Seminar Report

Application and System Benchmarks

Silin Zhao

MatrNr: 21569349

Supervisor: Marcus Merz

Georg-August-Universität Göttingen
Institute of Computer Science

March 26, 2023

Abstract

This report is about to discuss the tests of HPC performance for computational velocity with benchmarks, and is allowed to be published for academic purpose. As the begining of this report we are going to introduce the background and the purpose of the measurment for HPC performance in the Section 1, and such discussions will show us how such measurments indicate the HPC system performance. In Section 2 we are going to review some well-know benchmarks. For demonstration we will shortly recover the implementation and testing of 4 benchmarks. Then in Section 3 our topic is focusing on the special benchmark we interested, i.e. MiniBude benchmark. As we illustrate the performance in detail, we are going to go through the Serial, OpenMP and GPU implementation and performance with fine-turning different variables. In the Section 4 we will summary the measurments and talk about their drawback and weakness of this report, and we will end our report in Section 5 with conclusion.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC High-Performance Computing

HPL High-Performance Linpack

HPCG High Performance Conjugate Gradients

BUDE Bristol University Docking Engine

1 Background

Using benchmark to illustrate the HPC cluseter preformance has been widely accepeted, also has been treated as an indictor to estimate HPC system performance for ranking over the world.

1.1 The design of benchmark

Benchmark is such a application that people can utilize it for indicating the performance characters for HPC system. Such characters can be purely calculation capacity for CPU and GPU, or communication velocity based on parallel interaction etc [YFK22]. Normally if we want to estimate such a character, we have to go through those steps, implementation the application in our system, fine-turning some configuration varibles for different conditions, and summarizes the performance matrix. Usually the theoretical value can not be reached, because large application often works as a complete system, which need many kinds of instructions, such as numerical calculation, parallel communication, and IO etc. Hence people rather compare the resulte with the same implementation in different system than estimate the performance from the theoretical calculated value. They are some mechanisms to guarantee the implementated code is not modified for benchmark testing, such as in IO500.

The design of benchmark regular only focus on one character, such as IO500 is only designed for testing the IO operation performance, HPL is applied for testing the computational capacity.

About the problem that benchmark actually runs, people are inspired by the real scientific problems, such as from high energy physics or bio-physics. In order to scale the simulated problem, people are often allowed to use different size of input data, or controll the initialized value with configuration file.

1.2 The meaning of benchmark

HPC system is a complex system which contains massive hardware and software configurations. The question about whether those configuration are well defined and well matched is definitely not easy to estimate. By comparing the results of benchmark test from related systems can give us such a hint whether some potential capacity has been buried. Meanwhile, well-knowing the HPC preformance through benchmark can offer us a reasonable and efficient assignment of available computation resource for submited tasks.

1.3 The principles of Benchmark

We do not have some standardisation for designing benchmark, but we should take the popularization principle into consideration. The benchmark tests should be so designed to characterize the system in a well-defined and well-known way, that the results are easy to understand, meaningful and can be compared across different systems.

It's also important to ensure that the benchmark tests are scalable, so that they can accurately measure the performance of the system in different levels of parallelism. For example, to estimate the acceleration by using multi-threads or distributed system, in this way the benchmark tests should be designed to run on different numbers of threads or compute nodes.

Also we should play attention to the repeatablility of benchmark test, because it ensure that the results obtained from the tests are reliable and can be reproduced consistently over time[HCA16]. We can image that this case can be applied after the update or reconfiguration of system. Based on the repeatablility we are able to characterize the difference after changing the system.

We can also list some other relevances for the principles of benchmark, such as low energy cost, which means the execuation of the benchmark should be effective. Beside the transparency should also be considered, such as what is the test going for. In more detail, which problem is the benchmark implementated, and how the benchmark can be implementated in a easy way.

2 Benchmark introduction

In this section we are going to introduce some well-known benchmark, and give a short overview of how it works. Those benchmarks have been already well implementated in GWDG cluseter, and here we are going to give some instructions for self implementation.

2.1 IO500

This is a specific synthetic benchmark to measure the performance of storage systems, which use ior and mktest benchmark. Every year IO500 update its historic list, full list, IO500 list and 10-Node challenge list for different purpose. For the implementation we need to generate the configuration file[1], and in the generated configuration file we can change the storage system, fine-turning the transfered file size and block size. After loading the massage passing interface package for distributed system, such as MPI, we can execute IO500 with slurm. The results will indicate that how well the reading and writing operation perform with ior benchmark, and how fast the metadate operation [2] takes place with mdtest benchmark. As a example you can review the configuration process and results here[3], this site is also related to other 3 benchmarks, HPL, HPCG and Stream in following subsections.

2.2 HPL

HPL benchmark is a very famours benchmark for computational capacity and contribute to the TOP500 ranking list[Xu+20], which solves a random dense linear system in double precision arithmetic on distributed system using LU factorization. In the configuration file we can assige the problem size, and the initialized matrix will be block-cyclic distributed. The whole matrix will be recursive update based on the calculation with MPI transfered data. For the configuration we need to specific two liberaries, OpenBLAS and OpenMPI. We can use the compiled file by ourself or load them with module in SCC. If everything goes well, we can find the generated configuration file under /lib/bin directory, as the detail you can also review the link in footnote 3.

[1] Seeing the README file in the Github site: https://github.com/IO500/io500

[2] Management of file or directory, such as creating, rename, change ownership ect.

[3] https://pad.gwdg.de/s/w5_TJ9Yrp

2.3 HPCG and Stream

HPCG is a complementary to HPL, and it targets to a widely used patterns between computational resource and data access, which is more representative of real world scientific application. By default HPCG can automatically recognise the dependences, and the output is listed in two specialized files. Stream is special designed for testing the memory transfer rate for computational kernels[Kar+18]. This benchmark consists of the kernel functions that perform read, write, and copy operations on arrays of data only for the measurement for the memory bandwidth during the transfering. A briefly implementation and execuation of those two benchmarks can also be found with the above two benchmark at footnote 3.

3 Exploration Benchmark

From this section on we will foucs on we interested benchmark, MiniBude, which is also designed to measure the computational capacity for serial system, parallel system and GPU accelerated system. MiniBude consists of multiple implementations for running the screening the NDM-1 protein based on configurable varibles, such as iterations and poses of the protein[HNPF23]. Next we will go into the detail, from the description of the MiniBude, to how it is implementated for different variante, and then talk about the resulte for each implementation.

3.1 The description of MiniBude

This application is implementated in different HPC programming models[4]. Relying on the powerful computational capacity of HPC people try to simulate the process that we screen the possibe protein after multiple times of folding. Each folding can offer us many different kinds of poses and increase exponentially. We are going to simulate the process with difference implementations, in some implementations we can even use the GPU for acceleration.

Now let us review the implementation of MiniBude benchmark, the following is the all original MiniBude variante in Table 1, but because of the current limitations we are only be able to execute and discuss the 4 implementations in the first column of Table 1.

About the scalability of the size of the problem we have three options, i.e, *Small*, *Medium*, and *Large* as in the Table 2. For each of those option we can fine-turning by increasing the default iterations (default value is 8) for scaling our problem.

Well implementated	Defective implementated	Lack of dependences
OpenMP for CPUs	OpenMP target for GPUs	SYCL for CPUs and GPUs
Julia for CPUs	OpenACC for GPUs	Kokkos for CPUs and GPUs
CUDA for GPUs		
OpenCL for GPUs		

Table 1: All implementations from MiniBude are divided into 3 categories based on how well they are implementated in SCC cluseter.

[4]https://github.com/UoB-HPC/miniBUDE

	Small	*Medium*	*Large*
input data	mb1	mb2	mb2_long
poses	65536	65536	1048576

Table 2: Based on the input data and poses we have 3 (*Small*, *Medium* and *Large*) options of initial configaution to scale our problem size.

3.2 Defective Benchmarks

Before we go to the details about the first 4 well implementated benchmarks, let's make it clear that what is about the defective implementated and lacking of dependence benchmarks. For the defective implementated benchmarks which contains OpenMP target for GPUs and OpenACC for GPUs. About OpenMP target benchmark we need the clang compiler, so we have to load the llvm module. In GWDG SCC cluseter we can load llvm with spack as in Listing 1.

Listing 1: Load llvm module

```
module load spack-user
source $SPACK_USER_ROOT/share/spack/setup-env.sh
spack load llvm
```

And then another problem is prompted that we do not have nvptx64 dependence, and this is needed under llvm architecture. But I can't provide anything useful investigation for installing nvptx64 driver. So we comment line 45 in our implementation of OpenMP target for GPUs.

Listing 2: Disable nvptx64 driver because of lack of this dependence

```
#  TFLAGS_NVIDIA = -fopenmp -fopenmp-targets=nvptx64 -Xopenmp-target
```

In this way we are still able to compile this implementation, but the benchmark can only be executed on CPU with OpenMP-4 with computational velocity around 2.44 GFlops/s.

The other defective implementation, i.e, OpenACC for GPUs, can't launch GPU as well. Unlike OpenMP target implementation, which can only be compiled by llvm, OpenACC can be compiled by multiple architecture, and we choose GNU compiler.

Listing 3: Compile with GNU architecture

```
make COMPILER=GNU
```

The implementation can be well compiled without any error, but GPU is still not be launched. The result is illustrated in Figure 1, unlike the OpenMP target implementation use OpenMP of version 4, OpenACC implementation use OpenMP of version 5. Hence the computational speed is about 5.2 GFLOPs/s, which is much more faster than OpenMP 4. Because of the increasing the iteration, the consumption of time in totally is also increased likewise.

As in the last column of Table 1 we have also SYCL and Kokkos implementations, but from the instruction we can see that the SYCL and the Kokkos application are needed as dependences, for the simplification there is no such investigation for installing SYCL and Kokkos has been done for the report. In the following we will go into details about the well implementated benchmarks.

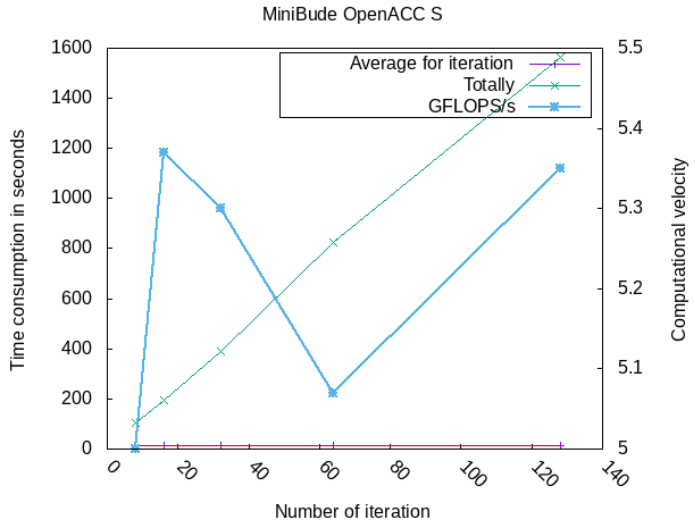

Figure 1: The computational velocity and time consumption for OpenACC implementation with *Small* input data.

3.3 OpenMP implementation

OpenMP benchmark of MiniBude has been implementated for shared memory system. From the instruction of the implementation we can customize the number of threads for acceleration. As in Listing 3, using GNU architecture we can compile this implementation smoothly, and the executable file is named bude and generated in the root directory of this implementation.

Because this benchmark is for shared memory system, we don't need to load the openmpi model, but in order to set the thread we have to customize the OMP_NUM_THREADS environment variable as in Listing 4 [5],

Listing 4: Execuation of OpenMP implementation

```
#!/bin/bash
#SBATCH --job-name B_openmp
#SBATCH -N 1
#SBATCH -p medium
#SBATCH -n 1
#SBATCH --time=3:00:00

export OMP_NUM_THREADS=1
./bude -i 8 -n 65536 --deck ../data/bm1
```

Listing 4 is the submited bash file for running this benchmark, for the problem size we use the *Small* size in Table 2, and the default iteration is set to be 8 as default. In Table 3 we list the numbers of threads for our report. Not only in this implementation, also in other implementations of MiniBude benchmark we scale the threads always in this way.

[5] To execute bude, -i is for the iteration, -n is the poses and –deck is for the inputed file path.

Threads	1	2	4	8	16	32

Table 3: The number of threads for fine-turning.

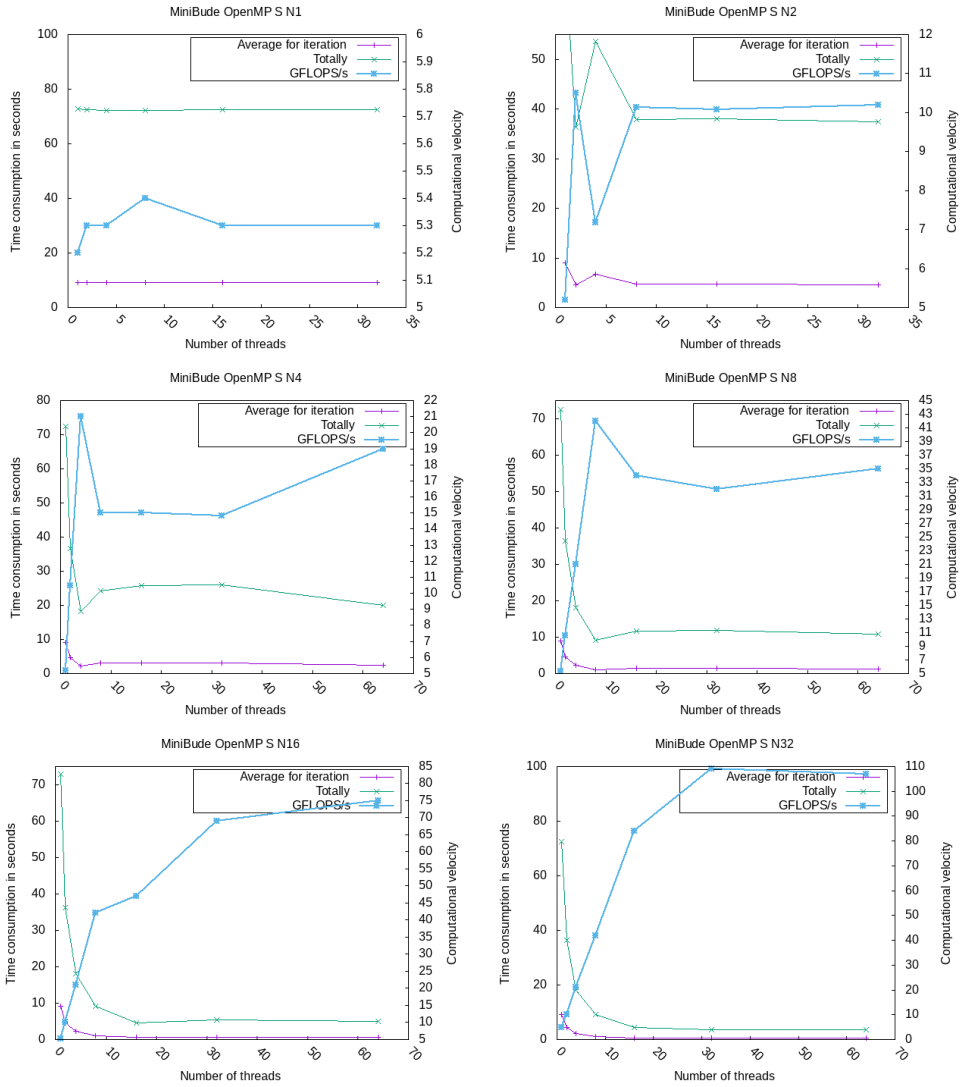

Figure 2: The time consumption for totally and average for each iteration illustration with increasing of threads in OpenMP implementation benchmark

The time consumption for totally and average for each iteration has been illustrated in Figure 2. From the title of each sub-image we see the number of CPUs that stands after the character N, and this number is also increased just like threads in Table 3. We know that the threads are running in CPU based on the rotating of execution. As the threads are swithed from the one to another, there will be a mount of time consuming for variables reloading from register. In order to gain the better performance, we want each thread is launched in different CPUs, this is the reason we increase the number of CPUs exactly as the increasing of threads.

In the first sub-image of Figure 2, we set the number of CPU to be 1, which means, all the computation is serial, no benefit of parallelism has to been contributed. No matter how many threads we try to scale as in Table 3, the computational speed stay almost the same around 5.3 GFLOps/s, hence the totally and average time consumption also

make no difference as the increasing of thread. We can understand in such a way, that it doesn't matter in which way the threads are switched from one to another, the serial implementation of task can not be scaled. We should keep this fact in mind that if we run multiple threads only in one CPU, the speed is limited at around 5.2 GFlops/s. This appearance also occurs in the rest sub-image in Figure 2 when we only run 1 thread for multiple CPUs.

As we move on to multiple CPUs in this OpenMP benchmark implementation. We can see a very clear fact that upto 16 CPUs, if the number of threads equal to the number of CPUs, the computational velocity has the best preformance. Before this point is reached, the speed is growing during the increasing of threads almost linearly. After this point the speed is decreasing a little bit, and stay approximately the same.

Above 16 CPUs, a fast increasing phase of speed before the number of threads reach the number of CPU, after that we don't see a decreasing phase comparing to less than 16 CPUs. But the continuous increasing of performance stoped, and no further benefit anymore for keeping growing threads.

Based on the fact that we obtain the best performance when each processor launchs each thread, we collect the data from above and combinate them together only for this case in Figure 3. Here we plot also the linear scalability of computational speed as yellow line and mark with 'Regression'. As this yellow line shows, the increasing of computational speed is in reality about 30% slower than linear scalability, and this contribution is mostly cased by the overhead while task assignment between the threads.

We explored the performance for *Small* problem size in detail, we do not repeat such a huge investigation again for *Medium* and *Large* problem size, because the basic computational speed of OpenMP implementation do not perform better than other problem size. Next we move to Julia implementation.

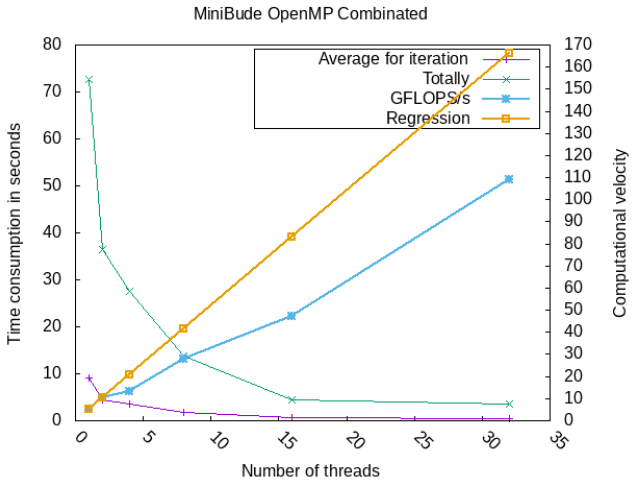

Figure 3: The best performed time consumption and computational velocity based on the number of threads matching the number of CPUs.

3.4 Julia implementation

In this subsection we are going to illustrate the performance of Julia implementation of MiniBude benchmark. Let's give a briefly introduction for Julia, based on the fact that it is designed to compile code just-in-time (JIT) at runtime, this allows Julia to achieve

performance levels that are compatible to Fortran and C[God+23]. Beside because of the productivity and nature spport for parallelism, this make Julia becomes a nonnegligible option for HPC.

Listing 5: Compiling Julia implementation

```
module load julia
./update_all.sh
julia --project=Threaded -e 'import␣Pkg;␣Pkg.instantiate()'
```

As in Listing 5 we need to compile julia code at first. For the execuation of julia implementation we submit the task with slurm as Listing 6. In order to save the energy we scale our threads as in Table 3 as we scale the CPU at the same time, which means for each test the number of CPUs is equal to the number of threads, and so that we can run this benchmark implementation for *Small*, *Medium* and *Large* problem size for best performance, and the final result is shown in Figure 4.

From left to right in Figure 4 we see that the time consumption for totally and average for each iteration rapidly grows with the increasing of problem size (*Small*, *Medium* and *Large*). We start with the discussion about the performance for the first sub-image for *Small*. With only one CPU and one thread (serial) we see that the basic computation velocity is about 36 GFlops/s, which is about 7 times larger than the speed in OpenMP implementation.

Listing 6: Execuation of Julia implementation

```
#!/bin/bash
#SBATCH --job-name julia
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -p medium
#SBATCH --time=3:00:00

export JULIA_NUM_THREADS=1
julia --project=Threaded src/Threaded.jl -i 8 -n 65536 --deck ../data/bm1
```

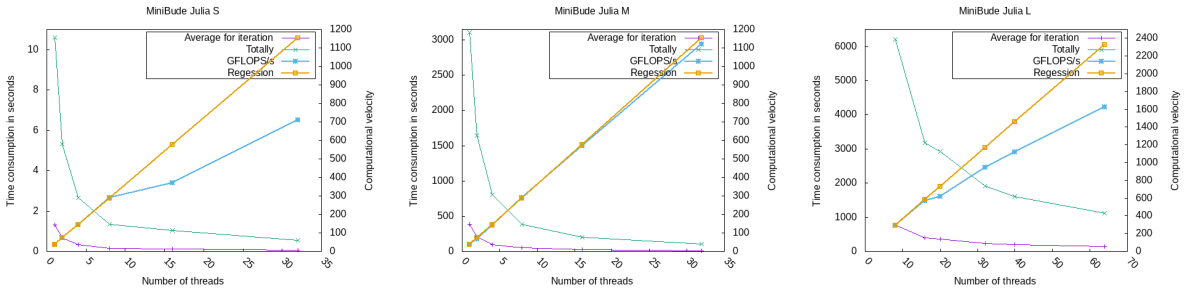

Figure 4: The time consumption for totally and average for each iteration illustration with increasing of threads in Julia implementation benchmark.

As the problem size increasing, we need more time to finish the benchmark. In the *Large* problem size we do not following the settings as in Table 3, because even only for the smallest case, i.e, 8 threads, we need about 2 hours. Instead of decrease the threads number to 4, 2, and 1, we increasing it [6] for saving the time consumption.

[6]The total threads number: 8, 16, 20, 32, 40, 64

Likewise as in Figure 3 we also plot the theoretical value of linear scalability based on the increasing of threads as yellow line, and mark it as 'Regression'. At this point we can see that for *Medium* problem size, the computational speed is perfect growing as theoretical scalability. The yellow line (Regression) matchs the green line (GFLOPS/s) upto 32 threads.

In MiniBude benchmark we have multiple implementation for Julia [7], but we only well-implementated here the multi-thread variant in SCC cluseter. That is because the effort of compiling Julia implementation with GPU doesn't work. After we load the CUDA and update the compilation as Listing 7, we still can't launch GPU implementation with Julia. According to the feedback we have to set the CUDA version using CUDA.set_runtime_version for giving the precompiled julia code an appropriate environment, but we don't find any instruction for the wanted CUDA driver version for this GPU implementation in Julia.

Listing 7: Loading CUDA and update Julia compilation

```
module load cuda
./update_all.sh
```

But in the following subsections we have 2 well-done GPU implementation as in Table 1.

3.5 CUDA implementation

In this subsection we are going to explore the successful GPU acceleration for computational speed in MiniBude benchmark, and let's start with CUDA implementation. As in Listing 8 we load the cuda environment and compile the source code.

Listing 8: Loading CUDA and compiling CUDA implementation

```
module load cuda
make COMPILER=GNU
```

Listing 9: Execuation of CUDA implementation

```
#!/bin/bash
#SBATCH --job-name B_cuda
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -p medium
#SBATCH -p gpu
#SBATCH -G v100:1
#SBATCH --time=1:00:00

./bude -n 65536 -i 8 --deck ../data/bm1
```

For the exploring the CUDA acceleration of our benchmark, we need some special commands to launch GPU as in Listing 9. $v100:1$ stand for using only one v100 GPU, and the available GPU of SCC in GWDG is listed here[8]. For this implementation we let the thread to be default as 1, and execuation is launched for 3 different problem size as in Figure 5. In the left side of all sub-image, our time consumption is increasing significantly.

[7] https://github.com/UoB-HPC/miniBUDE/tree/master/miniBUDE.jl
[8] https://hpc-neu.gwdg.de/hpc/systems/scc/

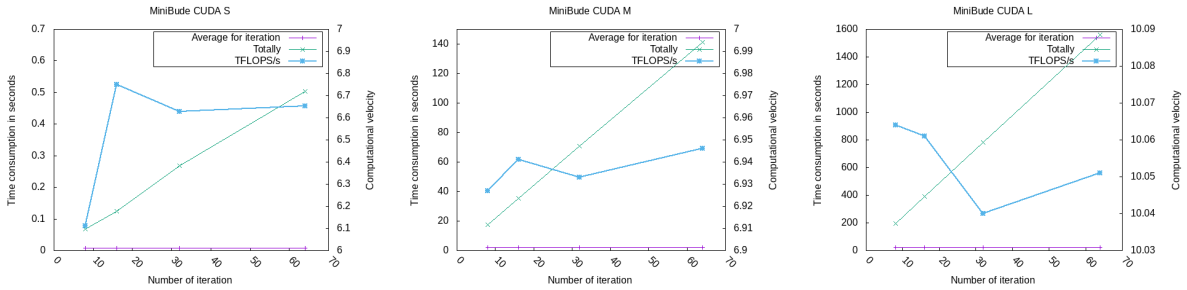

Figure 5: The time consumption for totally and average for each iteration illustration by increasing of iterations in CUDA benchmark.

To illustrate more detail about the performance about CUDA acceleration, we increase the computational demand by scaling its iterations[9]. In Figure 5 we can clear find that the time cost in total is linear increased by the increasing the iteration, and the average time consumption for each iteration stays the same.

The most interesting measurement is the computational speed, for *Small* and *Medium* problem size we see that the velocity is about 6.6 TFlops/s, and this is a massive improvement comparing to non-GPU. Meanwhile for *Large* problem size we see there is another improvement, from about 6.6 TFlops/s to about 10 TFlops/s.

3.6 OpenCL implementation

This is another well-implementated GPU benchmark of MiniBude, and the dependence load and compilation is the same as CUDA implementation as in Listing 8. To launch this implementation we use the same instruction as for CUDA implementation in Listing 9, and the exploration is also with the same problem size as before. The only difference is we increase the times of iteration upto 128[10].

From Figure 6 we also see that the enormous increasing of time consumption over different problem size from *Small* to *Large*, and with the increasing of iteration we see the linear increasing of time consumption for totally. Also the computational velocity is at level of TFlops/s. But we see a slightly reduction comparing to CUDA implementation, from 6.6 Tflops/s to 5.7 TFlops/s. Such reduction appears for all problem size, but the speed increasing at *Large* problem size over *Small* and *Medium* is also happens in this implementation.

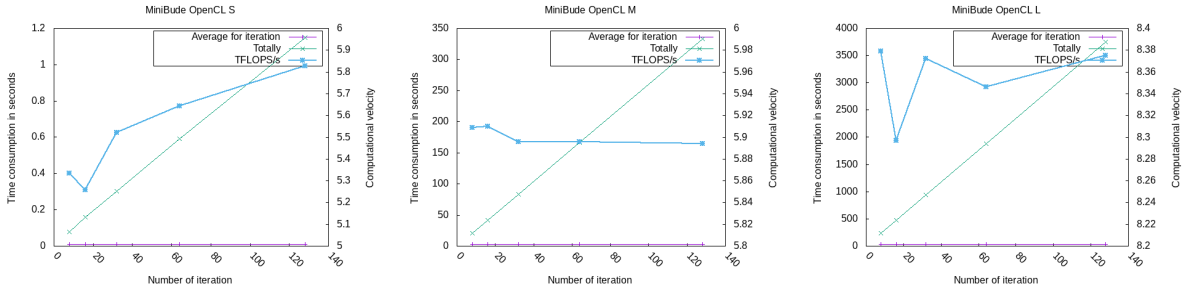

Figure 6: The time consumption for totally and average for each iteration illustration with increasing of iterations in OpenCL implementation benchmark.

[9] 8, 16, 32, 64
[10] 8, 16, 32, 64, 128

4 Summarizes and Drawback

In this report we started with the discussion about why and how benchmark should be designed. In HPC research area we are concerning about weather our system is able to expose its potential capacity. With benchmark testing we can compare the system performance with historic settings or over different cluseters. The core part of benchmark is to execute a scientific problem, such as from physics or biology. The execuation of such benchmark should satisfy some criterions. Such as the benchmark should be representative, in this way people can understand easyly what this benchmark stands for. Also the benchmark should be able to be repeated identity and verified. Meanwhile the implementation should obtains the scalability for different problem size. Those relevances and principles can guarantee the reproduction of the benchmark for different timestamps and comparsion between different system.

This report recalled 4 well implementated benchmarks in GWDG, IO500, HPL, HPCG and Stream. From compiling to execuation we briefly discuss those implementation in SCC cluseter, and mentioned some details for testing.

Next we detailly go through the MiniBude benchmark, which is a synthetic application and contains multiple implementations. Because of the lack of dependence we can't launch all the implementations, as the drawback of this report we should investigate more effort to install such dependence, such as Kokkos and SYCL. As for the defective implementation we are able to launch the benchmark, but GPU is not trigged.

As the core parts of this report we foucs on the 4 well-implementated benchmarks, i.e, OpenMP, Julia, CUDA and OpenCL. OpenMP and Julia is for CPU implementated and we explore the performance with increasing threads. In order to find the best configaution to launch the CPU benchmark, we fine-turning the number of CPUs and threads in OpenMP implementation. The fact shows us that if the number of CPU matches the number of threads, we have the best performance of computational speed. With this indication we configured our Julia implementation also in this way, and acqurie a interesting computational speed at 36 GFlops/s. Meanwhile as the drawback of this report, the attempt to launch other implementations with Julia, such as GPU, are failed. Highly because of the imcompatible GPU driver between compiling Julia code and executing julia code according to the failed attempt.

As the last part of this report we discuss the two well-implementated GPU benchmarks with CUDA and OpenCL. In GPU acceleration we run only one CPU and let the thread to be as default, and explore the computational velocity based on the scaling of iteration for different problem size. The computational speed is impoved from 6.6 Gflops/s to 5.7 TFlops/s comparing to non-GPU acceleration. For both GPU implementations we see a significante improvement of computational velocity in *Large* problem size over other different problem size, and as another drawback of this report, we don't know the reason for this improvement.

5 conclusion

In HPC area, no matter for academic or industrial, the benchmark testing is definite needed as indictor for operating the system. We introduced the background of designing, meaning and principles of benchmark and reviewed the 4 well-done benchmark in GWDG.

As a example we choosed MiniBude benchmark for demostration, from implementation to execuation, we illustrate the results of computational velocity measurements. At the same time as drawback of this report there are still many effort can be further investigated, we finish our report here, and hopefully this report can be valuable as reference for other people. Based on reseach purpose this report is allowed to be published and shared.

References

[God+23] William F. Godoy et al. *Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes*. 2023. arXiv: 2303.06195 [cs.DC].

[HCA16] Sascha Hunold and Alexandra Carpen-Amarie. *MPI Benchmarking Revisited: Experimental Design and Reproducibility*. 2016. arXiv: 1505.07734 [cs.DC].

[HNPF23] Pablo Herrera-Nieto, Adrià Pérez, and Gianni De Fabritiis. *Binding-and-folding recognition of an intrinsically disordered protein using online learning molecular dynamics*. 2023. arXiv: 2302.10348 [q-bio.BM].

[Kar+18] Jeyhun Karimov et al. "Benchmarking Distributed Stream Data Processing Systems". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018. DOI: 10.1109/icde.2018.00169. URL: https://doi.org/10.1109%2Ficde.2018.00169.

[Xu+20] Gen Xu et al. *Simulation-Based Performance Prediction of HPC Applications: A Case Study of HPL*. 2020. arXiv: 2011.02617 [cs.DC].

[YFK22] Yijing Yang, Hongyu Fu, and C. C. Jay Kuo. *Design of Supervision-Scalable Learning Systems: Methodology and Performance Benchmarking*. 2022. arXiv: 2206.09061 [cs.CV].