Critique on "Productivity, Portability, Performance: Data-Centric Python"

SYSU-SCC Team 12[1]

Sun Yat-sen University Guangzhou, China

Abstract

In SC21, Ziogas et al. proposed Data-Centric Python, or DaCe for short, which can enhance the performance, portability and productivity of Python. In the Reproducibility Challenge of the Student Cluster Competition, our task is to reproduce the results from the original work. In this paper, we perform experiments on single-node CPU/GPU and multiple nodes for the distributed version. All of our experiments are performed on the Azure Cycle Cloud platform. We reproduced the CPU, GPU, and distributed results from the paper and self-tuned several options to get better performance on heterogeneous machines.

Keywords: Reproducible computation, Student Cluster Competition

1. Introduction

Python is a language with great productivity. Thanks to the powerful module such as Numpy, Python can be used for more scientific areas. For HPC(High Performance Computing) applications, developed by Python can also be very productive. However, compared to traditional C and Fortran, Python cannot guarantee high performance. Hence, for HPC experts, it becomes urgent to enhance the performance of the Python language.

To deal with the existing challenges, Ziogas et al. propose DaCe Python. It's a toolbox to translate traditional Python code to the Data-Centric SDFG IR and adapt the SDFG IR to the architecture-specific code. Through this procedure, DaCe Python attains excellent performance, portability and further extends the original productivity of Python. This paper presents the results obtained by reproducing the DaCe results on CPU, GPU, and distributed nodes. We use the original NPBench dataset from the paper and the additional NPBench dataset provided by the author during Student Cluster Competition. Concerning the reproduction issues, we choose a hardware configuration similar to the configuration in the DaCe paper.

The remainder of this paper is organized as follows. section 2 provides details about our configuration for the experiments. section 3 describes our testing methodology. section 4, section 5, and section 6 show the results of our reproduction running on CPU, GPU, and distributed nodes. In section 7, we discuss the details of secret task. In section 8, we conclude our discussion.

2. Experimental Setup

To validate the performance of DaCe[5], we perform our experiments on the clusters as Table 1 and Table 2.

Node	GPU Node	CPU Node	Login Node
Type	ND40rs_v2	HB120r_v3	D4_v2
Processor	Intel® Xeon® Platinum 8168	AMD EPYC 7V73X	Intel® Xeon® Platinum 8171M
Num of cores	40	120	4
spec	2.70GHz	3.05GHz	2.1GHZ
Memory	32GB	448GB	14GB
Storage	672GB	1920GB	200GB
GPU Card	8 × V100 with 32 GB	None	None

Table 1: CPU and GPU Node

[1]huangh367@mail2.sysu.edu.cn

Node	**Distr Node**	**Login Node**
Type	HB120rs_v3	D4_v2
Processor	AMD EPYC 7V73X	Intel® Xeon® Platinum 8171M
Num of cores	120	4
spec	3.05GHz	2.1GHZ
Memory	4448GB	14GB
Storage	1920GB	200GB
NUMA nodes	4	1
RDMA Support	200 Gb/sec HDR InfiniBand	None

Table 2: Distributed Node

2.1. Hardware Selection

We carry out our experiments on three clusters concurrently because of the time limitation. For the CPU cluster, we choose one D4_v2 VM as our login node and one HB120_rs v3 VM as compute node. The parameters of them are listed in Table 1. For GPU experiment, the login node is D4_v2 VM while one ND40_rs v2 VM is enough for simulating the environment of paper. We list the parameters of the experiment in Table 1. To run distributed experiment on as many CPUs as possible, we choose D4_v2 VM as our login node and HB120_rs v3 VM as the compute node. Since the VM has four NUMA nodes, we can run four processes on one HB120_rs v2 for better performance, while one process per NUMA node. We set the number of threads 30 to take advantage of cores. The parameters of them are listed in Table 2.

2.2. Software Selection

For the reason of convinence, our operating system is the microsoft-dsvm:ubuntu-hpc:2004:latest image from the Azure market. We reuse GCC 9.3.0 and CUDA 11.4 as backend compiler, provided by the image. As for the other softwares, we select spack[3] to help us install and manage them. We install Intel-oneapi-compilers 2022.1.0, Intel-oneapi-mkl 2022.1.0, CPython 3.8.5, Cmake 3.23.1 and MPICH 4.0.2 by spack.

For the Python library, we mainly install Numpy 1.19.2 with Intel MKL support, DaCe 0.14[2] and npbench 0.1[4](branch scc22). What's more, we reuse the paint script from the previous artifac[1].

2.3. Code Modification

In the secret task of the SCC, we are required to modify the code of the following three benchmarks to the form applicable to DaCe Python: lstsqr, specialconvolve and wdist. The original version of these three benchmarks can not be analyzed by DaCe since they aren't written in expected format. The detailed modification will be metioned in section 7.

3. Description of experimental run

The design of our experiments consists of three parts: CPU experiments, GPU experiments, and distributed experiments. We use the original data set in the paper and the additional data set provided by the author.

Since the Reproducibility Challenge should be run on Azure, we have designed a parallel method to save time and budget. To run as many benchmarks as possible, we run the CPU, GPU, and distributed experiments simultaneously. In distributed experiments, we bind the processes to NUMA nodes for better performance.

The CPU and GPU experiments are performed using the run_framework.py provided by the authors. Since we use a more powerful CPU and GPU than the paper, we have achieved less run time in both cases. For the CPU version, we observe a better speedup in significant test cases while worse speedups in small test cases.

The distributed experiments are performed in a similar way to the single-node experiments. We use the script polybench.py to set up the experiment and bind cores to NUMA nodes to utilize the AMD-120-core CPUs better. Also, we use a hybrid MPI/OpenMP layout, with 30 threads per process in the experiments. What's more, we experiment the benchmark in other parameters, introduced in section 6.

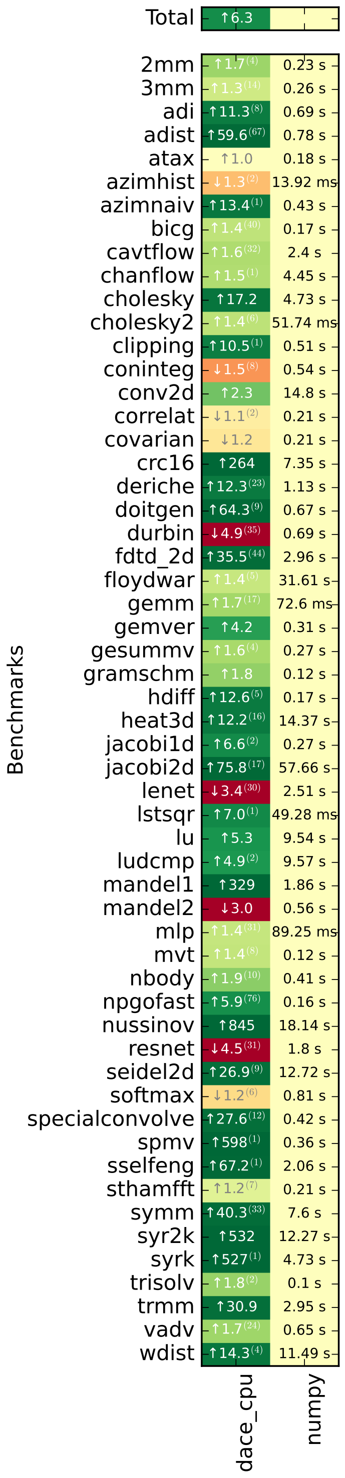

Figure 1: CPU Results

4. CPU Results

Our team performs CPU experiments on AMD CPUs from Azure. The hardware and software are listed in Table 1. We reuse the Python scripts from npbench(branch scc22)[4] with auto_opt, parallel, and fusion model. The auto_opt parallel and fusion models configure GCC compiler with flags(-std=c++14 -fPIC -Wall -Wextra -O3 -march=native -ffast-math -Wno-unused-parameter -Wno-unused-label). These flags could optimize the benchmarks according to the running machine. The figure 7 from original paper[2] shows that the speedup of DaCe over NumPy ranges from -1.1 to 948 times, which presents the high quality of DaCe to accelerate the Python codes. Compared with the original result, our reproducibility result ranges from -4.9 to 845 times.

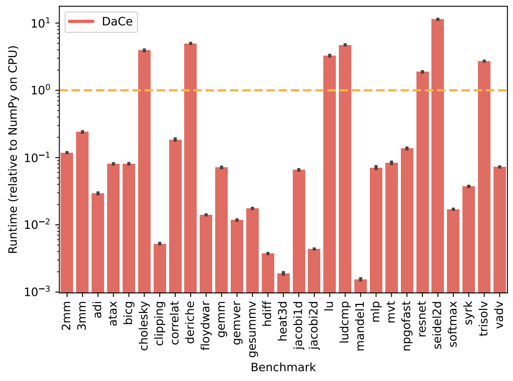

Figure 2: GPU Results

5. GPU Results

The GPU Node and login node we used are listed in Table 1. The software environment is similar to CPU Node other than CUDA 11.4 and Cupy 8.3.0. We experimented the DaCe_gpu only on 30 benchmarks, the same as that of paper[2].

We record every benchmark's runtime of DaCe_gpu framework and paint them on Figure 2, in analogy to Figure 8 from the original DaCe paper[2]. It's worth nothing that the results on benchmark deriche, adi and seidel2d are worse than

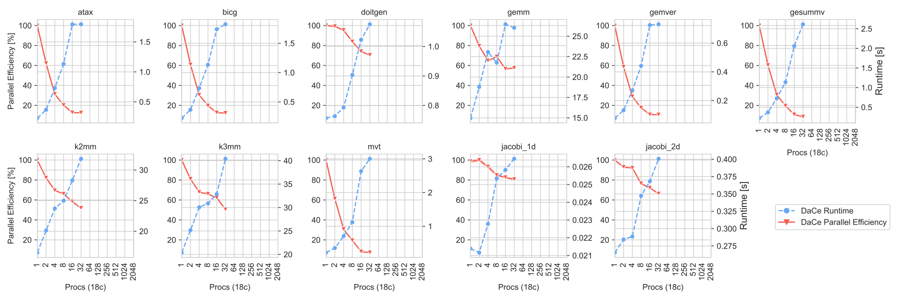

Figure 3: Distributed results

that from paper[2]. We find that their function name are all "kernel". So we infer the reason is that they need more kernel call.

6. Distributed Results

The DaCe paper performs scaling experiments with up to 23328 cores. However, due to the hardware limit of the Azure cluster, we choose HB120_rs v2 VM to ensure we can run on more cores. We choose process numbers 1,2,4,8,16,32.

Considering the performance, we configured the parameters listed in Section 3. However, due to limitation of available VM, we finally modify the parameters to 16 processes per node. As for the MPI, we selected MPICH to be as consistent as possible with the paper.

We paint the result for the previous parameter on Figure 3, only with DaCe. As shown in Figure 3, the curve of parallel efficiency shows that Dace works fine for distributed computation.

7. Secret task

In the secret task, three different benchmarks are given. Our task is to modify the DaCe version of the benchmarks, and set them applicable to DaCe Python. After carefully inspecting the three benchmarks, we describe the problems and the corresponding solutions in the following categories:

1. **Explicit Type Definition.** DaCe requires us to define the parameter type of the functions explicitly, but in all the three benchmarks provided, the parameters in the function are not decorated. Therefore, we add the *.dace* types to the functions and solve the problem.
2. **Nested function.** In the wdist benchmark, there's a nested function. Thus, we split the nested function to three individual parts and successfully solve the problem. In further experiments, we find out that the power operation can not be nested with the summation function.
3. **Method Adaptation.** In the lstsqr benchmark, the original average function does not fit the DaCe decorated type. Thus, we decide to change the average function. At the first time, we use a hand-written loop to calculate the average, but DaCe does not parallelize the loop and the run time is very long. Then, we use a sum function and successfully solve the problem.

8. Conclusion

As discussed above, DaCe Python can enhance the performance of Python codes and achieve tremendous speedups over the previous best solutions. Compared to DaCe, in CPU experiments, we achieve better speedups on a minority of test cases and worse on most of test cases. In GPU experiments, we achieve a shorter run time on most test cases. In distributed experiments, we use two Hybrid process/thread policies and find a different scaling result of the DaCe.

Despite our results, we have faced many issues and learned many lessons in our reproducing p rocedure.

For example, when we tried to resolve the dependencies of DaCe, we met many problems with the package versions. In order to compare the results of different versions of DaCe, we have run the newest version of DaCe and the artifact version of DaCe. However, the artifact version of DaCe had not been updated for over a year, so the package versions need to be updated. This issue shows that we should specify the pip package versions in our installation scripts.

For the DaCe project itself, we still found some points for improvement.

1. DaCe provide Python annotations for users to effectively accelerate Python code, while this method can inevitably reduce the programming productivity. In the future, users can avoid explicit type decorations with the SSA analysis being integrated into DaCe.
2. The compilation time is longer than expected to complete under some scenarios. In DaCe, the code goes through two steps of front-end: the first is to generate the SDFG IR, and the second is to use backend compilers to generate the IR. Although AoT compilation mitigate the cost of JIT in HPC applications, the compilation time started from a cold state should be paid attention. Even a short Python code will take a long time to compile in our practical uses. Even in the examples given in the article, we still find the example mandelbrot2 compilation timeout.
3. The performance evaluation is well performed, while it can be improved by comparing programming productivity. DaCe Python has achieved better performance than the commly used NumPy module. However, the DaCe Python has a lot of annotations to help enhance the performance, while the traditional C compilers require the compiler to optimize the code without any additional information. Therefore, this difference in semantics could bring improper comparison results.
4. DaCe will compile the Python code to a library node, like compiling a native library. This compiling procedure can be embedded information about function parameters and semantics, so as to conduct optimizations like constant propagation, dead code elimination and branch prediction. Also, this work can be further improved by considering the efficiency of inter-operations between functionalities.

References

[1] In: URL: `https://spclgitlab.ethz.ch/tim0s/ddace-lite-sc21/-/blob/master/README.md`.

[2] Tal Ben-Nun et al. "Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. 2019.

[3] T. Gamblin et al. "The Spack package manager: bringing order to HPC software chaos". In: *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2015, pp. 1–12. DOI: `10.1145/2807591.2807623`. URL: `https://doi.ieeecomputersociety.org/10.1145/2807591.2807623`.

[4] Alexandros Nikolaos Ziogas et al. "NPBench: A Benchmarking Suite for High-Performance NumPy". In: *Proceedings of the ACM International Conference on Supercomputing*. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021. DOI: `10.1145/3447818.3460360`. URL: `https://doi.org/10.1145/3447818.3460360`.

[5] Alexandros Nikolaos Ziogas et al. "Productivity, Portability, Performance: Data-Centric Python". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: `10.1145/3458817.3476176`. URL: `https://doi.org/10.1145/3458817.3476176`.