \times 104$ on GPU, invalid results were generated, as after 50 iterations of the first process do not converge and the error is larger than the tolerance. Therefore, deciding on the proper problem size is crucial for testing the authentic performance of GPU tackling with real applications. We have tried in varying problem size, problem dimension order and with the same problem size the combination of n_x, n_y, n_z.

Figure 4.15: Results of the memory used and the performance choosing different size of the problems (Left: Sort by memory used in descending order, Right: Sort by R_{max} in descending order)

Comparing the problem size, the outcome of R_{max} does not show direct relation with the choice of problem size, which means, the results do not rely on how much the memory is used.

Comparing the problem dimension order with the same parameter, the order of n_x, n_y, n_z sometimes largely affect the outcome.

Comparing different combination, it can be seen from the results that usually more balanced combination (the n_x, n_y, n_z values are balanced) produces better performance.

To sum up, we found that under the combination of $512 \times 512 \times 288$ will produce the best performance on our platform.

4.7.3 Power control effect on HPCG performance

During our competition, we faced the challenge of configuring a computing cluster to operate under a stringent total power constraint of 3000W. This necessitated a series of meticulous experiments to understand how various power caps could impact the performance of our HPCG benchmarks.

We have used `nvidia-smi -pl <target-limit>` to control the limits of the power. The results are shown in Figure 4.16.

Our testing regime involved progressively lowering the power limit from 400W down to 125W and observing the corresponding HPCG performance. As illustrated in Figure 4.16, we noted that the HPCG results remained relatively stable when the power cap was reduced from

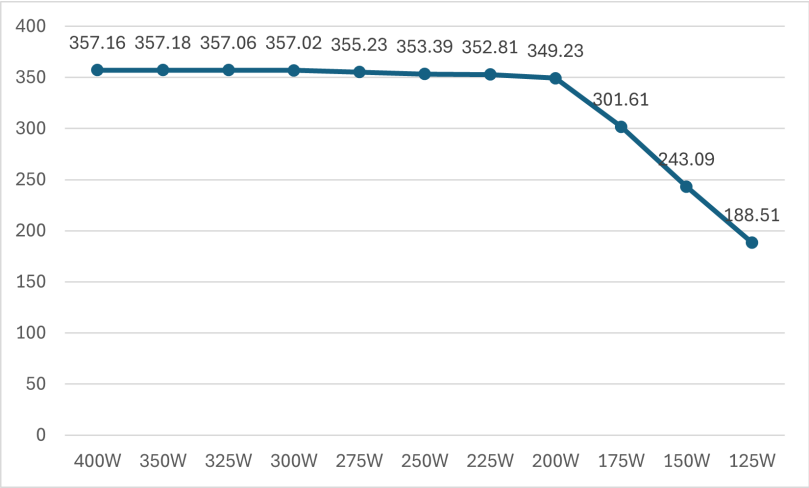


Figure 4.16: Results of the performance with different power limits

400W to 300W. This stability suggests that the HPCG benchmark, under our cluster configuration, has a baseline power consumption that hovers around 300W. Beyond this threshold, we hypothesize that the power is primarily utilized by components other than the HPCG computation, such as system overheads or other operational processes.

However, a stark performance degradation was observed as we breached the 200W limit. Below this threshold, the performance of HPCG plummeted drastically. This suggests that a power setting of 200W is a critical point for our cluster setup, below which the available power is insufficient to sustain the computational load of the HPCG benchmark.

The gradient of performance decline intensifies significantly beyond this critical point, indicating that the HPCG's computational units, possibly including CPU and memory, become starved of the necessary power to operate optimally. This is exemplified by the sharp drop in HPCG results as we moved from a 200W to a 150W power cap, followed by an even more pronounced decrease when the power limit was further reduced to 125W.

4.7.4 Optimization Results

During our benchmark, we find that when we take problem size 512×512×288, runtime 1810 on a single GPU, we can find a maximum efficiency of HPL GPU version is 356.605 Gflops on a single GPU. From the NVIDIA Ampere Architecture White-paper we know that the Peak FP32 TFlops(non-Tensor) is 19.5, which is 19500Gflops. And our maximum efficiency is that:

$$\eta = \frac{R}{R_{peak}} = \frac{356.605}{19500} = 1.829\% \tag{4.22}$$

5 Optimization for LLM inference

5.1 Introduction

The emergence of large language models (LLMs) like GPT [8, 9] and Llama 2 [10] has revolutionized the field of AI, demonstrating unprecedented capacity across tasks like programming assistant and universal chatbot. However, running these applications is very expensive due to the large cost in LLM inference. Firstly, the models are often too large to fit into memory. For example, the Llama2-70B model has 70 billion parameters and requires a large amount of memory to store its weights. Originally at least 140GB space will be used for the parameter and KV-cache of the model. Secondly, each inference for LLM can be time consuming, and thus the UTL of GPU will fluctuate so much that it will not be able to fully exert its computing power. Finally, as the number of parameters increases, the transformers model will store a larger KV-cache at inference, which takes up a lot of memory.

In order to solve the above problems, we need to combine the characteristics of clusters to design an inference schemes with high throughput. By profiling the baseline, we find that the UTL of the two GPUs shows alternating periodic fluctuations under the scheme of model parallelism (MP). To solve this problem, we turned to tensor parallelism (TP) to ensure that the load of the two cards is balanced in time and space. Next, in order to reduce the memory usage of KV-cache, we further leverage paged attention (PA). We found that there were only 996 output_len out of 10,000 samples, and thus we introduced a strategy to batch the data under the same output_len. Finally, we introduced the continuous batching strategy, combined with optimizations such as TP and PA, to achieve a speedup ratio of 8x.

5.2 Related Work

In this section, we review some of the previous works that are related to our topic of large-scale language model inference.

5.2.1 Tensor-Parallelism in Megatron-LM

Megatron-LM [11] is a framework for training and inferring large transformer language models using model parallelism. It provides efficient tensor, pipeline and sequence based model parallelism for pre-training transformer based Language Models such as GPT (Decoder Only), BERT (Encoder Only) and T5 (Encoder-Decoder). Megatron-LM implements tensor parallelism by splitting each tensor into multiple chunks with each shard residing on a separate GPU. This reduces the memory footprint without much additional communication on intra-node ranks. Megatron-LM also supports pipeline parallelism, which distributes the layers uniformly across pipeline stages and reduces the bubble of naive pipeline parallelism via different scheduling schemes. Moreover, Megatron-LM employs sequence parallelism, which reduces the activation memory required by sharding the outputs of each transformer layer along the sequence dimension. Megatron-LM has been used to train transformer models with up to 8.3 billion parameters using 512 GPUs, achieving state-of-the-art results on several language generation and understanding tasks.

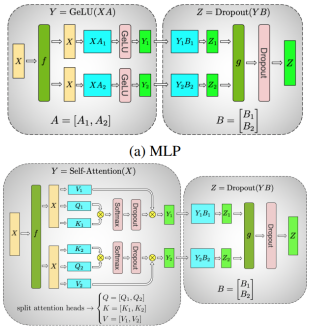


Figure 5.1: Illustration of tensor parallelism [11] for blocks of transformer. f and g are conjugate. f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass. MLP refers to multilayer perceptron.

5.2.2 Paged Attention in VLLM

VLLM [12] is the *very large language model* that uses paged attention to handle long sequences. Paged attention is a technique that divides the input sequence into pages and computes the attention only within each page. This reduces the quadratic complexity of the attention operation to a linear one, while preserving the global context by allowing the pages to communicate with each other via a gating mechanism. VLLM also uses a hierarchical softmax to speed up the output layer and a sparse embedding layer to reduce the memory consumption of the embedding matrix. VLLM has been trained on a large-scale text corpus with 1.2 trillion tokens, resulting in a 2.6 billion parameter model that outperforms GPT-2 on several downstream tasks.

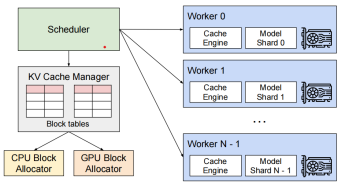


Figure 5.2: VLLM system overview.

5.2.3 Continuous Batching by Orca

Orca [13] is a distributed serving system for transformer-based generative models. It addresses the inefficiency of existing serving systems that have a multi-iteration characteristic, due to their inflexible scheduling mechanism that cannot change the current batch of requests being processed. Orca proposes iteration-level scheduling, a new scheduling mechanism that schedules execution at the granularity of iteration (instead of request) where the scheduler invokes the execution engine to run only a single iteration of the model on the batch. This allows requests that have finished earlier than other requests in a batch to return to the client, while newly arrived requests can join the current batch without waiting. This scheduling method can be also called continuous batching.

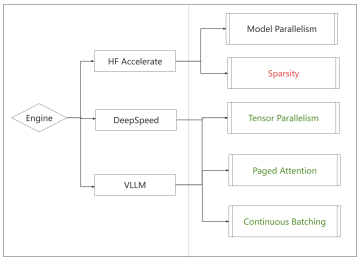


Figure 5.3: Framework of our methods.

5.3 Methodology

Our optimization focus is primarily on our dual-GPU clusters. In order to make the most of the computing power with limited resources, we use strategies such as tensor parallelism, paged attention, and continuous batching. At the same time, considering the special architecture of our GPU, we also carried out experiments on the impact of semi-structured sparse matrices on inference. For the detail of implementation, please refer the Experiments section.

5.4 Experiments

5.4.1 Experiments Setup

Cluster. We use the GPU cluster specified by Table 1.7 as the default experiment platform.

Software. The detailed software environment is specified in the README.md file of submitted code folder.

Test dataset. The dataset can be retrieved from Here.

Baseline: HF engine + MP. Accelerate [14] is a distributed inference engine for HuggingFace model. Since we don't have a machine capable of carrying the Llama2-70B on a single card, we use HuggingFace's default accelerate package to implement the model parallelism for baseline model. Model parallelism is a technique for training large and complex deep learning models that do not fit into a single device, such as a GPU or a CPU. It involves splitting the model into smaller parts and distributing them across multiple devices, so that each device only handles a portion of the model. By default, the parts are activated in a sequential order in the overall model inference.

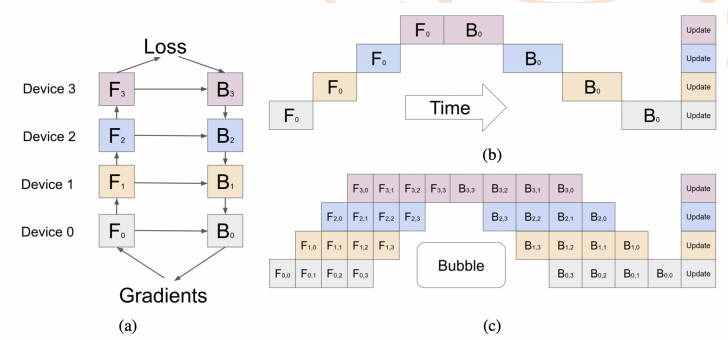


Figure 5.4: Illustration of model parallelism from Google.

TP engine. We use DeepSpeed to implement automatic tensor parallelism.

TP + paged attention engine. We use vLLM to leverage both the automatic tensor parallelism and the paged attention optimization.

Hyperparameter setting. All hyperparameters are set to use Greedy Search for token generation. The parameter settings may be slightly different for different engine interfaces, but using the settings shown in

Listing 5.1 and Listing 5.2 can ensure that their backend implementations are consistent.

```
batch_llm_outputs = llm.generate(
                    input_ids=batch_input_ids.cuda(),
                    do_sample=False,
                    num_return_sequences=1,
                    num_beams=1,
                    temperature=1.0,
                    top_p=1.0,
                    use_cache=True,
                    max_new_tokens=output_len,
                    use_tqdm=False
                )
```

Listing 5.1: Hyperparameter setting of Accelterate Engine and DeepSpeed engine.

```
sampling_params = SamplingParams(n=1,
                                 best_of=1,
                                 temperature=0,
                                 top_p=1.0,
                                 top_k=50,
                                 max_tokens=output_len
                                 )
```

Listing 5.2: Hyperparameter setting of VLLM engine.

Batch Strategy. In order to meet the requirements of the output_len of each sample, we form batches of samples with the same output_len for inference. The detailed implementation is shown in Listing 5.3.

```
"""Group requests by output length"""
requests_by_output_len = {}
for request in requests:
    _, _, output_len = request
    if output_len not in requests_by_output_len:
        requests_by_output_len[output_len] = []
    requests_by_output_len[output_len].append(request)
```

Listing 5.3: Batch strategy.

Sparsity. In the experiment, we used wanda as a sparsity tool to sparsify the model weights to semi-structure. At the same time, in order to activate the sparsity of the model, we also need to iterate through each layer of the model and convert the weights to sparse weights, which is shown in Listing 5.4.

```
from torch.sparse import SparseSemiStructuredTensor,
    to_sparse_semi_structured
SparseSemiStructuredTensor._FORCE_CUTLASS = True
def activate_sparsity(llm):
    """
    Activates sparsity for linear modules in the given LLM
        model.

    Args:
        llm (nn.Module): The LLM model.

    Returns:
```

```
        None
    """
    for fqn, module in llm.named_modules():
        if isinstance(module, nn.Linear) and "layer" in
            fqn:
            previous_weight: torch.tensor =
                module.weight.detach()
            module.weight = \
            nn.Parameter(to_sparse_semi_structured(
                                    previous_weight))
            del previous_weight
            gc.collect()
```

Listing 5.4: Function to activate sparsity.

5.4.2 Experiments Result

Baseline Analysis

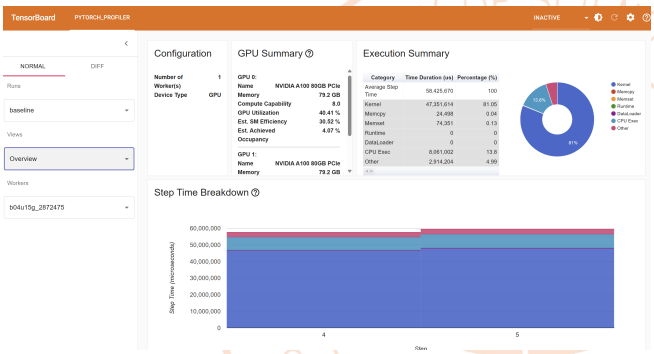


Figure 5.5: Tensorboard interface for torch.profiler data.

Using baseline for model parallelism to infer llama2-70B can only achieve a performance of 20.08 tokens/s. At first, we observed that in the process of inference, the UTL of the two cards will fluctuate alternately. To find out the inherent reason, we evaluated the model using torch.profiler, as shown in Figure 5.5.

	GPU 0	GPU 1	Average
GPU Utilization	40.41 %	40. 46 %	40.44 %
Est. SM Efficient	30.52 %	30.75 %	30.64 %

Table 5.1: GPU summary for baseline.

From Table 5.1, we can see that both the GPU Utilization[1] and the Est. SM Efficiency[2] are low. The left graph in Figure 5.6 shows that 13.8% of the computation comes from the CPU, indicating that the baseline has a large CPU overhead. The right figure in Figure 1 shows that the time consumption mainly comes from the computation of the attention layer.

1: GPU busy time / All steps time. The higher, the better. GPU busy time is the time during which there is at least one GPU kernel running on it. All steps time is the total time of all Kemer.

2: Estimated Stream Multiprocessor Efficiency. The higher, the better. This metric of a kernel, SM_Eff_K = min(blocks of this kernel / SM number of this GPU, 100%). This overall number is the sum of all kernels' SM_Eff_K weighted by kernel's execution duration, divided by all steps time.

Tensor Parallelism

We use DeepSpeed for automatic tensor parallelism:

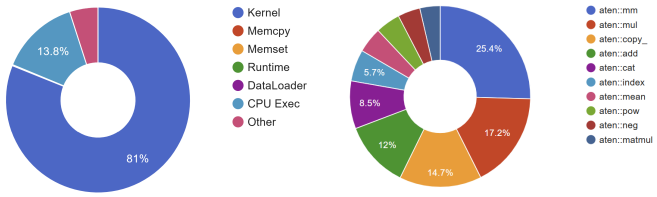


Figure 5.6: Execution summary and operator view of baseline. The left figure is the Execution summary and the right one is the operator view.

```
import deepspeed
llm = AutoModelForCausalLM.from_pretrained(
            args.model, torch_dtype=torch.float16,
            trust_remote_code=args.trust_remote_code,
            device_map='cpu')
world_size = int(os.getenv('WORLD_SIZE', '1'))
llm = deepspeed.initialize(model=llm,
                        mp_size=world_size,
                        dtype=torch.float16,
                        replace_with_kernel_inject=False)
```

Listing 5.5: Automatic tensor parallelism.

Using the tensor parallel strategy, two cards can be used synchronously for higher UTL, and the performance is increased to 27.62 tokens/s, which is 37.55% higher than that of the baseline. During the experiment, we observed that the UTL alternating fluctuation of the baseline no longer exists. After going through the profile, we can see that the average graphics card utilization is 37.3% higher than that of the baseline.

	GPU 0	GPU 1	Average
GPU Utilization	93.46 %	62.01 %	77.74 %
Est. SM Efficient	85.03 %	53.75 %	69.39 %

Table 5.2: GPU summary for TP parallelism.

Paged Attention + Tensor Parallelism

We use vLLM for paged attention implementation:

```
from vllm import LLM
llm = LLM(args.model, dtype=torch.float16,
    trust_remote_code=args.trust_remote_code,
    tensor_parallel_size=2)
```

Listing 5.6: Paged attention + Tensor Parallelism.

After using vLLM based on Paged Attention, the utilization rate of the graphics card has increased again. It makes full use of the idle resources of the graphics card to achieve efficient caching of KV tensors. The performance of the model also reached 39.11 tokens/s, which is 94.77% higher than that of the baseline.

	GPU 0	GPU 1	Average
GPU Utilization	94.68 %	95.44 %	95.06 %
Est. SM Efficient	85.63 %	86.07 %	85,85 %

Table 5.3: GPU summary for TP parallelism + Paged attention.

Continuous Batching + PA + TP

We use a Listing 5.3 based batch strategy and vLLM for continuous batching:

```
for output_len, requests in
    tqdm(requests_by_output_len.items()):
    sampling_params = SamplingParams(n=1, best_of=1,
        temperature=0, top_p=1.0, top_k=50,
        max_tokens=output_len)
    prompts = [request[0] for request in requests]
    input_ids = tokenizer(prompts,
        return_tensors="pt",padding=True).input_ids
    llm_outputs =
        llm.generate(prompt_token_ids=input_ids.tolist(),
        sampling_params=sampling_params, use_tqdm=False)
```

Listing 5.7: Continuous batching.

vLLM will automatically set the maximum batch to ensure the maximum throughput while the memory does not overflow. In the end, two A100-80G cards were used to achieve a performance of 158.55 tokens/s, which was 689.59% higher than that of the baseline.

Remark on Sparsity

Our cluster is equipped with two A100s of the ampere architecture, which naturally supports the operation of semi-structured sparse matrices[3]. Therefore, we want to observe the effect of semi-structure sparsity on the inference performance of the model. According to Nvidia's poster, using such matrix multiplication on the ampere architecture can achieve up to 2x the increase in computing power. Unfortunately, torch.sparse does not support batch inference for sparse models, and related functions are still under active development. We keep the corresponding interface in the code, and hope to use the sparse inference feature in the near future.

3: Accelerating Inference with Sparsity Using the NVIDIA Ampere Architecture and NVIDIA TensorRT

5.5 Conclusion

The overall result of the above 4 settings is illustrated in Figure 5.7. TP solves the problem of periodic fluctuations in the UTL of the GPU, PA improves the efficiency of memory usage for KV-cache, and finally CB makes the best use of GPU resources and significantly improves throughput. The meta data of output token shown in Table 5.4 shows that our optimization merely changes the output.

Model	Prompt Tokens	Total Tokens
Baseline	2518687	4694726
TP	2518687	4692431
PA+TP	2518687	4691827
CB+PA+TP	2518687	4691830

Table 5.4: Prompt and total tokens for different optimization methods.

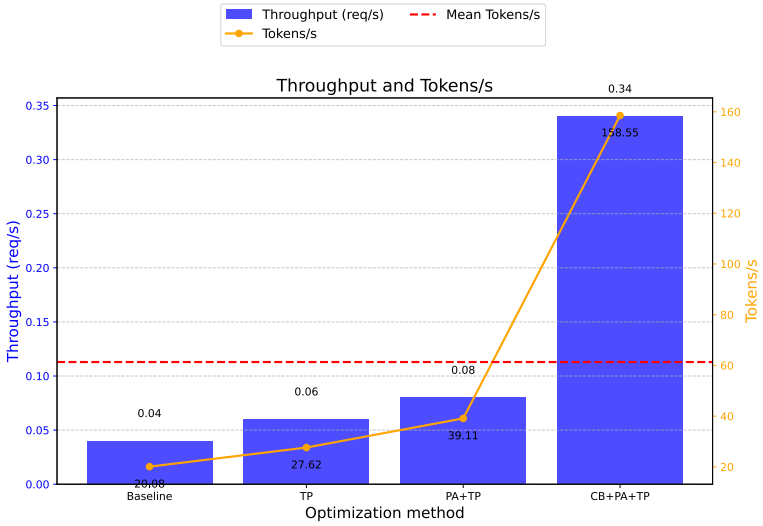


Figure 5.7: Overall result of the above 4 settings. CB refers to continuous batching.

In summary, addressing issues of large model sizes and memory constraints, we leverage tensor parallelism, paged attention, and optimized batch strategies. This comprehensive approach, combined with continuous batching, achieved an impressive 8x speedup ratio. These innovations not only overcome practical challenges but also enhance the efficiency and cost-effectiveness of deploying large language models in real-world applications.

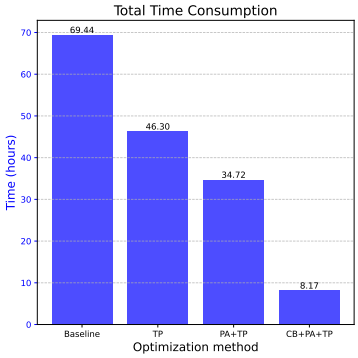


Figure 5.8: Overall time consumption of inference on 10000 samples.

6 OpenCAEporo

6.1 Background Intro

In the realm of computational engineering and physics, the simulation of multiphase flow in porous media stands as a significant challenge due to the complexity of the physical processes involved. OpenCAEPoro, emerges as a wonderful open-source software tool which supports hyperscale distributed parallel computing. It is specifically designed to facilitate the simulation of multicomponent multiphase flows within porous media. Developed in the C++ programming language, known for its efficiency and performance, OpenCAEPoro is well-suited for handling computationally intensive simulations. As an open-source project, OpenCAEPoro offers its source code freely for modification and distribution under the GNU Lesser General Public License. This aspect not only fosters a collaborative environment among users and developers but also encourages continuous improvement and innovation. The open-source nature significantly contributes to the adaptability of the software, allowing it to meet various specific requirements and research needs. OpenCAEPoro's current version specializes in simulating dynamic behaviors of fluids in porous structures, offering insights into how fluid properties and flow patterns evolve over time. This dynamic simulation capability is pivotal for researchers and professionals who deal with the challenges of multiphase flow in porous media. It also shows practical applications on many areas, such as energy crisis, dual carbon goal, pollution management.

Despite its robust capabilities, there is a continuous need for optimization to enhance OpenCAEPoro's performance, usability, and accuracy. In the field of porous media flow simulation, differential equations based on the physical properties of components can be established to describe the laws of multi-phase, multi-component percolation. However, these differential equations often do not have explicit solutions. In other words, we cannot intuitively determine the percolation conditions by simply inputting time and location information. Therefore, in the modeling phase, OpenCAEPoro adopts a meshes-based approach to divide the simulation area. This method assumes explicit hypothetical solutions for each meshes and seeks to satisfy the partial differential equations as closely as possible, thereby obtaining a discrete set of partial differential equations to approximate the true solution. Consequently, the method of grid division is particularly crucial. When processing information

for a mesh cell, it is essential to consider not only the geometrical characteristics of the reservoir but also the geological features of the rock layers and whether this division method is compatible with the numerical solution approach. In addition to the different conditions of various media, one must also consider changes in fluid properties due to variations in temperature and pressure. The flow capabilities of fluids in different phases vary, which in turn affects the fluid flow patterns. Therefore, the grid division approach must be determined based on the characteristics of the problem and the required precision of the solution. To ensure high-resolution grid-based simulations for finer modeling, stronger computational power and parallel distributed computing methods are needed. This highlights the significant role of high-performance computing in such numerical simulation problems.

At the level of numerical methods, the preprocessed set of partial differential equations is decoupled into discrete linear systems within a single time step, followed by linear iteration and updating. Therefore, a large amount of matrix operations during this phase is the primary computational expense in numerical simulation. How to set up an efficient solver is also a bottleneck and hot spot for optimizing applications. However, in this competition, our focus is on optimizing aspects other than numerical computational solutions, namely, the preprocessing and postprocessing stages. The ideal goal is to achieve load balancing. The preprocessing includes matrix assembly and the processing of physical information within mesh cells, which have strong computational independence and high parallelism. However, the computational cost changes constantly with the physical state of the mesh, making it difficult to achieve load balancing. The postprocessing output phase adopts a strategy of simultaneous computation and output, which also results in significant output delays.

6.2 Runtime Characteristics & Performance Analysis

In the ASC24 competition, we need to calculate based on the data in case1.data, with the goal of optimizing OBJECT TIME as much as possible while meeting accuracy requirements.

```
================================================
Final time:                   3600.000( Day )
 - Avg time step size .......   10.405 (346 steps)
 - Avg Newton steps .........    3.884 (1344 succeeded + 131 wasted)
 - Avg linear steps .........    5.796 (7790 succeeded + 1090 wasted)
Simulation time:                2348.917 (Seconds)
 - % Input & Partition .........    0.009 (0.202s)
 - % Partition - ParMetis ......    0.002 (0.047s)
 - % Input Reservoir ...........    0.004 (0.084s)
 - % Setup Simulator ...........    0.001 (0.027s)
 - % Initialization ............    0.003 (0.070s)
 - % Assembling ................    7.999 (187.894s)
 - % Assembling for LS interface    0.541 (12.699s)
 - % Linear Solver .............   87.718 (2060.429s)
 - % Newton Step ...............    0.484 (11.375s)
 - % Updating Properties .......    2.582 (60.641s)
 - % Output ....................    0.005 (0.111s)
================================================

OBJECT TIME:   414.366 s
```

Figure 6.1: 10 cores baseline

After our successful installation, we tested the program on queue 38 with 10 cores, and the results can be seen in Figure 6.1.

6.2.1 Multi node testing

We first conducted basic testing on the sample on queue 381.5, and the OBJECT TIME obtained from the testing is shown in Figure 6.2

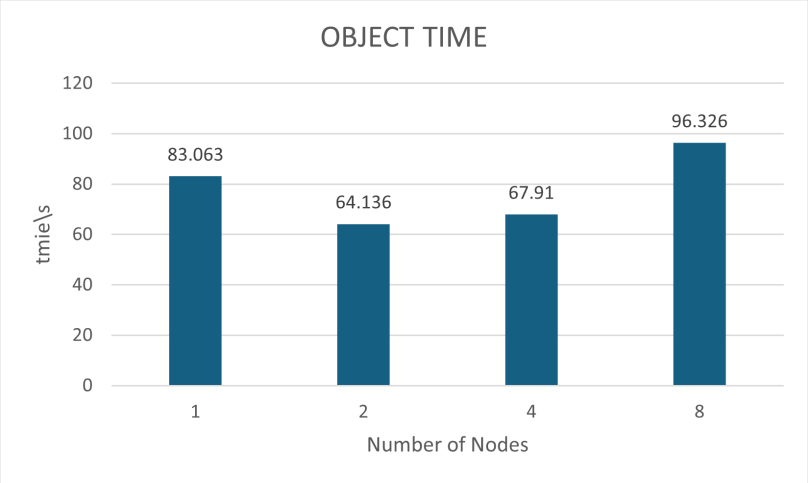


Figure 6.2: OBJECT TIME

There is no obvious trend of decreasing OBJECT TIME with the increase of the number of nodes. Therefore, we analyzed the composition of OBJECT TIME and analyzed its changes starting from its composition.

OBJUCT TIME is created by OCPTIME_Total plus MATRIX_VERSION_-TIME minus OCPTIME_LSOLVER obtained. Among them, OCPTIME_-Total is the total running time of the entire program, which means that all time except for OCPTIME_LSOLVER is included in the OBJECT TIME.Among all the components of OBJUCT TIME, Assembling, MATRIX_CONVERSION_TIME, Assembling for LS interface, Updating Properties and Output have a relatively large proportion and vary significantly with the number of nodes.The specific results obtained are shown in Figures 6.3 to 6.7.

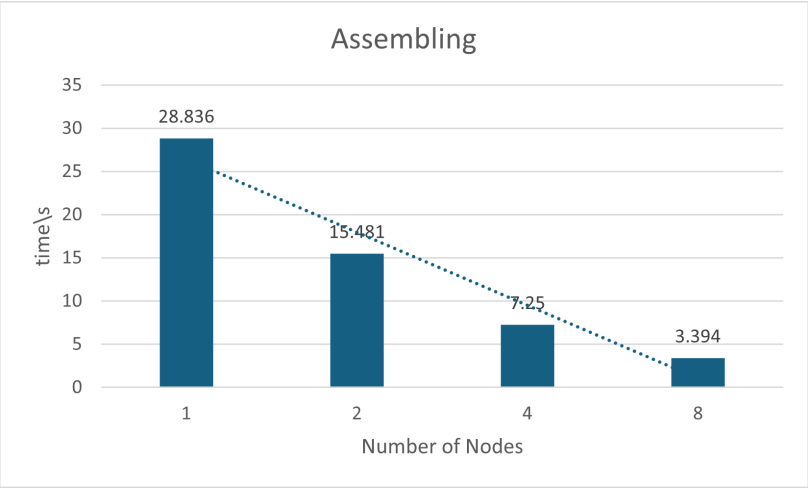


Figure 6.3: The time of Assembling

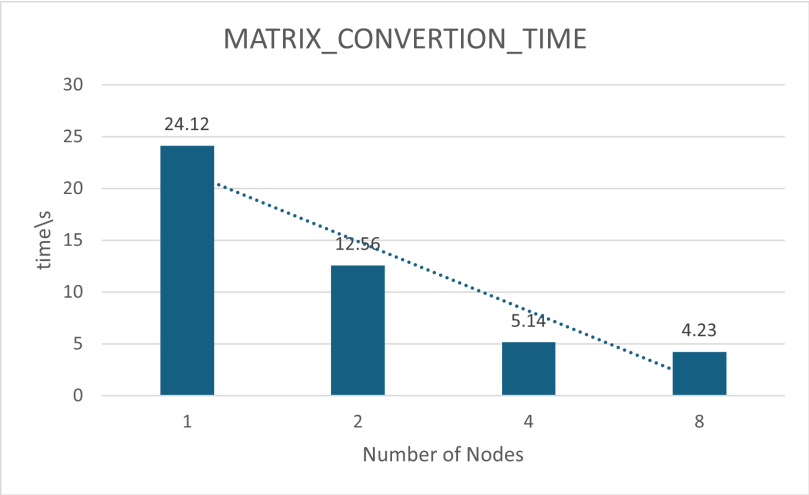


Figure 6.4: MATRIX CONVERSION TIME

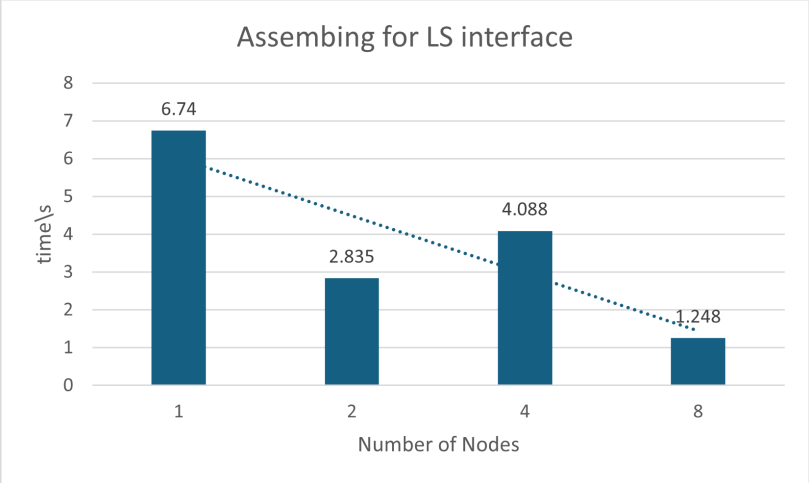


Figure 6.5: The time of Assembling for LS interface

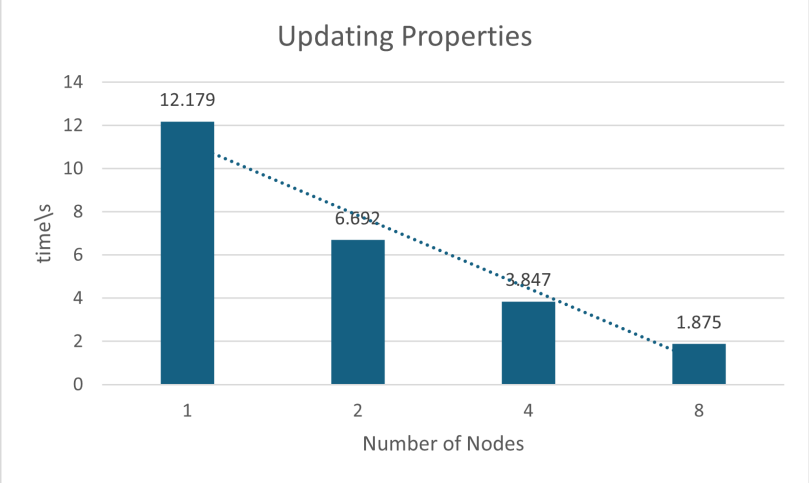


Figure 6.6: The time of Updating Properties

From the various components of OBJECT TIME, we can see that as the number of nodes increases, most of the OBJECT TIME is decreasing, but the output time is rapidly increasing, especially from single nodes to dual nodes.This phenomenon is understandable. When a program runs on multiple nodes, it incurs additional communication overhead, so we want to know what causes such a huge communication overhead.

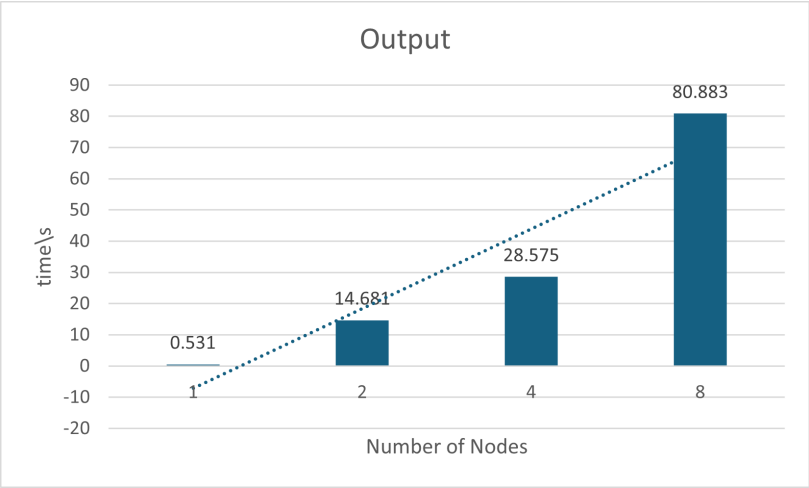


Figure 6.7: The time of Output

6.2.2 Hotspot Analysis

In order to obtain more detailed information about the OpenCAEporo at runtime, we analyzed the program using APS and VTune, respectively.

Figure 6.7 shows the results of using APS to test OpenCAE under single node operation.

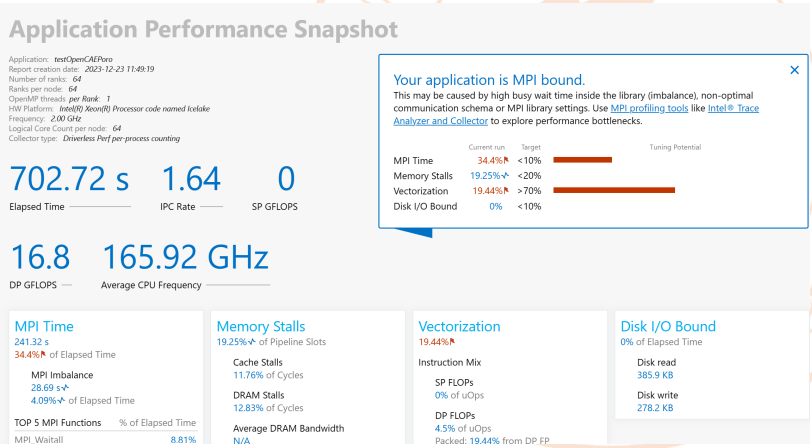


Figure 6.8: result of APS

From the analysis results of APS, we found that OpenCAEporo has a longer MPI time and a lower degree of vectorization. Meanwhile, we validated the results using VTunes to obtain more detailed information.

Function / Call Stack	CPU Time							Module
	Effective Time ▼	Spin Time	Overhead Time					
			Creation	Scheduling	Reduction	Atomics	Other	
ucp_worker_progress	16164.224s	0s	0s	0s	0s	0s	0s	libucp.so.0
PMPI_Allreduce	10535.045s	52.503s	0s	0s	0s	0s	0s	libmpi.so.12
MPIDI_SHMGR_release_generic	7195.904s	0s	0s	0s	0s	0s	0s	libmpi.so.12
hypre_BoomerAMGBuildCoarseOperatorKT	3715.618s	0s	0s	0s	0s	0s	0s	libHYPRE-2.28.0
PMPI_Testall	2714.447s	832.407s	0s	0s	0s	0s	0s	libmpi.so.12
MatSolve_SeqBAIJ_4_NaturalOrdering	2130.164s	0s	0s	0s	0s	0s	0s	libpetsc.so.3.19
hypre_BigBinarySearch	1791.393s	0s	0s	0s	0s	0s	0s	libHYPRE-2.28.0
MatMult_SeqBAIJ_4	1546.874s	0s	0s	0s	0s	0s	0s	libpetsc.so.3.19
PMPI_Iprobe	1490.913s	2676.205s	0s	0s	0s	0s	0s	libmpi.so.12
__intel_avx_rep_memset	1234.355s	0s	0s	0s	0s	0s	0s	testOpenCAEPor
PMPI_Isend	1212.147s	0s	0s	0s	0s	0s	0s	libmpi.so.12
MatSetValues_MPIBAIJ	1026.629s	0s	0s	0s	0s	0s	0s	libpetsc.so.3.19
MatLUFactorNumeric_SeqBAIJ_4_NaturalOrdering	1005.359s	0s	0s	0s	0s	0s	0s	libpetsc.so.3.19
PetscKernel_A_gets_inverse_A_4	912.951s	0s	0s	0s	0s	0s	0s	libpetsc.so.3.19
hypre_BigQsort0	881.131s	0s	0s	0s	0s	0s	0s	libHYPRE-2.28.0
calloc	867.076s	0s	0s	0s	0s	0s	0s	libc.so.6
__intel_avx_rep_memcpy	859.034s	0s	0s	0s	0s	0s	0s	testOpenCAEPor

Figure 6.9: result of VTune

From the results, we can see from Figure 6.8 that PMPI_Allreduce, MPIDI_SHMGR_Release_Generic, PMPI_Testall, PMPI_Iprobe, PMPI_-

Isend and other functions take a long time, which effectively validates the results of APS.

Meanwhile, we analyzed some data related to the communication part provided by the statistic.out file and obtained similar results.

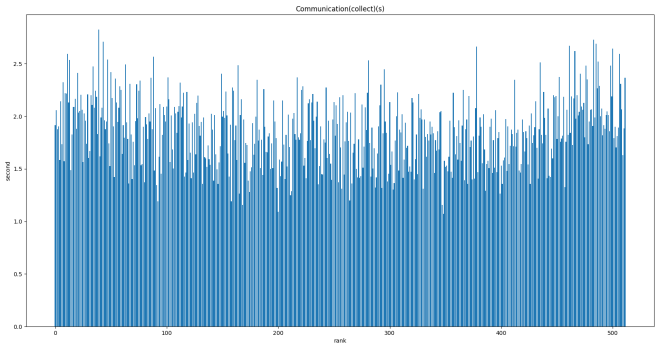


Figure 6.10: Communication (collect) of each process (s)

6.2.3 Source Code Analysis

After obtaining the above test results and calculating the proportion of time spent on each part of a single node and multiple nodes to the OBJECT TIME, we carefully reviewed the code, focusing mainly on the Output section that has the fastest and largest increase in time spent on multiple nodes.

The functions related to Output in OpenCAEporo are **OCPOutput::SetVal**, **OCPOutput::PrintInfo()**, **OCPOutput::PrintInfoSched**, and **OCPOutput::PostProcess()**. It is these four functions that determine the duration of Output. These four functions call more basic functions. From code analysis, the function related to multiple processes is summary.PostProcess, crtInfo.PostProcess summary.PrintInfo and crtInfo.PrintFastReview.

After timing these functions separately, it was found that summary.PostProcess and crtInfo.PostProcess took the longest time in multi node scenarios, accounting for almost 99% of the Output.

summary.PostProcess can be roughly divided into five parts, as shown in the following code block:

```
1
2  void Summary::PostProcess(const string& dir, const string&
       filename, const OCP_INT& numproc) const
3  {
4      //Define some variables
5      ......
6
7      //Read data from the results obtained from various
           processes
8      ......
9
10     // combine some values
11     ......
```

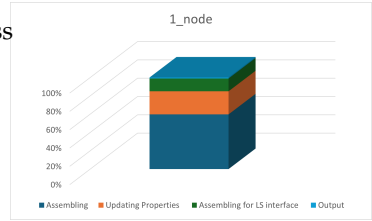


Figure 6.11: The proportion of time spent on each part of 1 node to OBJECT TIME

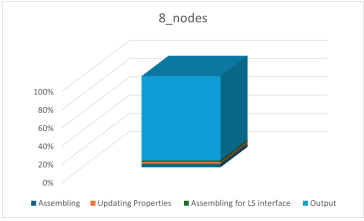


Figure 6.12: The proportion of time spent on each part of 8 nodes to OBJECT TIME

```
12
13      // set output precision
14      ......
15
16      // Print results
17      ......
18  }
```

Listing 6.1: summary.PostProcess

Using the same method as determining which function takes longer, we annotate each part separately before running to determine which part takes the longest.

We found that the step of Read data from the results obtained from various processes takes a long time, accounting for approximately 30 seconds in the output of 80 seconds.

```
1       for (USI p = 0; p < numproc; p++) {
2           const string myFile = dir + "proc_" + to_string(p)
                + "_SUMMARY.out";
3           ifstream ifs(myFile, ios::in);
4
5           if (!ifs) {
6               OCP_MESSAGE("Trying to open file: " <<
                    (myFile));
7               OCP_ABORT("Failed to open the input file!");
8           }
9
10          // Get time steps
11          ReadLine(ifs, buffer, OCP_FALSE);
12          for (USI i = 0; i < buffer.size(); i++) {
13              if (buffer[i] == "--") {
14                  rowNum = stoi(buffer[i + 1]);
15                  break;
16              }
17          }
18
19          while (!ifs.eof()) {
20              if (buffer[0] == "TIME") {
21                  // begin to input title of data
22                  flag = OCP_TRUE;
23                  bId += varlen;
24                  varlen = buffer.size();
25                  for (const auto& v : buffer) {
26                      mySum.push_back(SumItem(v, rowNum));
27                  }
28                  ReadLine(ifs, buffer, OCP_FALSE);
29                  OCP_ASSERT(varlen == buffer.size(),
                        "Mismatch Value");
30                  for (USI i = 0; i < varlen; i++) {
31                      mySum[bId + i].Unit = buffer[i];
32                  }
33                  ReadLine(ifs, buffer, OCP_FALSE);
34                  OCP_ASSERT(varlen == buffer.size(),
                        "Mismatch Value");
```

```
                for (USI i = 0; i < varlen; i++) {
                    mySum[bId + i].Obj = buffer[i];
                }
            }
            ReadLine(ifs, buffer, OCP_FALSE);
            if (flag && buffer.size() == varlen &&
                buffer[0] != "Row") {
                // begin to input value of data
                for (USI i = 0; i < varlen; i++) {
                    mySum[bId +
                        i].val.push_back(stod(buffer[i]));
                }
            }
            else {
                flag = OCP_FALSE;
            }
        }

        ifs.close();
        if (remove(myFile.c_str()) != 0) {
            OCP_WARNING("Failed to delete " + myFile);
        }
    }
```

Listing 6.2: Read data from the results obtained from various processes

In this section, the program runs on the main process, reads data from files generated by different processes, and stores it in **mySum** in order. Due to the sequential reading of a single process, as the number of nodes increases, communication overhead and running time inevitably increase. Moreover, due to slow communication between nodes, running time significantly increases from a single node to multiple nodes.

The longest process in **crtInfo.PostProcess** is also to read data and store it in **sumdata**, which incurs a lot of IO overhead.We can see from the code block below that the data reading and storage in this section are also carried out in parallel, which greatly slows down the running efficiency of the program under multiple nodes.

```
    for (USI p = 0; p < numproc; p++) {
        const string myFile = dir + "proc_" + to_string(p)
            + "_FastReview.out";
        ifstream ifs(myFile, ios::in);

        if (!ifs) {
            OCP_MESSAGE("Trying to open file: " <<
                (myFile));
            OCP_ABORT("Failed to open the input file!");
        }

        if (p == 0) {
            // Get time steps
            ......
            // Get Item
            ......
            // Get Unit
```

```
            ......
            // Get val
            ......
        }
        else {
            // skip first three lines
            ......
            // Get val
            ......
        }
        ifs.close();
        if (remove(myFile.c_str()) != 0) {
            OCP_WARNING("Failed to delete " + myFile);
        }
    }
```

Listing 6.3: Read data from the results obtained from various processes

6.3 Optimization

In our testing, we found that the program has significant issues in MPI communication and vectorization, so we optimized the program from these two aspects.

6.3.1 Intel oneAPI Implementation

Intel Suite Introduction

- **intel mkl**: the Intel® Math Kernel Library (Intel® MKL) is a highly optimized, extensively threaded, and vectorized math library that aims to maximize performance on Intel processors. It's part of Intel's suite of software development tools and is widely recognized for providing high-performance mathematical computing functions to applications.
- **intel MPI**: the Intel® MPI Library is a multi-facet communications library developed by Intel, focusing on enabling highly scalable, high-performance computing (HPC) applications. It is part of Intel's suite of tools for HPC and is designed to optimize the performance of MPI (Message Passing Interface) applications on Intel architecture, especially in cluster environments.

Intel Suite installation

we found that official recommand oneAPI suite to build OpenCAEPoro, so we load needed module

```
source /work/share/intel/oneapi/setvars.sh intel64
module load icc/2022.1.0
#module load mpi/2021.6.0
module load mkl/latest
module load compiler/2022.1.0
module load cmake/3.25.2
```

```
7 module load python
```

which has set oneapi env done and load mkl libraries. As intel mkl had provided well implemention of blaslib and lapacklib, so we simply build lapack. then follow the instruction we can successfully build openCAEPoro

6.3.2 HPCX Implementation

HPCX Introduction

HPCX, developed and maintained by Mellanox Technologies (now part of NVIDIA following an acquisition), is a comprehensive software suite designed to enhance the performance of high-performance computing (HPC) systems. HPCX is particularly tailored to leverage the advanced features of Mellanox/NVIDIA networking technologies, including InfiniBand and high-speed Ethernet.

Key Components and Features:

1. **Optimized MPI Library**: At the core of HPCX is its highly optimized implementation of the Message Passing Interface (MPI), which is the de facto standard for parallel programming in the HPC community. This MPI library is tailored for high-throughput and low-latency communication, crucial for large-scale simulations and data-intensive tasks.
2. **Advanced Communication Technologies**: HPCX takes full advantage of cutting-edge network technologies like InfiniBand, delivering superior performance through features like GPUDirect RDMA, which enables direct communication between GPUs across nodes, bypassing the CPU to reduce latency.
3. **Enhanced Collective Operations**: The toolkit includes specialized libraries for collective communication operations (e.g., HCOLL). These libraries provide optimized algorithms for collective tasks such as broadcasting and reduction, significantly improving communication efficiency in large-scale parallel applications.
4. **Support for SHMEM/PGAS Models**: Beyond MPI, HPCX offers support for Partitioned Global Address Space (PGAS) programming models, including OpenSHMEM, enabling more intuitive development of parallel applications with an alternative to the traditional message-passing paradigm.
5. **Multi-threading and Concurrency**: Recognizing the importance of multi-core and multi-threaded architectures in modern HPC systems, HPCX provides robust support for threading and concurrency, ensuring efficient utilization of available compute resources.
6. **Performance Analysis Tools**: To aid in identifying bottlenecks and optimizing performance, HPCX integrates with several performance analysis and debugging tools. These tools help in profiling and tracing HPC applications, offering insights into resource utilization and computational efficiency.

7. **Wide Compatibility and Portability**: HPCX is designed to be highly portable across various HPC platforms and is compatible with a broad range of hardware architectures, operating systems, and compilers. This flexibility makes it a versatile choice for diverse HPC environments.

HPCX_GNU version installation

get NVIDIA HPCX from the official website `https://developer.nvidia.com/networking/hpc-x`. we choose the newest 2.17.1-CUDA12.x-LTS, after unzip we run command to set vars.

```
cd /hpcx_directory/
export HPCX_HOME=PWD
source hpcx_init.sh
hpcx_load
```

External open-source libraries

then we clone external open-source libraries from official github repositories and unzip them and then begin to build them. firstly, we load cmake from module

```
module load cmake/3.25.1
```

Because we installed HPCX with GNU compiler suite, in order to link corresponding optimization libraries provided by HPCX, we changed compiler setting in the make.inc or CMakeList file or even add configure option while configuring, then we finished build all external libraries except petsc.

```
export CC=mpicc
export CXX=mpicxx
export FC=mpifort
```

petsc we met an error about fortran as new MPI implemention was used, and it needs mpifort as linker so we add `FC=mpifort` to solve this problem then we met a problem shows missing link to blas lib, we tried to add `-with-blaslib=directory_to_blas` but doesn't work, so we choose to add recommended option `-download-fblaslapack=1` to let it automatically download the suitable version of blaslib it needed. Because of the internet limit of the login node, so we fail to let blaslib download automatically, we choose to manually download lib to local and then use `sftp` to upload it to the login node. then with `-download-fblaslapack=download_path` we build petsc successfully.

It is worth mentioning that due to the competition with teammates on the same cluster, the direct modification of `.bashrc` led to the coverage and confusion of environment variables, so that when compiling, it did not link to the library built by himself but linked to the library linked by

teammates, resulting in a series of errors, which were finally solved by completely resetting the environment variables. The lesson is that the environment variables should be configured in their own scripts in the future to avoid confusion.

OpenCAEPoro while building openCAEPoro, after adding `-DBLAS_LIBRARIES=path_to_blaslib librefblas.a` we met dynamic linking error. Then we tried to make blaslib dynamically linking to OpenCAEPoro, so we after compiling all source code with mpifort, then we add `-fPIC` to FFLAGS on make.inc we run

```
mpicc -shared -o librefblas.so *.o
```

to build librefblas.so, but it failed because of FFLAGS_NOOPT make some file still compiling without `-fPIC` option, after modify `FFLAGS_NOOPT=$(FFLAGS)` we successfully build `librefblas.so`.

then after adding `-DBLAS_LIBRARIES=path_to_liblapack liblapack.a`, we met error about fortran linking error so we build lapack-lib again with `-lgfortran` and modified CMakeList of openCAEPoro

```
set(CMAKE_CXX_FLAGS_RELEASE "{CMAKE_CXX_FLAGS_RELEASE} -O3
    -lgfortran ")
```

then we successfully build OpenCAEPoro with HPCX

6.3.3 Comparison

We compared the data of openCAEPoro running under two MPI implementations on two different hardware platforms, and calculated their difference and optimization rate under different metric.

$$\text{Difference} = \text{HPCX Time} - \text{Intel Time} \tag{6.23}$$

$$\text{Ratio} = \frac{HPCX\ Time}{Intel\ Time} \tag{6.24}$$

Queue 38 (See table 1.5)

Queue 1t75c (See table 1.6)

Summary

Our report results show that HPCX library has a stable optimization effect compared with intel library in different environments. intel library has a stable and better performance in Newton Step, and there is no big difference between the two in linear solver. In other aspects, especially input and output, HPCX library has a stable and better optimization effect than Intel library. The HPCX library has obvious advantages.

Metric	HPCX(s)	Intel(s)	Difference(s)	Ratio
Simulation Time	610.833	630.296	-19.463	0.969
Input & Partition	0.206	3.591	-3.385	0.057
Partition - ParMetis	0.034	0.041	-0.007	0.829
Input Reservoir	0.073	0.114	-0.041	0.640
Setup Simulator	0.004	0.004	0.000	1.000
Initialization	0.024	0.025	-0.001	0.960
Assembling	22.180	28.836	-6.656	0.769
Assembling for LS	5.769	6.740	-0.931	0.861
Linear Solver	563.516	571.018	-7.502	0.987
Newton Step	2.986	2.718	0.268	1.099
Updating Properties	11.530	12.179	-0.649	0.947
Output	0.404	0.531	-0.127	0.761
OBJECT TIME	73.934	83.063	-9.129	0.890

Table 6.1: Performance Comparison Results of queue 38

Metric	HPCX(s)	Intel(s)	Difference(s)	Ratio
Simulation Time	485.288	433.914	51.374	1.184
Input & Partition	0.243	1.183	-0.940	0.205
Partition - ParMetis	0.080	0.116	-0.036	0.690
Input Reservoir	0.043	0.088	-0.045	0.489
Setup Simulator	0.002	0.004	-0.002	0.500
Initialization	0.022	0.022	0.000	1.000
Assembling	12.176	15.608	-3.432	0.780
Assembling for LS	5.126	7.835	-2.709	0.654
Linear Solver	456.717	396.927	59.790	1.151
Newton Step	1.830	1.465	0.365	1.249
Updating Properties	6.113	6.010	0.103	1.017
Output	0.341	0.578	-0.237	0.590
OBJECT TIME	41.539	49.019	-7.480	0.847

Table 6.2: Performance Comparison Results of queue 1t75c

6.3.4 Optimize compilation parameters

We adjusted the compilation parameters to attempt to optimize the program.To ensure consistency between the program and related dependency libraries, we used the same compilation parameters for all dependency libraries.

For Lapack, modify the compilation parameters as follows

```
1 make blaslib CFLAGS='-O3 -march=native -fopt-info-vec'
      FFLAGS='-O3 -march=native -fopt-info-vec'
      CXXFLAGS='-O3 -march=native -fopt-info-vec'
2 make cblaslib CFLAGS='-O3 -march=native -fopt-info-vec'
      FFLAGS='-O3 -march=native -fopt-info-vec'
      CXXFLAGS='-O3 -march=native -fopt-info-vec'
3 make lapacklib CFLAGS='-O3 -march=native -fopt-info-vec'
      FFLAGS='-O3 -march=native -fopt-info-vec'
      CXXFLAGS='-O3 -march=native -fopt-info-vec'
4 make lapackelib CFLAGS='-O3 -march=native -fopt-info-vec'
      FFLAGS='-O3 -march=native -fopt-info-vec'
      CXXFLAGS='-O3 -march=native -fopt-info-vec'
```

For Parmetis, we need to modify the Makefile, adding the following content to the relevant section.

```
    CONFIG_FLAGS += -DCMAKE_C_FLAGS_RELEASE="-O3 -xHost
        -qopt-zmm-usage=high -qopt-report=2"

    CONFIG_FLAGS += -DCMAKE_CXX_FLAGS_RELEASE="-O3 -xHost
        -qopt-zmm-usage=high -qopt-report=2"
```

Other dependency libraries can also directly add relevant compilation parameters based on the default compiler.After confirming that the program and all dependent libraries were successfully compiled, we tested the program, obtaining the following results

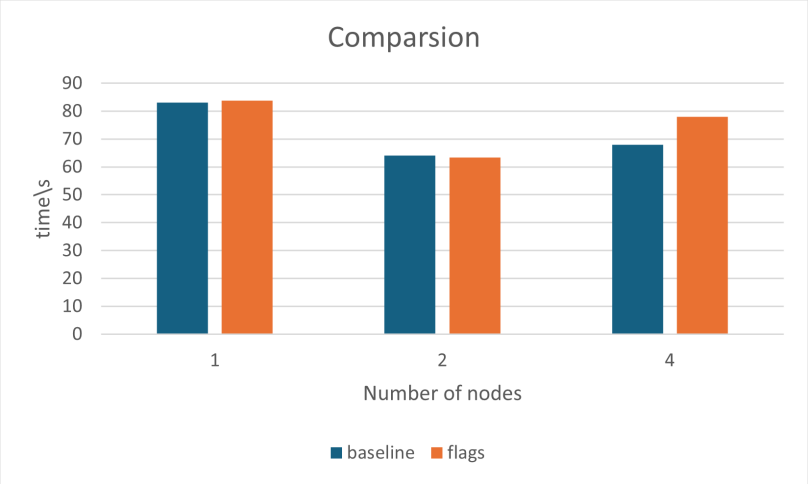


Figure 6.13: Comparison before and after adjusting compilation parameters

It can be observed that adjusting compilation options does not significantly optimize the program, and even in multi node scenarios, the total time may be even longer due to the additional increase in Output.

6.4 Correctness validation

Based on the preliminary notification, we have created `compare.py` to ensure the validity of our outcome. It is reuired that for the data in `FPR,FOPR,FGPR,FWPR,WBHP-INJE1, WBHP-PROD1`, we need to calculate for each attribute that satisfies

$$\frac{1}{|T|}\sum_{t\in T}\frac{|FPR(t)-FPR^{ref}(t)|^2}{FPR^{ref}(t)}<0.01 \tag{6.25}$$

The python script is as follows.

```
import numpy as np
import re
def read_output_file(file_path, time_points):
    #Read the contents of the output file and extract relevant data.
    data = {"TIME": [], "FPR": [], "FOPR": [], "FGPR": [], "FWPR": [],
         "WBHP-INJE1": [], "WBHP-PROD1" :[]}
    data["TIME"] = time_points
    with open(file_path, 'r') as file:
        lines = file.readlines()
        # Assuming that the header is fixed and contains the field names
        for index, line in enumerate(lines, start=1):
            if line.startswith("Row 1"):
```

```
12                line_row_1 = index
13            elif line.startswith("Row 2"):
14                line_row_2 = index
15
16        for line in lines[line_row_1 + 4 : line_row_2 - 2]:
17            print("Parsing line: {}".format(line))
18            processed_line = re.sub(r'\s+', ' ', line)
19            values = list(map(float, processed_line.strip().split()))
20            if values[0] in data["TIME"]:
21                data["FPR"].append(values[3])
22                data["FOPR"].append(values[5])
23                data["FGPR"].append(values[7])
24                data["FWPR"].append(values[9])
25                print("The values at time point {} are: FPR = {}, FOPR =
                      {}, FGPR = {}, FWPR = {}".format(values[0],
                      values[3], values[5], values[7], values[9]))
26            else:
27                continue
28
29        for line in lines[line_row_2 + 4: 2 * line_row_2 - line_row_1 -
               2]:
30            print("Parsing line: {}".format(line))
31            processed_line = re.sub(r'\s+', ' ', line)
32            values = list(map(float, processed_line.strip().split()))
33            if float(values[0]) in data["TIME"]:
34                data["WBHP-INJE1"].append(values[6])
35                data["WBHP-PROD1"].append(values[7])
36                print("The values at time point {} are: WBHP-INJE1 = {},
                      WBHP-PROD1 = {}".format(values[0], values[6],
                      values[7]))
37            else:
38                continue
39    return data
40
41 def compare_outputs(reference_data, test_data):
42    #Compare the values in the reference and test data dictionaries.
43    for field in reference_data:
44        reference_values = reference_data[field]
45        test_values = test_data[field]
46        count = 0
47        sum_deviation = 0
48        for ref_val, test_val in zip(reference_values, test_values):
49            if ref_val != test_val:
50                print(f"Difference in {field}: Reference={ref_val},
                      Test={test_val}, Difference = {ref_val - test_val}")
51
52            if ref_val != 0:
53                count += 1
54                sum_deviation += ((ref_val - test_val)*(ref_val -
                      test_val))/ref_val
55            else:
56                print(f"Reference value is zero for {field}, the sum
                      skips this values deviation")
57
58        print(f"The deviation for {field} is {sum_deviation / count}")
59        if(sum_deviation / count < 0.01):
60            print(f"The deviation for {field} is valid")
61        else:
62            print(f"The deviation for {field} is invalid")
63
64
65 if __name__ == "__main__":
66    # Replace these file paths with the actual paths of your reference
          and test output files
67    reference_file_path = "SUMMARY.out"
68    test_file_path = "SUMMARY_ref.out"
69    time_points_path = "time_points"
```

```
70
71      # Read data from the files
72      with open(time_points_path, 'r') as file:
73          content = file.read().strip()  # Read the content from the file
                and remove leading/trailing whitespaces
74          time_points = list(map(float, content.split(',')))  # Split the
                content using ',' as a delimiter and convert each element to
                an integer
75      time_points_length = len(time_points)
76      print(f"Time points: {time_points} and the number of time points are
            {time_points_length}")
77
78      reference_data = read_output_file(reference_file_path, time_points)
79      test_data = read_output_file(test_file_path, time_points)
80
81      # Compare the data
82      compare_outputs(reference_data, test_data)
```

Listing 6.4: compare.py

We have tested our results using the `compare.py` and all our results have passed the validation test.

6.5 Future Work

6.5.1 Improvement on CPR preconditioner

During the competition, we are given the permission to modify the file `PETSC_FIM_Solver.cpp`. We have analyzed the part that is free for modification.

The part that is free for modification accounts for the `Matrix_Conversion_Time` that will later be added into the object time. And during our analysis, this part accounts for 24s on one node and fall to 12s after using two nodes. Therefore, the parallelism of the FIM solver is functioning well. However, we try to utilize OpenMP to see if we can further accelerate the for loops inside the CPR preconditioner. There are mainly three loops inside the part that is free for modification.

```
for (int i = 0; i < nBlockRows; i++){
    nDCount[i] = 0;
    for (int j = rpt[i]; j < rpt[i + 1]; j++){
        if (cpt[j] >= Istart && cpt[j] <= Iend){
            nDCount[i]++;
        }
    }
    nNDCount[i] = rpt[i + 1] - rpt[i] - nDCount[i];
}
```

```
for (Ii = 0; Ii < nBlockRows; Ii++){
    for (i = 0; i < blockSize; i++){
        globalx[i] = (Ii + Istart) * blockSize + i;
    }
    for (int i = rpt[Ii]; i < rpt[Ii + 1]; i++){
        for (int j = 0; j < blockSize; j++){
            globaly[j] = cpt[i] * blockSize + j;
        }
```

Listing 6.5: The first loop in CPR preconditioner

```
        ierr = MatSetValues(A, blockSize, globalx,
            blockSize, globaly, valpt, INSERT_VALUES);
        CHKERRQ(ierr);
        valpt += blockSize * blockSize;
    }
}
```

```
for (i = 0; i < rowWidth; i++){
    bIndex[i] = Istart * blockSize + i;
}
```

Listing 6.6: The second loop in CPR preconditioner

For the first loop and the third loop, there are no data dependency in each iteration. So we can add `#pragma omp parallel for private ()` to parallelize them further. As in the following code:

Listing 6.7: The third loop in CPR preconditioner

```
#pragma omp parallel for private (nBlockRows, Istart, Iend)
for (int i = 0; i < nBlockRows; i++){
    nDCount[i] = 0;
    for (int j = rpt[i]; j < rpt[i + 1]; j++){
        if (cpt[j] >= Istart && cpt[j] <= Iend){
            nDCount[i]++;
        }
    }
    nNDCount[i] = rpt[i + 1] - rpt[i] - nDCount[i];
}
```

```
#pragma omp parallel for private(rowWidth, Istart,
    blockSize)
for (i = 0; i < rowWidth; i++){
    bIndex[i] = Istart * blockSize + i;
}
```

Listing 6.8: The modified first loop in CPR preconditioner

However, we need to add include files `#include<omp.h>` at the top to use OpenMP and we have to modify on CMakeLists to make the solver support OpenMP. We therefore give up on the implementation of OpenMP.

Listing 6.9: The modified third loop in CPR preconditioner

The MPI implementation of the CPR preconditioner is crucial for its implementation of partitioned matrix. Each process will be inserting values of different partition of matrix A. We fail to further parallelize the second loop above because of the data dependency issue. We are inserting into the matrix A in every loop, which may lead to concurrency issues. `MatSetValues()` is stated in `petsc/matrix.c` and as in the description, this method is used to insert values into a matrix using caches. After using the `MatAssemblyBegin()` and `MatAssemblyEnd()`, all the cached matrixes will be loaded into matrix A.

In our profiling, the function `MatSetValues()` is one of the hotspot functions. Therefore, we try to seek way to optimize the inserting process.

In the `matrix.c` file, there is another method called MatSetValuesBlocked(), which can insert or add a block of values into a matrix. This can greatly reduce the times we call on the function of `MatSetValues()`, which

theoretically can reduce the time needed. However, to insert using MatSetValuesBlocked(), we have to ensure that the inputs are valid:

```
   Example:
   Suppose m=n=2 and block size(bs) = 2 The array is

   1  2  | 3  4
   5  6  | 7  8
   - - - | - - -
   9  10 | 11 12
   13 14 | 15 16

   v[] should be passed in like
   v[] = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]

  If you are not using row oriented storage of v (that is
      you called
      MatSetOption(mat,MAT_ROW_ORIENTED,PETSC_FALSE)) then
   v[] = [1,5,9,13,2,6,10,14,3,7,11,15,4,8,12,16]
```

Listing 6.10: Requirements of `MatSetValuesBlocked()`

However, in the CPR preconditioner, we are storing the values by small blocks. Therefore, we need a method to convert the small blocks into rather big blocks. We anticipate that there might be some problems in this. First, as the `valpt` is stored in the order of small blocks by small blocks, converting it to a row-oriented way will lead to massive jumps of pointers, which might be another cost. Second, there are still loops when we try to convert the `valpt` in as row-oriented way. However, we will never know the trade-off if we never try it, so we try to use the following code to optimize our problems.

```
double *valpt = val;
int tot_size = nBlockRows * blockSize;
double *valpt_row = new double [tot_size * tot_size];
int *globalx = (int *)malloc(tot_size * sizeof(int));
int *globaly = (int *)malloc(tot_size * sizeof(int));
int blockRow;
int blockCol;
int rowInBlock;
int colInBlock;
int index;
//This Process is mainly setting the values of the Matrix.
for (int row = 0; row < tot_size; ++row) {
    for (int col = 0; col < tot_size; ++col) {
        blockRow = row / blockSize;
        blockCol = col / blockSize;
        rowInBlock = row % blockSize;
        colInBlock = col % blockSize;
        index = (blockRow * blockSize * tot_size) +
            (blockCol * blockSize * blockSize) +
            (rowInBlock * blockSize) + colInBlock;
        globalx[row] = Istart + row;
        globaly[col] = cpt[rpt[row]] + col;
        valpt_row[row * tot_size + col] = valpt[index];
    }
```

```
}
ierr = MatSetValuesBlocked(A, nBlockRows, globalx,
    nBlockRows, globaly, valpt_row, INSERT_VALUES);
CHKERRQ(ierr);
```

Listing 6.11: The modified third loop in CPR preconditioner

In this implementation, we are converting the `valpt` into an row-oriented way. However, we find that this modification will lead to segmentation errors and we try to find out the reasons. After outputing `val`, `rpt` and `cpt`, we found that the matrix is not storing all the elements. As the matrix is an symmetric sparse matrix, `val` only documents the crucial inputs in the matrix.

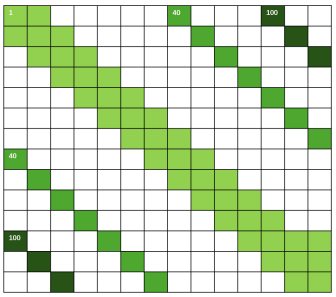


Figure 6.14: Illustration of the `val` storage

As in figure 6.14, the `val` only documents the green area, each box represents a small block with size `blockSize`.Therefore, it is not only hard but also space-consuming if we want to store the matrix in the row-oriented way. Therefore, we abandon this method.

Furthermore, we find that the `MatSetOption()` methods in `matrix.c` may provide help with the setup phase of the matrix. We `MatSetOption()` as follows before assembly and after setting the values. `MAT_SYMMETRIC` not really options of the matrix, they are knowledge about the structure of the matrix that users may provide so that they do not need to be computed (usually at a high cost).

```
ierr = MatSetOption(A, MAT_SYMMETRIC, PETSC_TRUE);
CHKERRQ(ierr);
```

Listing 6.12: `MatSetOption()`

However, when we tested in OpenCAEporo, the improvements are slight. The research into the CPR preconditioner code and relevant source codes have given us insight into how the preconditioner works in computer system. How to store a matrix and what it might cost to compute and operate on a matrix. Our future works can rely on the improvement of data structures and the data definition options it provide.

6.5.2 Parallel optimization of output

We mentioned in the code analysis section that the reason for the relatively long output time is that OpenCAEporo adopts a sequential reading method when reading files generated by various processes. In multi node scenarios, communication issues cause significant overhead.

We attempted to use OpenMP to enable multiple threads to speed up the reading process, but a series of issues arose. Meanwhile, if you directly write code for multi-threaded parts, there may be issues such as memory overflow.

In future work, we can achieve multi-threaded reading and writing of data by modifying the data structures of sumdata and mySum, adjusting the way data is read, and integrating the order of data, thereby accelerating the running speed of the program.

Bibliography

Here are the references in citation order.

[1] Massimiliano Fatica. 'Accelerating linpack with CUDA on heterogenous clusters'. In: *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU-2. Washington, D.C., USA: Association for Computing Machinery, 2009, pp. 46–51. DOI: 10.1145/1513895.1513901 (cited on pages 18, 19, 31).

[2] Je-Seok Ham et al. 'HPC LINPACK Parameter Optimization on Homo-/Heterogeneous System of ARM Neoverse N1SDP'. In: *The International Conference on High Performance Computing in Asia-Pacific Region*. HPCAsia '21. <conf-loc>, <city>Virtual Event</city>, <country>Republic of Korea</country>, </conf-loc>: Association for Computing Machinery, 2021, pp. 139–143. DOI: 10.1145/3432261.3439864 (cited on page 31).

[3] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture*. 2023. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/PB-10577-001_v02.pdf (cited on page 32).

[4] Michael Heroux and Jack. Dongarra. *Toward a new metric for ranking high performance computing systems.* en. Tech. rep. SAND2013-4744, 1089988, 456352. June 2013, SAND2013–4744, 1089988, 456352. DOI: 10.2172/1089988. (Visited on 01/05/2024) (cited on page 37).

[5] Michael A Heroux, Jack Dongarra, and Piotr Luszczek. 'HPCG Technical Specification'. In: () (cited on page 37).

[6] Vladimir Marjanović, José Gracia, and Colin W. Glass. 'Performance Modeling of the HPCG Benchmark'. en. In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Ed. by Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 172–192. DOI: 10.1007/978-3-319-17248-4_9 (cited on pages 38, 44).

[7] Xianyi Zhang et al. 'Optimizing and Scaling HPCG on Tianhe-2: Early Experience'. In: vol. 8630. May 2014. DOI: 10.1007/978-3-319-11197-1_3 (cited on page 44).

[8] Tom Brown et al. 'Language models are few-shot learners'. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901 (cited on page 54).

[9] OpenAI. *GPT-4 Technical Report*. 2023 (cited on page 54).

[10] Meta. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023 (cited on page 54).

[11] Mohammad Shoeybi et al. 'Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism'. In: *arXiv preprint arXiv:1909.08053* (2019) (cited on page 55).

[12] Geon-Woo Kim et al. 'VLLM: Very Large Language Model with Paged Attention'. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2020, pp. 6767–6777 (cited on page 55).

[13] Gyeong-In Yu et al. 'Orca: A Distributed Serving System for Transformer-Based Generative Models'. In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2022 (cited on page 55).

[14] Sylvain Gugger et al. *Accelerate: Training and inference at scale made simple, efficient and adaptable.* https://github.com/huggingface/accelerate. 2022 (cited on page 56).