Reproducing Performance of Data-Centric Python by SCC Team from National Tsing Hua University

Fu-Chiang Chang *, En-Ming Huang *, Pin-Yi Kuo, Chan-Yu Mou, Hsu-Tzu Ting, Pang-Ning Wu, Jerry Chou †,
Department of Computer Science, National Tsing Hua Univeristy, Taiwan
* contributed equally to this work as first authors
† corresponding author

***Abstract*—As part of the Student Cluster Competition at the SC22 conference, this work aims to reproduce the performance evaluations of the Data Centric (DaCe) Python framework by leveraging Intel MKL and NVIDIA CUDA interface. The evaluations are conducted on a single CPU-based node, NVIDIA A100 GPUs, and a eight-node cloud supercomputer. Our experimental results successfully reproduce the performance evaluations on our cluster. Additionally, we provide insightful analysis and propose effective methods for achieving higher performance when utilizing DaCe as an acceleration library.**

***Index Terms*—Student Cluster Competition, Parallel Computing, Reproducibility**

I. Introduction

PYthon has become one of the most widely used languages [1] for scientific code. Its popularity is attributed to the availability of numerous frameworks based on NumPy [2], such as SciPy [3], scikit-learn [4], and pandas [5], which make writing Python code easier. These frameworks are extensively employed by developers and users in scientific simulation [6], [7] and machine learning [8], [9] domains. However, existing Python libraries for high-performance computing (HPC) lack simultaneous support for the three Ps [1], [10]: Productivity, Portability, and Performance.

To bridge this gap, the Data-Centric (DaCe) Python framework [1] is proposed. DaCe offers several advantages. Firstly, it allows users to integrate it into existing programs using annotating statements, enhancing productivity. Secondly, DaCe provides compatibility with various computing backends, including CPUs, GPUs, FPGAs, and distributed computing, ensuring portability. Lastly, DaCe demonstrates significant performance improvements compared to state-of-the-art libraries. DaCe identifies data parallelism and data reusing patterns in dataflows, optimizing them and converting Python functions to C code. The resulting code can be compiled to run efficiently on CPUs, GPUs, or FPGAs.

In this work, as part of the Student Cluster Competition (SCC) at the SC22 conference, we aim to reproduce the performance evaluations of DaCe on CPUs, GPUs, and multi-node supercomputers. Our cluster includes state-of-the-art CPUs and GPUs. Despite running on different architectures (Intel Xeon 6130 CPU and NVIDIA V100 GPU in [1]), our results successfully reproduce the evaluation work presented by the authors [1]. The contributions of this paper are as follows: ① We reproduced the CPU experiments on NumPy and DaCe. ② We reproduced the DaCe GPU experiments against NumPy. ③ We reproduced the distributed experiments with the scale of at most eight nodes. ④ We provide a comprehensive analysis of the benchmarks to explain how DaCe achieves performance enhancements.

The remainder of the paper is organized as follows: Section II presents the hardware and software configurations used in the experiments. Section III describes the experimental procedures. Sections IV, V, and VI evaluate the experiments. Finally, Section VII concludes the paper.

TABLE I
THE HARDWARE CONFIGURATION OF OUR TESTBED

Type	Our GPU cluster	Azure Cloud HC
Number of nodes	2	8
CPU model	AMC EPYC 7763	Intel Xeon Platinum 8168
CPU architecture	Milan	Skylake
CPU frequency	2.45GHz	3.4 GHz
Cores per CPU	64	22
Number of CPU per node	2	2
L1d/i cache per core	32 KiB	32 KiB
L2 cache per core	512KiB	1 MiB
L3 cache	256 MiB	33 MiB
GPU model	NVIDIA A100*4	-
Interconnection bandwidth	200 Gb/s	100 Gb/s
Operating system environment	Bare-metal	Virtual machine
Experiments performed	GPU	CPU and distributed

II. Experimental Setup

A. Hardware configuration

Our experimental environment consists of two GPU nodes and a eight-node Azure cluster (HB44rs). Table I summarizes the hardware configurations used in our experiments.

The GPU cluster comprises two QuantaGrid D43N-3U nodes. Each node is equipped with two AMD EPYC "Milan" 7763 64-core processors, with hyperthreading enabled, resulting in a total of 128 cores per node. Each node has 512GB of memory and 3.84TB of NVMe storage. The cluster is equipped with two network connections: an InfiniBand capable

Fu-Chiang Chang and En-Ming Huang contributed equally to this work as first authors. They designed the study methodology, performed the experiments, analyzed the results, and drafted the paper. Pin-Yi Kuo, Chan-Yu Mou, Hsu-Tzu Ting and Pang-Ning Wu are the team members helped to review the text and provided comments for improving the articles. Jerry Chou is the corresponding author and team advisor who guided the team to compete this study and writing. All authors approve of the content of the manuscript and agree to be held accountable for the work.

TABLE II
THE SOFTWARE STACK CONFIGURATION OF OUR ENVIRONMENTS

OS & Libraries	Version
Operating system	Ubuntu 20.04
C Compiler & MPI	Intel oneAPI 2022.3.0
Intel ICC Compiler	20220726
GPU Compiler	CUDA 11.3 & gcc 9.4
DaCe	Commit with SHA 82314d8
NPBench	Commit with SHA 6fcd180

of transmitting at 200Gbps and a ConnectX-6 Ethernet for system control utilities.

The Azure cluster consists of up to 10 HB44rs instances. Each instance provides two Intel Xeon Platinum 8168 CPUs and has 352GB of memory. It should be noted that although the Intel CPU consists of 24 cores in a socket, the Azure platform only provide 22 virtual CPU cores. The cluster is connected using Mellanox EDR InfiniBand with a speed of 100Gbps in a fat tree topology.

B. Software Configuration

Table II lists the software installed on both the local cluster and cloud instances. We used the DaCe version released on September 26, 2022, with the commit SHA `82314d8` from the GitHub repository. The benchmarks we selected are downloaded from the NPBench GitHub repository. In our experiments, we employed the Intel oneAPI Math Kernel Library (MKL) for improved performance with NumPy, as also mentioned in the original work [1].

III. Description of Experimental Run

The benchmark applications used in our experiments were downloaded from the NPBench repository on GitHub. We evaluated all benchmarks using the "paper" dataset. For certain benchmarks (e.g., `syrk`), we made modifications to improve performance by annotating loops with `dace.map` to eliminate data dependencies across iterations. Please note that due to the potential bugs within DaCe, some benchmarks could not be compiled with NVCC when using the GPU as the backend.

To achieve better performance, we switched the C compiler from GCC to Intel ICC (included in oneAPI). Additionally, we analyzed the performance of AMD CPUs on our GPU node. However, the Intel CPUs on Azure consistently outperformed the AMD CPUs for the CPU benchmarks. Despite leveraging the AMD compiler (AOCC) and optimizing libraries (AOCL), the Intel CPUs demonstrated superior performance. Therefore, we only conducted GPU experiments on our GPU cluster.

The methods used to employ DaCe, as presented in NPBench, are as follows:

1) Variables representing loop sizes (e.g., `dace.map`) and arrays should be converted to DaCe symbols using `dace.symbol`.
2) Functions optimized by DaCe should be decorated with `@dace.program`.
3) To parallelize loops in an OpenMP-style manner, the Python `range()` function should be replaced with the `dace.map` construct.

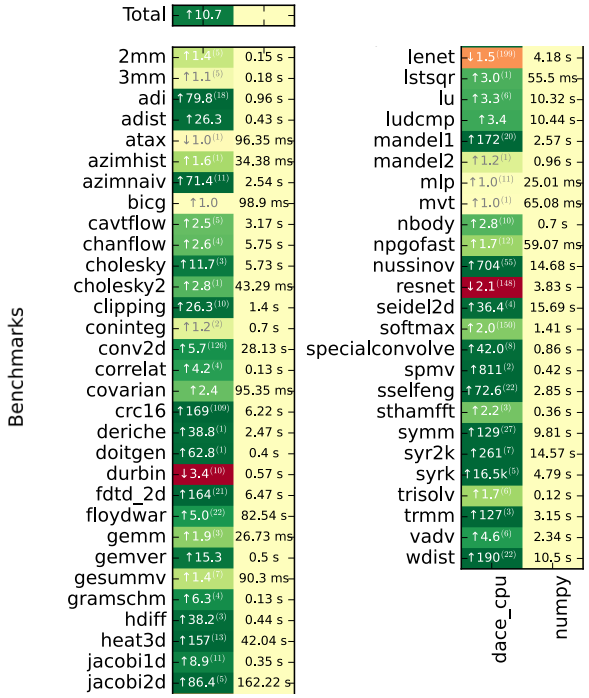

Fig. 1. Comparison of DaCe CPU Runtime to NumPy on CPU, showcasing speedup values for DaCe compared to NumPy. The first column number represent the speedup or slowdown factor of the DaCe run compared to the Numpy run. Performance improvements are highlighted in green and uparrows ↑, while performance degradation is indicated in red (or orange) and downarrows ↓ for each benchmark.

IV. CPU Results

We successfully reproduced a total of 56 CPU benchmarks on our Azure cluster, and the results are visualized in Figure 1. In general, DaCe's performance closely matches that reported in the original work [1]. Notably, in specific cases such as `heat3d`, `mandel1`, and `spmv`, DaCe demonstrates a remarkable performance improvement, achieving speedups of up to 100 times compared to NumPy. On the other hand, benchmarks like `atax`, `bicg`, and `durbin` do not exhibit such substantial enhancements.

Furthermore, this work includes a few additional benchmarks which were not included in [1], such as `lstsqr`, `specialconvolve`, and `wdist`. While the performance of these benchmarks is undocumented in the original work, our experimental results demonstrate that DaCe can indeed deliver performance improvements for these newly introduced benchmarks.

Benchmarks that experience significant speedup in DaCe typically involve operations performed on arrays, allowing DaCe to optimize them by leveraging OpenMP for parallel execution. Furthermore, applying the Stateful Dataflow multiGraphs [11] model to the calculations enables further optimization of the data flow, resulting in improved performance.

On the other hand, benchmarks such as `lenet`, `resnet`, and `durbin` are unable to benefit from the aforementioned optimizations for higher performance. The performance degradation observed in `lenet` and `resnet` is due to the representation of convolutions. The implementation of convolutions in

DaCe is translated into multiple atomic operations, resulting in decreased performance compared to other benchmarks. Such observations are also mentioned in the original work [1]. Regarding the `durbin` benchmark, the implementation utilizes `dace.map` to explicitly parallelize the `flip()` computation. Nontheless, in this specific case, the problem size is not sufficient to offset the overhead of parallelization. By explicitly configuring `OMP_NUM_THREADS` to 8, reducing the parallelism, DaCe can achieve a performance improvement of 2x compared to NumPy.

In the DaCe framework, parametric parallelism is supported to enable Python loops to be executed in parallel. There are two ways to take advantage of this feature. First, DaCe provides the LoopToMap transformation, which detects for-loops in the intermediate representation (IR) and uses symbolic affine expression analysis to validate the safety of parallel execution of iterations. Second, DaCe also offers explicit parallelism declaration through map scopes (`dace.map`), allowing programmers to identify data dependencies based on their knowledge.

Our reproduced results demonstrate that by replacing the Python built-in `range` iterator with `dace.map` in the `syrk` benchmark, the execution time can be significantly optimized, achieving a speedup of 16.5k times compared to NumPy and 30x faster than the original DaCe benchmark. This enhancement also extends to GPU performance. These findings highlight the importance of a programmer's effort in achieving optimal performance. If a programmer can accurately identify data dependencies, the benefits will be much greater than those achieved through the automatic LoopToMap transformation.

V. GPU Results

In this section, we present the evaluation results of 44 benchmarks conducted successfully on our GPU node. Figure 2 illustrates the performance comparison between DaCe and NumPy running on a single-node CPU. Notably, our experimental environment employs a more recent version of DaCe compared to the one referenced in the original work [1], enabling the execution of additional benchmarks on the GPU platform. The majority of our findings align with those reported in the original work [1]. For instance, benchmarks like `cholesky` and `ludcmp` exhibit slower performance on the GPU, while `clipping` and `heat3d` continue to demonstrate significant speedups of over 100x compared to NumPy. On the other hand, it is also important to note that the additional atomic operations introduced in `resnet` cause DaCe to perform more slowly on the GPU compared to NumPy. This issue is consistent with our observations in Section IV and is also documented in the original work [1].

In addition to observing similar results, we selected several benchmarks that exhibited either speedup or performance degradation to provide further analysis. Specifically, we analyze `cholesky`, `ludcmp`, `adist`, and `cholesky2`. The first two benchmarks demonstrate slower performance on the GPU, while the latter two show improved performance.

The reason behind the poorer performance of `cholesky` and `ludcmp` on the GPU is the communication and synchronization time between the CPU and GPU, which acts as a bottleneck. These benchmarks involve numerous scalar operations nested inside loops that cannot be parallelized. Since each scalar operation must be executed by a GPU kernel, significant communication overhead is introduced. Additionally, the problem size of each GPU kernel function is too small to effectively hide the communication overhead through computation.

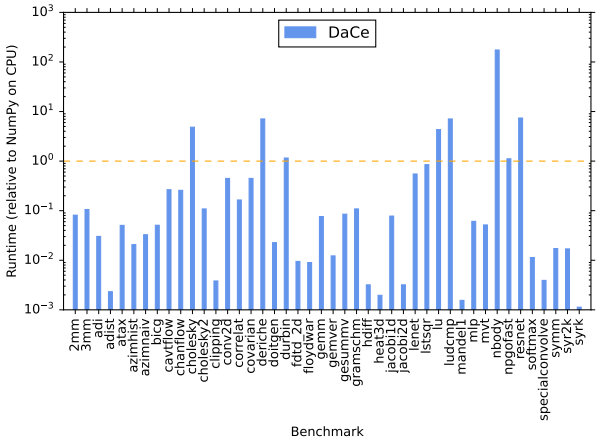

Fig. 2. DaCe GPU runtime normalized to NumPy on CPU (lower is better).

The benchmark `cholesky2` implements the same algorithm as `cholesky`, but it invokes the NumPy linear algebra function. DaCe recognizes this and ports it to the GPU by utilizing cuBLAS [12], the CUDA implementation of the Basic Linear Algebra Subprograms (BLAS) library. This optimization significantly improves the performance on the GPU.

Regarding the `adist` benchmark, we discovered that NumPy is unable to leverage Intel MKL and parallel processing for mathematical functions on arrays (e.g., `np.sin()`) [13]. However, DaCe is capable of exploiting data parallelism and dispatching computations on the GPU, leading to improved performance in this scenario.

VI. Distributed Results

We evaluated the scalability of DaCe on our Azure cluster due to smaller scale of the physical cluster used in the SCC. We choose the "hb44rs" instance type, which is equipped with the Intel CPUs. Despite AMD CPUs' prowess in highly scalable workloads, DaCe relies on Intel MKL-ScaLAPACK for distributed computations, and some ScaLAPACK optimizations are tailored to Intel-specific structures, which resulted in reduced performance on AMD CPUs.

Figure 3 illustrates our experimental results. Overall, our findings align with the original work. For instance, benchmarks that require minimal communication, such as `doitgen`, exhibit nearly perfect efficiency. However, we observed earlier degradation of parallel efficiency and significantly increased runtime variability in benchmarks with higher communication overhead. Notably, kernels like `atax`, `bicg`, `gemver`, `gesummv`, and `mvt`, responsible for matrix-vector products, exhibit optimal efficiency with 1 to 2 nodes but experience a rapid drop in efficiency when the scale exceeds 2 nodes.

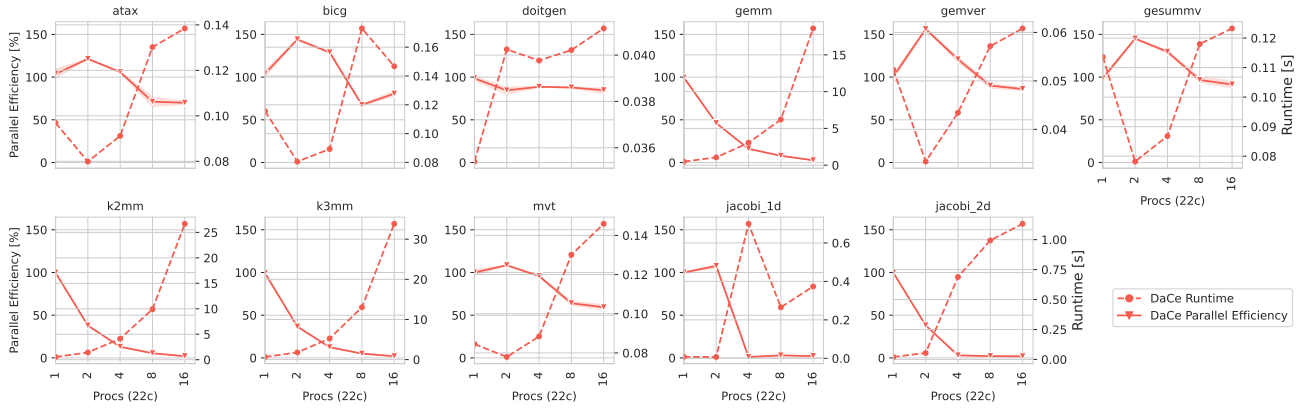

Fig. 3. Distributed performance of DaCe on the Azure Cycle Cloud. The X-axis represents the number of processes, where each process is assigned to a single CPU socket (22 CPU cores). The dashed lines represent the runtime, while the solid lines represent the scaling efficiency.

This performance variation is influenced by both CPU and interconnect differences. The Piz Daint supercomputer used by the authors employs the 'Aries' interconnect from Cray with a dragonfly network topology, while Azure instances use a fat-tree style. Additionally, our Azure environment runs within virtual machines, and the communication libraries within these virtual machines aren't as optimized as those on dedicated supercomputers. These factors contribute to the increased runtime variation compared to the original paper [1].

For the matrix-matrix product kernels, namely `gemm`, `k2mm`, and `k3mm`, the scalability is not ideal, which is an expected behavior of MKL-ScaLAPACK [14]. Such issues are also reported by the authors [1].

We reproduced the distributed experiments on our Azure cluster, providing insights into the scalability of DaCe. Despite observing efficiency degradation in certain scenarios, our results corroborate the original work and shed light on the behavior and limitations of MKL-ScaLAPACK in distributed computations.

VII. Conclusion

In this work, our reproducibility experiments on a single CPU node, a single GPU, and a multi-node distributed setup confirm the effectiveness of DaCe in enhancing application performance. We observed consistent improvements compared to the numpy library across various benchmarks. Additionally, we provided valuable insights into optimizing DaCe's performance.

By analyzing the experiments, we discovered important factors that impact DaCe's performance. Firstly, we found that excessive use of OpenMP threads for atomic operations in CPU benchmarks can lead to significant performance degradation. By determining the optimal number of OpenMP threads, we were able to mitigate this issue and improve overall performance. Secondly, we emphasized the critical role of identifying data dependencies. By annotating critical loops, we observed substantial speedup in certain benchmarks. Finally, in GPU computations, we highlighted the potential bottleneck of communication and synchronization between the CPU and GPU, especially in cases where benchmarks are not highly parallel or involve numerous scalar operations.

It is worth noting that the artifacts provided in [1] make it possible for others to conduct experiments on different architectures. However, one ongoing challenge in employing DaCe is deploying applications onto GPUs, as some benchmarks couldn't be successfully executed on the GPU. Further development to fully support GPUs would advance the maturity of the DaCe framework as an HPC library.

Overall, our reproducibility efforts not only validated the findings of the original work but also provided additional insights for optimizing DaCe's performance. These findings are valuable for researchers and practitioners seeking to leverage DaCe for high-performance computing, enabling them to make informed decisions and achieve better performance in their applications.

References

[1] A. N. Ziogas, T. Schneider, T. Ben-Nun, A. Calotoiu, T. De Matteis, J. de Fine Licht, L. Lavarini, and T. Hoefler, "Productivity, portability, performance: Data-Centric python," in *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2021.

[2] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.

[3] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[5] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proc. of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56 – 61.

[6] R. Gowers, M. Linke, J. Barnoud, T. Reddy, M. Melo, S. Seyler, J. Domański, D. Dotson, S. Buchoux, I. Kenney, and O. Beckstein, "MDAnalysis: A python package for the rapid analysis of molecular dynamics simulations," 01 2016, pp. 98–105.

[7] A. N. Ziogas, T. Ben-Nun, G. I. Fernández, T. Schneider, M. Luisier, and T. Hoefler, "A Data-Centric approach to Extreme-Scale ab initio dissipative quantum transport simulations," in *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2019.
[8] "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, accessed: 10 Jul. 2023. [Online]. Available: https://www.tensorflow.org/
[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, High-Performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.
[10] T. Ben-Nun, T. Gamblin, D. S. Hollman, H. Krishnan, and C. J. Newburn, "Workflows are the new applications: Challenges in performance, portability, and productivity," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 57–69.
[11] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: Association for Computing Machinery, 2019.
[12] NVIDIA, "cuBLAS documentation," accessed: 10 Jul. 2023. [Online]. Available: https://docs.nvidia.com/cuda/cublas/
[13] SciPy, "Parallel programming with numpy," accessed: 10 Jul. 2023. [Online]. Available: https://scipy-cookbook.readthedocs.io/items/ParallelProgramming.html
[14] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, "Red-Blue pebbling revisited: Near optimal parallel Matrix-Matrix multiplication," in *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: Association for Computing Machinery, 2019.

Fu-Chiang Chang obtained his B.S. degree in Computer Science from National Tsing Hua University (NTHU) in 2023. He is currently pursuing a Master's degree at NTHU. His research is centered around high-performance Large Language Model (LLM) training and inference, with a focus on harnessing hardware resources to optimize LLM performance. Fu-Chiang Chang became a contributor to the DaCe project after participating in the reproducibility challenge of SCC22.

En-Ming Huang is pursuing his B.S. degree in Computer Science at National Tsing Hua University (NTHU). Mr. Huang's current research focuses on GPU algorithms, performance, and hardware optimizations. Mr. Huang is also involved in high-performance computing events and has won the overall title in the SC22 student cluster competition.

Pin-Yi Kuo is pursuing his B.S. degree in Computer Science at National Tsing Hua University (NTHU). Mr. Kuo possesses extensive experience in cluster systems and networking management, and his research currently focuses on developing a novel framework for efficient and scalable GPU resource management in Kubernetes clusters.

Chan-Yu Mou is a Master's student in Computer Science at National Tsing Hua University, specializing in performance analysis and optimization. He excelled as a server manager in the SC22 Student Cluster Competition and as an application optimization specialist in the ASC20-21 Student Supercomputer Challenge, winning championship titles in both.

Hsu-Tzu Ting is pursuing her Master's degree in Computer Science at National Tsing Hua University (NTHU). Her research interests revolve around high-performance computing and disaggregated composable systems, with a particular focus on optimizing and leveraging Kubernetes for enhanced system performance.

Lawrence Wu is currently pursuing a B.S.'s degree in Computer Science at National Tsing Hua University, with a projected graduation in June 2024. Passionate about HPC, Mr. Wu has excelled in international computing competitions and is keenly interested in Machine Learning, Generative AI, and Computer Vision. He also embraces open-source technologies, with a particular fondness for Arch Linux and Proxmox VE.

Jerry Chou received Ph.D. degree from the Department of Computer Science and Engineering at University of San Diego (UCSD) in 2009. Dr. Chou joined the Department of Computer Science at National Tsing Hua University in 2011 as an Assistant Professor, and was promoted to Professor in 2022. Dr. Chou's research interests are in the broad area of distributed systems including high performance computing, cloud/edge computing, big data, storage systems, and resource or data management.