Automatic Code Decomposition, Parallelization, and Distribution on Heterogeneous Systems. A Case Study on Training a Neural Network for Image Classification.

Salvatore D'Angelo[1,*], Beniamino Di Martino[1,3,4], Pasquale Vassallo[1], Vito Alessandro Liccardo[1], Antonio Esposito[1], Andrea Carollo[2], Giacomo Corridori[2], Gianmarco Spinatelli[2] and Francesco Polzella[2]

[1]*University of Campania Luigi Vanvitelli, Via Roma 29, 81031 Aversa (CE), Italy*
[2]*Zerodivision Systems S.r.l., Piazza S. Francesco, 1 - 56127 Pisa (PI), Italy*
[3]*Department of Computer Science and Information Engineering, Asia University, Taiwan*
[4]*Department of Computer Science, University of Vienna, Austria*

Abstract

Modern software architectures are increasingly complex, often distributed across multiple computational and storage nodes. This complexity demands careful attention during both the design and implementation phases. While traditional compilers once automated certain parallelization tasks in simpler environments, today's heterogeneous infrastructures call for new and more sophisticated approaches. We introduce a novel system for automatically decomposing, parallelizing, and distributing Python code across heterogeneous systems. Our approach combines a skeleton-based compiler with lightweight decorators to annotate computational patterns, enabling automated translation into parallel workflows. This system is embedded within a custom Jupyter kernel and frontend, allowing interactive development and execution. The backend supports diverse environments, including Docker, Kubernetes, and Slurm-based HPC clusters. We demonstrate the effectiveness of our method by training a convolutional neural network for image classification, achieving near-linear speedup across multiple GPUs and nodes. Our results highlight the potential of this approach to democratize scalable computing for non-expert developers.

Keywords

HPC, Distributed Systems, Code Decomposition, Parallelization, Neural Networks, Image Classification, Heterogeneous Systems

1. Introduction

The growing complexity of modern software systems, often distributed across multiple computing and storage nodes, presents substantial challenges during both the design and implementation phases. These systems span heterogeneous architectures, including multi-core CPUs, many-core GPUs, and geographically distributed environments orchestrated via container-based platforms such as Docker and Kubernetes, or batch-scheduled clusters using systems like Slurm. Within this context, optimizing code for performance while preserving developer productivity is an increasingly critical concern.

Historically, compilers have played a central role in automating aspects of code transformation and parallelization. However, such tools were primarily designed for homogeneous, shared-memory environments and focused on local optimization techniques. As computing infrastructures evolve

ITADATA-WS 2025: The 4th Italian Conference on Big Data and Data Science – Workshops, September 9–11, 2025, Turin, Italy
*Corresponding author.
†These authors contributed equally.
salvatore.dangelo@unicampania.it (S. D'Angelo); beniamino.dimartino@unicampania.it (B. D. Martino); pasquale.vassallo@unicampania.it (P. Vassallo); vitoalessandro.liccardo@unicampania.it (V. A. Liccardo); antonio.esposito@unicampania.it (A. Esposito); a.carollo@zerodivision.it (A. Carollo); g.corridori@zerodivision.it (G. Corridori); g.spinatelli@zerodivision.it (G. Spinatelli); f.polzella@zerodivision.it (F. Polzella)
https://saldang.github.io/ (S. D'Angelo)
0000-0001-7185-3957 (S. D'Angelo); 0000-0001-7613-1312 (B. D. Martino)

toward highly heterogeneous and hybrid configurations that combine cloud resources, HPC clusters, and edge systems, these traditional compiler strategies are no longer sufficient.

Despite the availability of powerful infrastructure, developers and researchers face several persistent barriers when attempting to harness distributed and high-performance computing (HPC) resources. Writing parallel or distributed code remains a largely manual and error-prone process, requiring expertise in low-level synchronization, resource scheduling, and architecture-specific tuning. The learning curve for widely-used technologies such as MPI, Docker, Slurm, and Kubernetes is steep, and often misaligned with the priorities of domain experts in fields like data science, AI, or computational biology.

Moreover, development and execution environments frequently diverge, leading to issues with code portability, reproducibility, and deployment consistency. Code that runs locally may require extensive reconfiguration to operate correctly on a remote cluster or cloud platform, introducing friction into the software lifecycle and slowing down iterative development.

To address these challenges, we propose a novel system for the automatic decomposition, parallelization, and distribution of Python code targeting heterogeneous execution environments. At the core of our system is a skeleton-based compiler that uses lightweight Python decorators to annotate segments of code. These annotations encode high-level computational patterns that the compiler interprets to automatically construct an execution graph. This graph is then mapped to a backend-aware scheduler, which dispatches computation across available resources.

The system is tightly integrated into a custom Jupyter kernel and frontend, allowing developers to remain within a familiar, interactive notebook environment while transparently targeting a wide range of execution platforms—including Docker containers, Kubernetes clusters, and Slurm-managed HPC infrastructures. The use of decorators and cell-level metadata enables a clear separation between domain logic and orchestration semantics, reducing the need for users to manage low-level system details.

We evaluate our approach through a case study involving the distributed training of a convolutional neural network on the Fashion-MNIST dataset. Training is performed within Docker-based containers leveraging GPU acceleration and TensorFlow's distribution strategy. Results show that our system achieves near-linear speedup across multiple GPUs and compute nodes, while maintaining usability and minimal overhead for developers.

Unlike prior tools such as Dask, Ray, or Jupyter-Workflow, our system combines automatic compiler-based transformation with metadata-driven workflow construction and seamless backend abstraction. This combination enables rapid prototyping and scalable execution without requiring users to adopt new languages, workflow engines, or runtime frameworks.

By lowering the barrier to entry for scalable, distributed computing, our work aims to democratize access to HPC-level performance for a broader class of users—particularly those working in applied research and machine learning.

The remainder of this paper is organized as follows: Section 2 reviews related work in compiler-based parallelization and distributed notebook workflows. Section 3 presents the design and implementation of our system. Section 4 describes the case study and evaluates the performance of our approach. Finally, Section 5 discusses conclusions and future directions.

2. Related Work

The imperative to harness the full potential of modern computing architectures from on-premise multi-core and many-core processors to the vast, distributed resources of the cloud has positioned advanced compilers as a critical component in the software development landscape. This chapter provides a comprehensive overview of the current state of the art in compiler technology, focusing specifically on methodologies and tools available for automatic work decomposition. We will explore compilers designed not only for the traditional parallelization of sequential code on a single machine but also for the more recent challenge of decomposing applications into tasks that can be offloaded and executed

elsewhere, such as on remote cloud infrastructure.

As software and hardware systems continue to escalate in complexity, the ability of a compiler to autonomously identify and exploit concurrency, whether local or distributed, is paramount for enhancing application performance and reducing the intricate programming burden on developers.

The fundamental purpose of such a compiler is to analyse sequential source code, identify sections that can be executed concurrently, and transform them for efficient execution.

This transformation can manifest in two primary ways: as a parallelized form executed across multiple local processing units, or as discrete, independent tasks packaged for execution on remote computing resources. This automated process is crucial for unlocking the performance gains offered by both tightly coupled parallel hardware and distributed cloud environments. It aims to achieve this without necessitating that developers engage in the complexities of explicit parallel or distributed systems programming.

2.1. ROSE Compiler

A significant and distinct tool in the compiler landscape is ROSE [1], an open-source compiler infrastructure developed at Lawrence Livermore National Laboratory (LLNL). Unlike traditional compilers that translate source code into machine-executable object code, ROSE is a source-to-source framework. Its primary function is to build custom tools for program analysis and transformation. It reads source code, creates a detailed internal representation, allows tools to analyse and modify this representation, and then generates new, human-readable source code. This approach makes ROSE an exceptionally flexible platform for compiler research and the development of specialized tools.

The ROSE infrastructure is particularly well-suited for creating custom tools for static analysis, program optimization, domain-specific optimizations, and performance analysis. A notable tool included within the framework is AutoPar, an automatic parallelization compiler for C and C++ that inserts OpenMP directives into serial code.

Through several parsers it's frontend supports many languages such as C, C++, Fortran, Java, Python and many more. ROSE does not include code generation features specifically designed for the cloud, although it provides all the tools necessary for their implementation. In addition, because Rose is a C++ framework, you have maximum support by using the same language to write your own analysis tool as well.

2.2. PIPS

PIPS [2] is an open-source, source-to-source compilation framework whose development was initiated by MINES ParisTech and continued over time by several groups. It is engineered to analyse and transform numerical applications written in C and Fortran 77. The core philosophy of PIPS is to achieve effective automatic parallelization by first building a deep, "global" understanding of the entire program. This emphasis on comprehensive, inter-procedural analysis allows it to perform transformations such as loop optimizations and task parallelization.

2.3. OMNI Compiler

The Omni Compiler [3, 4] is a source-to-source compilation framework designed to transform C and Fortran programs annotated with XcalableMP and OpenACC directives into parallel code optimized for execution on high-performance computing systems. It enables the generation of code compatible with native compilers by linking against the Omni runtime library. Additionally, it supports the XcalableACC programming model, which combines directive-based parallelism with accelerator offloading for use in heterogeneous cluster environments. The project is actively developed by the Programming Environment Research Team at the RIKEN Center for Computational Science in collaboration with the HPCS Laboratory at the University of Tsukuba, Japan.

2.4. Cetus Compiler

Cetus [5] is a source-to-source compiler intended primarily for programs written in C. Its focus is automatic code parallelization by annotation with OpenMP directives to take advantage of execution on multicore systems.

2.5. Others

Additional tools such as **OpenUH** [6], and domain-specific languages like **Rascal** [7] and **TXL** [8], offer extensibility for compiler research, but their usage typically requires specialized language knowledge and is disconnected from mainstream Python-based workflows.

2.6. Cloud oriented compilers

The compilers and languages seen so far refer to shared-memory architectures and to the message-passing paradigm, so they do not support, for instance, the generation of microservices even though some offer function libraries to implement it.

In more recent times, attempts have been made to distribute the execution of Python code over multiple nodes and, at the same time, to improve performance by performing optimization steps before execution [9]. Generally, two approaches have been followed: the first is to organize the computation as a workflow [10] and point out its dependencies, so that an eventual Workflow Management System can organize the workload [11]; the second approach is to compile a subset of Python instructions, collected in a kernel function, into a lower-level language but with higher performance [12], or to simply optimize the code without changing languages [9].

Approaches involving the translation of a subset of instructions are frowned upon, as they require learning a new variant of the language, thus reducing productivity in the development of software solutions.

There are tools that provide a more user-friendly interface by integrating the compiler functionalities in the integrated development environment such as jupyter-workflow [13], on which our work is based, that provides the compiler as a jupyter kernel so that it can be accessed without leaving the current environment and, to a certain extent, the user is unaware of the compiler. Jupyter-workflow is in turn based on Streamflow.

The StreamFlow framework [14], developed and maintained by the Alpha research group at the University of Turin (UniTO), is a container-native Workflow Management System written in Python 3 that relies on the Common Workflow Language (CWL) standard. StreamFlow is designed around two principles: executing tasks in multi-container environments to support concurrent, communicating tasks and relaxing the requirement for a single, shared data space to enable hybrid executions on multi-cloud or hybrid cloud/HPC infrastructures.

Based on this foundation, the same research group has developed the Jupyter Workflow extension for the IPython kernel, which is designed to support distributed literate workflows directly within Jupyter Notebooks. The Jupyter Workflow kernel facilitates the description of intricate workflows and their distributed execution on hybrid cloud/HPC infrastructures. In this paradigm, code cells are regarded as the nodes of a distributed workflow graph, and cell metadata are used to express data dependencies, control flow and parallel execution patterns, such as scatter/gather, as well as target execution environments. This reliance on cell metadata offers several key advantages. Firstly, it maintains a clear separation between the host logic and coordination semantics, which improves the readability and maintainability of complex applications. Furthermore, it avoids technology lock-in, as the same metadata format can be interpreted by different Jupyter kernels to support various languages, execution architectures or commercial software stacks. This approach also eases the transition for users who are already familiar with Jupyter Notebooks, as they can scale their experiments without having to learn a completely new framework. Jupyter Workflow leverages the capabilities of the StreamFlow Workflow Management System for its underlying runtime support.

As shown in Table 1, the comparison of various tools and frameworks highlights differences in language support, parallelization strategies, and cloud integration capabilities.

Tool/Framework	Language Support	Parallelization Strategy	Execution Model	Cloud / HPC Support	Notebook Integration	Python Workflow Support
ROSE	C, C++, Fortran, Java, Python	Source-to-source, OpenMP	Local shared memory	No (cloud not natively supported)	No	Partial (Python not primary target)
PIPS	C, Fortran 77	Interprocedural, loop/task	Local shared memory	No	No	No
OMNI	C, Fortran	Directive-based (XcalableMP, OpenACC, XcalableACC)	HPC cluster + GPU offload	Yes (HPC support only)	No	No
Cetus	C	OpenMP annotations	Multicore systems	No	No	No
OpenUH	Fortran, C/C++ (OpenMP)	OpenMP compiler optimization	Local or cluster	Partial (HPC focused)	No	No
TXL / Rascal	DSLs for source analysis	Customizable transformation	Static/source analysis	No	No	No
Shirako et al.	Python	Automatic parallelization	Distributed (HPC/Cloud)	Yes	No	Yes
Snakemake + WMS	Python-based DSL	Workflow dependencies	Batch/workflow-based	Yes (Cloud/HPC)	Partial (script integration)	Yes
Jupyter-Workflow	Python (via IPython)	Metadata-driven cell graphs	Workflow + containers	Yes (Multi-cloud + HPC)	Yes (Jupyter-native)	Yes
StreamFlow	Python + CWL	Task orchestration, containers	Distributed workflow	Yes (Multi-cloud + HPC)	Partial (CLI or API only)	Partial (not Jupyter-based)
Our System	Python	Compiler-based, decorators	Interactive + distributed	Yes (Docker, Kubernetes, Slurm)	Yes (Jupyter-native)	Yes

Table 1
Comparison of Tools and Frameworks for Parallelization and Workflow Support

3. Neural Network Training on Heterogeneous Systems Using Skeleton-Based Compilation

This section presents an in-depth case study illustrating the end-to-end process of training a convolutional neural network (CNN) for image classification on a heterogeneous distributed execution environment. The experiment showcases how our **skeleton-based compiler**—integrated into a custom Jupyter kernel—can automatically decompose Python code, identify parallelizable segments, and distribute execution transparently across multiple GPUs hosted in Docker-based environments. The goal is to validate that compiler-assisted decomposition can not only simplify the development of distributed machine learning pipelines but also achieve *near-optimal hardware utilization* without requiring users to manually manage low-level synchronization, inter-process communication, or GPU affinity.

As shown in Figure 1, the proposed workflow unifies three tightly coupled components:

1. **A custom Jupyter kernel** capable of intercepting Python code and extracting structural metadata from developer-provided decorators.
2. **A compiler back-end** that transforms annotated blocks into optimized, parallelizable execution units with explicit data-dependency graphs.

3. **A distributed runtime** operating inside Docker containers, enabling GPU-aware execution and resource orchestration across multiple devices.

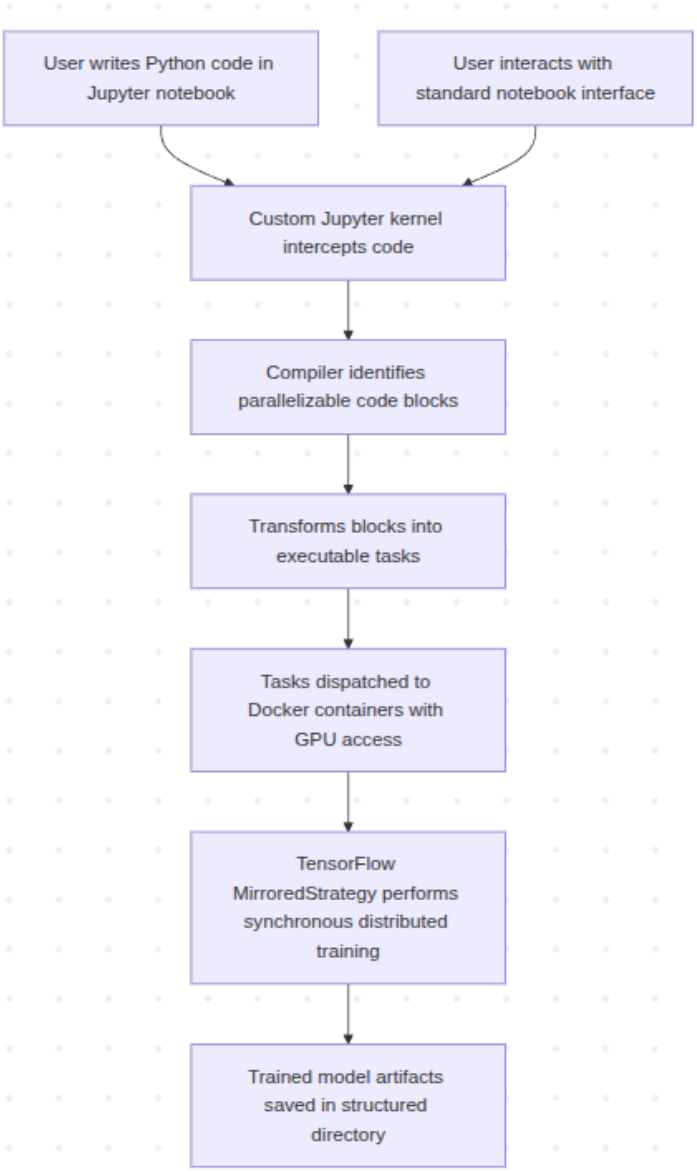

Figure 1: Distributed CNN training workflow using a custom Jupyter kernel with compiler-based task decomposition and Docker-based GPU execution.

3.1. Scenario Overview

The case study targets the **Fashion-MNIST** dataset, consisting of 70,000 grayscale images (28×28 pixels) spanning 10 fashion categories. The data is split into 60,000 training and 10,000 test samples. Preprocessing steps include:

- *Normalization* of pixel intensities to the $[0, 1]$ range.
- *Reshaping* into tensors compatible with TensorFlow's convolutional input format $(\text{batch}, \text{height}, \text{width}, \text{channels})$.
- *One-hot encoding* of labels for multi-class classification.

The training runs for **20 epochs** with a batch size tuned to saturate available GPU memory without inducing excessive communication overhead. Distribution is handled using `TensorFlow`'s `MirroredStrategy`, which replicates the model across all visible GPUs, performs synchronous gradient aggregation, and updates weights identically on each replica.

The custom Jupyter kernel captures standard Python cell code, with developers marking computationally significant functions or training loops using *lightweight decorators* (e.g., `@parallel_task`, `@gpu_task`). These annotations are translated into compiler metadata describing:

1. The task's role in the workflow (data loading, preprocessing, training step, evaluation).
2. Data dependencies and communication requirements.
3. Preferred execution targets (CPU, single GPU, multi-GPU).

3.2. Execution Strategy

The training notebook is **containerized** inside a GPU-enabled Docker image. The container:

- Exposes NVIDIA GPUs through the NVIDIA Container Toolkit (`–gpus all` flag).
- Pre-installs TensorFlow with GPU support, CUDA drivers, and NCCL for efficient multi-GPU collective operations.
- Contains the compiler runtime and backend scheduler.

The execution flow is as follows:

1. **User interaction** remains unchanged—commands are entered in a Jupyter Notebook.
2. **Kernel interception** parses decorated functions and training loops, passing them to the skeleton-based compiler.
3. The **compiler** generates a directed acyclic graph (DAG) of tasks, inserting implicit barriers and data transfers where needed.
4. The **backend scheduler** maps tasks to available GPUs, ensuring balanced workload distribution and overlapping computation with communication.
5. **TensorFlow's strategy scope** (`with strategy.scope():`) wraps model definition and training, enabling synchronous data-parallel execution.

This transparency is key: from the user's perspective, they write "normal" TensorFlow code, but behind the scenes, the compiler manages placement, communication, and execution.

3.3. Compiler-Based Optimization

The compiler performs a **three-stage optimization pipeline**:

1. Code analysis. Detects parallelizable loops and functions, identifies I/O bottlenecks, and locates potential data prefetch points. Recognizes reusable intermediate tensors to avoid redundant recomputation.

2. Skeleton mapping. Assigns each computational segment to a predefined "skeleton" pattern (e.g., *map-reduce*, *pipeline*, *data-parallel batch training*). For CNN training, the dominant skeleton is *data parallelism*: identical model replicas process distinct mini-batches.

3. Task transformation and scheduling. Generates intermediate code units with explicit device placement hints, produces an execution graph capturing data dependencies, and leverages NCCL's ring-allreduce for gradient synchronization. This minimizes inter-GPU latency and ensures efficient scaling.

The outcome is a compiled training workflow that is both *hardware-aware* and *backend-portable*, capable of running on:

- A single multi-GPU workstation.
- A Kubernetes-managed GPU cluster.
- A Slurm HPC node with Docker-in-Singularity.

3.4. Training Behavior and Observations

The CNN architecture is intentionally simple to focus on distribution behavior:

- `Conv2D` layer (32 filters, 3×3 kernel, ReLU activation).
- `MaxPooling2D` (2×2 pool size).
- `Flatten` layer to transition to dense layers.
- `Dense` layer with 128 units (ReLU).
- Output `Dense` layer with softmax activation for 10-class classification.

Figure 2 describes the complete model with all of its parameters. Key observations include:

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
flatten (Flatten)	(None, 5408)	0
dense (Dense)	(None, 128)	692,352
dense_1 (Dense)	(None, 10)	1,290

Total params: 693,962 (2.65 MB)
Trainable params: 693,962 (2.65 MB)
Non-trainable params: 0 (0.00 B)

Figure 2: CNN model for MNIST image recognition

- No manual synchronization was required—the compiler-generated execution graph and `MirroredStrategy` handled all gradient aggregation.
- GPU utilization stayed consistently above 90% on all devices, confirmed by execution logs and `nvidia-smi` profiling.
- Near-linear speedup was observed when scaling from 1 to 4 GPUs, with only minor diminishing returns due to synchronization overhead.
- Execution artifacts—trained model weights, logs, and performance metrics—were stored in a structured directory hierarchy, automatically versioned by the runtime.

Table 2 summarizes the main differences observed between sequential and distributed execution in our experiments. As expected, distributing training across four GPUs reduced the total runtime from approximately 30 minutes to just under 8 minutes, yielding a speedup factor of 3.75×. This near-linear scaling confirms that the compiler-generated execution graph and TensorFlow's `MirroredStrategy` effectively minimize idle time and communication overhead. The average GPU utilization remained consistently above 92%, indicating that the scheduler maintained a balanced workload across devices. A small overhead (6–8%) was measured for gradient synchronization via NCCL's ring-allreduce, but this was largely offset by the increase in effective batch size and the ability to overlap communication with computation.

Perhaps most notably, the transition from sequential to distributed execution required minimal code changes, limited to the addition of lightweight decorators and kernel metadata, whereas a traditional manual parallelization approach would require substantial boilerplate for synchronization, device placement, and inter-process communication. The high reproducibility of results across runs further supports the robustness of the proposed methodology.

This experiment confirms that **compiler-level decomposition combined with distributed runtime integration in Jupyter** enables scalable training without burdening the developer with HPC-level coding skills, paving the way for democratizing access to heterogeneous computing in AI research.

Table 2
Comparison between sequential and distributed execution of the CNN training using the proposed skeleton-based compilation methodology. Measurements obtained on the Fashion-MNIST dataset over 20 epochs.

Metric	Sequential Execution	Distributed Execution (4 GPUs)
Training Time (20 epochs)	$\sim$ 1800 s	$\sim$ 480 s
Speedup Factor	1.0$\times$	3.75$\times$
Average GPU Utilization	N/A (single GPU)	92–95% per GPU
Batch Size	64	256 (effective, per GPU: 64)
Memory Usage (per GPU)	$\sim$ 2.1 GB	$\sim$ 2.3 GB
Gradient Synchronization Overhead	N/A	$\sim$ 6–8%
Code Modifications Required	Extensive manual parallelization	Minimal (decorators only)
Reproducibility Across Runs	High	High (compiler-managed synchronization)

4. Conclusion and Future Work

This work has presented a compiler-assisted approach for the distributed training of convolutional neural networks on heterogeneous systems. By integrating a skeleton-based compilation mechanism directly into a custom Jupyter kernel, we demonstrated that it is possible to automatically decompose Python code, identify parallelizable segments, and schedule execution across multiple GPUs in a containerized environment. The case study on the Fashion-MNIST dataset confirmed that the proposed system can deliver near-linear scaling while preserving a high-level, interactive development experience for end-users. Importantly, the approach eliminates the need for manual synchronization, device management, or low-level configuration, thereby lowering the barrier to entry for leveraging HPC-class resources in machine learning workflows.

The proposed methodology not only achieves efficient hardware utilization but also ensures portability across diverse execution backends, including Docker, Kubernetes, and HPC environments managed by Slurm. This capability is particularly valuable for researchers and practitioners who require reproducibility, scalability, and minimal friction in transitioning between local prototyping and large-scale deployment.

Future Work

Building on the encouraging results obtained, several directions for further research and development are envisaged:

- **Support for more complex models and workflows.** Extending the compiler patterns to cover deeper neural networks, transformer-based architectures, and multi-modal learning pipelines.
- **Dynamic resource adaptation.** Incorporating runtime feedback mechanisms to dynamically adjust task scheduling, batch sizes, and data distribution based on current load, network latency, and GPU utilization.
- **Hybrid execution strategies.** Enabling mixed CPU–GPU and multi-node training with optimized data transfer strategies, including zero-copy memory sharing and gradient compression.
- **Integration with additional frameworks.** Extending support beyond TensorFlow to PyTorch and JAX, while retaining transparent user interaction in Jupyter.
- **Fault tolerance and checkpointing.** Implementing automatic recovery from hardware or network failures through fine-grained checkpointing and workflow resumption.
- **Broader heterogeneous support.** Adapting the compiler for emerging accelerators such as TPUs, IPUs, and FPGA-based inference engines.

By pursuing these enhancements, we aim to transform the proposed system into a fully general-purpose platform for distributed computing in AI and scientific applications, capable of bridging the gap between high-performance computing and accessible, notebook-based development environments.

Acknowledgments

This work was supported by the FLUENDO project, a subgrantee of the National HPC, Big Data, and Quantum Computing Center - ICSC, under the Italian NRRP MUR program, funded by the European Union - Next Generation EU, Mission 4, Component 1, CUP J33C22001170001.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] D. Quinlan, C. Liao, The ROSE source-to-source compiler infrastructure, in: Cetus users and compiler infrastructure workshop, in conjunction with PACT, volume 2011, Citeseer, 2011, p. 1.

[2] R. Keryell, C. Ancourt, F. Coelho, B. Eatrice, C. Frann, F. Irigoin, P. Jouvelot, Pips: a workbench for building interprocedural parallelizers, compilers and optimizers technical paper (1996).

[3] H. Murai, M. Sato, M. Nakao, J. Lee, Metaprogramming framework for existing hpc languages based on the omni compiler infrastructure, in: 2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW), IEEE, 2018, pp. 250–256.

[4] Omni Compiler Project, https://omni-compiler.org/, ???? Accessed: 2025-08-07.

[5] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, S. Midkiff, CETUS: a Source-to-Source compiler infrastructure for multicores, Computer 42 (2009) 36–42. URL: https://doi.org/10.1109/mc.2009.385. doi:10.1109/mc.2009.385.

[6] C. Liao, O. Hernandez, B. Chapman, W. Chen, W. Zheng, Openuh: an optimizing, portable openmp compiler, Concurrency and Computation: Practice and Experience 19 (2007) 2317–2332. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1174. doi:https://doi.org/10.1002/cpe.1174. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1174.

[7] P. Klint, T. van der Storm, J. J. Vinju, Rascal: A domain specific language for source code analysis and manipulation, in: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society, 2009, pp. 168–177. doi:http://doi.ieeecomputersociety.org/10.1109/SCAM.2009.28.

[8] J. R. Cordy, Txl - a language for programming language tools and applications, Electron. Notes Theor. Comput. Sci. 110 (2005) 3–31.

[9] J. Shirako, A. Hayashi, S. R. Paul, A. Tumanov, V. Sarkar, Automatic parallelization of python programs for distributed heterogeneous computing, 2022. URL: https://arxiv.org/abs/2203.06233. arXiv:2203.06233.

[10] J. Köster, S. Rahmann, Snakemake—a scalable bioinformatics workflow engine, Bioinformatics 28 (2012) 2520–2522.

[11] M. Bux, U. Leser, Parallelization in scientific workflow management systems, CoRR abs/1303.7195 (2013). URL: http://arxiv.org/abs/1303.7195. arXiv:1303.7195.

[12] O. Castro, P. Bruneau, J.-S. Sottet, D. Torregrossa, Landscape of high-performance python to develop data science and machine learning applications, ACM Comput. Surv. 56 (2023). URL: https://doi.org/10.1145/3617588. doi:10.1145/3617588.

[13] I. Colonnelli, M. Aldinucci, B. Cantalupo, L. Padovani, S. Rabellino, C. Spampinato, R. Morelli, R. Di Carlo, N. Magini, C. Cavazzoni, Distributed workflows with jupyter, Future Generation Computer Systems 128 (2022) 282–298. URL: https://www.sciencedirect.com/science/article/pii/S0167739X21003976. doi:https://doi.org/10.1016/j.future.2021.10.007.

[14] I. Colonnelli, B. Cantalupo, I. Merelli, M. Aldinucci, Streamflow: Cross-breeding cloud with hpc, IEEE Transactions on Emerging Topics in Computing 9 (2021) 1723–1737. doi:10.1109/TETC.2020.3019202.