17th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures

15th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms

PARMA-DITAM 2026, January 27, 2026, Krakow, Poland

Edited by
Davide Baroffio
Paola Busia
Lev Denisov
Nitin Shukla

Editors

Davide Baroffio
Politecnico di Milano, Italy
davide.baroffio@polimi.it

Paola Busia
University of Cagliari, Italy
paola.busia@unica.it

Lev Denisov
Politecnico di Milano, Italy
lev.denisov@polimi.it

Nitin Shukla
CINECA, Casalecchio di Reno, Italy
n.shukla@cineca.it

ACM Classification 2012
Computing methodologies → Distributed algorithms; Software and its engineering → Distributed systems organizing principles; Computing methodologies; Computer systems organization → Embedded software; Software and its engineering → Compilers; Computer systems organization → Quantum computing; Software and its engineering → Just-in-time compilers; Software and its engineering → Retargetable compilers; Computing methodologies → Parallel programming languages; Computing methodologies → Simulation tools; Computer systems organization → Parallel architectures; Computer systems organization → Cloud computing; Computer systems organization → Heterogeneous (hybrid) systems; Hardware → Emerging architectures; Computer systems organization → Reconfigurable computing; Hardware → Modeling and parameter extraction; Computing methodologies → Parallel computing methodologies; Software and its engineering → Parallel programming languages; Software and its engineering → Distributed programming languages; Software and its engineering → Runtime environments; Computer systems organization → Distributed architectures

ISBN 978-3-95977-416-1

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-416-1.

Publication date
April, 2026

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists all publications of this volume in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

License
This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): https://creativecommons.org/licenses/by/4.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASIcs.PARMA-DITAM.2026.0
ISBN 978-3-95977-416-1 **ISSN 1868-8969** **https://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs is a series of high-quality conference proceedings across all fields in informatics. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Daniel Cremers (TU München, Germany)
- Barbara Hammer (Universität Bielefeld, Germany)
- Marc Langheinrich (Università della Svizzera Italiana – Lugano, Switzerland)
- Dorothea Wagner (*Editor-in-Chief*, Karlsruher Institut für Technologie, Germany)

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

17th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 15th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2026).
Editors: Davide Baroffio, Paola Busia, Lev Denisov, and Nitin Shukla
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume contains the papers selected for presentation at the 17th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 15th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2026), held in Krakow (Poland), on January 27, 2026, in conjunction with the 21st edition of the High Performance and Embedded Architecture and Compilation (HiPEAC) conference.

Each anonymously submitted paper was reviewed by at least three members of the program committee, who did not have any conflicts of interest with the authors of the paper, and was evaluated for significance, novelty, technical quality, and potential to spark discussion among attendees from the academic and industry research communities. The program committee's work was carried out electronically, and, where needed, additional reviews were solicited. The final acceptance decision of all 7 accepted works was made by consensus among the reviewers.

The workshop program consists of two sessions, entitled "Compilers and Cloud HPC Tools" and "Heterogeneous Deployment".

The first session consists of 4 papers, exploring the integration of advanced compiler techniques and orchestration tools to enhance performance and manageability in cloud and High-Performance Computing (HPC) environments. The first paper introduces an open-source tool that integrates the Asynchronous Successive Halving Algorithm (ASHA) with the SLURM Experiment Management Library (SEML) to enable scalable and fault-tolerant hyperparameter optimization on shared clusters. The second paper presents a framework that integrates the VisIVO visualization tool with cloud-based services at Cineca, enabling researchers to perform GPU-accelerated, real-time, interactive visualization of large astrophysical simulations via Jupiter notebooks. The third paper proposes an inter-procedural extension to the strength reduction compiler optimization, replacing expensive operations, like exponentiation, with incremental updates across function calls to improve the efficiency of code generated for embedded platforms. The fourth and last paper of the first session, instead, describes a library for MLIR to translate quantum dialects into machine-learning-oriented primitives, facilitating the integration of quantum accelerators into hybrid HPC workflows.

The second session consists of 3 papers, describing solutions for optimizing simulation and orchestration of heterogeneous hardware, focusing on GPGPUs and FPGAs to improve resource utilization. The first paper addresses the issue of long runtime cycle-accurate GPGPU simulators, exploiting SIMT-Core cluster execution using a thread pool to achieve significant speed-ups with minimal code modifications. The second paper introduces a framework leveraging Pareto-optimal designs shorten the gap between High-Level Synthesis (HLS) and runtime orchestration, increasing the efficiency of FPGA-as-a-service, balancing the trade-off between performance and resource management. The third and last paper of the second session provides an end-to-end framework for modeling and mapping LLM inference on Coarse-Grained Reconfigurable Architectures (CGRAs), achieving high performance per Watt through accurate micro-architectural design space exploration.

In addition to the themed sessions, the workshop included three invited talks from Paul Carpenter (Barcelona Supercomputing Center), Ivy Peng (KTH Royal Institute of Technology), and Daniele Gregori (E4). The proceedings include an extended abstract of the invited talk by Paul Carpenter, detailing recent advances in the OmpSs-2@Cluster framework, focusing on how task offloading and dynamic load balancing can mitigate

17th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 15th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2026).
Editors: Davide Baroffio, Paola Busia, Lev Denisov, and Nitin Shukla

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

performance bottlenecks in distributed-memory systems. The talk from Ivy Peng examined the current landscape of schedulers and runtimes, shedding the light on the critical needs for orchestrating data movement and power dynamics, outlining the requirements for achieving system-level gains in future accelerator-rich systems. The talk from Daniele Gregori discussed the advancements of E4 in the adoption of RISC-V in HPC via the Monte Cimone cluster, the first RISC-V ISA cluster built for co-design activities to enable its use in scientific and engineering applications.

Putting together a successful PARMA-DITAM 2026 was the result of the time and effort devoted by many individuals. We are thankful to the program committee and external reviewers for their work in evaluating several papers in a short time-frame and their dedication to the quality of the program. Moreover, we are also grateful to Sasha Uhrig (HiPEAC 2026 Workshops & Tutorials Chair) and Tomasz Kryjak (HiPEAC 2026 General Chair) for their support during workshop organization and logistics. Finally, we thank the authors of all submitted papers, as well as the workshop's attendees, for contributing to the program and the discussions surrounding it. We hope you will find this program interesting, stimulating, and inspiring for your future research.

List of Authors

Giovanni Agosta (4)
Politecnico di Milano, Italy; Consorzio Interuniversitario Nazionale per l'Informatica, Roma, Italy

Georgios Alexandris (8)
National Technical University of Athens, Greece

Umer Arshad (3)
Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Italy

Hendrik Borras (2)
Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University, Germany

Paul Carpenter (1)
Barcelona Supercomputing Center, Spain

Panagiotis Chaidos (8)
National Technical University of Athens, Greece

Anusha Chattopadhyay (2)
Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University, Germany

Henk Corporaal (8)
Eindhoven University of Technology, Netherlands

Juliette Fournis d'Albiat (1)
Barcelona Supercomputing Center, Spain

Marco De Pascale (5)
Leibniz Supercomputing Centre, München, Germany

Jorge Echavarria (5)
Leibniz Supercomputing Centre, München, Germany

Aggelos Ferikoglou (7)
Microprocessors and Digital Systems Laboratory, ECE, NTUA, Athens, Greece

Holger Fröning (2)
Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University, Germany

Manil Dev Gomony (8)
Eindhoven University of Technology, Netherlands

Mario Hernández Vera (5)
Leibniz Supercomputing Centre, München, Germany

Dionysios Kefallinos (8)
National Technical University of Athens, Greece

Bernhard Klein (2)
Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University, Germany

Sohan Lal (6)
Hamburg University of Technology, Germany

Tim Lühnen (6)
Hamburg University of Technology, Germany

Alexis Maras (8)
National Technical University of Athens, Greece

Dimosthenis Masouros (7)
Microprocessors and Digital Systems Laboratory, ECE, NTUA, Athens, Greece

Giuseppa Muscianisi (3)
CINECA, Bologna, Italy

Muhammad Nufail Farooqi (5)
Leibniz Supercomputing Centre, München, Germany

Isabel Piedrahita (1)
Barcelona Supercomputing Center, Spain

Fabio Pitari (3)
CINECA, Bologna, Italy

Jakob Sachs (6)
Hamburg University of Technology, Germany

Laura Schulz (5)
Argonne National Laboratory, Lemont, IL, USA

Martin Schulz (5)
Technical University of Munich, Germany

Eva Sciacca (3)
INAF Astrophysical Observatory of Catania, Italy

Omar Shaaban (1)
Barcelona Supercomputing Center, Spain

Dimitrios Soudris (7, 8)
Microprocessors and Digital Systems Laboratory, ECE, NTUA, Athens, Greece

Despoina Tomkou (7)
Microprocessors and Digital Systems Laboratory, ECE, NTUA, Athens, Greece

17th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 15th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2026).
Editors: Davide Baroffio, Paola Busia, Lev Denisov, and Nitin Shukla
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Nicola Tuccari (3)
Department of Mathematics and Informatics,
University of Catania, Italy

Sotirios Xydis (7, 8)
Microprocessors and Digital Systems Laboratory,
ECE, NTUA, Athens, Greece

Distributed Task Execution: Opportunities, Challenges and Lessons Learnt from OmpSs-2@Cluster

Paul Carpenter
Barcelona Supercomputing Center, Spain

Omar Shaaban
Barcelona Supercomputing Center, Spain

Juliette Fournis d'Albiat
Barcelona Supercomputing Center, Spain

Isabel Piedrahita
Barcelona Supercomputing Center, Spain

Abstract

This talk will present recent advances in extending OmpSs-2 to distributed-memory systems, highlighting three contributions and the associated challenges. OmpSs-2@Cluster employs a common address space and weak accesses to support concurrent task creation and dataflow execution across nodes. Achieving good performance and scalability on 16 to 32 nodes requires detailed performance analysis together with a set of optimizations and runtime techniques, which I will outline in the talk. Second, I will describe how task offloading, in combination with BSC's Dynamic Load Balancing (DLB), enables OmpSs-2@Cluster to mitigate load imbalance in MPI + OmpSs-2 programs with minimal application changes. Third, I will explain how the runtime can exploit the iterative structure of certain task dependency graphs to precompute communications and execute iterative regions efficiently, yielding performance and scalability comparable to state-of-the-art asynchronous MPI+X. Together, these results indicate that distributed tasking can combine productivity, adaptability, and high performance in modern HPC applications.

2012 ACM Subject Classification Computing methodologies → Parallel computing methodologies; Software and its engineering → Parallel programming languages; Software and its engineering → Distributed programming languages; Software and its engineering → Runtime environments; Computer systems organization → Distributed architectures

Keywords and phrases Task-based programming, distributed-memory clusters, programming models, runtime systems, task scheduling, data dependency management, load balancing, asynchronous communication

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2026.1

Category Invited Talk

Funding This work is funded by the Barcelona Zettascale Laboratory (REGAGE22e00058408992), backed by the Spanish Ministry for Digital Transformation and of Public Services, within the framework of the Recovery, Transformation, and Resilience Plan – funded by the European Union – NextGenerationEU. It is also supported by the Spanish State Research Agency – Ministry of Science and Innovation under contract PID2019-107255GB-C21/MCIN/AEI/10.13039/501100011033 and Ramon y Cajal fellowship RYC2018-025628-I/MCIN/AEI/10.13039/501100011033 and by "ESF Investing in your future", as well as by the Generalitat de Catalunya (2017-SGR-1414).

© Paul Carpenter, Omar Shaaban, Juliette Fournis d'Albiat, and Isabel Piedrahita;
licensed under Creative Commons License CC-BY 4.0

17th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 15th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2026).
Editors: Davide Baroffio, Paola Busia, Lev Denisov, and Nitin Shukla; Article No. 1; pp. 1:1–1:7

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Task-based programming has become a powerful abstraction for expressing parallelism and managing complexity in modern HPC, and it is increasingly accepted for node-level parallelism. Tasks were introduced in OpenMP 3.0 in 2008 and substantially strengthened in OpenMP 4.0 (2013) with explicit task dependencies, enabling dependency-driven asynchronous execution. Later OpenMP revisions added more advanced tasking capabilities, e.g. taskgroups, task reductions and detached tasks, and improved the integration with accelerators. Task-based execution is also widely used in libraries and runtimes such as Intel Threading Building Blocks (TBB) [16] and in systems including Cilk-derived frameworks, HPX, StarPU [2], PaRSEC [7, 12], Legion [6] and OmpSs [10].

Task-based approaches have likewise proven successful in workflow systems such as COMPSs [13] and Pegasus [9], where tasks naturally correspond to coarse-grained units of work and communication costs can be amortized over longer execution times. Nevertheless, despite over fifteen years of research and clear benefits at both the finest scale (node-level parallelism) and the coarsest scale (workflows), task-based programming has not displaced message passing in the intermediate regime of distributed HPC applications. In this setting, the overheads of task graph management, dependency tracking and data versioning can become prohibitive for fine- to medium-grained tasks on distributed memory, limiting scalability. Moreover, dynamic scheduling and implicit communication can reduce performance predictability, leading to performance anomalies and unexpected bottlenecks that are difficult to diagnose.

OmpSs-2@Cluster [1, 5] is a research platform for exploring distributed task-based execution at a moderate granularity, building on the refined semantics of OmpSs-2 [4] and a runtime designed for scalable cluster execution. It evolves earlier OmpSs@Cluster work by Bueno et al. [8] and incorporates lessons from earlier efforts in distributed task execution. While retaining tasks and dependencies as the core abstraction, OmpSs-2@Cluster mitigates the scalability challenges that arise when task creation, dependency tracking and data management span multiple nodes.

A key design element is support for weak accesses (also known as weak dependencies), as introduced by OmpSs-2 [15]. A weak access indicates that the task does not directly access the data region but its nested subtasks may do so. This allows a parent task to begin execution before the completion of any data transfers required by its children, thereby avoiding unnecessary synchronization and overlapping communication with subtask creation and related dependency management. Weak accesses are a mechanism that supports grouping of tasks into a coarser-grained unit to be offloaded to another node. OmpSs-2@Cluster also employs fragmented region dependencies to interoperate between coarse-grained accesses passed among nodes and fine-grained accesses manipulated on each node. Together these mechanisms aim to make task-based execution more scalable on distributed-memory clusters.

The remainder of this extended abstract provides an overview of the substantial effort devoted over the years to performance analysis and optimizations in OmpSs-2@Cluster. It also discusses the opportunities, challenges and recent progress along two complementary directions: first, inter-node load balancing in MPI + OmpSs-2 programs; and second, exploiting iterative program structure to amortize the costs of task graph construction and management.

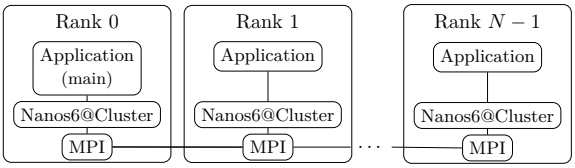
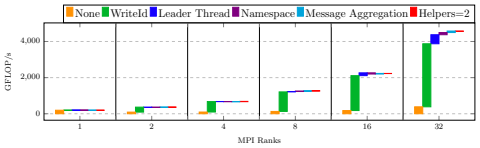
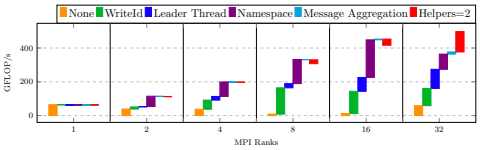


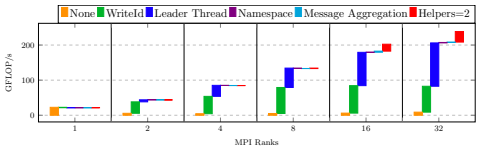
Figure 1 OmpSs-2@Cluster architecture: each rank is a peer and `main` runs as a task on Rank 0.



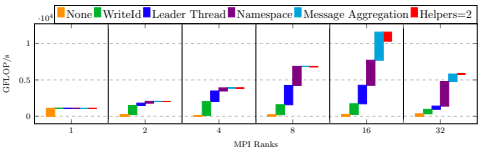
(a) Matrix–matrix multiplication 16384 × 16384.



(b) Matrix–vector multiplication 32768 × 32768.



(c) Jacobi benchmark 32768 × 32768.



(d) Cholesky factorization 32768 × 32768.

Figure 2 Performance impact of key optimizations on MareNostrum 4. Reproduced from [14].

2 Runtime and optimizations

OmpSs-2@Cluster uses the same compiler as regular OmpSs-2 and relies on an open-source fork of the Nanos6 runtime known as Nanos6@Cluster. Early development of Nanos6@Cluster was carried out in a branch of the Nanos6 code base, with regular upstreaming of changes. This approach was later abandoned, as the significantly higher maturity of Nanos6 and its requirement for stable shared-memory–oriented internal interfaces made it difficult to accommodate the experimental and rapidly evolving features needed for distributed execution, some of which involved intrusive changes to these internal APIs. Moreover, maintaining a separate fork allowed the small research-focused OmpSs-2@Cluster team to delay certain technical transitions, most notably the migration from the legacy source-to-source Mercurium compiler to LLVM, in order to concentrate on core runtime development.

As shown in Figure 1, each MPI rank runs an independent instance of Nanos6@Cluster, with all instances communicating as peers via MPI. To simplify data management across ranks, each process establishes an identical virtual address space using `mmap`, allowing tasks to refer to the same memory addresses regardless of the rank on which they execute.

While the basic mechanism for task offloading was relatively straightforward to implement and completed within a few months, achieving satisfactory performance required substantial runtime optimizations developed over several years. Figure 2 illustrates the cumulative impact of these optimizations on performance. As the figure suggests, different benchmarks benefit from different subsets of optimizations. In practice, performance was often sensitive to low-level implementation details, and any such cumulative view depends on the order in which optimizations are introduced in the figure, which is to some extent arbitrary and chosen for explanatory purposes.

The main optimizations implemented in Nanos6@Cluster include WriteID, a form of data versioning used to avoid redundant data transfers; LeaderThread, which dedicates a thread to handle incoming MPI messages such as newly offloaded tasks and to process message

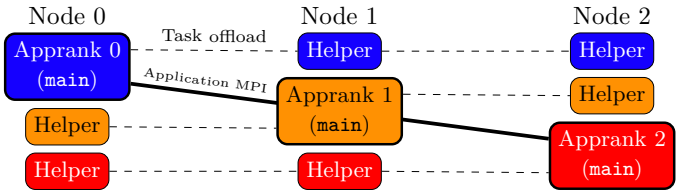


Figure 3 Architecture of MPI+OmpSs-2@Cluster. Application ranks (appranks) communicate via MPI and helper ranks on some other nodes can execute tasks from heavily loaded appranks.

completions; and Namespace, which eliminates unnecessary host-mediated messages between consecutive tasks offloaded to the same rank. Additional improvements include message aggregation, which coalesces control messages when multiple accesses become ready, and multiple low-priority Helper tasks that assist with message handling and runtime progress when compute resources would otherwise be idle. Together, these optimizations substantially reduce overheads and enable scaling to approximately 16–32 nodes for the evaluated small-scale benchmarks.

3 Dynamic Load Balancing (DLB)

Load imbalance is a long-standing source of inefficiency in high-performance computing. It is commonly addressed at application level through techniques such as mesh partitioning, domain decomposition, or manual work redistribution, often guided by problem-specific heuristics. While effective, these approaches entangle load-balancing concerns with application logic and may require substantial code refactoring, complicating development and long-term maintenance. Although OmpSs-2@Cluster does not scale sufficiently to serve as the primary distributed-memory programming model for large-scale HPC applications, it is well suited to addressing residual load imbalance in hybrid MPI+OmpSs-2 programs. In this context, OmpSs-2@Cluster complements static partitioning by redistributing work at runtime.

The basic approach is illustrated in Figure 3. Each MPI rank visible to the application (hereafter referred to as an application rank or apprank) is shown in a different colour, with a single apprank per node in this example. To mitigate load imbalance in an apprank, additional helper ranks are deployed on a subset of other nodes. These helper ranks are full runtime instances that execute tasks offloaded from a given apprank within a dedicated process, providing isolation between appranks while enabling dynamic redistribution of work at runtime.

Load balancing is done at three levels. First, at coarse granularity, helper ranks are activated based on a prediction of upcoming load imbalance. The prediction is calculated by the runtime and passed to an external solver, which determines the minimum number of helpers required for each apprank and allocates these helpers to lightly-loaded nodes. The decisions are implemented by the runtime. Second, at medium granularity, the runtime employs BSC's Dynamic Load Balancing (DLB) [11] library to assign CPU cores to the appranks and active helpers on the same node. Finally, at fine granularity, the runtime instances dynamically offload tasks to helper ranks in order to fully utilize the allocated cores.

4 Distributed Taskiter

The main limits to the scalability of OmpSs-2@Cluster arise from the sequential creation of tasks and computation of their dependencies on Rank 0, as well as the centralized resolution of top-level task dependencies on the same rank. These bottlenecks are partially mitigated through strong support for task nesting, which increases effective task granularity, and through the *Namespace* optimization, which reduces the need for centralized dependency management. However, these mechanisms have largely been pushed to their practical limits within the current runtime design. A complementary approach is therefore to exploit structural regularities in the task graph itself, under programmer direction, enabling substantial reductions in the cost of task creation and dependency management.

Many scientific applications employ iterative methods or multi-step simulations in which the same directed acyclic task graph is executed repeatedly at each timestep or iteration. To address this common pattern, the *taskiter* construct was proposed in 2023 [3]. A loop can be annotated with `taskiter` provided that each iteration generates the same top-level dependency graph and the program remains valid if the code inside the loop body but outside any task is executed just once. The runtime instantiates the tasks once and represents the repeated execution of this acyclic structure as a cyclic task graph across iterations.

Distributed taskiter [18] extends this concept to OmpSs-2@Cluster. When the runtime encounters a loop annotated with `taskiter`, the loop is offloaded to all ranks, each of which locally instantiates the full task dependency graph. The runtime then partitions this cyclic graph across nodes, and each rank precomputes the MPI transfers in which it participates. Compared with MPI + OmpSs-2, the only overhead is the one-time initialization cost, after which the loop body is executed without any control messages. By integrating MPI communications directly into the application's task graph, distributed taskiter naturally overlaps computation and communication. Experimental results show that this approach achieves throughput matching or exceeding that of MPI + OpenMP. In some cases, for example 3D wave parallelism in the Gauss–Seidel heat equation, the asynchronous tasking approach exposes substantially more parallelism than fork–join MPI + OpenMP, and distributed taskiter achieves performance on par with state-of-the-art TAMPI [17] + OmpSs-2 (see [18]).

5 Conclusions

While task-based programming has proven effective at node level and workflow scale, our experience with OmpSs-2@Cluster confirms that extending fine-grained task graphs to distributed memory quickly encounters scalability limits related to centralized task creation and dependency management. Addressing these issues required substantial runtime engineering effort and a sequence of optimizations to enable practical scalability to tens of nodes.

The paper highlights two complementary directions in which distributed tasking provides tangible benefits. First, OmpSs-2@Cluster can be used selectively to mitigate residual load imbalance in hybrid MPI+OmpSs-2 applications. By combining task offloading with BSC's DLB library, our approach improves resource utilization with minimal disruption to existing application structure. Second, for applications with regular iterative structure, distributed taskiter demonstrates that exposing and exploiting task-graph regularity can fundamentally reduce runtime overheads. Similar ideas may apply to other kinds of task graph structure.

Overall, these results suggest that distributed tasking is most effective when applied judiciously, either as a targeted mechanism to address specific inefficiencies such as load imbalance, or in conjunction with programmer-provided structure that enables the runtime to avoid repeated control overheads. While OmpSs-2@Cluster cannot replace MPI as the

dominant distributed-memory programming model for large-scale HPC, it demonstrates that task-based abstractions can deliver productivity, adaptability, and competitive performance when their limitations are explicitly acknowledged and addressed.

References

1 Jimmy Aguilar Mena, Omar Shaaban, Vicenç Beltran, Paul Carpenter, Eduard Ayguadé, and Jesus Labarta. OmpSs-2@Cluster: Distributed memory execution of nested OpenMP-style tasks. In *European Conference on Parallel Processing*, 2022. doi:10.1007/978-3-031-12597-3_20.

2 Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, 2017. doi:10.1109/TPDS.2017.2766064.

3 David Álvarez and Vicenç Beltran. Optimizing iterative data-flow scientific applications using directed cyclic graphs. *IEEE Access*, 2023. doi:10.1109/ACCESS.2023.3269902.

4 Barcelona Supercomputing Center. OmpSs-2 specification, 2021. URL: https://pm.bsc.es/ftp/ompss-2/doc/spec/.

5 Barcelona Supercomputing Center. OmpSs-2@Cluster releases, 2022. URL: https://github.com/bsc-pm/ompss-2-cluster-releases.

6 Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012. doi:10.1109/SC.2012.71.

7 George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pitior Luszczek, and Jack Dongarra. Dense linear algebra on distributed heterogeneous hardware with a symbolic DAG approach. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2012. URL: https://www.osti.gov/servlets/purl/1173290.

8 J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *IEEE 26th International Parallel and Distributed Processing Symposium*, 2012. doi:10.1109/IPDPS.2012.58.

9 Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Berriman, John Good, Anastasia Laity, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3), 2005. doi:10.1155/2005/128026.

10 Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 21(02), 2011. doi:10.1142/S0129626411000151.

11 Marta Garcia, Julita Corbalan, R.M Badia, and Jesus Labarta. A dynamic load balancing approach with SMPSuperscalar and MPI. *Facing the Multicore - Challenge II. Lecture Notes in Computer Science, vol 7174*, 2012.

12 Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. Dynamic task discovery in PaRSEC: a data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 2017. doi:10.1145/3148226.3148233.

13 Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Alvarez, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M Badia. ServiceSs: An interoperable programming framework for the cloud. *Journal of grid computing*, 12(1), 2014. doi:10.1007/s10723-013-9272-5.

14 Jimmy Aguilar Mena. *Methodology for malleable applications on distributed memory systems*. PhD thesis, Universitat Politècnica de Catalunya, 2022. doi:10.5821/dissertation-2117-380814.

15 J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé. Improving the integration of task nesting and dependencies in OpenMP. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017. doi:10.1109/IPDPS.2017.69.

16 James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., USA, 2007.

17 Kevin Sala, Xavier Teruel, Josep M Perez, Antonio J Peña, Vicenç Beltran, and Jesus Labarta. Integrating blocking and non-blocking MPI primitives with task-based programming models. *Parallel Computing*, 2019. doi:10.1016/j.parco.2018.12.008.

18 Omar Shaaban Ibrahim ali, Juliette Fournis d'Albiat, Isabel Piedrahita, Vicenç Beltran, Xavier Martorell, Paul Carpenter, Eduard Ayguadé, and Jesus Labarta. Leveraging iterative applications to improve the scalability of task-based programming models on distributed systems. *ACM Transactions on Architecture and Code Optimization*, 22(3):1–27, 2025. doi:10.1145/3743134.

SLURM-Managed HyperParameter Optimization

Anusha Chattopadhyay
Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University, Germany

Hendrik Borras
Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University, Germany

Bernhard Klein
Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University, Germany

Holger Fröning[1]
Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University, Germany

Abstract

Hyperparameter optimization (HPO) is essential for achieving state-of-the-art performance in machine learning, yet it is computationally demanding, particularly on shared or resource-constrained clusters. We present a system that integrates the Asynchronous Successive Halving Algorithm (ASHA) with SEML, the SLURM Experiment Management Library – an experiment orchestration layer that provides declarative configuration, provenance, metric tracking, and robust SLURM job management. The resulting open-source tool enables scalable, fault-tolerant HPO on SLURM-managed infrastructure: SEML handles experiment specification, versioning, and scheduling, while ASHA performs asynchronous early stopping and resource reallocation to concentrate computation on promising configurations. Overall, the system streamlines experiment lifecycle management, enables distributed evaluations with minimal manual effort, and reduces the time required to reach high-quality configurations compared to conventional Grid and Random Search methods under similar compute budgets.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Software and its engineering → Distributed systems organizing principles

Keywords and phrases Hyperparameter optimization, Asynchronous Successive Halving Algorithm (ASHA), Experiment management, SLURM, SEML, Open Source

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2026.2

Supplementary Material *Software (Source Code)*: `https://github.com/TUM-DAML/seml/pull/160`

Acknowledgements The authors acknowledge support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG.

1 Introduction

Machine learning models are now widely used across multiple industries, ranging from healthcare to weather forecasting. During model development, hyperparameters play a critical role. Some hyperparameters control the structure of the model, such as the number of layers or hidden units per layer, while others govern the training process, such as the choice of optimizer, learning rate, and batch size. These hyperparameters differ from model parameters in that they are not learned from the data but are instead predefined model configurations set before training. A poor choice of hyperparameters can lead to overfitting and inefficient training. Inversely, an optimal combination can lead to well performing results.

[1] corresponding author


© Anusha Chattopadhyay, Hendrik Borras, Bernhard Klein, and Holger Fröning;
licensed under Creative Commons License CC-BY 4.0

Hyperparameter Optimization (HPO) algorithms are used to automate the process of finding such optimal combinations. Besides manual tuning, a common baseline approach is Grid Search, which exhaustively evaluates all parameter combinations. However, more efficient algorithms such as Random Search [2], Asynchronous Successive Halving [11], and Hyperband [12] have been developed to reduce computational cost. These algorithms often require multiple concurrent model evaluations, which can be computationally intensive and challenging to manage in shared or resource-constrained environments.

This raises the challenge of effectively using such algorithms on shared systems, such as high-performance computing (HPC) clusters in industry and research alike. HPC systems are usually orchestrated by workload managers. With the Simple Linux Utility for Resource Management (SLURM) [17] being a popular choice as a workload manager. It efficiently manages and schedules jobs across large-scale, shared computing resources. This allows multiple users to submit and run tasks in parallel while handling resource allocation, job prioritization, and load balancing.

This work focuses on selecting and implementing an HPO framework for experiments conducted on such a SLURM-managed compute cluster. The key objectives are efficient resource utilization and seamless SLURM integration. The framework must support parallelized optimization while maintaining predictable performance and minimal scheduling delays. Additionally, it should be well maintained and suitable for long-term use in a shared research computing environment. Keeping these factors in mind, we build on top of the SLURM Experiment Management Library (SEML) [18], due to its native compatibility with SLURM.

We further select the Asynchronous Successive Halving Algorithm (ASHA) [11] as the core HPO algorithm, due to its reliability and efficiency in distributed computing systems. As ASHA is stable and inherently asynchronous, each configuration makes promotion decisions independently based on the metrics it can access, eliminating the need for centralized coordination and scheduling delays. Its ability to cancel less promising jobs early while maintaining simple and predictable execution makes ASHA particularly well-suited for scalable hyperparameter optimization in SLURM-managed distributed environments. In detail, we make the following overarching contributions:

1. Identification of ASHA as promising HPO algorithm for SLURM-managed compute clusters, based on its efficiency, parallelizability, and robustness to resource contention.
2. Extension of SEML [18] to support ASHA, enabling scalable hyperparameter optimization on SLURM-managed compute clusters. We open-source our implementation as an extension to the official SEML GitHub repository.[2]
3. Evaluation by testing the implementation on standard models such as ResNet18, demonstrating the practical effectiveness of ASHA in realistic scenarios.

2 Background and Related Work

In machine learning, a hyperparameter is a parameter that is set before training to configure the model's training process. These can include settings such as the learning rate or the type of optimizer used during training. Others may refer to topological features such as the number of layers. Notably hyperparameters are different from parameters that are learned from data, such as weights and biases, which change during training. Most machine learning models are highly sensitive to their hyperparameters. Thus, a choice of suboptimal hyperparameters can lead to issues such as lack of convergence, inefficient training, and low quality results.

[2] https://github.com/TUM-DAML/seml/pull/160

Manual tuning is when model developers manually search for optimal hyperparameters. This process can be time-consuming, and its effectiveness relies on the developers' intuition. In contrast, *Hyperparameter Optimization (HPO) Algorithms* are algorithms designed to automate the process of hyperparameter tuning. These algorithms automatically search through often vast, non-differentiable search spaces (e.g., 5 hyperparameters with each 10 choices = 100,000 combinations). The automation of this process allows for better generalization and scalability.

Several libraries, such as Ray Tune [13] and Optuna [1], provide a wide range of hyperparameter-optimization algorithms. However, they do not integrate natively with SLURM. Ray Tune, built on the Ray [16] framework for distributed machine learning, assumes a cluster architecture with a single head node orchestrating multiple worker nodes, whereas SLURM launches independent jobs without maintaining such hierarchy. This mismatch makes Ray's structure undesirable in shared, distributed SLURM-based systems. Optuna likewise lacks direct SLURM integration and requires users to manually manage job submission when distributing trials.

2.1 Selected HPO Algorithms

Grid Search is a brute-force optimization algorithm that evaluates all possible combinations of hyperparameters within a predefined grid. While computationally expensive, it guarantees that an optimal solution will be found within the predefined discrete set of hyperparameter values. The results are also more likely to be reproducible during experimentation, as the process is generally deterministic.

Random Search [2] selects hyperparameter values randomly rather than exhaustively evaluating every possible combination. It is often more efficient than Grid Search, particularly as the dimensionality of the search space increases. However, this randomness makes it harder to reproduce results unless the random number generator is carefully seeded.

The **Successive Halving Algorithm (SHA)** [7] is an evolution of Random Search that employs early stopping to conserve computational resources. This algorithm uses a *bracket*, which consists of a sequence of evaluation *rungs*. Each *rung* within the bracket represents a stage where the algorithm evaluates whether a configuration should continue training or be terminated. The decision is based on whether the configuration ranks within the top $\lfloor n/\eta \rfloor$ configurations, where n is the number of active configurations and $\eta > 1$ is the reduction factor that determines the proportion of configurations to retain. All active configurations must reach a predefined rung before any elimination occurs, making this algorithm inherently synchronous.

Hyperband [12], similarly is an early-stopping algorithm that eliminates poorly performing trials. Hyperband is essentially a collection of multiple rounds of SHA, where each round is referred to as a stage. While this improves the inherent exploration/exploitation tradeoff, it also adds even more synchronization requirements. Making Hyperband inefficient in highly parallel and distributed settings.

The **Asynchronous Successive Halving Algorithm** (ASHA) [11] is an asynchronous version of SHA designed for parallel hyperparameter optimization. Unlike Hyperband or SHA, ASHA does not wait for all configurations in a bracket to complete before promoting the best ones at a decision *rung*. It thus evaluates configurations asynchronously, allowing for better utilization of computational resources. While this entails making decisions on incomplete information, which may lead to non-optimal choices, the issue can be compensated by algorithm configuration adjustments. This makes ASHA more efficient in distributed environments.

For its efficiency and scalability, this algorithm was chosen for hyperparameter optimization tasks. Unlike traditional methods like Random Search or Bayesian Optimization [5], it can handle a large number of configurations simultaneously due to its asynchronous nature. This allows for better utilization of computational resources, as it does not require waiting for all configurations to complete before making decisions on promotions. Additionally, ASHA's successive halving strategy effectively allocates resources to the most promising configurations, leading to faster convergence towards optimal hyperparameters. This makes it particularly well-suited for scenarios where training models can be time-consuming and resource-intensive.

2.2 SLURM and SEML

SLURM is a widely adopted, free, and open-source workload manager and job scheduler for Linux and other Unix-like based high-performance computing (HPC) clusters. It provides resource allocation, job scheduling, accounting, and monitoring to coordinate workloads across shared compute infrastructures.

Implications for multi-job algorithms. Because queued jobs start opportunistically rather than in lockstep, workflows that launch multiple jobs must handle asynchronous execution – start times, runtimes, and completion order are not guaranteed to align.

SLURM Experiment Management Library (SEML) [18] is an open-source library designed to manage SLURM jobs. It integrates with Sacred [9] for experiment management and MongoDB [15] for experiment storage. SEML generates job scripts that specify experiment details, resource requests, and commands, which are then submitted to SLURM. Once submitted, SEML tracks job statuses, logs, and results, simplifying reproducibility and debugging. The library includes implementations of Grid Search and Random Search for hyperparameter optimization. As it is specifically designed for SLURM-managed clusters, SEML is an ideal framework for extending with the selected HPO algorithm.

Sacred [9] is a tool to manage computational experiments. It aids in logging and reproduction. SEML is built on top of Sacred and thus leverages some of Sacred's capabilities in its experiment management.

SEML stores experiment information in a *MongoDB* [15] database. This meant that much of the final framework development was dependent on interacting with MongoDB.

3 HPO Algorithm Selection and System Architecture

In this work, we selected ASHA as our HPO algorithm due to its efficiency and effectiveness in hyperparameter optimization tasks, as we will elaborate on in the following.

3.1 Details of ASHA

The following section describes the ASHA [11] algorithm. In Algorithm 1, processes run in parallel, starting whenever resources are available (line 2).

Algorithm 1 Asynchronous Successive Halving Algorithm (ASHA).

1: **Input:** $r_{\max}$: maximum resource per config (e.g., epochs), η: elimination rate, $r_{\min}$: minimum resource to train a configuration for, N: number of ASHA jobs to start
2: **while** compute resources are available and $N < n$ **do** ▷ Run in parallel
3: Sample new configuration c and increment n
4: Train c for $r_{\min}$ resources
5: **for** $k = 0$ **to** $\left\lfloor \log_\eta \left(\frac{r_{\max}}{r_{\min}} \right) \right\rfloor$ **do**
6: $r \leftarrow r_{\min} \cdot \eta^k$
7: Wait until c finishes training for r resources ▷ Configuration reaches evaluation point(rung)
8: Evaluate c and store its performance ▷ Evaluation occurs here
9: **if** c ranks in top $1/\eta$ fraction at rung r **then**
10: Promote c to next rung with resource $r \cdot \eta$
11: **else**
12: Stop training c
13: **break**
14: **end if**
15: **end for**
16: **end while**

In Algorithm 1, line 5 is responsible for deciding the times to check for promotion. This formula counts how many times one can multiply the minimum resource $r_{\min}$ by the factor η before reaching the maximum resource $r_{\max}$.

After a configuration completes training at a given resource level, it is evaluated and compared to others at the same rung. If it ranks in the top η or $\lfloor n/\eta \rfloor$, it is promoted to the next rung with more resources; otherwise, training is stopped early. As this process is asynchronous, each evaluation is done only with processes that have already reached this rung.In scenarios where compute restrictions force ASHA to execute each configuration serially, each successive process can benchmark against prior ones, thereby enabling early stopping.

It is possible to further extend ASHA with a tie-breaking strategy when two configurations achieve the same accuracy. For example, the algorithm could choose which configuration to promote based on factors such as which one arrived first, or select randomly if they arrived simultaneously. However, in our implementation, we avoided tie-breaking, by simply promoting all tied configurations.

3.2 Integration with SEML

Figure 1 depicts the overall system architecture. It describes the overall program and data flow for the presented ASHA extension. When the experiment configuration reaches a stage marked as an evaluation rung, the ASHA class compares it with its neighbors by communicating with the shared MongoDB collection. The implementation is designed to seamlessly integrate ASHA within the existing SEML framework, extending SEML with efficient hyperparameter optimization.

An overview of the integration of ASHA in SEML is given in Algorithm 2. This integration leverages SEML's existing capabilities while introducing ASHA's efficient hyperparameter optimization strategy. The process begins with the configuration file `experiment.yaml`,

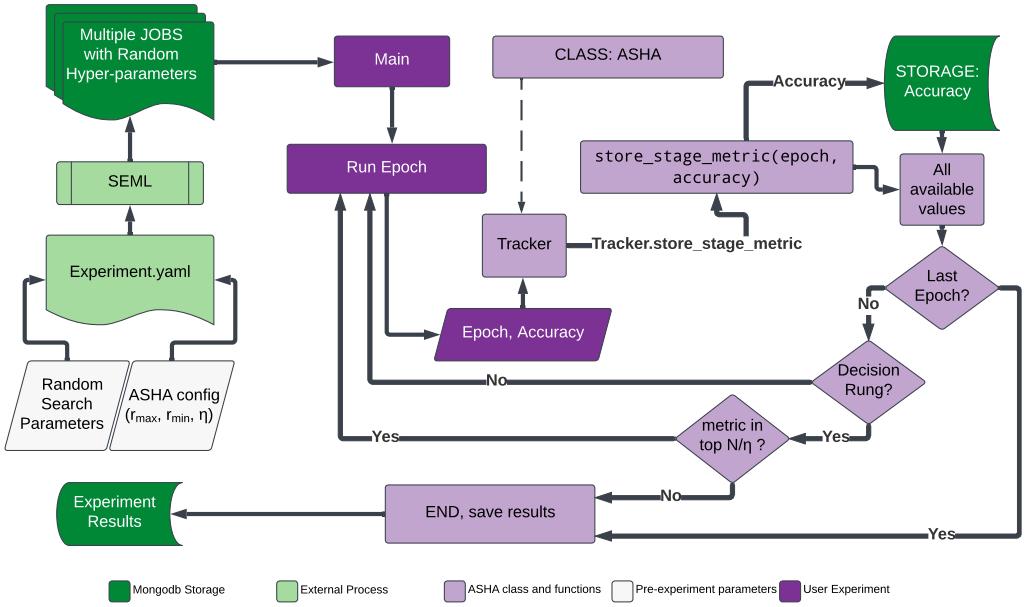


Figure 1 System Architecture of the integration of ASHA with SEML. The components are colour coded according to functionality. The overall process takes the already existing feature of Random Search in SEML to extend it to ASHA. This diagram uses accuracy as the evaluation metric and epochs as the promotion stages.

which defines the hyperparameter search space using Random Search and includes ASHA-specific parameters such as η, $r_{\min}$, and $r_{\max}$. SEML initializes multiple processes based on the defined hyperparameter configurations. Each process imports the `ASHA.py` file, which contains the ASHA class and its associated functions. During training, each process periodically calls the `tracker.store_stage_metric()` function to log the current metric (e.g., accuracy) to a shared MongoDB collection. When a process reaches a decision rung, it queries the MongoDB to retrieve metrics from peer processes that have also reached this rung. Using this data, the ASHA decision-making logic is executed to determine whether the process should be promoted to the next rung or terminated. If the process ranks within the top N/η or meets the promotion criteria, it is promoted; otherwise, a stop signal is sent to terminate the process. This integration allows ASHA to function effectively within the SEML framework, enabling efficient and scalable hyperparameter optimization.

Setting up a Hyperparameter Optimization run with ASHA revolves around 3 basic files and the SEML base libraries:

- **Configuration file (usually named `experiment.yaml`):** mandatory file defining all SLURM and SEML settings, except for MongoDB configurations, which are set separately. It also specifies additional ASHA parameters, such as minimum and maximum resources, while SEML's Random Search configurations define the hyperparameters for the experiments.
- **`main.py`:** is the main experiment script, where the model is implemented. It communicates with the ASHA class, sending experiment data and receiving stop signals. Users may act on these signals to halt the process or perform other cleanup tasks.
- **`asha.py`:** contains the ASHA class along with all functions for ASHA's stopping criteria and logging logic.

Algorithm 2 Overview of the ASHA Process Integrated with SEML.

Input: `experiment.yaml` contains Random Search hyperparameters, ASHA setup: η, $r_{\min}$, $r_{\max}$.
SEML initializes processes via Random Search defined in `experiment.yaml`.
Each process imports the `ASHA.py` file as a `tracker` object, which contains all required ASHA-related functions.
`tracker.store_stage_metric()` logs accuracy (or any other metric) to a shared MongoDB collection at each stage (e.g epoch).
if the current stage corresponds to a decision rung **then**
    Query MongoDB for peer metrics at this rung.
    Perform ASHA decision-making **only** based on available data.
    if the process ranks in the top $\lfloor N/\eta \rfloor$ or meets cutoff **then**
        Promote to the next rung.
    else
        Send stop signal to terminate the process.
    end if
end if

3.3 Storage and Access

As a centralized controller isn't ideal and sometimes non-viable in a shared cluster enviroment, we rely on a MongoDB collection in which each trial logs its progress, enabling coordination across processes.

This is achieved by using the `store_stage_metric()` function to store the current tasks accuracy (metric) in a database collection for each epoch (stage). This means that when a task reaches a decision rung, it checks the database to find the results of all tasks which have already reached and/or crossed this rung (in this case at an epoch in a predefined set calculated using $r_{\min}$ and $r_{\max}$). Following, the performance of all available tasks is compared and the corresponding ASHA promotion action is performed.

To uniquely identify each configuration and avoid conflicts, a Universally Unique Identifier (UUIDv4) is assigned to every job. Each entry stores the epoch, the metric value, and the associated job's UUID. This design prevents multiple jobs from reading and writing the same object simultaneously, ensuring concurrency safety.

With this system, once a job stores its evaluated metric, it no longer needs to modify it; as the metrics are solely used by other jobs for promotion decisions.

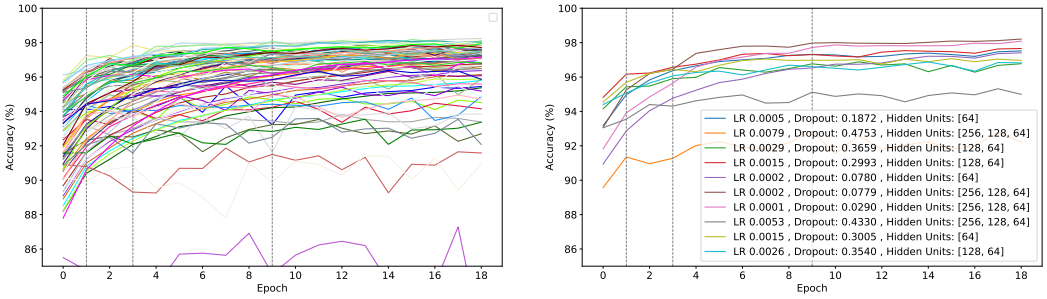
3.4 Configurations

ASHA is configured in the experiment.yaml file with the following metrics: η is the reduction factor and N/η specifies the number of jobs promoted per rung. $r_{\min}$ and $r_{\max}$ set the minimum and maximum rungs at which a job can be promoted. Metrics with "bigger is better" are specified by `metric_increases = True`, while metrics with "less is better" are specified by `metric_increases = False`.

The parameter allocations are derived from the Random Search configurations already defined in SEML's setup. For each job, the full history of the chosen metric (such as accuracy or any other performance measure) is stored in MongoDB. This allows configurations to be compared at each decision rung, enabling ASHA to make promotion or early-stopping decisions based on the recorded metrics.

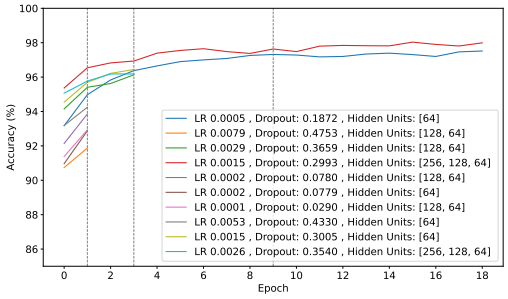
4 Results

We evaluated our ASHA implementation on two widely used image classification benchmarks, MNIST [4] and CIFAR-10 [10], using neural networks of differing complexity with no image augmentation. For MNIST, we used a configurable multilayer perceptron (*SimpleNN*) with a variable number of fully connected layers, ReLU activations, optional dropout, and a ten-class output layer. Each SimpleNN configuration was trained with ADAM [8] optimizer for 20 epochs with evaluation intervals $r \in [1, 20]$. Hyperparameters were tuned over 1 to 3 hidden layers with variable unit sizes, dropout rates and learning rates. For CIFAR-10, we used ResNet-18 [6] from Torchvision [14], training for 50 epochs with $r \in [1, 50]$. In all experiments, the reduction factor was fixed at $\eta = 3$, and the progression parameter was set to *increasing* to reflect that higher accuracy indicates better performance. Experiments were run on NVIDIA TITAN X GPUs with a memory capacity of 14 GB.



(a) Grid Search (90 trials, 20 epochs).

(b) Random Search (10 trials, 20 epochs).



(c) ASHA (10 trials, 20 epochs).

Figure 2 Comparison of search strategies on MNIST with *SimpleNN*. Random and Grid Search train all configurations to completion, whereas ASHA adaptively allocates resources by early-stopping underperforming trials and promoting only the most promising ones. In our experiments, ASHA allocated full training budgets to only two high-performing configurations, achieving accuracies comparable to the exhaustive methods while using substantially less resources. All methods used the same search space configuration.

4.1 Comparison of Hyperparameter Optimization Methods

To evaluate the behaviour of different hyperparameter optimization algorithms within the SEML framework, we conducted experiments using Random Search, Grid Search, and ASHA on the *SimpleNN* model. Random Search and ASHA were each executed for 10 trials, while Grid Search exhaustively explored a discrete configuration space of size $3 \times 6 \times 5$. The

Table 1 Comparison of total compute time, achieved accuracy, and relative speedups of different hyperparameter optimization methods on the *SimpleNN* model using MNIST. Grid Search (GS) explores $3 \times 6 \times 5 = 90$ configurations, while Random Search (RS) and ASHA each ran for $N = 10$ trials. Notably, ASHA avoided unpromising configurations through early stopping, while Grid and Random Search completed all runs regardless of performance.

Algorithm	Acc. [%]	Total Time [s]	Speedup over RS	Speedup over GS
Grid Search	98.1	20361.26	–	1.00×
Random Search	97.5	3296.61	1.00×	6.18×
ASHA	97.0	642.31	5.13×	31.70×

grid comprised three possible hidden layer configurations ranging from [64], [128, 64], and [256, 128, 64], six dropout values uniformly spaced between 0.0 and 0.5; and five learning rates sampled log-uniformly between 10^{-4} and 10^{-2}. For Random Search and ASHA, the same parameter ranges were used, but the exact hyperparameter combinations were determined stochastically rather than by enumerating a fixed set of configurations.

Figure 2 shows that Grid Search and Random Search train all configurations to completion, whereas ASHA focuses resources on promising trials by early-stopping underperforming ones. This selective allocation allows ASHA to reach comparable accuracy with far less compute resources. As summarized in Table 1, ASHA is substantially faster than both alternatives. While Grid Search achieved the highest accuracy (97.5%), both Random Search and ASHA exceeded 96% within ten trials. Overall, ASHA provides an excellent balance between accuracy and efficiency, making it well suited for scenarios with limited computational resources.

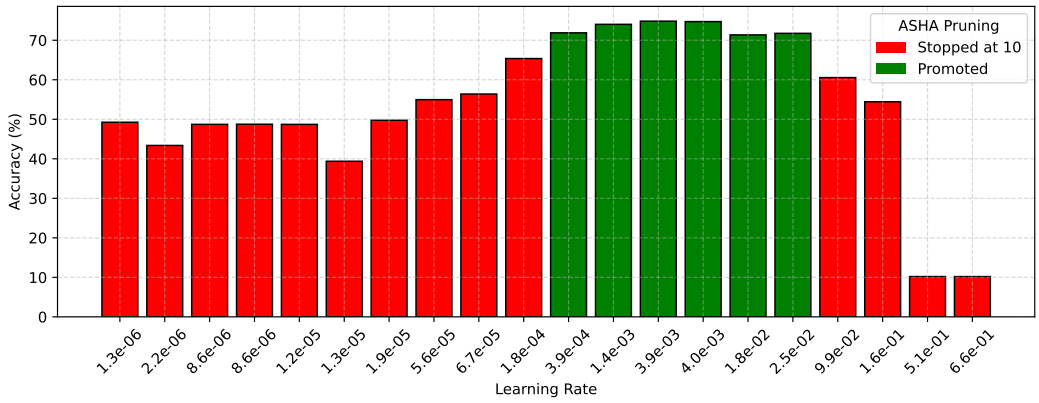


Figure 3 Summary of ResNet-18 training on CIFAR-10 over 20 sampled configurations. The learning rate was drawn log-uniformly from the range 10^{-6} to 1. Early trial termination began after $r_{\min} = 10$. It is observed that the trials with a learning rate set in the range $3.9 \cdot 10^{-4}$ to $2.5 \cdot 10^{-2}$ were promoted after the first elimination round. These results line up with intuition, in that a learning rate should be neither too small nor too large. Further decisions after the first rung are not shown, as they only reinforce this observation.

4.2 Hyperparameter Search over a Wide Learning Rate Range

To examine ASHA's behaviour under an extremely broad hyperparameter range, we conducted a search over learning rates spanning six orders of magnitude, from 10^{-6} to 1. ResNet-18 was trained on CIFAR-10 across 20 configurations sampled from this log-uniform distribution. To

avoid premature pruning given the large range, the minimum promotion threshold was set to $r_{\min} = 10$, ensuring that all configurations completed at least ten epochs before potential cancellation.

As shown in Figure 3, ASHA quickly differentiated between promising and unpromising configurations during training. Runs with learning rates that were either too high or too low were stopped after the first evaluation stage, whereas configurations with moderate learning rates advanced further through the training process. These results line up with intuition, as a learning rate that is too low leads to inefficient training, whereas an overly large learning rate will lead to overfitting. This pattern shows that ASHA can automatically identify an effective region of the learning rate space without exhaustive evaluation and while working asynchronously.

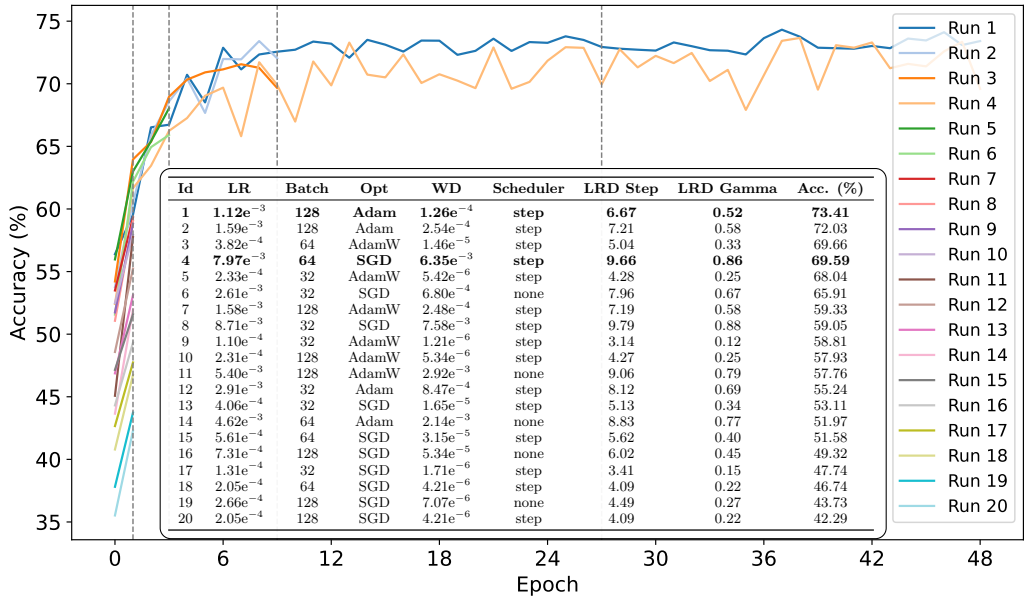
Overall, the experiment demonstrates how ASHA efficiently allocates computational resources by focusing on configurations with strong early performance while eliminating unpromising ones. The trials that progressed beyond the first rung converged consistently to higher accuracies, confirming ASHA's ability to concentrate computation on the most relevant hyperparameter ranges.

Table 2 Hyperparameter search space for ResNet-18 on CIFAR-10 (50 epochs; $r_{\min} = 1$, $r_{\max} = 50$).

Hyperparameter	Search Space	Distribution	Note / Purpose
`learning rate`	$[10^{-4}, 10^{-2}]$	log-uniform	Step size for gradient updates; controls learning stability and convergence speed.
`batch size`	{32, 64, 128}	categorical	Samples per update; larger batches improve stability but increase memory use.
`optimizer`	{SGD, Adam, AdamW}	categorical	Gradient update algorithm; AdamW includes decoupled weight decay.
`weight decay`	$[10^{-6}, 10^{-2}]$	log-uniform	L2 regularization strength to reduce overfitting.
`lr scheduler`	{none, step}	categorical	Learning rate schedule type.
`lr decay step`	[3, 10]	uniform	Interval (in epochs) for step-based LR decay.
`lr decay gamma`	[0.1, 0.9]	uniform	Multiplicative LR reduction factor per decay step.

4.3 Evaluation of ASHA on ResNet-18 with Multiple Hyperparameters

To evaluate ASHA's performance in a higher-dimensional search space, we trained ResNet-18 while jointly tuning seven hyperparameters, which corresponds to a search space dimensionality wholly unrealistic for standard Grid Search. The corresponding parameters and their search spaces are summarized in Table 2. As illustrated by Figure 4, the top-performing configurations did not share a single common hyperparameter setting, underscoring the sensitivity of the model to the joint optimization of multiple parameters. The experiment confirms that ASHA scales effectively to complex search spaces involving many interacting hyperparameters. Through its integration with the SEML framework, ASHA can now be seamlessly deployed on SLURM-managed compute clusters, enabling efficient hyperparameter optimization and helping researchers identify well-performing models with significantly reduced compute cost.



Id	LR	Batch	Opt	WD	Scheduler	LRD Step	LRD Gamma	Acc. (%)
1	**1.12e^{-3}**	**128**	**Adam**	**1.26e^{-4}**	**step**	**6.67**	**0.52**	**73.41**
2	1.59e^{-3}	128	Adam	2.54e^{-4}	step	7.21	0.58	72.03
3	3.82e^{-4}	64	AdamW	1.46e^{-5}	step	5.04	0.33	69.66
4	**7.97e^{-3}**	**64**	**SGD**	**6.35e^{-3}**	**step**	**9.66**	**0.86**	**69.59**
5	2.33e^{-4}	32	AdamW	5.42e^{-6}	step	4.28	0.25	68.04
6	2.61e^{-3}	32	SGD	6.80e^{-4}	none	7.96	0.67	65.91
7	1.58e^{-3}	128	AdamW	2.48e^{-4}	step	7.19	0.58	59.33
8	8.71e^{-3}	32	SGD	7.58e^{-3}	step	9.79	0.88	59.05
9	1.10e^{-4}	32	AdamW	1.21e^{-6}	step	3.14	0.12	58.81
10	2.31e^{-4}	128	AdamW	5.34e^{-6}	step	4.27	0.25	57.93
11	5.40e^{-3}	128	AdamW	2.92e^{-3}	none	9.06	0.79	57.76
12	2.91e^{-3}	32	Adam	8.47e^{-4}	step	8.12	0.69	55.24
13	4.06e^{-4}	32	SGD	1.65e^{-5}	step	5.13	0.34	53.11
14	4.62e^{-3}	64	Adam	2.14e^{-3}	none	8.83	0.77	51.97
15	5.61e^{-4}	64	SGD	3.15e^{-5}	step	5.62	0.40	51.58
16	7.31e^{-4}	128	SGD	5.34e^{-5}	none	6.02	0.45	49.32
17	1.31e^{-4}	32	SGD	1.71e^{-6}	step	3.41	0.15	47.74
18	2.05e^{-4}	64	SGD	4.21e^{-6}	step	4.09	0.22	46.74
19	2.66e^{-4}	128	SGD	7.07e^{-6}	none	4.49	0.27	43.73
20	2.05e^{-4}	128	SGD	4.21e^{-6}	step	4.09	0.22	42.29

Figure 4 Accuracy histories for 20 ResNet-18 trials on CIFAR-10 with an embedded table summarizing hyperparameters and final accuracies.

5 Conclusion

In this paper, we presented a framework for running hyperparameter optimization on SLURM-managed clusters by integrating the ASHA algorithm into the SEML experiment management library. The framework coordinates large numbers of concurrent trials through a shared MongoDB backend and can be deployed on existing SLURM infrastructures without modifying the underlying scheduler.

Our experiments show that ASHA substantially reduces compute time compared to simple baseline methods. On the workloads considered, it achieved up to a 5.13× speed-up over Random Search and a 31.7× speed-up over Grid Search, while reliably identifying well-performing hyperparameter configurations, including in high-dimensional search spaces that are infeasible for naive methods.

By providing an open-source implementation tailored to SLURM-based clusters, we make efficient, asynchronous hyperparameter optimization readily accessible to practitioners. Future work includes extending the framework with additional schedulers and search algorithms and exploring more advanced resource-allocation policies for heterogeneous cluster environments.

The framework is already in regular use on production SLURM clusters at our institution, where it has successfully optimized models for “Uncertainty Reasoning with Photonic Bayesian Machines” [3], which are notably sensitive to hyperparameters, further demonstrating its practical utility.

References

1 Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631, 2019. doi:10.1145/3292500.3330701.

2 James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012. doi:10.5555/2503308.2188395.

3 F. Brückerhoff-Plückelmann, H. Borras, S. U. Hulyal, L. Meyer, X. Ji, J. Hu, J. Sun, B. Klein, F. Ebert, J. Dijkstra, L. McRae, P. Schmidt, T. J. Kippenberg, H. Fröning, and W. Pernice. Uncertainty reasoning with photonic bayesian machines, 2025. doi:10.48550/arXiv.2512.02217.

4 Li Deng. The MNIST database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Process. Mag.*, 29(6):141–142, 2012. doi:10.1109/MSP.2012.2211477.

5 Roman Garnett. *Bayesian Optimization*. Cambridge University Press, 2023.

6 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi:10.1109/CVPR.2016.90.

7 Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In Arthur Gretton and Christian C. Robert, editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pages 240–248, Cadiz, Spain, 09–11 May 2016. PMLR. URL: https://proceedings.mlr.press/v51/jamieson16.html.

8 Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, volume abs/1412.6980, 2015. doi:10.48550/arXiv.1412.6980.

9 Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. The Sacred Infrastructure for Computational Research. In Katy Huff, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 49–56, 2017. doi:10.25080/shinma-7f4c6e7-008.

10 Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

11 Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Bentzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of the Third Conference on Machine Learning and Systems, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, volume 2, pages 230–246. mlsys.org, 2020. URL: https://proceedings.mlsys.org/paper_files/paper/2020/hash/a06f20b349c6cf09a6b171c71b88bbfc-Abstract.html.

12 Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: a novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(1):185:1–185:52, 2017. URL: https://jmlr.org/papers/v18/16-558.html.

13 Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A Research Platform for Distributed Model Selection and Training, July 2018. doi:10.48550/arXiv.1807.05118.

14 TorchVision maintainers and contributors. Torchvision: Pytorch's computer vision library. https://github.com/pytorch/vision, 2016.

15 MongoDB, Inc. *MongoDB Manual*, 2025. Accessed: 2025-07-10. URL: https://www.mongodb.com/docs/.

16 Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, OSDI'18, pages 561–577, USA, 2018. USENIX Association. URL: `https://www.usenix.org/conference/osdi18/presentation/nishihara`.

17 Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. `doi:10.1007/10968987_3`.

18 Daniel Zügner, Johannes Gasteiger, Nicholas Gao, and Dominik Fuchsgruber. SEML: Slurm Experiment Management Library, 2023. URL: `https://github.com/TUM-DAML/seml`.

High Performance Visualization with VisIVO Across Cloud and HPC

Umer Arshad
Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Italy

Eva Sciacca
INAF Astrophysical Observatory of Catania, Italy

Nicola Tuccari
Department of Mathematics and Informatics, University of Catania, Italy

Fabio Pitari
CINECA, Bologna, Italy

Giuseppa Muscianisi
CINECA, Bologna, Italy

Abstract

The rapid growth of data in Astrophysics and Cosmology creates significant challenges that require scalable computing and advanced visualization solutions. Cineca is Italy's largest supercomputing center and a leading global provider of high-performance computing (HPC) services. This paper shows that the integration of the VisIVO scientific visualization framework is with the cloud-based InterActive Computing (IAC) service at Cineca. This integration enables GPU-accelerated, real-time visualization on HPC resources via a Jupyter interface in their browser. A new dedicated Python wrapper and a custom Jupyter kernel enable VisIVO to run smoothly from interactive notebooks, avoid command-line operations, and visualize data directly on HPC compute nodes. Furthermore, we enabled cloud-oriented RESTful APIs, built with the Flask framework, to perform VisIVO operations remotely via simple web services. This setup hides the backend's complexity and simplifies connections with other applications. Our framework increases system accessibility, ensures reproducibility of results, and supports rapid data exploration for large astrophysical simulations. The system was evaluated using real-world cases, including visual analysis of cosmological simulations generated using the OpenGadget3 code. Results indicate that the system is scalable and reliable, and that it facilitates interactive scientific discovery on high-performance computing (HPC) infrastructures.

2012 ACM Subject Classification Computing methodologies

Keywords and phrases High-performance computing, HPC, VisIVO, Scientific visualization, Interactive visualization, Cloud computing, Jupyter, Flask, REST API

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2026.3

Funding The work is supported by the EuroHPC JU under grant agreement No 101093441 (SPACE CoE) and by the Spoke 1 "Future HPC & BigData" of the ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing, funded by NextGenerationEU.
Umer Arshad: ICSC – Spoke 1.
Eva Sciacca: SPACE CoE, ICSC – Spoke 1.
Nicola Tuccari: ICSC – Spoke 1.
Fabio Pitari: ICSC – Spoke 1.
Giuseppa Muscianisi: ICSC – Spoke 1.

© Umer Arshad, Eva Sciacca, Nicola Tuccari, Fabio Pitari, and Giuseppa Muscianisi;
licensed under Creative Commons License CC-BY 4.0

17th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 15th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2026).
Editors: Davide Baroffio, Paola Busia, Lev Denisov, and Nitin Shukla; Article No. 3; pp. 3:1–3:11

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Astrophysical observations and simulation codes on high-performance supercomputers generate petabytes of data [22]. Managing this data – storing, retrieving, and analyzing it – presents major challenges critical to scientific progress [17]. Pre-exascale systems enable new ways to scale High-Performance Computing (HPC) for Astrophysics and Cosmology. Researchers now need both powerful computation and interactive methods to visualize results [7]. VisIVO[1], the Visualization Interface for the Virtual Observatory is a suite of tools for high-performance, multidimensional data analysis and visualization in astrophysics [11]. It was first built for distributed systems like grids and clouds [5], and later updated for the European Open Science Cloud (EOSC) [20, 21] using containerized science gateways.

VisIVO supports interactive and portable workflows [19]. Most recently, it has been integrated with the cloud-oriented InterActive Computing (IAC[2]) service [4], which is available on the Galileo 100 cluster (G100[3]) [9], at Cineca. This IAC service enables users to access compute nodes on demand via Jupyter notebooks in a web browser. The IAC service fundamentally transforms the conventional batch-based HPC access model into a fully interactive, web-accessible environment. This platform enables users to initiate notebook sessions directly on compute nodes equipped with GPUs and high-speed interconnects, thereby facilitating interactive data analysis, real-time visualization, and dynamic simulation control [3]. By enhancing usability and reducing technical barriers, IAC addresses critical challenges in HPC accessibility and user engagement. The integration of VisIVO with IAC further empowers researchers to conduct 3D visualization workflows seamlessly within the HPC ecosystem, eliminating the need for complex command-line configurations or manual data transfer. This integration extends the scalability and computational power of HPC to Jupyter-based environments, promoting reproducible, accessible, and user-centric scientific workflows.

We embed VisIVO within IAC via lightweight Python wrappers and a custom Jupyter kernel that automatically loads modules, configures environment variables, and transparently calls the underlying C++ binaries [22]. This design hides system complexity and delivers consistent execution across distributed nodes. Additionally, we have developed cloud-oriented RESTful API services that expand VisIVO's capabilities beyond the notebook interface. Built with lightweight Python frameworks, these APIs enable remote execution of visualization tasks and provide programmatic and web-based access for large-scale data rendering and analysis. This work complements the interactive IAC integration by enhancing automation and laying the foundation for future cloud–HPC hybrid environments.

The paper is organized as follows: Section 2 provides an overview of related works in interactive and HPC-driven visualization; Section 3 introduces the VisIVO Server modules and its architecture; Section 4 introduces the InterActive Computing (IAC) service; Section 5 describes the methodology used to integrate VisIVO into the IAC framework, while Section 6 focuses on the development of REST-based visualization APIs; Conclusions and future developments are discussed in Section 7.

[1] VisIVO, `https://visivo.readthedocs.io/`
[2] IAC, `https://jupyter.g100.cineca.it/`
[3] Galileo 100 infrastructure: `https://www.hpc.cineca.it/systems/hardware/galileo100/`

2 Related Works

Scientific visualization has progressed markedly at scale – especially in astrophysics and cosmology – where petascale simulations and surveys demand real-time analysis and visual insight. Although existing tools and platforms provide a solid foundation, just a few let you do data analysis and visualization in real-time, in one reproducible HPC workflow. Early pioneers in large-scale HPC visualization, VisIt [8] and ParaView [1] established powerful frameworks for parallel and remote rendering across distributed systems. Although both support HPC-friendly client–server models, they typically demand expert configuration and do not offer native web integration, which constrains usability for many domain scientists.

Interactive web-based platforms have become popular through science gateways and cloud-centric systems. For instance, [6] gateways have been shown to provide researchers access to sophisticated astrophysical analysis tools like VisIVO through web portals. Following this, the NEANIAS Visualization Gateway [21] was developed, incorporating VisIVO into the European Open Science Cloud (EOSC) using containerized services and Jupyter interfaces. This work demonstrated that deploying advanced astrophysical visualization on federated cloud resources is feasible, ensuring reproducibility and cross-platform portability. Nevertheless, these gateways generally do not deeply integrate with HPC schedulers or support real-time session control, operating primarily in batch mode or with limited interactivity.

As Jupyter Notebooks have become standard tools for reproducible scientific workflows, various projects have investigated combining them with high-performance computing resources. The work of [16] introduced Jupyter as a user interface for scientific computing, and newer efforts, such as the Fenix infrastructure [4], have adapted this approach specifically for neuroscience and brain studies. These platforms demonstrated that HPC systems could be accessed via notebook-driven web portals, though they primarily focused on static data analysis rather than interactive, GPU-powered visualization.

Here, combining VisIVO with Cineca's InterActive Computing (IAC) service, as demonstrated in this study, addresses an important shortfall. This method enables interactive 3D rendering on HPC nodes and packages VisIVO's functionality within a specialized Jupyter kernel via Python wrappers, making it easier for users to access high-performance GPU resources. Unlike [21], this solution removes the necessity for manual module management, SSH file transfers, or command-line interface rendering. Additionally, it offers live visualization through VisIVO Viewer and supports complex processing workflows by using pre-configured, scriptable notebooks. Spack [10] and Ansible [13] are commonly used tools for automating deployments in HPC software environments, and their application here to merge Python and C++ tools into unified notebook setups highlights an advanced degree of composability. This approach converts traditional static command-line workflows into interactive, reproducible sessions, thereby enhancing both the reproducibility and accessibility of scientific visualization methods in astrophysics. This method has been applied specifically to astrophysical simulations such as GADGET and to workflows involving density visualization and the examination of cosmological structures [22], but it might be generalized to any command-line-based backend tool. Additionally, it enables the use of cloud-based RESTful services and serverless visualization backends, simplifying HPC complexity without sacrificing efficiency.

3 VisIVO Server

VisIVO Server is a collection of software tools built to produce customized 3D visualizations using astrophysical datasets, handling very large amounts of data with no restrictions on data dimensions [23]. The primary modules available via the command line are:

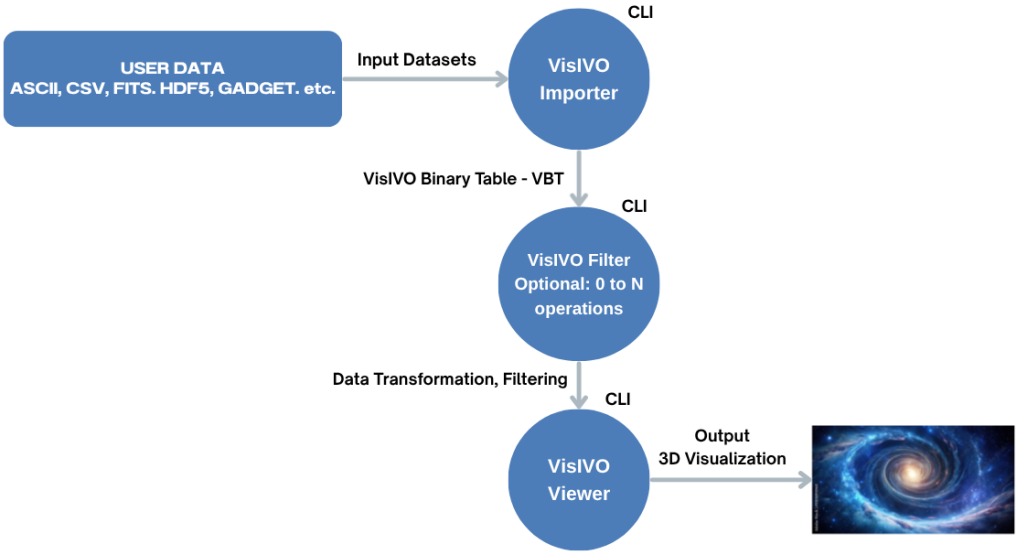


Figure 1 VisIVO Server modular architecture and basic workflow involving importer, filter and view operations.

- VisIVO Importer: transforms user datasets into an efficient internal format called VisIVO Binary Tables (VBT), supporting a wide range of formats from common data types like ASCII and CSV to specialized astronomy formats such as FITS, HDF5, and GADGET.
- VisIVO Filter: allows users to process VBT files by applying data transformations, performing filtering, and other modifications needed before visualization.
- VisIVO Viewer: creates interactive 3D graphics from VBTs, enabling users to visually analyze and explore datasets efficiently.

A standard VisIVO Server workflow progresses through three stages – data preparation, processing, and visualization (as shown in Figure 1).

4 InterActive Computing service (IAC)

As the demands of modern scientific workflows increase, there is a need for capabilities beyond traditional batch-based processing. Fenix[4] addresses this need for interactive access to large-scale computing resources by providing a federated set of infrastructure services that complement traditional batch-oriented HPC systems [2]. Fenix introduces Interactive Computing Services (IAC) as a core constituent of its architecture. IAC enables users to access a persistent, low-latency interactive environment, typically via web-based interfaces such as Jupyter. IAC has been developed with a focus on supporting neuroscience work in the Human Brain Project (HBP[5]) and European Brain Research INfrastructure (EBRAINS[6]) communities. This service lets researchers interact with data and run applications directly, facilitating rapid feedback and accelerating scientific progress. The development

[4] Fenix, `https://fenix-ri.eu/`
[5] Human Brain Project, `https://humanbrainproject.eu/`
[6] EBRAINS, `https://ebrains.eu/`

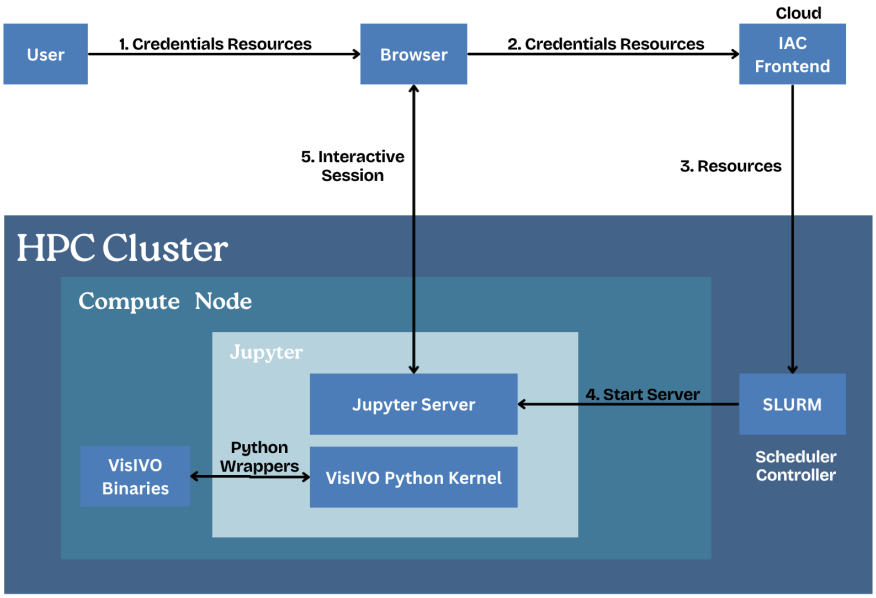


Figure 2 InterActive Computing VisIVO service implementation at Cineca.

and management of IAC services were a collaborative effort between Cineca and E4, using the ICE4HPC[7] software suite. Currently, IAC is available on Cineca's Galileo 100[8] supercomputing platform [22].

An interactive, browser-based interface to HPC delivers two key benefits.

1. Users can launch sessions directly on compute nodes at `https://jupyter.g100.cineca.it` with near-instant startup, bypassing the traditional SSH, batch-queue workflow and gaining seamless access to graphical visualization tools that are cumbersome in command-line-only environments.
2. It enables real-time control of running workflows – monitoring resource usage [15], inspecting intermediate results, and adjusting parameters on the fly – avoiding the unpredictable start times and slow feedback cycles inherent to batch processing.

The Interactive Access and Computing (IAC) framework delivers a secure, end-to-end interactive HPC experience by routing user authentication and resource requests from a local browser to a cloud-hosted frontend that isolates the web surface, which then forwards requests to Slurm to allocate a compute node and launch a Jupyter server with a VisIVO-enabled Python kernel, as shown in the UML diagram of Figure 2. Users work continuously in notebooks on that node – running heavy computations directly on HPC resources while monitoring usage and visualizing intermediate results in real time. Jupyter is chosen for its single-user server–client model, rich plugin ecosystem (e.g., monitoring tools, featured dashboards, and interface with other services via jupyter-server-proxy), and broad user familiarity. Security is enforced by executing orchestration on a dedicated VM in Cineca's cloud, running HPC operations as standard user-space Slurm jobs (no root privileges required on the cluster side), and applying two-factor authentication alongside continuous monitoring, vulnerability assessments, and penetration testing.

[7] ICE4HPC, `https://www.e4company.com/en/ice4hpc/`
[8] Galileo 100 infrastructure: `https://www.hpc.cineca.it/systems/hardware/galileo100/`

5 Methodology

Integrating a dedicated VisIVO Jupyter kernel into the IAC replaces the old ssh-and-module-load workflow on G100, eliminating command-line setup and extra steps to fetch PNG outputs. Users now launch an IAC session, select "VisIVO," and run commands immediately; outputs are saved to the session and viewable in the web interface. This streamlines visualization, improves accessibility, and reduces friction – even though VisIVO, written in C++, isn't a native Jupyter target – details of which are described next.

5.1 VisIVO wrappers

VisIVO is implemented in C/C++, and its compilation generates standalone executable binaries. Since Jupyter cannot directly embed such binaries in its interface; integration is achieved through a dedicated Python environment. To enable this, we developed a custom Python wrapper[9] that bridges Python with the VisIVO binaries, allowing users to execute VisIVO commands seamlessly within a Jupyter session. The Python wrapper is intended to execute a specific command while also handling essential tasks, such as invoking the CLI, logging activities, and verifying the function arguments.

The wrapper provides command-specific functions – VisIVOImporter, VisIVOFilter, and VisIVOViewer – that share a common design pattern (here below we will show the importer as an example), enabling seamless execution of VisIVO commands from a Python session.

```
def importer(input_file, *flags, mpi=False, tasks=None, fformat=None, out=None,
    volume=None, compx=None, [...]):
```

The design deliberately hides the VisIVO command-line from users delegating the VisIVO command line execution to an ad-hoc function (`_corefunction`):

```
command = "VisIVOImporter"
_corefunction(command, input_file, *flags, mpi=mpi, tasks=tasks, options=options)
```

The `_corefunction` validates the types of options passed from the outer wrapper, converts booleans into flag-only options and all other parameters into key–value options, then appends the input file to finalize the command; execution runs either with MPI or in serial mode based on the `mpi` and `tasks` settings. Additionally, integrating the *Pillow* library with the VisIVO kernel improves image handling – especially PNGs – by rendering outputs directly inside the Jupyter session, eliminating command-line viewers and manual file retrieval. This makes image generation and inspection faster and simpler, so users can stay focused on analysis rather than setup.

5.2 IAC Deployment

To deploy the framework, VisIVO was first compiled system-wide, with a Spack [10]. A Spack module enforces the OSMesa backend across graphical libraries such as VTK and GLEW. This enforcement ensures correct image output in IAC through off-screen rendering with OSMesa, which is crucial because the system lacks an active windowing system and only the web front end is operational. Then, a *conda* environment was provisioned with *Ansible* [13] on the IAC backend nodes to host the VisIVO Python wrappers (described in the previous Section 5.1), optionally adding Pillow, NumPy, and Matplotlib for smoother

[9] VisIVO Python Wrapper: `https://github.com/VisIVOLab/VisIVOPythonWrapper`

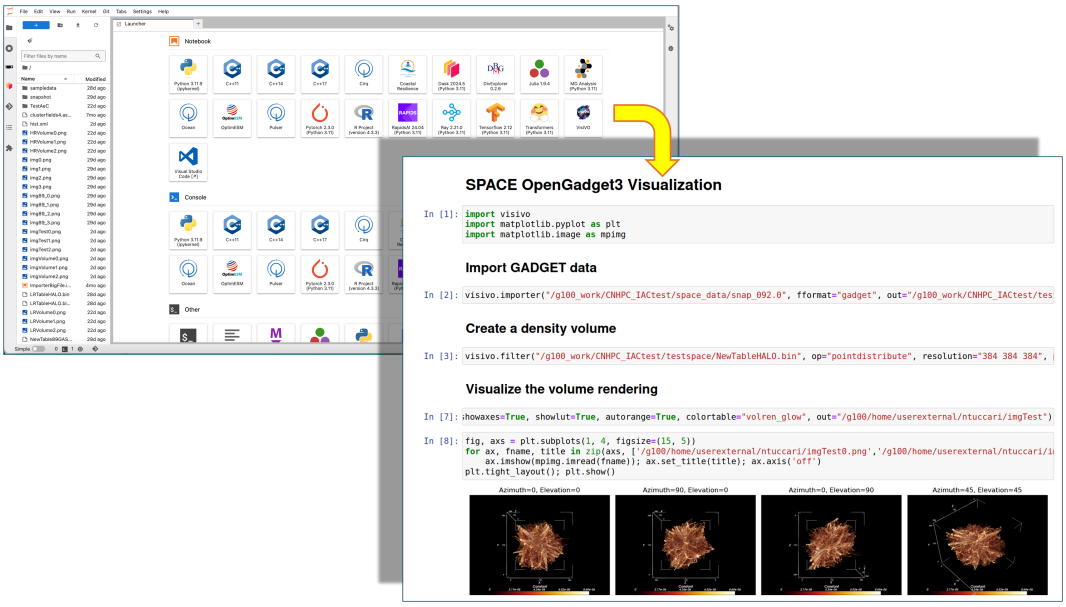


Figure 3 IAC VisIVO framework deployment and sample notebook to visualize OpenGadget3 data.

visualization. Ansible deploys the conda environment to each backend node's local storage (not the parallel file system) to avoid login slowdowns from many small file reads during Jupyter initialization; the inventory explicitly lists all service nodes, and the login node acts as the Ansible control host.

After that, the VisIVO Python wrappers were installed. With the wrappers in place, users can invoke VisIVOImporter, VisIVOFilter, and VisIVOViewer directly from Python, simplifying analysis and visualization workflows. Finally, the Jupyter ipykernel was customized to execute a bash prolog that loads the Spack VisIVO module and exports the necessary environment variables, making the VisIVO kernel immediately available in notebook and console sessions.

This deployment is being tested within the SPACE[10] project to interactively visualize the cosmological snapshots of two simulation codes, namely OpenGadget3 [12] and ChaNGa [14], in GADGET and Tipsy format respectively, as shown in Figure 3.

6 Automated pipeline using VisIVO

A second target for our codebase is pipeline-style automation, which is essential for high-throughput science (e.g., weather forecasting, tsunami alerts, satellite data management), where multiple servers interact in cascades that eventually trigger HPC jobs. Because initiating these interactions depends on site-specific HPC access policies, no standard method exists; in our earlier IAC use case, we overcame this by using Cloud–HPC coupling, which allows us to expose web services typically disallowed on HPC clusters. Extending that idea, we propose embedding standard web REST APIs into HPC workflows to provide secure, scalable, and reproducible automation from submission to result retrieval. REST makes

[10] SPACE project: `https://www.space-coe.eu/`

services client-agnostic, so requests can be sent from the command line (e.g., curl) or any web front end without server-side changes. Similar needs arise in projects like Terabit [24, 25] (rapid earthquake simulations) and GAIA [17] (post-processing large astronomical datasets), where pipelines coordinate data flows from remote sources (telescopes, seismic sensors) through interconnected steps but still lack a general approach – REST APIs are a strong candidate standard. To prototype this, we used our VisIVO codebases to simulate a pipeline that automatically post-processes simulation data using the cloud-hosted VisIVO REST API.

6.1 Deployment as a service

To achieve integration with pipelines via the cloud-based REST API, we integrated Flask [18] into our original code, i.e., the VisIVO Python wrappers described in Section 5.1. The choice of Flask is a perfect match for the implementation. Flask enables the rapid deployment of a web server using minimal Python code.

```
app = Flask(__name__)
if __name__ == "__main__":
app.run(host='0.0.0.0', port=5000, debug=True)
```

It processes requests and generates responses based on the application's configuration. Developers can specify the server's network address and port. In development mode, the application reloads automatically upon code modification, ensuring immediate updates, and provides detailed error messages through an interactive debugger.

```
@app.route('/some_endpoint', methods=['POST', 'GET'])
def some_endpoint():
```

Flask provides function decorators that make it easy to implement REST API endpoints. This feature aligns well with our starting code, which is organized around three core functions. Using Flask decorators, we can clearly map each function to a specific endpoint, simplifying the creation of REST APIs for the cloud-hosted server.

The REST API runs on a virtual machine in the OpenStack-based Cineca ADA cloud[11]; this is because bare-metal execution on the HPC cluster of web services is typically discouraged due to an higher probability of security issues that might affect such services, with the risk of compromising the security of the cluster. We set up VisIVO inside a Docker container dedicated to this service; while not strictly necessary, this cloud-friendly approach offers greater convenience and portability compared to a traditional installation. Alongside the container, we created a dedicated Python virtual environment to host the server, ensuring that REST API dependencies are cleanly managed. Together, these steps deliver an isolated, efficient backend layer specifically designed to support the reliable, scalable use of the application's REST API.

The primary goal is to provide a language-agnostic interface that abstracts system complexity via standard REST APIs. Currently, the "Import" action operates only on the cloud web server which serves as the system's control layer for handling user input, request validation, and secure REST interactions. To fully realize our aim, we plan to offload these operations to the HPC cluster. In the future, with shared storage, computation will be colocated with data – improving performance, scalability, and reliability on high-performance infrastructure.

[11] ADA cloud, `https://www.hpc.cineca.it/systems/hardware/ada-cloud/`

7 Conclusions and Future Works

This study describes how the VisIVO scientific visualization framework was integrated into Cineca's InterActive Computing (IAC) service to create an interactive, scalable, and reproducible visualization environment directly on HPC systems. The integration transformed VisIVO from a command-line, batch-based tool into a web-accessible, browser-driven platform that supports complex 3D visualizations within Jupyter notebooks using GPU nodes. Python wrappers and a dedicated Jupyter kernel make it easier for users to interact and run VisIVO commands manual configuration or module loading. These updates speed up visualization, provide clear HPC access, and help scientists analyze data more quickly. Cloud-based RESTful APIs built with Flask now let users run VisIVO remotely through web services, which simplifies the backend and makes it easier to connect with other applications. Overall, this paper showcases how conventional visualization software can be adapted into modern HPC-native solutions that combine high performance with user-friendly accessibility. The framework was tested on real astrophysics workloads, such as the ones from the SPACE project, and handled them smoothly, efficiently scaling and reliably visualizing very large, complex datasets. These trials demonstrated its ability to connect traditional batch supercomputing with real-time, interactive analysis.

In the future, the next step is to exploit the cloud-hosted REST API with VAST's multi-protocol shared storage so the web front end can read cluster-generated data directly – no data movement – possibly via VAST's object-storage interface for extra flexibility. The REST API will require proper authentication, likely token-based. Once shared storage is active, a REST workflow similar to IAC can submit VisIVO jobs to the cluster's batch system, moving computation to the data. This should boost throughput and I/O efficiency while keeping security tight by not exposing any web server from the cluster.

Declaration on Generative AI

During the preparation of this work, the authors made use of OpenAI's ChatGPT and Google's Gemini for paraphrasing, rewording, and checking grammar and spelling. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

1 James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. In *Visualization Handbook*, pages 717–731. Elsevier, 2005. doi:10.1016/B978-012387582-2/50038-1.

2 Sadaf Alam, Javier Bartolome, Sanzio Bassini, Michele Carpene, Mirko Cestari, Frederic Combeau, Sergi Girona, Stefano Gorini, Giuseppe Fiameni, Björn Hagemeier, Andreas Herten, Nikoleta Kiapidou, Wouter Klijn, Dorian Krause, Jacques-Charles Lafoucriere, Cerlane Leong, Thomas Leibovici, Thomas Lippert, Colin McMurtrie, Pavel Mezentsev, Anne Nahm, Boris Orth, Dirk Pleiter, Thomas Schulthess, Benedikt von St. Vieth, Debora Testi, and Gilles Wiber. Fenix: Distributed e-infrastructure services for ebrains. In Katrin Amunts, Lucio Grandinetti, Thomas Lippert, and Nicolai Petkov, editors, *Brain-Inspired Computing*, pages 81–89, Cham, 2021. Springer International Publishing.

3 Sadaf R Alam, Javier Bartolome, Sanzio Bassini, Michele Carpenè, Mirko Cestari, Frederic Combeau, Sergi Girona, Stefano Gorini, Giuseppe Fiameni, Björn Hagemeier, et al. Archival data repository services to enable hpc and cloud workflows in a federated research e-infrastructure. In *2020 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud)*, pages 39–44. IEEE, 2020.

4 Shafqat Alam, Juan Bartolome, Stefano Bassini, Massimo Carpene, Marco Cestari, Franck Combeau, Santi Girona, Stefano Gorini, Giuseppe Fiameni, Benedikt Hagemeier, et al. Fenix: Distributed e-infrastructure services for ebrains. In Katrin Amunts, Lucio Grandinetti, Thomas Lippert, and Nikolay Petkov, editors, *Brain-Inspired Computing*, pages 81–89. Springer International Publishing, 2021. doi:10.1007/978-3-030-70710-8_7.

5 Ugo Becciani, Eva Sciacca, Alessandro Costa, Piero Massimino, Costantino Pistagna, Simone Riggi, Fabio Vitello, Catia Petta, Marilena Bandieramonte, and Mel Krokos. Science gateway technologies for the astrophysics community. *Concurrency and Computation: Practice and Experience*, 27(2):306–327, 2015. doi:10.1002/CPE.3255.

6 Ugo Becciani, Eva Sciacca, Antonio Costa, Paola Massimino, Caterina Pistagna, Salvatore Riggi, Fabio Vitello, Claudio Petta, Massimo Bandieramonte, and Michalis Krokos. Science gateway technologies for the astrophysics community. *Concurrency and Computation: Practice and Experience*, 27(2):306–327, 2015. doi:10.1002/cpe.3280.

7 Anthony GA Brown, Antonella Vallenari, TJDBJH Prusti, Jos HJ De Bruijne, Carine Babusiaux, Coryn AL Bailer-Jones, Michael Biermann, Dafydd Wyn Evans, Laurent Eyer, Femke Jansen, et al. Gaia data release 2-summary of the contents and survey properties. *Astronomy & astrophysics*, 616:A1, 2018.

8 Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean M. Favre, and Paul Navrátil. Visit: An end-user tool for visualizing and analyzing very large data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pages 357–372. Chapman and Hall/CRC, October 2012. doi:10.1201/b12985.

9 Antonio Corradi, Giuseppe Di Modica, Luca Evangelisti, Anna Fiorini, Luca Foschini, and Luca Zerbini. Hs-autofit: A highly scalable autofit application for cloud and hpc environments. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6. IEEE, 2020. doi:10.1109/ISCC50000.2020.9219556.

10 Todd Gamblin, Matthew LeGendre, Michael R Collette, Gregory L Lee, Adam Moody, Bronis R De Supinski, and Scott Futral. The spack package manager: bringing order to hpc software chaos. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–12, 2015. doi:10.1145/2807591.2807623.

11 C Gheller, M Comparato, and U Becciani. Visivo: interoperable visualization and data analysis for the vo. In *Astronomical Data Analysis Software and Systems XV*, volume 351, page 29, 2006.

12 Frederick Groth, Ulrich P Steinwandel, Milena Valentini, and Klaus Dolag. The cosmological simulation code opengadget3–implementation of meshless finite mass. *Monthly Notices of the Royal Astronomical Society*, 526(1):616–644, 2023.

13 Lorin Hochstein and Rene Moser. *Ansible: Up and Running: Automating configuration management and deployment the easy way.* " O'Reilly Media, Inc.", 2017.

14 Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V Kale, and Thomas Quinn. Massively parallel cosmological simulations with changa. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12. IEEE, 2008.

15 Morris A Jette and Tim Wickberg. Architecture of the slurm workload manager. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 3–23. Springer, 2023. doi:10.1007/978-3-031-43943-8_1.

16 Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks, 2016.

17 Timo Prusti, JHJ De Bruijne, Anthony GA Brown, Antonella Vallenari, C Babusiaux, CAL Bailer-Jones, U Bastian, M Biermann, Dafydd Wyn Evans, L Eyer, et al. The gaia mission. *Astronomy & astrophysics*, 595:A1, 2016.

18 Kunal Relan. Building rest apis with flask. *Building REST APIs with Flask*, 2019.

19 Eva Sciacca, Valentina Cesare, Nicola Tuccari, Fabio Vitello, Iacopo Colonnelli, Camelita Carbone, Emiliano Tramontana, and Ugo Becciani. Toward a portable and reproducible high-performance visualization for astronomy & cosmology, September 2024. doi:10.5281/zenodo.13858498.

20 Eva Sciacca, Mel Krokos, Cristobal Bordiu, Carlos Brandt, Fabio Vitello, Filomena Bufano, Ugo Becciani, Mario Raciti, Giuseppe Tudisco, Simone Riggi, et al. Scientific visualization on the cloud: the neanias services towards eosc integration. *Journal of Grid Computing*, 20(1):7, 2022. doi:10.1007/S10723-022-09598-Y.

21 Eva Sciacca, Mario Raciti, Mel Krokos, Benjamin Kyd, et al. Science gateways in eosc: The neanias visualisation gateway. In *Proceedings of the 14th International Workshop on Science Gateways*, 2023.

22 Eva Sciacca, Nicola Tuccari, Umer Arshad, Fabio Pitari, Giuseppa Muscianisi, and Emiliano Tramontana. Interactive high-performance visualization for astronomy and cosmology. *arXiv preprint arXiv:2510.04665*, 2025. doi:10.48550/arXiv.2510.04665.

23 Nicola Tuccari, Eva Sciacca, Fabio Vitello, Iacopo Colonnelli, Yolanda Becerra, Enric Sosa Cintero, Guillermo Marin, Milan Jaros, Lubomir Riha, Petr Strakos, et al. High performance visualization for astrophysics and cosmology. In *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 443–450. IEEE, 2025.

24 Elisa Zuccolo, Giorgio Bolzon, Fabio Pitari, Ileana Elizabeth Monsalvo Franco, Chiara Scaini, Valerio Poggi, Chiara Smerzini, Stefano Salon, et al. Urgentshake: An hpc system for near real-time physics-based ground shaking simulations to support emergency response efforts. In *EGU2025*. Copernicus Meetings, 2025.

25 Elisa Zuccolo, Giorgio Bolzon, Fabio Pitari, Lucía Rodríguez Muñoz, Chiara Scaini, Manuela Vanini, Valerio Poggi, and Stefano Salon. Advancing rapid response to earthquakes with tiered physics-based ground-shaking simulations: The urgentshake system. *Seismological Research Letters*, 2025.

Inter-Procedural Strength Reduction for Embedded Systems

Giovanni Agosta
Politecnico di Milano, Italy
Consorzio Interuniversitario Nazionale per l'Informatica, Roma, Italy

Abstract

Embedded systems often rely on code generated from model-based design tools, which can result in inefficient implementations due to the loss of high-level semantic information during code generation. This paper explores an inter-procedural extension of the strength reduction transformation, traditionally applied within loops, to optimize repeated computations across function calls. The proposed technique identifies parameters acting as counters and replaces costly operations – such as exponentiation or multiplication – with incremental updates based on recurrence relations, using static variables to preserve state between calls. We formalize the transformation, discuss its applicability conditions, and analyse tradeoffs between computation and memory access costs. Experimental evaluation on ARMv8, AVR microcontrollers, and x86_64 platforms demonstrates significant speedups for power operations (up to 9 ×), while highlighting limitations for simpler operations due to memory overhead.

2012 ACM Subject Classification Computer systems organization → Embedded software; Software and its engineering → Compilers

Keywords and phrases Compiler Optimization, Strength Reduction

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2026.4

1 Introduction

The design of embedded system is a complex task [4], that requires specialized knowledge both on the programming of resource constrained platforms, often not supporting the more widely known APIs used in general purpose programming (e.g., POSIX), and of a specific application domain, usually dealing with sensing or actuation within a particular environment or device – e.g., appliances, white goods, or vehicles. As a result, the development process requires a small team of experts with in depth knowledge and wide expertise to achieve good performance and/or energy consumption [3, 11]. This approach however does not scale to large corporate environments, where development is distributed across large teams where each member has a focused competence and limited insight on the overall design process [3, 4, 11]. To support this latter scenario, model-based design tools, visual programming and code generation environments are employed, such as Simulink or LabVIEW [16]. These tools generate code (typically in the C language) requiring significant post-generation optimization. This is often hampered by the loss of information originally expressed in the high level model but not preserved in the generated C code. While usually these tools are closed source, it is possible to observe such effects on similar open source tools and language, such as Modelica, for which it is known that compiling through C language leads to a loss of information that is damaging for performance [1]. At the embedded software level, another example is that of physical meaning of data types, which can provide useful information for value range analysis [12].

A solution to these issues would be to perform code optimization on the high-level representation of the model. This is indeed a solution that would be supported by the rise of MLIR [15], which enables domain-specific dialects to represent specific semantics of the high-level model. While the idea has been proven at least at an exploratory level [6], the most


© Giovanni Agosta;
licensed under Creative Commons License CC-BY 4.0
17th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 15th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2026).
Editors: Davide Baroffio, Paola Busia, Lev Denisov, and Nitin Shukla; Article No. 4; pp. 4:1–4:10

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

commonly employed tools in the industry are proprietary, and do not have MLIR back-ends. Thus, at the present time, alternative options should be investigated, working at the level of the C compiler.

Often, embedded systems perform repetitive tasks [13, 18] in the form of a "main loop", within which a series of activities, represented by subprograms, are carried out. The "function call within a loop" pattern makes code less convenient to optimize, as compilers rarely apply inter-procedural optimization, yet many loop-based optimizations are available, e.g, loop-based strength reduction [5]. When *function inlining* is convenient, it is usually the easiest way to enable inter-procedural optimization [19]. However, several factors, particularly code size expansion, can hamper the widespread use of inlining in embedded software compilation.

In this paper, we explore the tradeoffs in applying inter-procedural loop-based strength reduction, showing how, under certain conditions, it can actually provide significant speedups, but also highlighting the need to perform careful profiling or static analysis to avoid de-optimizing effects due to trading off computation for memory access costs.

The rest of this paper is organized as follows. In Section 2 we introduce the background notions of strength reduction and its variants. In Section 3 we discuss the proposed inter-procedural strength reduction transformation, while in Section 4 we provide experimental data to support the proposed transformation. Finally, in Section 5 we draw some conclusions and highlight future research directions.

2 Background

Strength reduction is a well-known compiler transformation that consists in replacing an operation with a semantically equivalent, but less costly, one [5]. In its simplest form, a direct replacement of a costly operation with a less costly one can be directly performed, as shown in the following examples:

$$\begin{aligned} x \leftarrow y^2 &\Rightarrow x \leftarrow y \times y \\ x \leftarrow y/2^n &\Rightarrow x \leftarrow y >> n \quad (n > 0) \end{aligned}$$

The above examples show how the transformation is clearly optimizing, but also highlight some limitations – in both case, the transformation can be performed only for particular values – small constant exponents in the first case, and powers of 2 in the second.

A more complex, but also more widely applicable, extension is *loop-based strength reduction* [5], which is applied to arithmetic computations dependent on an induction variable i:

$$\begin{aligned} \forall i \in [i_0, i_n] : x \leftarrow y^i &\Rightarrow \forall i \in [i_0, i_n] : x \leftarrow x \times y \\ \forall i \in [i_0, i_n] : x \leftarrow y \times i &\Rightarrow \forall i \in [i_0, i_n] : x \leftarrow x + y \end{aligned}$$

Where an appropriate initial value of x, x_0, must be provided before the loop start. It is worth noting that, in the loop-based version, there is a tradeoff between computation and memory: the value of x at iteration j is now used to compute the value of x at iteration $j+1$, thus introducing a loop-carried dependency which was not present in the original code. The practical impact is, usually, an increased register pressure, since one register is employed to store the value of x and avoid memory accesses, which would lower the performance gain.

3 Inter-procedural Strength Reduction

The proposed inter-procedural version of the strength transformation is applied to a subprogram $f(p_0, \ldots, p_n, i)$ that is called repeatedly, with parameter i being computed in a way such that for successive calls f_n and f_{n+1}, $i_{f_{n+1}} = i_{f_n} + \delta$, where δ is a constant stride. In practice, it is usually sufficient to support the transformation for the unitary stride $\delta = 1$, but in principle is can be stated in a more general way. Also, while the simplest case is that of f being called within a loop, this is not necessary – the key point is that i behaves as a counter. In the case of a loop, this can be directly detected by a scalar evolution analysis (e.g., in LLVM [14]), or from inspection of the loop structure (e.g., in a higher-level compiler framework such as MLIR [15]). Otherwise, it may be inferred from an annotation coming from the programmer, or from a code-generation tool, allowing a wider applicability.

Assuming the compiler has identified f and i as potential targets for the transformation, the condition under which the transformation can be applied is that, within the body of f, there exists an operation $x \leftarrow g(v_0, \ldots, v_n, i)$, where $v_0, \ldots, v_n$ are constants. It must be also possible to define a recurrence-based transformation to replace repeated evaluations of g with an incremental update. Thus, a recurrence computation of g must be defined as:

$$g(v_0, \ldots, v_n, i_{n+1}) = g(v_0, \ldots, v_n, i_n + \delta) = T(g(v_0, \ldots, v_n, c_n), v_0, \ldots, v_n, \delta)$$

where T is a suitable operation.

As an example, if $g(c, i) = c^i$ with c constant, then T can be defined as $x_n \times c$ where $x_n = g(c, i_n)$.

Thus, given the original computation:

$$x_n \leftarrow g(v_0, \ldots, v_n, i_n)$$

it can be transformed by replacing the direct computation of g with the application of T, as long as the value of the previous computation of g is preserved:

$$\begin{aligned} x_0 &\leftarrow g(v_0, \ldots, v_n, i_0) \\ x_{n+1} &\leftarrow T(x_n, v_0, \ldots, v_n, \delta) \end{aligned}$$

where x must be a static variable – i.e., its contents must survive between different invocations of the same function.

3.1 Limitations

If subprogram f has other call sites that do not participate in the same counting, the a copy of f, f' must be generated. While this may appear as a contrived case, code generated by model-based tools may reuse the same block (for which code is generated in the form of a subprogram) across both related and unrelated calls. However, at the model level, the presence or absence of relation between the different call is usually detectable – whereas this might not be the case after the code is generated.

As mentioned above, the identification of the behaviour of parameter i might be difficult to perform at the level of the compiler for a general purpose language. However, the designer of an embedded system would often know such opportunities, and could add hints to the source code. Even in a proprietary model-based system, such a hint could be embedded in a comment or other metadata that is not discarded before code generation, and then detected by a suitably modified compiler.

The transformation also has the same limitations as the underlying strength reduction – e.g., even if the relation between g and T is mathematically correct, it might incur in inaccuracies due to floating point rounding errors. This, however, is generally accepted in compilers (e.g., the `-ffast-math` optimizations in `gcc`).

3.2 Performance Issues

The proposed transformation is in principle as effective as intra-procedural strength reduction, but suffers from an added effect, due to the nature of the stored temporary value. While in intra-procedural strength reduction the temporary value can be stored in a register, this is generally not possible in our case (except under the assumption of inter-procedural register allocation, which is not generally performed by the compiler). Instead, the value is stored as a static variable, thus requiring access to the memory. The cost of this access depends on the data type and the microprocessor architecture – it is minimal if the data size is the same or smaller than the word size, but can be larger otherwise. In this case, there may be interactions with transformations such as precision tuning, which alter the data type and size [21, 9].

Furthermore, the performance relation between the two operations g and T may be complex to understand. This is generally not the case in RISC architectures, where instructions have generally simpler semantics and performance are somewhat easier to understand, but it may easily happen in CISC architectures. However, since the vast majority of embedded software runs on RISC-based microcontrollers, this is a relatively minor issue.

Still, we investigate experimentally both issues in the Section 4.

3.3 Generalization to variable δ

While the standard strength reduction is applied with constant values of δ, it is possible to generalize it to some extent. However, this requires T to be designed for the corresponding g. For this purpose, we take into account the power function as g. We can perform a rewrite according to the fragment reported in Listing 1, which performs the strength reduced version of the power function only when the current parameter i_{n+1} is within $[-3, +3]$ of i_n.

Listing 1 Strength reduction applied to the pow function, with support for non-unitary induction variable steps.

```
DTYPE pow_sred(int i){
    static DTYPE x;
    static int old_i;
    if (i==0) { x=1; old_i=0; }
    switch (i-old_i) {
        case 0: return x;
        case 1: x*=C; break;
        case 2: x*=C*C; break;
        case 3: x*=C*C*C; break;
        case -1: x/=C; break;
        case -2: x/=C*C; break;
        case -3: x/=C*C*C; break;
      default : x=pow(C,i);
    }
    old_i=i;
    return x;
}
```

Intuitively, this transformation works as long as consecutive calls have values that are close – i.e., if the step in the induction variable is small. We will show this dependence in Section 4.3.1. In principle, the code could be adapted for larger steps, but the beneficial effects will be soon lost, since extending the range of supported steps increases the code size.

4 Experimental Evaluation

In this section we perform an experimental evaluation of the proposed transformation. The goal is to understand the opportunities for speedups compared to code optimized with existing transformations (i.e., those enabled by the -O3 flag), as well as to explore the interaction with precision tuning and different architectural models.

4.1 Benchmark Definition

To test the proposed transformation, we employ three kernels, representative of computations involving arithmetic operations of significant cost (powers and multiplications), both on floating point and integer numbers. Specifically, the three kernels model a sine wave generation, an amplitude modulation with exponential decay, and a cryptographic key evolution.

Sine wave generation. This kernel generates a sinusoidal waveform employing a tabulated sin function driven by a phase Φ computed as the product of a constant phase increment δ_Φ and the time step i.

$$\begin{aligned} \Phi &\leftarrow i \times \delta_\Phi \\ w &\leftarrow sin(\Phi) \end{aligned}$$

The computation is performed using double or single precision floating point numbers (we test both cases), and a 32-bit integer counter.

Amplitude modulation with exponential decay. This kernel applies an exponentially decaying envelope to a sine wave carrier, modelling a variety of fade-out controls:

$$\begin{aligned} \Phi &\leftarrow \Phi + 2\pi \times f \times \tfrac{i}{f_s} \\ \Phi &\leftarrow \Phi - 2\pi \, if \, \Phi \geq 2\pi \\ x &\leftarrow 100 \times 0.95^i \times sin(\Phi) \end{aligned}$$

where Φ is the phase, i is the induction variable representing discrete time steps, f is the carrier frequency and f_s is the sample rate. The operation to which strength reduction is applied is the power 0.95^i. The computation is performed using double or single precision floating point numbers, and a 32-bit integer counter.

Cryptographic key evolution. This kernel employs an integer exponentiation to derive a new key at every step i, starting from an initial prime p and the base key k_0.

$$k_{i+1} \leftarrow k_0 \oplus p^i \, mod \, 0xFFFFFFFF$$

This computation is performed employing a 32-bit integer counter and 64-bit integers for intermediate values.

Each kernel is deployed as a function, where i is one of the parameters. The function is repeatedly called from the main loop, with values of i increasing by 1 at each call.

Strength reduction is applied employing static variables to hold the partial computation, as described in Section 3.

4.2 Experimental Setup

For our experiments, we use a range of different platforms, both embedded and general purpose. While the general purpose platforms are not as interesting as embedded ones from the application perspective, they still provide some useful insights in understanding the behaviour of the transformation.

4.2.1 ARMv8 and x86_64 platforms

For ARM, the benchmarks have been compiled using `gcc` 12.2.0 (with target triplet arm-linux-gnueabihf and -O3 optimization level) on a Broadcom ARMv8 CPU with Cortex-A72 cores, mounted on a Raspberry PI 4 platform with the Raspbian operating system.

For x86_64, the benchmarks have been compiled using `gcc` 9.4.0 (with target triplet x86_64-linux-gnu and -O3 optimization level) on a Intel(R) Core(TM) i7-8750H CPU, mounted on a laptop with the Ubuntu operating system.

Each benchmark runs the main loop multiple 10000 times, to minimize the impact of initialization effects and other disturbances on the time measurement. All benchmarks have been run 36 times, reporting values for the mean case, to further remove disturbances.

Results are reported in seconds, measured as the user time with the Linux `time` utility.

4.2.2 AVR microcontrollers

To provide insights on the behaviour of the optimization on smaller microcontrollers, the benchmarks have been also compiled and simulated on the *Simulavr* simulator[1]. Simulavr is instruction-accurate and cycle accurate, and was successfully used as the starting point for tasks that require significant accuracy, such as side channel analysis [20].

For this experiment, we have employed `avr-gcc` version 5.4.0, targeting two cores from the "classic" AVR2 family (at90s4433) and the "enhanced" AVR5 (atmega328) family[2]. In the case of AVR devices, i was set to run in the $[0, 1000)$ range, and the outer loop is run only once, since the simulator does not suffer from disturbances.

Results are reported in simulated clock cycles from the start of the program to the reaching of the program exit point.

4.3 Experimental Results and Analysis

Table 1 reports the results for the ARMv8 platform. It can be seen that a speedup is produced in all cases. While the general expectation would be to see some further improvement when using smaller data types, this is not the case for ARMv8, since this is a 64-bit architecture. Reducing the data type size only adds some cast operations, which degrade the performance in the case of the amplitude modulation benchmark. `gcc` does not support quadruple precision floating points on ARMv8, which would likely show a degradation of performance due to the increased access to memory-stored static variables.

Table 2 reports the results for the Intel x86_64 platform. This platform offers the opportunity to also compare the use of long double. The interesting result here is the major slowdown obtained with double precision floating point compared to both `long double` and single precision `float`. A more in-depth analysis is required, but preliminarily the additional costs can be attributed to the memory moves which are performed here using mostly `movq` and `movsd`, whereas the single precision `float` version employs mostly `movss`.

[1] `http://www.nongnu.org/simulavr/`
[2] The compiler version employed is the one shipped with Simulavr.

Table 1 Experimental results on ARMv8, with precision tuning (double vs single precision floating point).

Benchmark	Precision	Baseline	Strength Reduction	Speedup
Sine Wave	Double	1.19s	1.15s	1.03
Amplitude modulation	Double	5.32s	2.03s	2.67
Sine Wave	Float	1.19s	1.15s	1.03
Amplitude modulation	Float	5.43s	2.32s	2.34
Key evolution	Int64	3.48s	0.36s	9.67

Table 2 Experimental results on x86_64, with precision tuning (quadruple vs double vs single precision floating point).

Benchmark	Precision	Baseline	Strength Reduction	Speedup
Sine Wave	Long Double	0.83s	0.89s	0.93
Amplitude modulation	Long Double	3.21s	1.64s	1.95
Sine Wave	Double	1.09s	1.08s	1.01
Amplitude modulation	Double	2.78s	8.60s	0.32
Sine Wave	Float	1.07s	1.08s	0.99
Amplitude modulation	Float	2.92s	1.37s	2.13
Key evolution	Int64	1.40s	0.16s	9.37

Table 3 shows the results for the two AVR platforms, which are essentially the same in terms of achievable speedups. The interesting point is that the speedup is below 1 (thus, a slowdown) for the sine wave generation example, where the target operation is a multiplication. This slowdown is driven by the need to store and load the 4-word floating point temporary needed for storing the old Φ, thus demonstrating the negative impact of memory-stored static variables.

Table 3 Experimental results on AVR (single precision only), reported in clock cycles.

Benchmark	CPU model	Baseline	Strength Reduction	Speedup
Sine Wave	atmega328	9995	12748	0.78
Amplitude modulation	atmega328	720350	232804	3.09
Key evolution	atmega328	588796	64788	9.08
Sine Wave	at90s4433	9790	12843	0.76
Amplitude modulation	at90s4433	1337724	441186	3.03
Key evolution	at90s4433	1160673	233912	4.96

4.3.1 Analysis of the generalized strength reduction

To understand the dependence of the generalized version of the transformation introduced in Section 3.3, we run on the ARMv8 platform a simple loop performing 0.95^v for 1M randomly generated values. The values v change with δ that are generated uniformly in a range $[-x, x+1]$, i.e., $v_{n+1} \leftarrow v_n + choice([-x, x+1])$, thus leading a slowly increasing value of v. Smaller ranges lead to higher likelihood of falling into one of the optimized cases, whereas larger ranges will lead to higher chances of taking the defaults, non-optimized case. The loop is further iterated 10k times, accumulating the total value to amplify execution times and errors (which in the end are negligible). Latency is measured with the standard C `time` library.

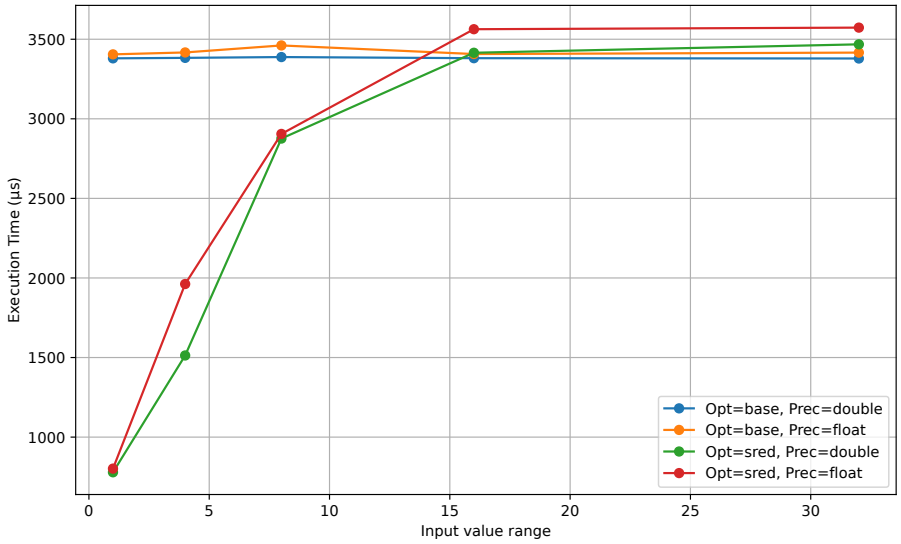


Figure 1 Variation of the execution time as a function of the input value range. The plot shows how the generalized strength reduction (Opt=sred) effectiveness depends on the careful identification of input range variability, whereas the non-optimized code (Opt=base) does not exhibit this dependency.

The graph in Figure 1 reports the execution time (in μs), as a function of the value of the input range extreme x. As expected, the non-optimized version (base) does not depend on the increment value range, whereas the optimized version behaves increasingly better as the increment range shrinks. The data type precision does not impact significantly in this case.

5 Conclusions

In this work, we explored the opportunity of applying the strength reduction transformation in an inter-procedural way in the context of embedded systems. The research is motivated by the need to optimize code that is automatically generated from proprietary model-based and visual programming tools, which often produce redundant code. We show, through an experimental analysis on synthetic benchmarks and a range of hardware platforms, how the proposed transformation can obtain significant speedups (3 to 9 $\times$ in strength reduction of power operations), but also some limitations, particularly the difficulty to predict performance in CISC architectures and interactions with precision tuning.

The latter issue, together with the identification of input value ranges, may benefit from integration with automated precision tuning tools at compiler level such as TAFFO [7, 17], leveraging either value range analysis [8] or profiling [10].

Future research will aim at exploring the impact on actual code generated from model-based tools, as well as integration in such a toolchain. Furthermore, a tradeoff comparison with an approach based on function inlining would be helpful to better understand the limitations of the proposed approach. Finally, strength reduction does affect the type of operations that are performed, so in the case of application to cryptographic operations, one

must also consider whether there is an impact on timing, to ensure that no side-channel information leakage is added. More in general, the interactions with compiler transformations aiming at implementation security are non-trivial [2].

References

1 Giovanni Agosta, Emanuele Baldino, Francesco Casella, Stefano Cherubin, Alberto Leva, Federico Terraneo, et al. Towards a high-performance modelica compiler. In *Proceedings of the 13th International Modelica Conference*, pages 313–320, 2019. doi:10.3384/ecp19157313.

2 Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. Compiler-based techniques to secure cryptographic embedded software against side-channel attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1550–1554, 2019. doi:10.1109/TCAD.2019.2912924.

3 Deniz Akdur. Skills gaps in the industry: Opinions of embedded software practitioners. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5):1–39, 2021. doi:10.1145/3463340.

4 Deniz Akdur, Bilge Say, and Onur Demirörs. Modeling cultures of the embedded software industry: feedback from the field. *Software and Systems Modeling*, 20(2):447–467, 2021. doi:10.1007/s10270-020-00810-9.

5 David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994. doi:10.1145/197405.197406.

6 Jiahong Bi, Guilherme Korol, and Jeronimo Castrillon. Leveraging the mlir infrastructure for the computing continuum. In *CPS Workshop 2024*, September 2024. URL: https://ceur-ws.org/Vol-3960/short5.pdf.

7 Daniele Cattaneo, Michele Chiari, Giovanni Agosta, and Stefano Cherubin. Taffo: The compiler-based precision tuner. *SoftwareX*, 20:101238, 2022. doi:10.1016/j.softx.2022.101238.

8 Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. Architecture-aware precision tuning with multiple number representation systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 673–678. IEEE, 2021. doi:10.1109/DAC18074.2021.9586303.

9 Stefano Cherubin and Giovanni Agosta. Tools for reduced precision computation: a survey. *ACM Computing Surveys (CSUR)*, 53(2):1–35, 2020. doi:10.1145/3381039.

10 Lev Denisov, Gabriele Magnani, Daniele Cattaneo, Giovanni Agosta, and Stefano Cherubin. The impact of profiling versus static analysis in precision tuning. *IEEE Access*, 12:69475–69487, 2024. doi:10.1109/ACCESS.2024.3401831.

11 Jack Ganssle. *The art of designing embedded systems*. Newnes, 2008.

12 W.H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, 1977. doi:10.1109/TSE.1977.231133.

13 Yew Ho Hee, Mohamad Khairi Ishak, Mohd Shahrimie Mohd Asaari, and Mohamad Tarmizi Abu Seman. Embedded operating system and industrial applications: a review. *Bulletin of Electrical Engineering and Informatics*, 10(3):1687–1700, 2021.

14 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004. doi:10.1109/CGO.2004.1281665.

15 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021. doi:10.1109/CGO51591.2021.9370308.

16 Peter Liggesmeyer and Mario Trapp. Trends in embedded software engineering. *IEEE Software*, 26(3):19–25, 2009. doi:10.1109/MS.2009.80.

17 Gabriele Magnani, Daniele Cattaneo, Michele Chiari, Giovanni Agosta, et al. The impact of precision tuning on embedded systems performance: A case study on field-oriented control. In *Proceedings of PARMA-DITAM 2021 (OASICS)*, volume 88, pages 1–13, 2021. doi:10.4230/OASIcs.PARMA-DITAM.2021.3.

18 John A Stankovic. Real-time and embedded systems. *ACM Computing Surveys (CSUR)*, 28(1):205–208, 1996. doi:10.1145/234313.234400.

19 Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 977–989, 2022. doi:10.1145/3503222.3507744.

20 Nikita Veshchikov and Sylvain Guilley. Use of simulators for side-channel analysis. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 104–112, 2017. doi:10.1109/EuroSPW.2017.59.

21 Yutong Wang and Cindy Rubio-González. Predicting performance and accuracy of mixed-precision programs for precision tuning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3623338.

LAMINA: An MLIR-Based Translation Library for Heterogeneous Quantum-Classical Compilation

Marco De Pascale[1,2]
Leibniz Supercomputing Centre, München, Germany

Mario Hernández Vera[1,2]
Leibniz Supercomputing Centre, München, Germany

Jorge Echavarria
Leibniz Supercomputing Centre, München, Germany

Muhammad Nufail Farooqi
Leibniz Supercomputing Centre, München, Germany

Martin Schulz
Technical University of Munich, Germany

Laura Schulz
Argonne National Laboratory, Lemont, IL, USA

Abstract

Quantum computing is increasingly integrated into High-Performance Computing (HPC) environments, where quantum processors act as specialized accelerators within hybrid workflows. The Munich Quantum Software Stack (MQSS) – a unified compilation and runtime framework for hybrid quantum–classical computing – provides the foundation for this integration. However, the growing heterogeneity of applications demands more flexible compilation tools. This work introduces an Multi-Level Intermediate Representation (MLIR)-based translation library that extends MQSS by enabling the conversion of CUDA-Quantum (CUDA-Q) (`quake`) dialects into machine learning–oriented MLIR representations compatible with modern compiler ecosystems. Leveraging MLIR's dialect-driven design, the library enables hardware-agnostic transformations, device-specific optimizations, and seamless integration with MQSS components. The proposed approach bridges quantum compilation and contemporary machine learning frameworks, facilitating GPU-accelerated circuit simulation, hybrid quantum–classical workflows, and heterogeneous execution, thereby advancing a unified compiler infrastructure for quantum computing.

2012 ACM Subject Classification Computer systems organization → Quantum computing; Software and its engineering → Just-in-time compilers; Software and its engineering → Retargetable compilers

Keywords and phrases HPCQC, MLIR, Quantum Computing, Heterogeneous Computing

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2026.5

Supplementary Material *Software (Source Code)*: `https://github.com/QCT-UEA-management/quake2iree.git` [7], archived at `swh:1:snp:c7b8f26bbeb0273ac7e029a8bba5845db219bdec`

Funding This work is supported by the German Federal Ministry of Research, Technology and Space (BMFTR) with the grants 13N15689 (DAQC), 13N16063 (Q-Exa), 13N16078 (MUNIQC-Atoms), 13N16187 (MUNIQC-SC), 13N16690 (Euro-Q-Exa), 13N16894 (MAQCS), European fundings 101136607 (CLARA), 101114305 (Millenion), 10111394-6 (OpenSuperQPlus), 101194491 (QEX), and the Bavarian State Ministry of Science and the Arts (StMWK) through funding, as part of MQV, Q-DESSI.

[1] Corresponding authors
[2] These authors contributed equally to this work.

© Marco De Pascale, Mario Hernández Vera, Muhammad Nufail Farooqi, Martin Schulz, and Laura Schulz;
licensed under Creative Commons License CC-BY 4.0

Acknowledgements The authors acknowledge Fabian Mora Cordero and Sunita Chandrasekaran from the University of Delaware for the initial discussion on this topic. The authors thank the anonymous reviewers for their constructive feedback and insightful suggestions.

1 Introduction

Quantum Computing (QC) is rapidly evolving from experimental systems into a critical component of modern HPC ecosystems. In this paradigm, quantum processors are envisioned as specialized accelerators that complement classical compute nodes, much like Graphical Processing Units (GPUs) or FPGAs, to solve computationally demanding subproblems in scientific and industrial applications. Realizing this hybrid model requires a robust software infrastructure that bridges high-level quantum programming frameworks with diverse quantum and classical hardware back-ends. The MQSS [4] has been developed to address this need. It provides a modular, open-source software ecosystem that integrates quantum devices into HPC environments and enables users to execute hybrid quantum-classical workflows efficiently through unified compilation, scheduling, and runtime management. Building on this foundation, the MQSS is evolving toward a fully dialect-agnostic, MLIR-based compilation framework that can integrate multiple quantum programming front-ends within a unified infrastructure. At present, MQSS supports the `quake` and `catalyst` dialects through its Quantum Resource Manager & Compiler Infrastructure (QRM & CI) component, which hosts the current `quake`-based compiler implementation. The QRM & CI repository provides the foundation for this compiler and demonstrates the ongoing transition toward a more general, extensible MLIR-driven design.

Despite this progress, the increasing heterogeneity of both hardware and software in quantum computing poses new challenges. Current compilers and runtime systems must not only support a variety of quantum technologies – such as superconducting qubits, trapped ions, and neutral atoms – but also interface with heterogeneous classical resources, such as CPUs, GPUs, and distributed HPC systems. Moreover, quantum circuit simulation and hybrid quantum–classical workflows typically require high-performance classical acceleration, particularly on GPU-based systems [13]. Existing compilation approaches, however, are typically tailored to specific platforms and lack the flexibility to dynamically target or optimize for different architectures and technologies. This limits their capacity to adapt dynamically to heterogeneous HPC infrastructures, preventing the efficient utilization of GPUs and other accelerators essential for quantum circuit simulation.

In addition to software and architectural heterogeneity, practical constraints from the physics of quantum simulation must be considered when targeting CPU or GPU back-ends. For state-vector simulators, the memory requirement scales as $M \approx 16 \times 2^n$ bytes for n qubits in double precision, limiting a 40 GB GPU to about 31 qubits and requiring distributed memory for larger systems; scalability here is bounded mainly by qubit count and inter-device bandwidth. Tensor-network methods become advantageous when the simulated states exhibit limited entanglement. Quantum states obeying the so-called *area law* for entanglement entropy – where subsystem entropy grows at most with boundary size – can be efficiently approximated by tensor-network states with moderate bond dimension [6]. In this regime, the memory cost of a matrix product state simulator scales as $O(nd\chi^2)$, with n the number of qubits, $d = 2$ the local dimension, and χ the bond dimension controlling retained singular values. For fixed χ, the memory grows linearly with n, enabling simulations of around 60 qubits on a single high-end GPU – such as an NVIDIA H100 with 96 GB of memory – compared to roughly 33 qubits for state-vector methods [13]. Because χ and accuracy depend

on circuit entanglement, highly entangled states still demand exponentially larger resources. Thus, circuits that generate weak or localized entanglement can be efficiently simulated on GPUs or CPUs using tensor-network techniques. In contrast, strongly correlated circuits remain limited by memory and communication costs.

To address these challenges, we observe that both Machine Learning (ML)-based models and quantum algorithms can be expressed in terms of linear-algebraic operations. This shared mathematical structure enables the use of those machine learning compilers such as Intermediate Representation Execution Environment (IREE) [15], which take tensor-based intermediate representations of ML models and, through successive MLIR passes, automatically map its execution to different hardware back-ends, including GPUs. Leveraging this commonality, we propose a new MLIR-based translation library for MQSS with the capability to translate quantum programs expressed in the `quake` dialect into a combination of MLIR dialects compatible with the IREE compiler. This extension leverages the modular, dialect-driven design of MLIR to enable hardware-agnostic transformations and produce intermediate representations compatible with IREE, which in turn performs device-specific optimizations and code generation. By integrating seamlessly with MQSS components such as the QRM & CI and the Quantum Device Management Interface (QDMI), the library extends the existing support of MQSS – which already interfaces with Quantum Processing Units (QPUs) based on superconducting, ion-trap, and neutral-atom technologies, as well as emulation platforms – by enabling workloads also to leverage GPU acceleration. In doing so, it enhances MQSS's ability to efficiently parallelize and accelerate quantum circuit simulation across heterogeneous compute resources, paving the way for a unified heterogeneous compiler that seamlessly combines quantum and classical compilation paths within a single software framework. Through the integration of MLIR-based lowering into the IREE compiler and the Open Accelerated Linear Algebra (OpenXLA) [16] ecosystem within the MQSS, this work advances the broader vision of building an interoperable, and extensible software framework that connects end users, HPC infrastructures, and diverse quantum technologies.

The remainder of this paper is organized as follows. Section 2 provides the necessary technical background, introducing the key compiler technologies – LLVM, MLIR, and the `quake` dialect – and describing the architecture of the MQSS together with the roles of IREE and OpenXLA in heterogeneous compilation. Section 3 presents the design goals and architecture of the proposed Linear Algebra MLIR Interface for Neutral Acceleration (LAMINA) translation library, explaining its integration within the MQSS framework and illustrating its operation through an example workflow. Finally, Section 4 (current implementation) details the `quake2iree` prototype module that realizes the cross-dialect translation from quantum to standard MLIR dialects.

2 Technical Background

2.1 LLVM, MLIR, and `quake`: Foundational Technologies for Hybrid Compilation

The design of our translation library builds on modular compiler infrastructure that underpins modern heterogeneous compilation workflows. To situate our contribution within this landscape, we briefly introduce three key components: LLVM, MLIR, and the `quake` dialect.

The LLVM project [11] established the foundation for reusable, language-agnostic compiler technology. At its core lies a typed, language-independent Intermediate Representation (IR) that enables extensive compile-time and runtime optimizations across diverse targets. LLVM's modular architecture allows the coexistence of multiple front-ends and back-ends, while its

static single-assignment (SSA) form and rich type system facilitate precise program analysis and transformation. Over time, LLVM has become the *de facto* substrate for modern compiler ecosystems, serving as the foundational back-end for languages such as C/C++/Objective-C (via the Clang frontend), Rust, and Swift; moreover, its concept of IR as a common language for all middle-layers is so powerful that it has been extended to the Quantum Itermediate Representation (QIR) [19] to support QC. QIR is used in MQSS as an exchange format between MLIR-based quantum dialects such as `quake` or `catalyst` and hardware-specific software back-ends, since it provides a hardware-agnostic format for expressing quantum operations.

While LLVM excels at low-level optimizations, it lacks mechanisms for representing higher-level program structure or domain-specific semantics. The MLIR framework [12] extends LLVM by introducing a multi-level approach that supports multiple abstraction layers within one infrastructure. MLIR defines dialects – domain-specific IR extensions that capture specialized semantics while remaining interoperable. This enables progressive lowering, where high-level representations are incrementally transformed into lower-level forms (e.g., LLVM IR) while preserving source structure and metadata. MLIR's design thus allows compiler developers to compose transformations that span from algorithmic descriptions to device-specific instructions, drastically reducing engineering effort and improving maintainability across heterogeneous targets.

Within the MLIR ecosystem, the `quake` dialect – developed as part of NVIDIA's CUDA-Q framework [14] – extends MLIR with a unified representation for quantum-classical programs. Serving as the quantum kernel IR in CUDA-Q, `quake` bridges high-level constructs (from C++ and Python) with low-level execution through a dual semantic model: *memory semantics* for runtime-managed qubits and *value semantics* where qubits behave as first-class SSA values. This structure enables efficient analysis of data dependencies and supports transformations such as circuit optimization and hybrid control flow integration. As an MLIR dialect abstracting quantum operations and types, `quake` has been adopted by MQSS as one of its primary intermediate representations for quantum kernels in its current implementation, forming the basis for target-independent optimization and lowering.

2.2 The Munich Quantum Software Stack

The MQSS constitutes the core software infrastructure of the Munich Quantum Valley (MQV) initiative, aiming to bridge quantum and classical computing within a unified high-performance environment. Developed jointly by the Leibniz Supercomputing Centre (LRZ) and the Technical University of Munich (TUM), the MQSS provides a modular, extensible, and open-source framework that connects users, compilers, and hardware through a common software layer. Its architecture is designed to integrate a diverse range of quantum back-ends, including superconducting, ion-trap, and neutral-atom devices – alongside high-performance classical systems, enabling the integration of QC into HPC, which is referred to as HPCQC. This allows running hybrid quantum-classical workflows that are tightly coupled to existing supercomputing infrastructures such as LRZ's SuperMUC-NG cluster. The MQSS embodies the broader MQV vision of integrating quantum computers as disaggregated accelerators within HPC facilities, moving beyond the current paradigm of isolated, cloud-based quantum resources.

At its core, the MQSS comprises three tightly interconnected layers, as illustrated in Fig. 1. The front-end layer currently provides adapters for major quantum programming frameworks such as Qiskit, Pennylane, CUDA-Q, and its own C/C++-based QPI [10], while its modular design allows additional front-ends to be integrated in the future. These

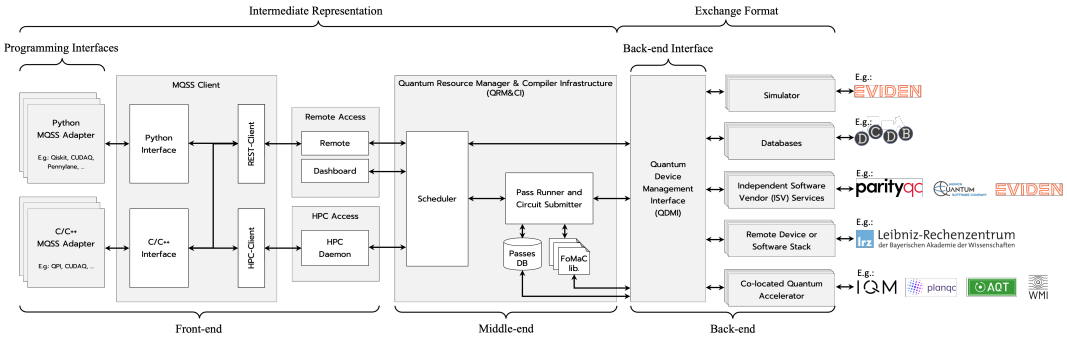


Figure 1 MQSS architecture overview: *MQSS Adapters* (e.g., Qiskit, CUDA-Q, PennyLane, and its native C-based Quantum Programming Interface (QPI)) submit gate- and pulse-based jobs to the *MQSS Client*, which handles automatic routing for both local HPC jobs and remote submissions. The QRM & CI encompasses MQSS's second-level scheduler, its Just In Time (JIT) LLVM-based compiler, and supporting libraries. QDMI exposes device capabilities, timing/granularity, and constraints to QRM & CI during JIT compilation and to the *MQSS Adapters* via the *MQSS Client* during runtime execution. Example *QDMI Devices* shown for illustrating target diversity: classical simulators, databases, ISVs, data centers, and superconducting, neutral atom, and trapped-ion quantum accelerators.

MQSS Adapters decouple programming models from the underlying runtime, allowing users to describe quantum circuits and hybrid workflows without depending on specific hardware or compilation paths. The middle-end layer – implemented through the QRM & CI – handles the quantum portion of submitted workloads, performing their compilation, optimization, and scheduling. It leverages the MLIR framework to represent quantum programs at multiple abstraction levels, thereby separating target-agnostic optimizations (such as circuit simplification, decomposition, and layout-independent rewriting) from target-specific transformations that adapt circuits to individual hardware back-ends. In its current form, the QRM & CI relies primarily on receiving from the Adapters the quantum circuits expressed in `quake` and `catalyst` [8] MLIR Dialects; QRM & CI lowers that representation to QIR or to device-oriented exchange format (such as OpenQASM). The back-end layer, implemented via the hardware-agnostic QDMI [18], is responsible for submitting the optimized circuits to the selected hardware executor and managing device interaction and feedback. The development of the present LAMINA library builds directly upon this foundation, extending the QRM & CI with new dialect conversion passes that connect the MQSS quantum compilation pipeline to heterogeneous classical back-ends through IREE and OpenXLA; see Sec. 2.3 for a description of both.

Finally, QDMI, as MQSS back-end layer provides a unified communication channel, with real hardware back-ends, made available as abstract devices, abstracting device-specific details through standardized queries, job, and session interface, and is complemented by the Figures-of-Merit and Constraints (FoMaC) libraries, which expose live device characteristics and support hardware-aware compilation.

This layered architecture enables MQSS to act as a flexible bridge between diverse programming frameworks and an expanding ecosystem of quantum technologies. However, hybrid workloads increasingly depend on heterogeneous compute resources. Large-scale quantum circuit simulations, variational hybrid quantum-classical algorithms, and Quantum Machine Learning (QML) applications rely heavily on GPU-accelerated classical computation. GPUs are particularly well-suited for state-vector evolution and tensor-network contraction,

where high-throughput linear algebra operations dominate runtime complexity. For example, simulators such as NVIDIA cuQuantum [1] and Google's TensorFlow Quantum [3] demonstrate that GPU-based acceleration can outperform CPU-only implementations by orders of magnitude for moderate circuit sizes, making them essential for algorithm benchmarking and noise modeling in near-term quantum research. Likewise, hybrid optimization loops – as used in algorithms such as the Variational Quantum Eigensolver (VQE) or the Quantum Approximate Optimization Algorithm (QAOA) – require tight coupling between classical optimizers running on HPC nodes or GPUs and quantum kernels executed on physical or emulated quantum devices. Furthermore, QC simulation remains a critical component for the quantum computing community at this stage of developmental maturity. These workloads highlight the growing need for a compilation and execution framework that can transparently target both quantum and classical accelerators while maintaining semantic consistency across architectures.

2.3 The IREE and OpenXLA Ecosystem

The IREE compiler and runtime framework, together with the OpenXLA ecosystem – which encompasses components such as `XLA` and the `StableHLO` dialect – are designed to enable efficient deployment of compute-intensive ML workloads across heterogeneous hardware. Both frameworks are built on the MLIR infrastructure and share a design philosophy centered on modularity, retargetability across heterogeneous hardware, and efficient execution. Initially developed for ML workloads, they provide an ideal foundation for compiling and executing quantum–classical hybrid applications, thanks to the fact that both ML models and quantum circuits can be expressed in terms of linear algebra.

IREE is an MLIR-native compiler and runtime environment that unifies compilation, optimization, and execution across diverse hardware targets. It supports both ahead-of-time (AOT) and JIT compilation, enabling flexible deployment on CPUs, GPUs, and dedicated accelerators such as Tensor Processing Unit (TPU) or emerging AI cores. At its core, IREE lowers MLIR modules through a sequence of dialect transformations and back-end-specific passes that progressively translate high-level operations into executable machine code for the selected target. IREE supports several input dialects, including `TOSA`, `MHLO`, and the standards `linalg`, `arith`, `math`, `tensor`, and `scf`, which form a rich foundation for representing numerical and control-flow operations common to both machine-learning and quantum workloads. Its modular runtime enables seamless integration with existing HPC infrastructures and supports heterogeneous CPU/GPU architectures – including NVIDIA and AMD – while providing low-latency execution via dynamic scheduling of overlapping compute and data-transfer workloads across devices.

Complementary to IREE, OpenXLA is an open ecosystem of compiler and runtime components designed for large-scale ML workloads, providing a bridge between MLIR-based front-ends and high-performance back-ends such as Accelerated Linear Algebra (XLA) and StableHLO. OpenXLA provides a rich ecosystem of optimizers for tensor and matrix operations, enabling sophisticated graph-level optimizations, automatic differentiation, and device-specific tuning. While initially developed for frameworks such as TensorFlow, PyTorch, and JAX, OpenXLA's modularity and MLIR-based design make it equally suitable for quantum pipelines. In particular, it enables seamless integration between parameterized quantum circuits and classical ML models, treating quantum kernels as differentiable computational primitives within machine-learning workflows. This is a key enabler for QML and the Variational Quantum Algorithm (VQA) [2], where optimization and training loops require tight coupling between quantum and classical computations.

3 Library Design and MLIR Translation

3.1 Design Goals

The LAMINA library is conceived as a modular bridge between quantum-oriented MLIR dialects and heterogeneous classical back-ends within the MQSS. Its principal goal is to enable the translation of CUDA-Q's `quake` dialect operations into MLIR representations compatible with IREE and OpenXLA. This allows quantum kernels to leverage the same optimization and deployment pipelines such as those provided by IREE and OpenXLA.

The design of LAMINA follows three main principles:

1. **Portability**: Support multiple compute architectures (Central Processing Unit (CPU), GPU, Tensor Processing Unit (TPU), or other accelerators) while remaining agnostic to vendor-specific details. The library should allow the same quantum kernel to be compiled and executed efficiently across heterogeneous HPC environments.
2. **Modularity and Extensibility**: Use MLIR's dialect-based design to implement a clean separation between front-end quantum dialects, intermediate transformation passes, and back-end lowering stages. This enables incremental extension to dialects such as `XQSS Pulse` [5], and supports external optimization passes developed by the community.
3. **Interoperability and Integration**: Ensure tight coupling with existing MQSS components – particularly the QRM & CI and QDMI – so that hardware-aware compilation and runtime control remain consistent across quantum and classical targets.

Through these design principles, LAMINA aims to expand the MQSS compilation pipeline to support targeting both quantum hardware and high-performance classical accelerators under a unified MLIR-based abstraction.

3.2 Architecture of the Translation Library

At a high level, LAMINA operates as a translation and optimization layer that consumes MLIR modules expressed in the `quake` dialect, produced by the MQSS CUDA-Q Adapter, the extended MQSS's QPI, or by translation passes applied to circuits expressed in the Catalyst MLIR dialect. This `quake` representation is then progressively lowered to MLIR dialects compatible with IREE or OpenXLA back-ends. The quantum compiler in QRM & CI applies translation passes as the final lowering before submission via QDMI; this allows all optimization and transpilation passes to be applied beforehand, which is helpful in transpiling the circuit to a specific gate set or for benchmarking use cases. The translation process consists of two stages:

1. **Cross-Dialect Mapping**: LAMINA performs a dialect-to-dialect transformation that maps `quake` operations to a composition of MLIR dialects. When targeting IREE, the set of MLIR dialects is composed of the standards `arith`, `math`, `tensor`, `scf`, and `linalg`. When targeting OpenXLA, `quake` is translated into first `scf`, `arith`, `memref` and `linalg`, the latter then lowered to the Stable High-Level Optimizer (`stablehlo`) Dialect [17]. Both mappings express quantum gates and measurement operations as tensor or linear-algebra primitives suitable for execution on classical hardware.
2. **Back-end Lowering and Code Generation**: then, LAMINA invokes IREE's or OpenXLA's compilation pipelines to lower the translated MLIR module to target-specific code. Depending on the execution context, this may result in GPU kernels or vectorized CPU instructions. The binaries are generated by the dedicated LAMINA MQSS component (See 3.3), which also handles submission via the HPC cluster scheduler.

This architecture ensures that the translation path from quantum kernels to classical accelerators remains transparent, verifiable, and reusable, enabling reproducible hybrid workflows where identical quantum programs can be benchmarked or simulated across different hardware back-ends.

3.3 Integration with MQSS Components

LAMINA integrates seamlessly into the existing MQSS infrustructure as three separate components, as a set of MLIR translation passes, by implementing the QDMI specifincation into the so called LAMINA QDMI Device and as a Figures-of-Merit and Constraints (FoMaC) library. As translation passes, LAMINA is available as a pass the MLIR dialect agnostic pass runner within QRM & CI that can apply to a circuit, alternatively to lower `quake` to QIR or to the device-native exchange format (see Fig 2).

During compilation QRM & CI communicates with the LAMINA FoMaC library to know whether or not there is a GPU with enough memory to fit the number of qubits in the circuit, and the estimated time to access it; the first information is dispatched to the LAMINA QDMI Device, the second back to the MQSS Client.

The LAMINA QDMI Device receives the translated circuit, the target GPU architecture, and cluster partition to which it belongs (both provided by the LAMINA FoMaC). It subsequently compiles to the target architecture using IREE or OpenXLA, then submits the resulting binary for execution via the available HPC resource scheduler. The LAMINA QDMI Device is also responsible for routing the results back to the MQSS Client that requested execution.

Via MLIR passes, FoMaC library and the LAMINA QDMI Device, LAMINA enables QDMI to serve as a unified interface for communicating with both quantum and classical execution resources. In the traditional MQSS workflow, quantum kernels compiled through the standard pipelines are dispatched to physical quantum devices via QDMI. LAMINA enriches this pipeline by interfacing to it classical compute resources in the same way as quantum resources. The powerful abstraction layer provided by QDMI handles both real and simulated execution targets through a consistent API. Users and system components can therefore submit, monitor, and retrieve results from quantum processors and classical accelerators uniformly, without requiring separate interfaces or modification to their source code. From an operational perspective, this allows classical HPC resources to serve as drop-in replacements for real devices, enabling reproducible benchmarking, regression testing, and performance studies under controlled simulation conditions.

LAMINA integration with MQSS is hence made in a way to reinforce MQSS philosophy of hardware-agnostic design, ensuring that all computational resources – quantum or classical can be orchestrated coherently within the same ecosystem.

3.4 Example Workflow: Pure Quantum Algorithm

Let us illustrate the ideas above with a simple example: submitting a pure quantum algorithm to a single GPU without the need to sync with classical computation, which we would call a hybrid algorithm. In the following, we assume the access to classical resources is managed by Slurm [9], a resource scheduler widely used in HPC centers. In our simple example, the workflow proceeds as follows.

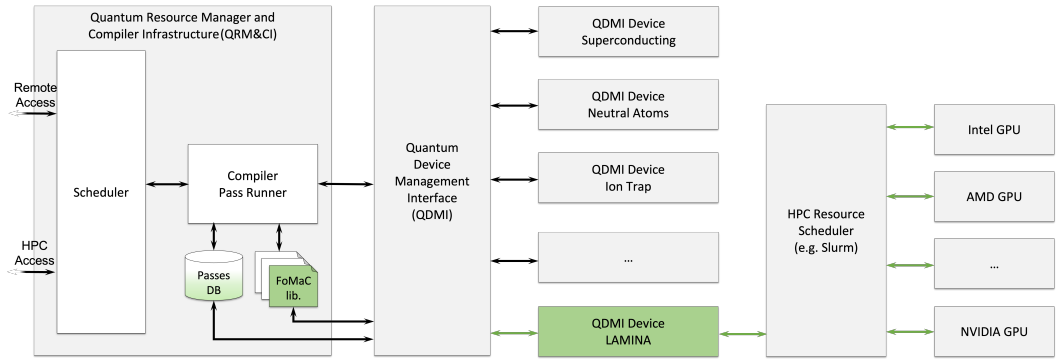


Figure 2 LAMINA integration within MQSS. The circuit reaches the Scheduler in QRM & CI via one submission channel. The dedicated translation passes in the Pass DB are applied by the Compiler Pass Runner to translate from `quake` to the input dialects for the LAMINA QDMI Device; the dedicated FoMaC gets information on the better GPU for the circuit at hand. Translated circuit and GPU details are dispatched to the LAMINA QDMI Device, to perform compilation and submission to the HPC resource scheduler.

1. **Front-end to `Quake`**: the user submits a quantum algorithm (quantum job), composed of a single quantum circuit, via one of the MQSS Adapters; as part of its responsibility, the MQSS Adapter lowers the circuit to `quake` representation and passes it to the QRM & CI component.
2. **Optimization and MLIR Dialect Translation**: in QRM & CI the quantum compiler optimizes the circuit. It applies all the passes required by the use-case, and as the latest MLIR-pass, it applies the translation from `quake` to the set of standard MLIR dialects compatible with IREE. During this step, thanks to the LAMINA FoMaC library which exploits the HPC cluster scheduler functionalities (e.g., the Slurm REST API), the target GPU and how to reach it via the HPC resource scheduler is identified.
3. **Loading the LAMINA QDMI Device and dispatching**: at this stage QRM & CI loads LAMINA QDMI Device related library and dispatches the MLIR representation of the circuit to it for further compilation to machine code.
4. **Target Architecture Identification and Compilation**: on receipt of MLIR circuit representation and target GPU architecture, the LAMINA QDMI Device executes the IREE compilation pipeline.
5. **Job Submission**: once the binary file has been prepared by IREE, it is ready for execution; the LAMINA QDMI Device submits its execution to the HPC cluster scheduler, specifying the resource selected for compilation.
6. **Returning results**: much like all the QPUs dedicated QDMI devices, the LAMINA QDMI Device waits for the job to complete; once that happens, it reports the results back to the QDMI Client which submitted the job (QRM & CI in this example), or, in case of errors, the related error messages.

4 Current Implementation: Cross-Dialect Mapping and the `Quake2IREE` Module

At the current stage of development, the implementation of the **cross-dialect mapping layer** within LAMINA is being prototyped through the standalone module `Quake2IREE`. This module provides a minimal but functional infrastructure to transform quantum programs expressed in the `quake` dialect – originating from the MQSS Adapters – into standard MLIR dialects that can participate in IREE's compilation and optimization pipeline.

The `Quake2IREE` project builds directly on the MLIR/LLVM framework and defines a complete conversion pass (`QuakeToStandard`) that lowers quantum-specific constructs to conventional MLIR representations. The pass is implemented as an MLIR *Conversion Pattern* combined with a dedicated *Type Converter* and is designed to integrate seamlessly into the LAMINA pipeline as its first transformation stage. Its purpose is to decouple quantum-specific abstractions from their physical semantics and re-express them in terms of tensor-based operations compatible with IREE's back-ends.

The core of the implementation resides in three modules that correspond to the standard structure of MLIR-based compiler projects:

- **Dialect Module:** Defines the intermediate representations used in the pipeline. This includes the `quake` dialect for quantum computation and the `Classical Compute` (`CC`) dialect for classical control and data structures, defined by NVIDIA as part of the CUDA-Q project. The module also provides shared utilities for type handling, attribute registration, and cross-dialect interoperability.
- **Conversion Module:** Implements the `QuakeToStandard` conversion pass, which performs type conversion and operation rewriting. The `QuakeToStandardTypeConverter` maps quantum-specific types such as `!quake.ref` (the type for a single qubit reference) and `!quake.veq` (the type for a vector of qubits) to tensor-based representations (e.g., `tensor<i1>` for the first and `tensor<?xi1>` for the second, using the MLIR `Tensor` Dialect). Conversion patterns such as `ConvertAlloca`, `ConvertVeqSize`, and `ConvertInitState` translate allocation (`quake.alloca`), size query (`quake.veq_size`), and initialization (`quake.init_state`) operations into their classical analogues.
- **Tool Module:** Provides the command-line utility `quake-to-standard-opt`, which serves as the standalone driver for the conversion pass and allows developers to test the transformation pipeline on individual MLIR modules. This component will later be incorporated into the LAMINA toolchain as the entry point for classical–quantum lowering workflows.

The transformation realized by the `Quake2IREE` module enables quantum programs to be represented entirely in terms of standard MLIR dialects (`arith`, `tensor`, `complex`, `func`, `scf`) and the `CC` dialect. By doing so, the resulting IR becomes compatible with existing MLIR optimization passes, facilitating the use of IREE's lowering infrastructure for heterogeneous targets. Conceptually, this step transforms quantum IR into a form that can be reasoned about using the same optimization strategies applied to classical ML workloads, paving the way for unified compilation across quantum and classical domains.

To ensure the correctness and robustness of the current implementation, a lightweight Python-based test suite has been developed. The script collects per-test results and reports aggregated statistics, automatically failing the CI pipeline if any individual test fails. This testing framework provides an early validation layer for new conversion patterns and will later be extended with semantic checks comparing `quake` and lowered IR behaviors. It is integrated into the project's build system and serves as the foundation for continuous integration testing as additional dialect mappings are added.

Although `Quake2IREE` currently operates as an independent prototype, it represents the foundational stage of LAMINA's cross-dialect translation layer. The next development milestone will integrate this module directly into the MQSS compilation pipeline via the QRM & CI, enabling end-to-end workflows from CUDA-Q front ends to IREE-compatible MLIR modules.

5 Conclusions and Outlook

This work presented the design and early implementation of LAMINA, a new MLIR-based translation library that extends the MQSS toward heterogeneous quantum-classical compilation. By introducing a translation path from CUDA-Q (`quake`) dialects to MLIR representations compatible with IREE and OpenXLA, LAMINA enables hybrid workloads to target both quantum and classical accelerators through a unified compiler infrastructure. The approach leverages MLIR's modular dialect architecture to bridge quantum abstractions with mature machine-learning compiler ecosystems, fostering cross-domain interoperability between quantum computing, classical HPC, and AI-oriented hardware.

The integration of LAMINA within MQSS represents an essential step toward realizing the broader vision of the MQV initiative: building a software ecosystem in which quantum devices function as integral components of HPC infrastructures. Functionally, LAMINA extends the existing QRM & CI of MQSS, introducing cross-dialect translation capabilities that connect its MLIR-based compilation pipeline with classical back-ends and ML-oriented runtimes. Through its coupling with the QRM & CI and QDMI, LAMINA provides a coherent runtime model where real and virtual devices can coexist, enabling developers to benchmark, simulate, and deploy quantum workloads on heterogeneous resources such as CPUs and GPUs without changing workflow semantics. The resulting interoperability creates opportunities for accelerating quantum circuit simulation, hybrid algorithm optimization, and QML applications on classical accelerators, while maintaining full compatibility with MQSS's existing runtime ecosystem.

From a developer's perspective, current quantum software workflows are often fragmented and tightly coupled to specific execution back-ends. In practice, developers frequently rely on different simulators for CPU and GPU execution, use back-end-specific intermediate representations, or maintain separate code paths for simulation and hardware execution. This fragmentation complicates rapid prototyping and benchmarking, as developers often need to adapt or rewrite quantum circuits when moving from local CPU-based simulation to GPU-accelerated simulation or to execution on physical QPUs. The approach presented in this work directly addresses these challenges by providing a unified compilation path for quantum circuits that is agnostic to the underlying classical execution backend. By lowering quantum kernels to a common MLIR-based representation and leveraging on ML compilers, developers can target CPU- or GPU-based simulation through the same interface and reuse the same workflow within the broader MQSS stack.

At the current stage, the `quake2iree` prototype demonstrates the feasibility of translating quantum-specific constructs into standard MLIR dialects, allowing their integration into IREE's and OpenXLA's compilation pipelines. Future work will focus on several key directions. First, the ongoing integration of `quake2iree` into the MQSS compilation pipeline will enable automated end-to-end workflows from front-end quantum languages to executable IREE modules. Second, LAMINA will be extended with hardware-aware optimization passes that exploit information from FoMaC and QDMI to perform dynamic scheduling and device-specific tuning. Third, we will extend IREE to support a hardware back-end for which it can compile, including the Intel Data Center GPU Max 1550 (formerly known as the Intel Ponte Vecchio GPU), thereby furthering its vendor-agnostic characteristic. Benchmarks will accompany each of the directions listed above to assess how well the MQSS with LAMINA performs in comparison with other frameworks accelerating quantum simulation on GPUs, such as NVIDIA's CUDA-Q. We will publish the results of these benchmarks in follow-up

papers. Finally, as part of MQV's broader roadmap, LAMINA will serve as a foundation for exploring unified compiler techniques that couple quantum kernels with AI-oriented computation graphs, contributing to the convergence of quantum, HPC, and ML ecosystems.

References

1 Harun Bayraktar, Ali Charara, David Clark, Saul Cohen, Timothy Costa, Yao-Lung L. Fang, Yang Gao, Jack Guan, John Gunnels, Azzam Haidar, Andreas Hehn, Markus Hohnerbach, Matthew Jones, Tom Lubowe, Dmitry Lyakh, Shinya Morino, Paul Springer, Sam Stanwyck, Igor Terentyev, Satya Varadhan, Jonathan Wong, and Takuma Yamaguchi. cuquantum sdk: A high-performance library for accelerating quantum science. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 1050–1061, 2023. doi:10.1109/QCE57702.2023.00119.

2 Kishor Bharti, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alperin-Lea, Abhinav Anand, Matthias Degroote, Hermanni Heimonen, Jakob S. Kottmann, Tim Menke, Wai-Keong Mok, Sukin Sim, Leong-Chuan Kwek, and Alán Aspuru-Guzik. Noisy intermediate-scale quantum algorithms. *Rev. Mod. Phys.*, 94:015004, February 2022. doi:10.1103/RevModPhys.94.015004.

3 Michael Broughton, Guillaume Verdon, Trevor McCourt, Antonio J. Martinez, Jae Hyeon Yoo, Sergei V. Isakov, Philip Massey, Ramin Halavati, Murphy Yuezhen Niu, Alexander Zlokapa, Evan Peters, Owen Lockwood, Andrea Skolik, Sofiene Jerbi, Vedran Dunjko, Martin Leib, Michael Streif, David Von Dollen, Hongxiang Chen, Shuxiang Cao, Roeland Wiersema, Hsin-Yuan Huang, Jarrod R. McClean, Ryan Babbush, Sergio Boixo, Dave Bacon, Alan K. Ho, Hartmut Neven, and Masoud Mohseni. Tensorflow quantum: A software framework for quantum machine learning. *arXiv preprint arXiv:2003.02989*, 2020. last revised 26 Aug 2021 (v2). arXiv:2003.02989.

4 Lukas Burgholzer, Jorge Echavarria, Patrick Hopf, Yannick Stade, Damian Rovara, Ludwig Schmid, Ercüment Kaya, Burak Mete, Muhammad Nufail Farooqi, Minh Chung, Marco De Pascale, Laura Schulz, Martin Schulz, and Robert Wille. The xqss: Connecting end users, integrating diverse quantum technologies, accelerating hpc, 2025. arXiv:2509.02674.

5 Jorge Echavarria, Muhammad Nufail Farooqi, Amit Devra, Santana Lujan, Léo Van Damme, Hossam Ahmed, Martín Letras, Ercüment Kaya, Adrian Vetter, Max Werninghaus, Martin Knudsen, Felix Rohde, Albert Frisch, Eric Mansfield, Rakhim Davletkaliyev, Vladimir Kukushkin, Noora Färkkilä, Janne Mäntylä, Nikolas Pomplun, Andreas Spörl, Lukas Burgholzer, Yannick Stade, Robert Wille, Laura B. Schulz, and Martin Schulz. Tackling the challenges of adding pulse-level support to a heterogeneous hpcqc software stack: Xqss pulse. *arXiv e-prints*, page arXiv:2510.26565, October 2025. doi:10.48550/arXiv.2510.26565.

6 J. Eisert, M. Cramer, and M. B. Plenio. Colloquium: Area laws for the entanglement entropy. *Rev. Mod. Phys.*, 82:277–306, February 2010. doi:10.1103/RevModPhys.82.277.

7 Mario Hernandez Vera and Marco De Pascale. QCT-UEA-management/quake2iree. Software, swhId: swh:1:snp:c7b8f26bbeb0273ac7e029a8bba5845db219bdec (visited on 2026-03-17). URL: https://github.com/QCT-UEA-management/quake2iree.git, doi:10.4230/artifacts.25678.

8 David Ittah, Ali Asadi, Erick Ochoa Lopez, Sergei Mironov, Samuel Banning, Romain Moyard, Mai Jacob Peng, and Josh Izaac. Catalyst: a python jit compiler for auto-differentiable hybrid quantum programs. *Journal of Open Source Software*, 9(99):6720, 2024. doi:10.21105/joss.06720.

9 Morris A. Jette and Tim Wickberg. Architecture of the slurm workload manager. In Dalibor Klusáček, Julita Corbalán, and Gonzalo P. Rodrigo, editors, *Job Scheduling Strategies for Parallel Processing*, pages 3–23, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-43943-8_1.

10 Ercüment Kaya, Burak Mete, Laura Schulz, Muhammad Nufail Farooqi, Jorge Echavarria, and Martin Schulz. QPI: A programming interface for quantum computers. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 02, 2024. doi:10.1109/QCE60285.2024.10293.

11 C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi:10.1109/CGO.2004.1281665.

12 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi:10.1109/CGO51591.2021.9370308.

13 Gabin Schieffer, Stefano Markidis, and Ivy Peng. Harnessing CUDA-Q's MPS for Tensor Network Simulations of Large-Scale Quantum Circuits . In *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 94–103, Los Alamitos, CA, USA, March 2025. IEEE Computer Society. doi:10.1109/PDP66500.2025.00022.

14 The CUDA-Q development team. CUDA-Q. URL: https://github.com/NVIDIA/cuda-quantum.

15 The IREE Authors. IREE, September 2019. URL: https://github.com/iree-org/iree.

16 The OpenXLA Community. The OpenXLA Ecosystem. URL: https://openxla.org.

17 The StableHLO Community. The StableHLO Operation Set. URL: https://openxla.org/stablehlo.

18 Robert Wille, Ludwig Schmid, Yannick Stade, Jorge Echavarria, Martin Schulz, Laura Schulz, and Lukas Burgholzer. Qdmi – quantum device management interface: A standardized interface for quantum computing platforms. In *IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2024.

19 Yannick Stade and Lukas Burgholzer and Robert Wille. Towards Supporting QIR: Steps for Adopting the Quantum Intermediate Representation, 2025.

Accelerating GPGPU Simulation by Strategically Parallelizing the Compute Bottleneck

Jakob Sachs
Hamburg University of Technology, Germany

Tim Lühnen
Hamburg University of Technology, Germany

Sohan Lal
Hamburg University of Technology, Germany

Abstract

Cycle-accurate GPGPU simulators like GPGPU-Sim provide invaluable insights for hardware architecture research but suffer from extremely long runtimes, hindering research productivity. This paper addresses this critical bottleneck by proposing a strategy to accelerate GPGPU-Sim. We first perform a holistic profiling analysis across diverse GPGPU benchmarks to identify the primary performance bottleneck, pinpointing the SIMT-Core cluster execution within the CORE-clock cycle. Based on this, we implement a parallelization scheme that strategically targets this hotspot, utilizing a thread pool to manage concurrent execution of SIMT-Core clusters. Our approach prioritizes minimal modifications to the existing GPGPU-Sim codebase to ensure long-term maintainability. Evaluation of a simulated NVIDIA H100 model demonstrates an average simulation wall-time speedup of 3.58× with 8 worker threads, and a maximum up to 4.38×, while incurring a maximum cycle count error of 3.22%, with some other benchmarks exhibiting no error at all.

2012 ACM Subject Classification Computing methodologies → Parallel programming languages; Computing methodologies → Simulation tools; Computer systems organization → Parallel architectures

Keywords and phrases GPGPU, CUDA, Simulation, Computer Architecture, GPGPU-Sim, Parallel Simulation, Cycle-Accurate Simulation, Thread Pool

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2026.6

Supplementary Material *Software (Source-Code)*: `https://github.com/Tim453/ClusterSim` [11]

1 Introduction

Since the introduction of the GPGPU (*General Purpose Computing on GPUs*) paradigm, the need for accelerated computing has only grown, both in terms of scale and applications. This has increased the pressure on hardware architecture research to deliver higher performance/efficiency solutions and faster iteration on these new architectures. Simulation of hardware architectures is a widely used tool in the toolbox of hardware architecture researchers. For GPGPU, there are several established tools, of which a prominent one is GPGPU-Sim [8], which aims to be a cycle-accurate GPGPU simulator, meaning it directly models the target hardware's behavior on a clock-cycle-by-clock-cycle basis, offering deep insights into performance and internal operations (e.g., pipeline states, resource utilization, and cache behavior), particularly for NVIDIA GPUs. For our implementation, we build upon ClusterSim [12], an extension of GPGPU-Sim supporting modern architectural features.

However, such detailed simulation comes with significant performance impact, both due to the overhead inherent to all hardware simulations, and due to the shift in computational architecture from a SIMD processor (GPU) to a sequential CPU, leading to runtimes of hours and even days, as the parallelism of the GPU must be serialized for execution on the CPU.

© Jakob Sachs, Tim Lühnen, and Sohan Lal;
licensed under Creative Commons License CC-BY 4.0

To illustrate the scale of this challenge, one of our test cases (a 2048×2048 single-precision floating-point matrix multiplication kernel running on a simulated H100) originally took ≈ 9 hours to complete, while the real device runs the same kernel in 1.2 milliseconds. This dramatic difference in execution time represents a critical bottleneck in the research workflow and severely limits the search space of architectural parameters that researchers can explore with cycle-accurate simulators.

This critical bottleneck motivates our primary objective in this work: To reduce the "real" runtime (wall-time) of these simulations. Our empirical approach is based on profiling the original simulation and pragmatically parallelizing the identified hotspot, allowing us to reduce the overall amount of changes made to the code base, as opposed to a top-down strategy mandating large-scale structural refactoring from the outset.

This strategy offers two key advantages: First, it helps contain the project's scope, ensuring feasibility. Secondly, it significantly enhances the long-term maintainability of our parallelization extension. By limiting invasive changes, we simplify the process of integrating future updates or features and also leave open the possibility of future attempts at parallelizing other parts of the simulator. In this paper, we contribute the following:

- A holistic profiling of the GPGPU-Sim program across a diverse set of GPGPU benchmarks, to identify and verify the primary performance-limiting component: specifically the SIMT-Core execution inside the *CORE*-clock section.
- A parallelization scheme, based on our profiling results for the SIMT-Core execution. Our implementation distinctively prioritizes minimal modifications to the core GPGPU-Sim code base, to ensure long-term maintainability and ease of integration with future updates, while significantly decreasing simulation wall-times by an average $3.58\times$ with a maximum error of 3.22%.
- A comprehensive evaluation of our parallelization approach conducted on a simulated NVIDIA H100 model, which quantifies achieved simulation speedups against the original sequential implementation, measures, and analyzes the performance scaling of our implementation across different thread counts and measures the incurred error in simulated clock-cycle count across our set of benchmarks.

2 Background and Related Work

In this section, we will quickly cover the internal architecture of GPGPU-Sim and previous research on the same or related subjects.

2.1 GPGPU-Sim

GPGPU-Sim is a cycle-accurate simulator for GPUs that supports simulating the execution of applications written using both CUDA and OpenCL. For the purposes of this work, we focus exclusively on CUDA programs. The simulator functions by intercepting the application's CUDA calls and replacing them with its own calls. This allows it to simulate the target GPU architecture in detail and collect performance statistics. This interception approach provides significant ease of usability, often requiring minimal application modification. GPGPU-Sim also includes a functional-only mode, which bypasses detailed timing simulation. It also incorporates a power model for energy analysis and tools for visualization. GPGPU-Sim's hardware model is designed to closely model contemporary NVIDIA GPUs. The main architecture features of the model are:

- **SIMT cores**: These are the main execution units on which thread blocks/CTAs are scheduled and are thus the equivalent of NVIDIA's Streaming Multiprocessors (*SMs*).

- **SIMT core clusters**: Clusters of multiple SIMT cores that share an interface to the memory interconnect, occupying the same place in the organizational hierarchy as NVIDIA's texture processing clusters (*TPCs*).
- **GPC**: A collection of SIMT core clusters. This level incorporates an internal high-speed network (the SM-to-SM network) connecting its constituent clusters without needing to go through the global interconnect. This addition of the GPC structure and its internal network enables the simulation of features like Hopper's distributed shared memory and thread block clusters.
- **Memory Partition**: Represents a slice of the device's memory subsystem, typically including a portion of DRAM and its associated L2 cache slice.
- **Interconnect**: The interconnect between the SIMT core clusters and the memory partitions. GPGPU-Sim utilizes BookSim [7] to simulate this interconnect.

The component layout is shown in Figure 1. The main driver for larger computing power is the increased component counts, especially the number of SMs, which is the dominant contributor to simulation time (Subsection 3.1). Historically, NVIDIA's flagship SM count grew from up to 16 (Fermi, 2010) [19] to up to 144 (Hopper, 2022) [1]. This trend is advantageous for our scheme, as it is the dimension along which we parallelize (Subsection 3.2). The simulation is controlled by a central clock, which serially steps through sub-clocks (*CORE*, *L2*, *DRAM*, *ICNT*, *SM_NETWORK*). Our optimizations focus entirely on the *CORE*-clock section, as this handles instruction execution and scales the most with grid size.

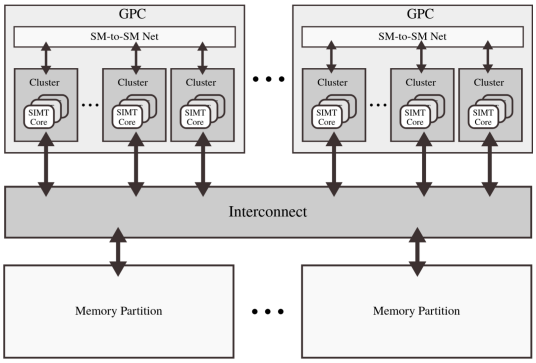


Figure 1 A simplified representation of the hierarchical abstract hardware model described in this section.

2.2 Related Work

The GPGPU simulation landscape extends beyond cycle-accurate simulators like GPGPU-Sim. Machine-learning-based models predict kernel performance [20] or the performance gain of porting applications from CPU to GPGPU [2]. Analytical approaches also statistically model architecture behavior based on execution traces [4, 9, 18]. Despite their utility, researching novel architectural designs often requires the higher fidelity of computationally intensive, cycle-accurate simulators. While valuable for research [6, 13, 15], the slow speed of cycle-accurate simulators like GPGPU-Sim limits study scope and motivates efforts to improve their performance. Early attempts to parallelize GPGPU-Sim, such as by Lee et al. [10], split hardware into shared (memory, interconnect) and parallel (SIMT-Cores) components. This achieved up to 4.15× speedup (3.39× average with 6 threads) but incurred substantial cycle count errors of up to 5.78%.

Zhao et al. [21] implemented a two-level scheme, parallelizing "intra-kernel" (across SIMT-core-clusters) and "inter-kernel" (across machines), estimating a $10 - 40\times$ speedup with a 0.16% cycle error, though these results are just a theoretical combination of their two separate results without empirical verification of the combined approach. Wang et al. [17] proposed a two-step parallelization: first, evaluating SIMT cores in parallel ($1.61 - 2.24\times$ speedup, $\leq 0.22\%$ error), and second, separating *CORE*, *L2*, *Interconnect*, and *DRAM* clock domains using a "shadow-clock". However, such strategies often involve accuracy trade-offs or extensive code changes, hindering adaptability and upstream integration. More recently, Huerta et al. [5] accelerated Accel-Sim using OpenMP with minimal code modification, achieving an average speedup of 5.8× (up to 14×) with 16 threads.

3 Profiling and Parallelization Strategy

While from a purely architectural point of view, there are some clear ideas about which parts of the simulation can be easily parallelized, we decided to focus on a highly empirical approach, by profiling the original implementation to ensure that our beliefs are correct and we don't fall into the trap of premature optimization. So next we will briefly describe our profiling setup and the hotspot analysis results.

3.1 Profiling and Bottleneck Analysis

The original implementation was profiled using Linux's `perf` tool. We profiled our benchmarks on an AMD EPYC 9124 running at 3 GHz and up to 3.70 GHz in turbo and then evaluate the results by examining the percentage of samples recorded inside each function as a proxy for time spent inside. As anticipated, our profiling results reveal that almost the entirety of the runtime is spent with cycling the simulator clock. On average 98.3% of all recorded samples are within the `gpgpu_sim::cycle()` function and its child calls (*SGEMM*: 99.2%, *SM2SM_HIST*: 98.1%, *MONTE_CARLO*: 98.6%, *VECTOR_ADD*: 97.2%).

To get a clearer understanding of this, we observe the child calls of `gpgpu_sim::cycle()`, given its role as the primary simulation clock. Analyzing these child calls reveals a consistent pattern: the same set of functions are the primary contributors to the runtime of `gpgpu_sim::cycle()` across all benchmarks. Among these, one function consistently emerges as the dominant contributor, although its exact percentage contribution varies per benchmark.

Table 1 Percentage of total sample count for the top three child calls of `gpgpu_sim::cycle()`, highlighting the largest hotspot across all benchmarks.

Benchmark	*SGEMM*	*SM2SM_HIST*	*MONTE_CARLO*	*VECTOR_ADD*
`clst.core_cycle()`	**61.36**	**87.93**	**92.91**	**86.03**
`clst.getCacheStats()`	27.49	6.15	4.24	5.83
`LocalIcnt_transfer()`	5.17	1.18	0.65	1.36

Profiling the major child calls of `gpgpu_sim::cycle()` (as detailed in Table 1) confirms that the *CORE*-Clock, handled by `simt_core_cluster::core_cycle()`, is the dominant contributor to runtime. This confirmation validates our focus and allows us to proceed with the design and implementation of the parallelization scheme.

3.2 Parallelization Scheme

We'll now outline our approach to parallelizing the *CORE*-clock cycle and explain why our method is preferable. Given that profiling identified `simt_core_cluster::core_cycle()` as the primary contributor within the *CORE*-clock's execution breakdown, we must first examine its surrounding code structure to inform our parallelization strategy. The main simulator clock is composed of different sub-clocks, which execute based on which one is active. `gpgpu_sim::cycle()` represents this main simulation clock in code, managing the execution of the separate sub-clocks, as structurally simplified in Algorithm 1.

Algorithm 1 Simplified pseudo-code showing the approximate structure inside the `gpgpu_sim::cycle()` function, where the functions are meant as placeholders for the actual code.

```
1:  clock = get_next_clock_domain()
2:  if clock == L2 then
3:      L2_cycle()
4:  if clock == CORE then
5:      core_cycle()                                  ▷ hotspot section
6:  if clock == DRAM then
7:      dram_cycle()
8:  if clock == ICNT then
9:      icnt_cycle()
10: if clock == SM_NETWORK then
11:     sm2smnet_cycle()
```

Based on the profiling results in Subsection 3.1, the *CORE*-clock section represented by `core_cycle()` contains the primary hotspot function. We will now examine the actual internal structure of this `core_cycle()` section.

Algorithm 2 Simplified pseudo-code showing the insides of the placeholder `core_cycle()`, and displaying the context around the hotspot `simt_core_cluster::core_cycle()`.

```
1: for clst in clusters do
2:     if clst.is_active or work_remaining() then
3:         clst.core_cycle()                          ▷ hotspot function
4:     update_stats_for_cluster(clst)
```

As illustrated in Algorithm 2, the compute-heavy portion of each cycle is largely independent across the different SIMT-clusters, with dependencies limited primarily to control logic (scheduling and statistics). Similar to related work covered in Subsection 2.2, we chose to parallelize along this cluster dimension: the set of all clusters is split, and the `clst.core_cycle()` function for each is evaluated in parallel. This approach is the least invasive method for addressing the identified hotspot and minimizes synchronization overhead compared to parallelizing individual SIMT-Cores within the clusters.

Parallelizing each individual `<...>::core_cycle()` by launching a new software thread is infeasible due to the substantial overhead from frequent thread creation and destruction in latency-sensitive code. To quantify this overhead, consider the *SGEMM* 1024×1024 benchmark, which runs for 2.4 hrs and involves 1.5×10^6 *CORE*-cycles.

Assuming an optimistic 10 μs overhead per thread-launch, the total wall-time added solely by thread creation across the simulation's 57 clusters is:

$$57 \text{ clusters} \cdot 10\,\mu\text{s} \cdot 1.5 \times 10^6 \text{ cycles} \approx 14.25\,\text{min}$$

This overhead represents approximately 10% of the total runtime and becomes a bottleneck; any performance improvements must first overcome this initial $\approx$ 10% slowdown, which is likely underestimated. To address this bottleneck in thread management, we instead opted for a thread reuse scheme, implemented using the C++ thread pool by [16]. This dynamic thread pool approach allows us to control the number of software threads used for cycle execution, enabling precise tuning based on both the host hardware and the simulated hardware model. The pool is initialized once at startup and currently handles only the SIMT-Cluster parallelization.

Algorithm 3 Simplified pseudo-code showing the actual dispatching of the cluster-cycles to the worker threads, as mostly handled inside the thread pool.

```
1: chunks = split_up_evenly(clusters)
2: for chunk in chunks do
3:     dispatch_to_thread(chunk)                ▷ following code is ran in worker thread
4:     for clst in chunk do
5:         if clst.is_active or work_remaining() then
6:             clst.core_cycle()
                                                                        ▷ in main-thread
7: wait_for_worker_threads()
8: for clst in clusters do
9:     update_stats_for_cluster(clst)
```

Algorithm 3 shows the simplified thread pool structure, detailing how tasks are dispatched to worker threads and which components are parallelized. We explicitly wait for all clusters to complete their core cycle before updating statistics based on their finished state. This statistics collection step is kept sequential, as it operates on small pieces of shared state and parallelization is unlikely to provide benefit while potentially introducing race conditions. Access to the shared state manipulated by `simt_core_cluster::core_cycle()` is primarily managed using `std::mutex` from the C++ standard library. This static, even partitioning of clusters can be suboptimal under partial system loads. When only a fraction of SIMT-Clusters are active, threads assigned to inactive clusters are underutilized. While detecting and reassigning empty cycling clusters to a single thread could improve speedup in these non-full occupancy scenarios, the additional scheduling logic required would add latency, potentially reducing speedup in the optimal, near-full occupancy case. Given that high-occupancy scenarios correspond to the longest sequential wall-times and are the primary optimization target, we chose to optimize for the highest occupancy rather than implementing dynamic load balancing.

4 Experimental Setup

To evaluate our parallelization scheme, we implemented our proposed solution into Cluster-Sim [12], which is a modified version of GPGPU-Sim, that extends the base simulator with modern features, such as the SM-to-SM-network introduced in the Hopper architecture, to support functionality like thread block clusters and distributed shared memory.

4.1 Modeled GPU configuration

The model configuration used for all tests is designed to replicate the NVIDIA H100 architecture. Specifically, the simulation aimed for a replica of the H100 PCIe 80 GB configuration, which features 114 SMs, as briefly discussed in Subsection 2.1. The primary focus is the number of SIMT clusters and SIMT cores (SMs) in the model. Since parallelization occurs across these clusters, the high cluster count is expected to positively influence speedup results.

4.2 Hardware & Software environment

The benchmarks were executed on a compute node equipped with dual Intel Xeon Gold 6130 CPUs. Unlike the AMD login node used for profiling, these nodes offer a SMP configuration of their cores, which is more suitable for our workload. These nodes provide a total of 32 physical cores (2 sockets × 16 cores/socket, with hyper-threading disabled) operating at a 2.10 GHz base clock speed (3.70 GHz turbo), supported by 192 GB of RAM split across the two NUMA nodes. For configurations with $N \leq 16$ worker threads, we pinned the process to a single NUMA node using `numactl --cpunodebind=0 --membind=0`. For $N > 16$, threads were allowed to span both NUMA nodes under default Linux scheduling.

4.3 Benchmarks

To evaluate the performance impact of our parallelization strategy across diverse computational patterns, we selected four CUDA kernels. These kernels represent a range of common GPGPU workload characteristics, from compute-intensive tasks to memory-bound operations.

- *MONTE_CARLO*: Estimates π via Monte Carlo using a simple LCG generating single-precision pseudo-random (x, y) pairs. Results are aggregated within thread-blocks via a shared memory reduction, and final summation happens on the host CPU.
- *SGEMM*: For our evaluation, we employ an SGEMM (Single-Precision General Matrix Multiplication) kernel based on [3], which operates on two randomized square matrices using 2D shared-memory tiling.
- *SM2SM_HIST*: A histogram implementation adapted from NVIDIA's official guide [14] for using the distributed-shared-memory feature. This is our benchmark targeted to check for incurred cycle error in heavy use of the SM-to-SM net model. The thread-block-clusters are statically configured to run at the maximum size possible on the *sm90* architecture (16 thread-blocks per cluster).
- *VECTOR_ADD*: A kernel that performs element-wise addition on vectors of single-precision floating-point values. It was selected primarily because it exhibits high memory intensity, requiring only one arithmetic operation per element.

These benchmarks should equip us to make relatively general statements about the scaling behavior of our implementation of the parallelization scheme.

4.4 Metrics

The primary objective of this work is reducing the wall-time T_{wall} of GPGPU-Sim simulations, making it the main performance metric. Although factors like total CPU time, utilization, or energy efficiency may be relevant, they are considered secondary to minimizing T_{wall}.

To evaluate the effectiveness and scaling of our parallelization approach, we calculate the speedup S_N achieved using N threads:

$$S_N = \frac{T_{\text{seq}}}{T_N}$$

where T_N is the wall-time of the parallelized simulator running with N threads. To ensure functional correctness and verify that our parallelization does not alter the simulation semantics, we measure the total number of simulated GPU cycles C reported by the simulator. We compare the cycle-count from the parallel execution C_N with the reported count by the original sequential execution C_{seq} and measure the absolute percentage deviation:

$$\text{Absolute Relative Error (\%)} = \frac{|C_N - C_{\text{seq}}|}{C_{\text{seq}}} \times 100\%$$

While GPGPU-Sim reports many more statistics than just the total simulated GPU cycles, this is a good proxy for the other statistics, and also allows us to compare our results with the other works covered in Subsection 2.2.

4.5 Experimental Design

Table 2 Benchmark input sizes for both configurations. The exponent denotes the input parameter passed to each benchmark; the resulting workload size is shown alongside.

Benchmark	T_8 Evaluation	Scaling Study
MONTE_CARLO	$2^{27} = 134$M samples	$2^{24} = 16.8$M samples
SGEMM	$2^{11} = 2048 \times 2048$ matrix	$2^{10} = 1024 \times 1024$ matrix
SM2SM_HIST	$2^{23} = 8.4$M elements	$2^{21} = 2.1$M elements
VECTOR_ADD	$2^{24} = 16.8$M floats	$2^{23} = 8.4$M floats

Using the hardware, software, simulated GPU configuration, and benchmarks described above, we performed the following experiments to evaluate our parallelized implementation and the base version:

1. We measured the wall-time for the original sequential version (T_{seq}) and our implementation with 8 worker threads (T_8). We set the input-sizes of each benchmark so that (T_{seq}) was at least 60 minutes. We also record the simulated cycles so we can quantify the incurred cycle-error (as described in Subsection 4.4).
2. We measured the wall-time for the sequential version (T_{seq}) and our implementation with different worker thread counts, from 2 up-to and including 32 in increments of 2 ($T_2, T_4, \ldots, T_{32}$). Due to the increased amount of runs for this benchmark, we reduced the input-sizes slightly. This experiment allows us to observe the performance scaling characteristic of our implementation and potential impacts of the NUMA architecture of our dual-socket machine (as detailed in Subsection 4.2). We also measure the cycle-error in the same way as described above.

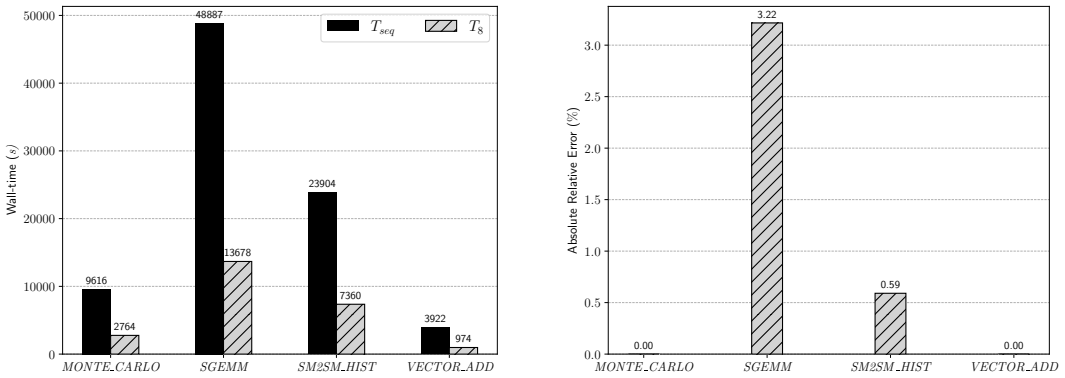
The input sizes for each benchmark can be seen in Table 2. For both sets, we average the resulting metrics over three separate runs.

5 Results and Discussion

In this section, we cover the results of our measurements as described in Subsection 4.4, beginning with the longer benchmarks for a fixed worker thread count of 8.

5.1 Comparing Sequential and Parallel implementations

We begin by examining the results for our long-running benchmarks, comparing the base version and our implementation with 8 worker threads.



(a) The measured Wall-Times comparing T_{seq} and T_8.

(b) Absolute-Relative-Cycle error (%) of the parallelized implementation.

Figure 2 Results for both the wall-time measurement (sub-figure 2a) and the error analysis (sub-figure 2b) of the long running benchmarks.

Figure 2a presents the measured wall-times for both of these (T_{seq} and T_8) across the different benchmarks. These are averaged over three separate runs, as detailed in Subsection 4.4.

Our parallelization scheme achieved a notable reduction in wall-time over all the benchmarks. The speedups ($S_8 = T_{\text{seq}}/T_8$), calculated from these average wall-times, were 3.48× for *MONTE_CARLO*, 3.57× for *SGEMM*, 3.25× for *SM2SM_HIST*, and 4.03× for *VECTOR_ADD*. The difference between benchmarks is likely explained by the different computational and data-access patterns of the different benchmarks.

Across these four benchmarks, our implementation demonstrated an average speedup of 3.58×. This speedup, while remarkable, is below an ideal linear value of 8×. This is to be expected due to factors such as inherent sequential portions of the simulation, synchronization overheads, and communication costs between threads, as discussed in Subsection 3.2. The exact scaling behavior of our implementation is discussed below (in Subsection 5.2).

Error discussion

While parallelization improved speed, we must quantify accuracy using the absolute relative error in total simulation cycles, as shown in Figure 2b. The results vary by benchmark complexity. *SGEMM* shows the highest error (3.22%), which we hypothesize is an inherent consequence of the parallel execution ordering altering L2-cache access patterns, but deeper investigation into this is needed for a definite confirmation.

Similarly, the minor error in *SM2SM_HIST* (0.59%) appears to stem from unsynchronized access to the intra-GPC SM-to-SM-network, potentially changing the order in which messages are sent/received. While stricter synchronization could likely mitigate these deviations, the current error margins are deemed acceptable trade-offs for rapid architectural design space exploration. *MONTE_CARLO* and *VECTOR_ADD* notably exhibit 0.0% error. We attribute this stability to their specific computational characteristics: *MONTE_CARLO* relies on intra-core shared memory and order-independent global atomics, while *VECTOR_ADD*'s single-use access pattern effectively allows it to bypass the L2-cache.

5.2 Scaling with thread-count

To assess how effectively our parallelization strategy utilizes increasing computational resources, we evaluated its scaling behavior. Figure 3 illustrates the measured speedup S_N, relative to the sequential base version, for all the benchmarks as the number of worker threads N was varied from 2 up to 32.

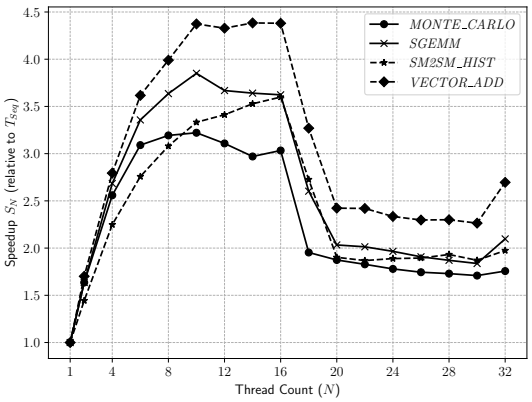


Figure 3 The measured speedup (from S_1 up to S_{32}) over the base implementation for each benchmark.

We can see in Figure 3 that our implementation scales well up to a certain amount of worker threads, but starts delivering diminishing returns around $N \geq 8$. The benchmarks peak at different thread counts: *MONTE_CARLO* and *SGEMM* both top out at $N = 10$ with 3.22× and 3.85× respectively, while *VECTOR_ADD* doesn't top out until $N = 14$ with a speedup of 4.38× and *SM2SM_HIST* only tops out at $N = 16$ with a speedup of 3.60× over the sequential base version.

We can also see a drop in speedup as the thread count increases above 16. The reason for this is quite clearly the nature of our hardware platform (described in Subsection 4.2) having 2 NUMA Nodes, each with 16 CPU cores. So what we are likely observing here is the increase in NUMA-latency as the program is spread across the two NUMA nodes (we manually restricted the program to a single NUMA node if $N \leq 16$).

The diminishing returns effect we observe around 8-10 worker threads is likely due to the increase in synchronization overhead beginning to outweigh the additional speedup of more threads. As mentioned above, we would not expect perfect speedup for any worker thread count, since there are necessary sequential sections to the simulator, and also the mentioned synchronization and communication overhead from the parallelization scheme itself.

5.3 Comparison to Related Approaches

As discussed in Subsection 2.2, there have been previous efforts in parallelizing GPGPU-Sim. The comparison reveals distinct trade-offs between performance and accuracy across different parallelization strategies.

Among prior approaches, *Work-Group Parallel* achieves the highest average speedup at 3.39×, but exhibits significantly higher error rates, with maximum errors reaching 5.78%. This suggests that while aggressive parallelization can yield substantial performance gains, it may compromise simulation fidelity.

Table 3 Comparison of GPU Simulation Parallelization Approaches.

Implementation	Average Speedup	Min. Error	Max. Error
Work-Group Parallel [10]	3.39	0.05%	5.78%
gpusim-para1 [17]	1.91	0.00%	0.18%
gpusim-para2 [17]	2.1	0.04%	0.59%
Our Approach	3.58	0.00%	3.22%

In contrast, both *gpusim-para1* and *gpusim-para2* prioritize accuracy, maintaining maximum errors below 0.6%. However, this reduced error comes at the cost of reduced speedup, with *gpusim-para1* achieving 1.91× and *gpusim-para2* reaching 2.1× median speedup (the paper only reports median for this specific result). The *gpusim-para1* implementation demonstrates exceptional precision with some benchmarks showing zero error, indicating that highly accurate parallel simulation is achievable but requires more conservative parallelization strategies.

We exclude the work of Zhao et al. [21] from this comparison, because the reported 10 – 40× speedup is not empirically measured but rather a theoretical product of their separate intra-kernel and inter-kernel results.

Comparing implementation complexity, our approach required only 10 lines of critical path changes versus Wang et al.'s [17] 170 lines for their simpler version. This minimal modification approach, combined with our balanced performance-accuracy profile, is particularly valuable for research requiring long-term maintainability and frequent updates to the code base.

6 Future Work

While this work demonstrates significant performance improvements, several promising directions exist for future research and development. One potential area involves enhancing the simulation model's fidelity. Specifically, investigating how stricter synchronization within the simulated GPCs could help reduce the cycle error, particularly for benchmarks sensitive to inter-SM communication (e.g., *SM2SM_HIST*).

This would, as previously discussed, likely come with a noticeable decrease in speedup. Additionally, testing with more complex benchmarks could help further narrow down the error source for the *SGEMM* and *SM2SM_HIST* benchmark, if so desired. Alternatively, exploring more relaxed synchronization strategies for the worker threads presents another avenue. While potentially complex to implement, allowing worker threads greater asynchronicity (as demonstrated by [10]) in completing their tasks might bring even greater speedup in simulation time, especially on highly parallel host systems. Testing different simulated model configurations might also provide more insight into possible optimizations, especially regarding the impact that SIMT core cluster counts can have on the amount of speedup achieved through parallelization.

7 Conclusion

In this paper, we demonstrate the effectiveness of a relatively simple and non-invasive parallelization scheme for GPGPU-Sim. We benchmarked our implementation across varied GPGPU workloads, measuring both wall-time and overall simulation cycles to quantify the error introduced by parallelization. Our implementation achieves an average speedup of 3.58× over the sequential version, with a maximum relative cycle error of 3.22%, while

some benchmarks display no error at all. We also analyzed the scaling behavior across different worker thread counts on a dual-node NUMA system, providing guidelines for optimal hardware/software configuration.

Our approach parallelizes the CORE-cycle execution phase by distributing work across worker threads managed by a thread pool. We based our approach on thorough profiling that identified the execution hotspots, particularly in the SIMT-Core cluster execution within the CORE-clock cycle. By reducing simulation wall-times, our parallelization enables researchers to explore significantly larger architectural design spaces within practical time constraints. This acceleration is particularly valuable for iterative design processes requiring multiple parameter sweeps. Our strategy of minimal code modifications ensures these benefits can be maintained as GPGPU-Sim evolves, while serving as a foundation for further parallelization attempts.

While achieving substantial speedups, there are some limitations. The cycle count errors in benchmarks like *SGEMM* (3.22%) likely stem from different execution orderings impacting shared components like L2-cache or the SM-to-SM network, representing the trade-off between simulation speed and determinism. Our scaling analysis reveals diminishing returns beyond 16 worker threads due to NUMA architecture constraints, suggesting that optimal performance requires careful consideration of the underlying hardware platform. The code for this paper can be found in the ClusterSim [12] code base.

References

1 Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito, and Sridhar Ramaswamy. NVIDIA Hopper Architecture In-Depth, March 2022. URL: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/.

2 Newsha Ardalani, Urmish Thakker, Aws Albarghouthi, and Karu Sankaralingam. A Static Analysis-based Cross-Architecture Performance Prediction Using Machine Learning, 2019. arXiv:1906.07840 [cs]. doi:10.48550/arXiv.1906.07840.

3 Simon Boehm. How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog, December 2022. URL: https://siboehm.com/articles/22/CUDA-MMM.

4 Jen-Cheng Huang, Joo Hwan Lee, Hyesoon Kim, and Hsien-Hsin S. Lee. GPUMech: GPU Performance Modeling Technique Based on Interval Analysis. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 268–279, Cambridge, United Kingdom, December 2014. IEEE. doi:10.1109/MICRO.2014.59.

5 Rodrigo Huerta and Antonio González. Parallelizing a modern gpu simulator. *arXiv preprint arXiv:2502.14691*, 2025. doi:10.48550/arXiv.2502.14691.

6 Vishwesh Jatala, Jayvant Anantpur, and Amey Karkare. Improving GPU Performance Through Resource Sharing, 2015. arXiv:1503.05694 [cs]. URL: http://arxiv.org/abs/1503.05694.

7 Nan Jiang, James Balfour, Daniel U. Becker, Brian Towles, William J. Dally, George Michelogiannakis, and John Kim. A detailed and flexible cycle-accurate Network-on-Chip simulator. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 86–96, Austin, TX, USA, April 2013. IEEE. doi:10.1109/ISPASS.2013.6557149.

8 Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020. doi:10.1109/ISCA45697.2020.00047.

9 Jounghoo Lee, Yeonan Ha, Suhyun Lee, Jinyoung Woo, Jinho Lee, Hanhwi Jang, and Youngsok Kim. GCoM: a detailed GPU core model for accurate analytical modeling of modern GPUs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 424–436, New York New York, June 2022. ACM. doi:10.1145/3470496.3527384.

10 Sangpil Lee and Won Woo Ro. Parallel GPU architecture simulation framework exploiting work allocation unit parallelism. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 107–117, Austin, TX, USA, April 2013. IEEE. doi:10.1109/ISPASS.2013.6557151.

11 Tim Lühnen. Tim453/ClusterSim. Software (visited on 2026-03-17). URL: https://github.com/Tim453/ClusterSim, doi:10.4230/artifacts.25679.

12 Tim Lühnen, Jyotirman Behera, Devashree Tripathy, and Sohan Lal. Clustersim: Modeling thread block clusters in hopper gpus. In *IEEE International Symposium on Workload Characterization, IISWC*, 2025. doi:10.15480/882.15858.

13 Shuai Mu, Yandong Deng, Yubei Chen, Huaiming Li, Jianming Pan, Wenjun Zhang, and Zhihua Wang. Orchestrating Cache Management and Memory Scheduling for GPGPU Applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(8):1803–1814, August 2014. doi:10.1109/TVLSI.2013.2278025.

14 NVIDIA. CUDA C++ Programming Guide (for CUDA version 12.8.1), March 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#distributed-shared-memory.

15 Yunho Oh, Keunsoo Kim, Myung Kuk Yoon, Jong Hyun Park, Yongjun Park, Won Woo Ro, and Murali Annavaram. APRES: improving cache efficiency by exploiting load characteristics on GPUs. *ACM SIGARCH Computer Architecture News*, 44(3):191–203, 2016. doi:10.1145/3007787.3001158.

16 Barak Shoshany. A C++17 Thread Pool for High-Performance Scientific Computing. *SoftwareX*, 26:101687, 2024. arXiv:2105.00613 [cs]. doi:10.1016/j.softx.2024.101687.

17 Jun Wang and Jiaquan Gao. Parallelizing GPGPU-Sim for Faster Simulation with High Fidelity. *IEEE Design & Test*, 37(4):83–91, August 2020. doi:10.1109/MDAT.2020.2986738.

18 Lu Wang, Magnus Jahre, Almutaz Adileho, and Lieven Eeckhout. MDM: The GPU Memory Divergence Model. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1009–1021, Athens, Greece, October 2020. IEEE. doi:10.1109/MICRO50266.2020.00085.

19 Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi gf100 gpu architecture. *IEEE Micro*, 31(2):50–59, 2011. doi:10.1109/MM.2011.24.

20 Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. GPGPU performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 564–576, Burlingame, CA, USA, February 2015. IEEE. doi:10.1109/HPCA.2015.7056063.

21 Xia Zhao, Sheng Ma, Wei Chen, and Zhiying Wang. Exploiting parallelism in the simulation of general purpose graphics processing unit program. *Journal of Shanghai Jiaotong University (Science)*, 21(3):280–288, June 2016. doi:10.1007/s12204-016-1723-2.

Linking High-Level Synthesis with FPGA Runtime Orchestration

Despoina Tomkou
Microprocessors and Digital Systems Laboratory, ECE, NTUA, Athens, Greece

Aggelos Ferikoglou
Microprocessors and Digital Systems Laboratory, ECE, NTUA, Athens, Greece

Dimosthenis Masouros
Microprocessors and Digital Systems Laboratory, ECE, NTUA, Athens, Greece

Sotirios Xydis
Microprocessors and Digital Systems Laboratory, ECE, NTUA, Athens, Greece

Dimitrios Soudris
Microprocessors and Digital Systems Laboratory, ECE, NTUA, Athens, Greece

Abstract

FPGAs are increasingly being adopted across the edge-to-cloud continuum due to their ability to provide both high performance and energy efficiency. However, the complexity of programming FPGAs often leads to deployed designs that underutilize available resources. FPGA multi-tenancy has been proposed to enhance resource utilization, yet monolithic designs and dynamic workload demands continue to challenge efficient FPGA usage and compliance with Quality of Service requirements. To address these issues, we propose a novel framework for the optimal orchestration of FPGAs across the edge-to-cloud continuum while meeting user demands. The framework generates approximations of Pareto-optimal designs for each application, capturing trade-offs between performance and resource usage with minimal bitstream generation. This information allows the runtime orchestrator to select the most suitable design based on available PR regions and the QoS requirements of each user. Experimental results demonstrate that the proposed approach achieves an average reduction of QoS violations by a factor of 8.1× across diverse workloads and baseline configurations. Overall, the framework offers a practical and effective solution for realizing FPGA-as-a-Service across the edge-to-cloud continuum.

2012 ACM Subject Classification Computing methodologies; Computer systems organization → Cloud computing; Computer systems organization → Heterogeneous (hybrid) systems; Hardware → Emerging architectures

Keywords and phrases FPGA, Orchestration, Partial Reconfiguration, FPGAaaS

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2026.7

Supplementary Material
Software (Source Code): `https://github.com/aferikoglou/FPGAScheduler` [13]
archived at `swh:1:dir:2941cf80f4a5b396622da5ebb3f1fb76ffae7b8b`

Funding This work has been partially funded by the EU Horizon 2022 program under grant agreement No 101096698 REFMAP (`https://www.refmap.eu/`).

1 Introduction

The rapid growth and increasing capabilities of IoT devices, together with the extensive deployment of high-speed 5G networks, have led to a massive rise in data generation. These technological advancements are driving a diverse range of applications, including autonomous vehicles [20], smart urban infrastructures [2], and intelligent industrial systems [32]. Such


© Despoina Tomkou, Aggelos Ferikoglou, Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris; licensed under Creative Commons License CC-BY 4.0

applications demand ultra-low latency, high reliability, and real-time data processing, requirements that traditional cloud-based architectures alone cannot effectively fulfill. To address these challenges, computing resources are being positioned closer to the data sources, leading to the emergence of edge computing. As a result, multi-layered architectures are being developed to distribute computational resources and applications seamlessly from the network edge to centralized cloud platforms. This integrated approach establishes the edge-cloud computing continuum, enabling efficient workload distribution, reduced latency, and improved resource utilization across a wide range of application domains [31].

Although edge-cloud architectures expand computing capabilities by providing access to a large pool of additional resources, conventional CPUs often fall short for fast, near real-time processing. To address this limitation, hardware accelerators such as GPUs, FPGAs, and ASICs are increasingly integrated across the computing continuum. GPUs, with thousands of small cores enabling massive parallelism, are particularly effective for computationally intensive tasks like Deep Learning [5], but they come with high power consumption and can be inefficient on workloads with complex parallelism [26]. In contrast, ASICs deliver outstanding performance and energy efficiency for specific tasks through customized circuit designs, though they involve high development costs and lack flexibility after fabrication [11]. FPGAs provide a middle ground between these approaches, offering hardware acceleration while retaining post-deployment reconfigurability [30]. Their flexible architecture allows custom implementations of critical operations, from general-purpose datapaths to specialized components such as attention mechanisms and matrix operations in transformer-based models [7], achieving both high performance and energy efficiency.

The increasing adoption of FPGAs is evident from their integration into the infrastructures of major cloud providers. Microsoft's Project Brainwave [25], an FPGA-based real-time AI platform, delivers pre-trained DNN models as online services and operates on Catapult-enhanced servers [27]. This integration illustrates how vendors leverage FPGAs to accelerate large-scale workloads efficiently while preserving the flexibility needed to support evolving AI applications. Beyond serving as accelerators for computationally intensive tasks within provider infrastructures, FPGAs are increasingly offered as a service. Under the FPGA-as-a-Service (FPGAaaS) paradigm, they are made available as computing resources, allowing users to allocate, program, and deploy them through existing management frameworks for on-demand hardware acceleration [1]. For instance, Amazon provides FPGAs as on-demand cloud resources [3], illustrating the growing adoption FPGAaaS.

Despite the advantages offered by the FPGAaaS paradigm, significant challenges remain in efficiently utilizing available FPGA resources. From the user's perspective, programmability continues to be a major obstacle. Unlike traditional computing resources, FPGAs require substantial manual effort and specialized expertise to achieve efficient designs [9]. Although High-Level Synthesis (HLS) tools simplify development, they still demand considerable hardware knowledge to achieve effective performance optimization [28]. As shown by Chi et al. [8], insufficient optimization can even cause FPGA accelerators to perform worse than CPUs. Moreover, exploring the vast FPGA design space is constrained by the lengthy hardware generation process, making comprehensive optimization both time-consuming and resource-intensive [10]. The use of unoptimized designs also poses challenges for providers, as inefficient workloads lead to under-utilization of FPGA resources and, consequently, effective under-provisioning. To enhance utilization, providers often employ multi-tenancy, allowing multiple users to share FPGA resources [1]. However, this approach introduces additional complexities. In a multi-tenant environment, the concurrent sharing of limited resources presents major challenges to satisfying users' Quality of Service (QoS) requirements. Efficient

orchestration of these workloads must also adapt to dynamic and fluctuating demands while maintaining effective resource allocation. Together, these factors make the delivery of FPGAaaS a highly complex and demanding task.

Over the years, numerous solutions have been proposed to streamline the FPGA development process and enable more efficient sharing of FPGA resources. Within the context of HLS, many studies have focused on automating the identification of optimal directive configurations for a given design. Traditional approaches explore the configuration space by evaluating Quality of Result (QoR) metrics obtained from HLS-synthesized designs [34], while modern data-driven methods leverage knowledge from previously optimized designs to guide configurations for unseen applications [15]. Other techniques aim to accelerate the design process by building predictive models that estimate QoR metrics for unexplored designs [24]. Despite these advances, most existing approaches focus on higher-level design aspects, limiting their applicability for rapid hardware generation and deployment in real-world scenarios. In the context of FPGAaaS, Partial Reconfiguration (PR) [36] has emerged as a key enabler of FPGA multi-tenancy, allowing multiple users to share resources and improve overall utilization. In this paradigm, the FPGA is partitioned into multiple PR regions, each functioning as a virtual FPGA to which a user can allocate their custom hardware design. PR-based virtualization permits idle fabric to be reassigned to other users, effectively transforming the FPGA into a multi-tenant device. PR regions may be configured as asymmetric [41], to accommodate hardware modules of varying sizes, or symmetric [6] with fixed dimensions. However, the hardware designs provided by users or obtained from FPGA-accelerated libraries are typically monolithic, requiring specific resources that may not be available at deployment, which can result in violations of user requirements until the necessary resources become available.

In this work, we propose a novel framework for the efficient utilization of multi-tenant FPGAs. Its primary objective is to minimize QoS violations for applications deployed on the FPGA while ensuring optimal use of device resources. Our framework generates, for each deployed application, an approximation of the Pareto-optimal designs representing different trade-offs between performance and resource usage. This is achieved through a modeling approach capable of constructing the Pareto front with a minimal number of hardware generations. At runtime, the orchestrator selects the most appropriate design based on the available PR regions to meet the application's QoS requirements. Experimental results demonstrate that our framework reduces QoS violations by 8.1× on average across various baseline mechanisms and workloads with differing arrival rates and resource demands, underscoring its effectiveness in realizing the FPGAaaS paradigm.

2 Related Work

Recent years have seen significant research efforts dedicated to improving FPGA utilization, addressing both the challenges faced by users in developing efficient accelerators and by providers in managing FPGAs as shared computational resources. These efforts can be broadly classified into two categories: i) *programming-oriented approaches*, which focus on automating the identification of optimal HLS directives and streamlining the design space exploration process, and ii) *orchestration-oriented approaches*, which aim to efficiently share FPGA resources among multiple users, with a particular emphasis on techniques based on partial reconfiguration.

Programming-oriented Approaches. The exploration of directive design spaces has long been a major challenge in the field of HLS. Early approaches sought to efficiently navigate the directive configuration space by analyzing QoR metrics obtained from HLS tools, employing techniques such as simulated annealing [22], evolutionary algorithms [16], and particle swarm optimization [29]. Other studies introduced specialized heuristics to enhance the search process [37]. More recent frameworks have extended these efforts by applying source-level transformations through modern compiler infrastructures such as MLIR [40] or custom compilers built on top of HLS [34]. Nevertheless, achieving accurate results in these methods typically requires invoking the HLS tool to obtain QoR metrics, which is a time-consuming process that can take several minutes per design. To address this limitation, data-driven approaches have emerged, leveraging knowledge from large collections of previously optimized designs to guide configurations for unseen applications [14, 15]. Despite these advancements, most existing methods remain focused on high-level design abstraction, reducing their applicability for rapid hardware generation and deployment in practical orchestration scenarios.

Another research direction focuses on replacing the HLS process itself by developing predictive models trained on large collections of existing designs. During exploration, these models are queried to estimate QoR metrics, greatly reducing the need for time-consuming synthesis runs. The increasing availability of design data has enabled the use of various ML methods, including XGBoost [10], ANNs [10, 23], and CNNs [21], for predicting latency, resource utilization, and power consumption on target FPGAs. Recently, GNNs [35, 33] have attracted attention for their ability to capture the structural characteristics of HLS applications when predicting QoR metrics. Despite their improved representational power, most of these approaches remain limited to HLS estimations, which often diverge from the actual hardware performance in terms of latency, resource usage, and power consumption of the implemented design [10]. Bridging this gap between high-level prediction accuracy and low-level hardware realization remains a critical challenge for enabling truly efficient and deployable FPGA design automation frameworks.

Orchestration-oriented Approaches. The increasing density and capacity of modern FPGAs have made it possible to virtualize devices effectively, enabling multiple users to share resources efficiently. Partial Reconfiguration (PR) is a widely used technique for achieving this goal, as it divides an FPGA into several reconfigurable regions, each functioning as a virtual FPGA. Research on PR-based FPGA partitioning can generally be divided into asymmetric and symmetric approaches [36]. Asymmetric PR regions provide enhanced flexibility by supporting hardware modules of different sizes without requiring full device reconfiguration. One representative work in this category is hCODE2.0 [41], an open-source FPGAaaS platform that leverages asymmetric PR to enable flexible multi-tenancy. The framework simplifies accelerator design and deployment while incorporating an intelligent scheduler that optimizes performance and utilization by considering both resource availability and communication bandwidth.

In contrast, symmetric PR regions, often referred to as resource slots or tiled regions, have fixed and predefined sizes that cannot be altered at runtime. Adjacent regions can be combined to host larger accelerators, helping to reduce internal fragmentation and improve overall resource utilization. Several research frameworks and commercial platforms have adopted symmetric PR partitioning to enable FPGAaaS. Chen et al. [6] proposed a cloud FPGA framework that divides devices into symmetric PR regions for accelerator deployment, achieving low virtualization latency, though limited by PCIe bandwidth constraints. The

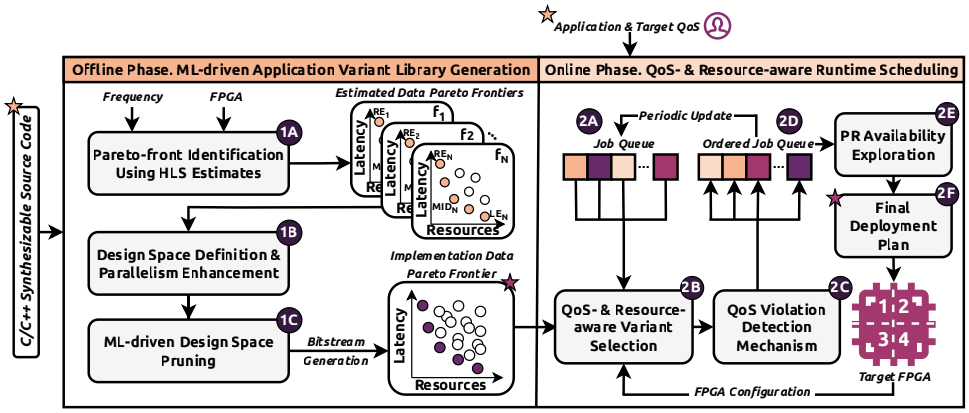


Figure 1 Overview of the Offline and Online Phases of the Proposed Framework.

RC3E FPGA hypervisor [19] allows users to deploy custom hardware designs through full or partial FPGA allocation, improving utilization while maintaining strict security controls. Similarly, a datacenter-oriented reconfigurable accelerator framework [12] connects FPGAs via PCIe and leverages a hypervisor to manage PR regions, effectively reducing redundant reconfigurations. Commercial solutions such as Amazon's EC2 F1 instances [3] extend this model by providing reusable FPGA images and development toolkits for hardware and software developers, while Accelize QuickStore introduces a pay-per-use marketplace for third-party FPGA accelerators and IP cores. Despite these advancements, most accelerators, whether user-designed or sourced from FPGA libraries, remain monolithic in nature, requiring fixed resource configurations. This inflexibility can limit deployment adaptability, potentially leading to QoS violations when suitable resources are not immediately available at runtime.

3 Proposed Scheduling Framework

The proposed methodology enables efficient orchestration of FPGA resources across the edge-to-cloud continuum. Its central objective is to satisfy application-level QoS constraints by minimizing QoS violations while simultaneously maximizing utilization of the available FPGA resources. As illustrated in Figure 1, the framework operates in two complementary phases: (i) the *ML-driven Application Variant Library Generation (Offline)* phase, detailed in Section 3.1, and (ii) the *QoS- & Resource-aware Runtime Scheduling (Online)* phase, presented in Section 3.2.

3.1 Offline Phase: ML-driven Application Variant Library Generation

The objective of the offline phase is to take a given HLS-based application with a synthesizable C/C++ kernel and generate a library of hardware variants, each offering different trade-offs between latency and resource usage. This library of bitstreams serves as a foundation for the online orchestrator, enabling informed design selection at runtime.

The process begins by approximating the Pareto frontier of the computationally intensive kernel 1A. Given a set of target operating frequencies and the chosen FPGA device, this stage generates the latency–resource Pareto frontier for each frequency configuration. This step relies on pre-synthesis information, specifically the reports generated during the HLS process. The rationale for this decision stems from the substantial body of prior work capable

of efficiently deriving Pareto-optimal configurations at the HLS level, as discussed in Section 2. Moreover, publicly available design collections provide Pareto-optimal configurations for well-studied applications, allowing us to directly leverage these results without performing full design-space exploration from scratch. We realize this step using the GNΩSIS dataset [18], the largest HLS design collection currently available, comprising implementations from over 55 applications drawn from established benchmark suites and open-source repositories. For applications not included in the dataset, we employ an NSGA-II–based optimization engine [17] that automatically derives the latency–resource Pareto front for the specified frequency and target FPGA.

The next phase of the methodology focuses on identifying the subset of designs that will be physically implemented to build the application variant library 1B. From each Pareto frontier generated across the evaluated operating frequencies, we select three representative design points: (i) the latency-optimized configuration (LE), (ii) the resource-optimized configuration (RE), and (iii) an intermediate design balancing both objectives (MID). Instead of implementing the entire Pareto set, we choose only these key configurations for two reasons. First, synthesizing and implementing every Pareto candidate would lead to excessive compilation time and computational overhead [24]. Second, offering the runtime scheduler clearly differentiated design points provides meaningful flexibility when mapping workloads to available FPGA resources. Beyond the directive-based variants, we further enrich the design space by incorporating coarse-grained parallelism. For each selected design at each operating frequency, we instantiate multiple Compute Units (CUs), meaning multiple copies of the kernel on the Programmable Logic (PL). This simple yet effective strategy can significantly reduce execution latency when sufficient hardware resources are available. To ensure feasibility, we scale the number of CUs and keep only those configurations whose estimated resource usage, as reported by the HLS tool, fits within the target FPGA.

Introducing CU exploration significantly increases the number of designs that must be physically generated. For example, consider a scenario where only a single operating frequency is evaluated. Assume the LE configuration can support 1, 2, 4, and 6 CUs, the MID configuration can support 1, 2, 4, 6, and 8 CUs, and the RE configuration can support 1, 2, 4, 6, 8, and 10 CUs on the target FPGA. In this case, the design space expands to a total of 15 variants that must be implemented. Additionally, prior research has shown that pre-synthesis estimates often differ from post-implementation results [10], meaning that the actual resource utilization of a multi-CU design may vary, and some configurations may even be not possible to generate. To address this problem, step 1C introduces two mechanisms that leverage post-implementation data from a single CU, along with a small subset of the design space, to assess the feasibility of multi-CU configurations on the FPGA and to accurately predict the design's latency as well as the utilization of BRAMs, DSPs, FFs, and LUTs. This step enables the prediction of feasibility, latency, and resource usage for the remaining configurations. Further details can be found in the paragraph titled "Feasibility Classifier and Performance–Resource Modeling". Using these predictions, the actual Pareto frontier can be identified, allowing only the Pareto-optimal designs to be implemented and thereby significantly reducing the number of required implementations. The resulting Pareto frontier subsequently serves as the application variant library for the online phase.

Feasibility Classifier & Performance–Resource Modeling. To train our models, we utilize applications from the GNΩSIS dataset [18]. Specifically, we use several applications such as KNN and KMeans and obtain their Pareto frontiers at 100, 200, and 300 MHz on the Alveo U200 FPGA. For each target frequency, we select representative designs including LE, MID,

and RE, and expand the design space by incorporating multiple CUs. Each variant is implemented, and for every valid design, we collect post-implementation metrics including latency, BRAM, DSP, FF, and LUT utilization. For each application, the collected implementation data are used to train both the feasibility classifier and the performance–resource models. We build upon established approaches from the literature [10], employing a Decision Tree Classifier (DTC) for feasibility prediction, Linear Regression (LR) for latency estimation, and XGBoost Regression (XGB) for predicting BRAMs, DSPs, FFs, and LUTs. Model performance for each application is evaluated using 5-fold cross-validation, applying the R^2 metric for regression tasks and the F_1 score for classification. After feature engineering and hyperparameter tuning, DTC achieves an average F_1 score of 0.967, while the LR model attains an R^2 of 0.97. The XGB-based models also exhibit strong predictive capability, achieving R^2 scores of 0.99 for BRAMs, 0.98 for DSPs, 0.97 for FFs, and 0.97 for LUTs. Having established high accuracy across all models, we reduce the dataset to identify the minimal portion of the design space that must be implemented beyond the single-CU configurations. The remaining points are selected via random sampling. Our results indicate that utilizing only 20% of the design space leads to an average accuracy degradation of less than 3% across all models. These findings demonstrate that the proposed models can be effectively trained using a small subset of the design space, substantially decreasing the number of bitstream generations required to construct the application variant library.

3.2 Online Phase: QoS- & Resource-aware Runtime Scheduling

The objective of the online phase is to manage a stream of FPGA-accelerated applications, each with a user-defined execution-time requirement, in order to minimize QoS violations while efficiently utilizing the available FPGA resources. The scheduler maintains a queue 2A, where each entry represents an application along with its maximum acceptable execution time. For every queued job, the system tracks both the specified execution deadline and the time elapsed since the job entered the queue. Based on this information, it computes the remaining allowable time before a QoS violation occurs, defined as the difference between the target execution time and the accumulated waiting time. As the job waits longer, this remaining time decreases, signaling increased urgency for scheduling. This parameter is continuously updated and used to guide job prioritization and reduce QoS violations.

Step 2B is responsible for selecting the most suitable application variant to meet the target application's requirements. It operates using the application variant library generated during the offline phase, along with knowledge of the current FPGA configuration. Here, the FPGA configuration refers to the number of available PR regions. In this work, we assume symmetric PR regions, in line with the prevailing practice in literature [1]. The FPGA is divided into these PR regions such that the total resources are evenly distributed, giving each region an equal share of the device's resources. To identify the best candidate, this step first verifies that the application variant can fit within the available PR regions. Specifically, it excludes any variants that require more BRAM, DSP, FF, or LUT resources than are available in a single PR region. Among the remaining bitstreams, it further eliminates any variants that cannot meet the QoS constraint based on the remaining allowable time, as these would result in a violation. From the filtered set of candidates, the scheduler then selects the variant with the shortest execution time to maximize performance while ensuring compliance with the QoS requirements.

If none of the available application variants that fit within a single PR region can satisfy the QoS constraint, the QoS violation detection mechanism is triggered 2C. This step identifies jobs that cannot be accommodated under the current conditions and temporarily

de-prioritizes them by placing them at the end of the queue, allowing feasible jobs to be scheduled first. Once the urgency scores for all applications are computed, the queue is reordered so that the most urgent jobs are processed first, forming the ordered job queue 2D. When a job becomes ready for execution, the framework explores the available PR regions 2E to determine whether a suitable region is available for allocation. If a region is available, the application is assigned to it; otherwise, the job remains in the queue until a region becomes free. Once assigned, the job is mapped to the corresponding PR region of the FPGA, finalizing the deployment plan 2F.

4 Experimental Evaluation

In this section, we present the experimental evaluation of the proposed framework. We begin by describing the experimental setup and the baseline mechanisms used for comparison, followed by a detailed presentation and analysis of the obtained results and the impact of the proposed approach.

4.1 Experimental Overview & Baselines Description

We assess the efficiency of the proposed scheduling framework through a comprehensive set of experiments designed to capture realistic workload dynamics and system behaviors. The experimental workloads are derived from applications in the GNΩSIS dataset [18], focusing on KNN and KMeans, which are representative algorithms widely used in large-scale data analytics and real-time processing and are particularly well suited for FPGA acceleration [38]. All implementations were developed using the AMD Vitis Unified Software Platform 2021.1 [4] and executed on the Alveo U200 FPGA. The experiments were conducted on a Linux-based server equipped with an Intel Xeon Silver 4210 CPU @ 2.20 GHz, featuring 32KB of L1D/I, 1MB of L2, and 14MB of L3 cache. To evaluate Partial Reconfiguration (PR), we employed a simulation-based approach instead of deploying physical bitstreams, providing the flexibility to explore a broad range of partitioning schemes and FPGA configurations. T3he experiments involve a set of 30 applications, each defined with specific QoS constraints representing maximum allowable execution times. Application demands were modeled by aggregating the execution times of all available design variants and selecting the 25%-ile values, emulating a high-load operational scenario. Application arrivals were generated using a bursty distribution derived from Alibaba accelerator cluster traces [39], capturing realistic cloud workload patterns where requests arrive in concentrated bursts. This setup ensures a realistic yet analytically manageable representation of production-scale system conditions.

The proposed framework is evaluated along two complementary dimensions: (a) the scheduling strategy, which determines the order in which applications are executed, and (b) the management strategy, which governs bitstream selection and FPGA partitioning through PR. At the scheduling level, the proposed scheduler is compared against a conventional FIFO scheduler that serves applications strictly in their order of arrival without accounting for workload characteristics or system state. This baseline provides a simple yet informative reference point for assessing the benefits of the proposed policy under dynamic and heterogeneous workloads. At the management level, four configurations are examined to capture the effects of Pareto-awareness and FPGA partitioning. The first configuration (`M1`) represents a Pareto-agnostic manager without access to the Pareto frontier. It consistently selects a fixed bitstream that, on average across applications, is 30% slower and utilizes 24% more resources than the MID design, emulating the behavior of a non-expert user choosing a

suboptimal configuration. The FPGA operates as a single monolithic region in this setup. The second configuration (M2) introduces Pareto-awareness, enabling the manager to select implementations from the Pareto frontier while still using a single FPGA region. The third configuration (M3) combines Pareto-awareness with PR by dividing the FPGA into two reconfigurable regions, allowing concurrent execution of multiple Pareto-optimal designs. The fourth configuration (M4) further extends this concept by partitioning the FPGA into four PR regions, enhancing parallelism and flexibility. The number of PR regions is limited to four, as further partitioning would excessively constrain the available hardware area, preventing many bitstreams from fitting within smaller regions and reducing the system's practical usability. The performance of each configuration is evaluated using four key metrics: (a) total number of QoS violations, (b) average waiting time, representing the duration an application remains in the queue before execution, (c) end-to-end execution time, which includes waiting time, execution time, and scheduling overhead, and (d) average excess time, defined as the mean duration by which a job exceeds its QoS target. For jobs meeting their QoS constraints, the excess time is clipped to zero.

4.2 Detailed Analysis of Proposed Framework versus Defined Baselines

We analyze the system behavior under two workload scenarios. The first corresponds to a homogeneous case, where a single application, in this case KNN, is scaled across multiple executions, while the second represents a heterogeneous case, where multiple distinct applications are executed concurrently.

4.2.1 Homogeneous Workload Evaluation

Figure 2 presents the results of the homogeneous workload experiment. Focusing on manager M1, which represents the Pareto-agnostic configuration where the FPGA operates as a single region, we can observe the baseline impact of the proposed scheduling policy. The proposed scheduler results in 22 QoS violations compared to 27 under FIFO. In terms of end-to-end execution time, it achieves a distribution ranging from 31 seconds to 13.3 minutes, with an average of 7.1 minutes. The waiting time exhibits a range from 2 milliseconds to 12.7 minutes, averaging 6.6 minutes. Overall, the proposed scheduler achieves about a 1% reduction in both end-to-end execution and waiting times, with a modest 2% increase in average excess time. These results demonstrate that even at the scheduling level alone, prioritizing jobs with tighter deadlines effectively reduces violations and improves timing efficiency. When examining manager M2, the benefits of Pareto-awareness become more pronounced. Under this configuration, the proposed scheduler records 18 QoS violations, improving upon its performance in M1. The end-to-end execution time distribution ranges from 23 seconds to 8.9 minutes, with an average of 4.9 minutes, while waiting times span from 2.7 milliseconds to 8.6 minutes, averaging 4.5 minutes. This corresponds to a 1.45× reduction in both end-to-end and waiting times, accompanied by a 1.6× decrease in average excess time. These improvements underscore the value of Pareto-awareness, as M2 leverages latency-optimized designs from the Pareto frontier, resulting in more efficient execution and a substantial reduction in QoS violations.

Next, we evaluate the performance of the M3 and M4 managers, which leverage PR to divide the FPGA into multiple independent regions. As expected, increasing the number of PR regions reduces the number of QoS violations, since more requests can be executed concurrently. In the proposed approach, M3 results in 13 violations, whereas M4 further decreases this number to only 3, demonstrating the clear advantages of finer-grained reconfig-

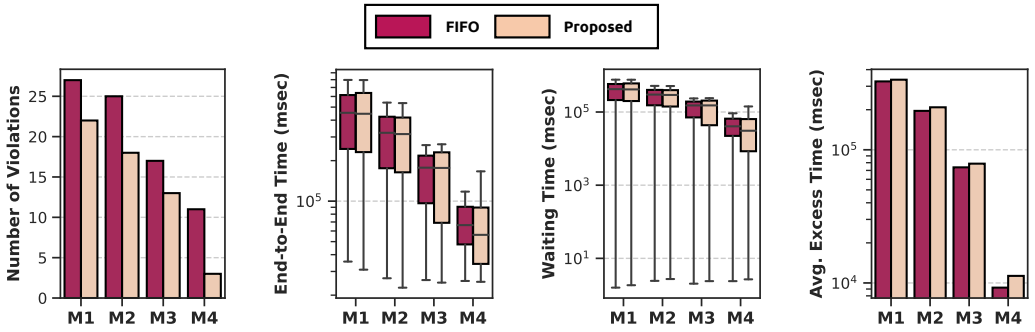


Figure 2 Evaluation of the proposed approach under homogeneous workloads, showing (Left) the number of QoS violations, (Middle Left) the distribution of end-to-end execution time, (Middle Right) the distribution of waiting time, and (Right) the average excess time.

uration. For `M3`, the end-to-end execution time ranges from 24.8 seconds to 4.4 minutes, with an average of 2.6 minutes, while the waiting time spans from 2 milliseconds to 4 minutes, averaging 2.2 minutes. In the case of `M4`, the end-to-end time varies between 25 seconds and 2.8 minutes, averaging 1.5 minutes, and the waiting time ranges from 2 milliseconds to 2.4 minutes, with an average of 43 seconds. When compared to `M2`, `M3` achieves a 1.9× improvement in end-to-end execution time, while `M4` achieves a 4.3× improvement. Similarly, the waiting time is reduced by 2.1× with `M3` and by 6.4× with `M4`. The average excess time, which quantifies how long jobs exceed their QoS targets, also decreases significantly – by 2.7× for `M3` and 18.5× for `M4` compared to `M2`. These results demonstrate that increasing the number of PR regions consistently enhances overall system performance. They further highlight the effectiveness of the proposed scheduling framework in exploiting PR to achieve high resource utilization and strong QoS compliance. An additional observation is that the proposed scheduler shows an 8% higher average excess time, even though it achieves fewer QoS violations. This behavior is attributed to the *QoS Violation Detection Mechanism*, which de-prioritizes jobs predicted to miss their deadlines. Consequently, while this strategy helps reduce the number of QoS violations, it also increases the waiting time for certain jobs, slightly raising the average excess time.

Ablation Study. In this section, we conduct an ablation study based on Figure 2 (Left) to examine how each mechanism introduced in the proposed framework contributes to reducing QoS violations. The FIFO scheduler paired with the `M1` manager serves as the baseline for this analysis. Introducing the proposed scheduler, which prioritizes applications that are close to violating their QoS constraints and de-prioritizes those that have already missed them, leads to a 19% reduction in violations compared to the baseline. Adding Pareto-awareness, which enables the manager to identify and select latency-efficient configurations from the Pareto frontier, provides a further 20% decrease in violations. The most substantial improvement is achieved with the `M4` configuration, which incorporates both Pareto-awareness and PR by dividing the FPGA into four independent regions. This setup results in a total reduction of 77% in violations, demonstrating the combined benefits of adaptive scheduling, Pareto-aware management, and fine-grained FPGA partitioning. As expected, exploiting the full potential of partial reconfiguration delivers the greatest performance gains by enabling concurrent execution and improving resource utilization.

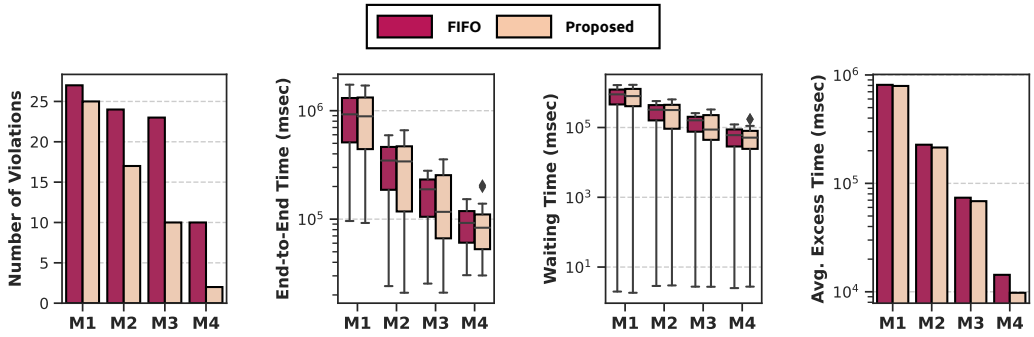


Figure 3 Evaluation of the proposed approach under heterogeneous workloads, showing (Left) the number of QoS violations, (Middle Left) the distribution of end-to-end execution time, (Middle Right) the distribution of waiting time, and (Right) the average excess time.

4.2.2 Heterogeneous Workload Evaluation

For the heterogeneous workload, we evaluate a diverse set of applications, each generating requests with distinct execution completion requirements, providing a more realistic representation of deployment scenarios. Figure 3 presents the results of this experiment. The observed results align with the trends seen in the homogeneous workload, where the proposed scheduler combined with the `M4` manager achieves the fewest QoS violations. Specifically, it results in only 2 violations. The end-to-end execution time distribution ranges from 30 seconds to 3.4 minutes, with an average of 1.5 minutes, while the waiting time varies between 2.7 seconds and 2.9 minutes, averaging 57.3 seconds. The average excess time is measured at 9.8 seconds. In terms of violations, this configuration achieves substantial improvements, showing 13.5×, 8.5×, and 5× fewer violations compared to the FIFO scheduler with `M1` and the proposed scheduler with `M2` and `M3`, respectively. It also outperforms the same baselines in terms of average end-to-end execution time, achieving improvements of 10.6×, 3.6×, and 1.8×, respectively. In terms of waiting time, it demonstrates reductions of 14.9× and 2.3× compared to the corresponding configurations. Furthermore, the average excess time shows an order-of-magnitude reduction relative to the FIFO scheduler with `M1` and `M2`, and it remains 7.6× lower than that of `M3`. These findings indicate that the proposed scheduler, when combined with Pareto-awareness and full utilization of four PR regions, consistently surpasses all baseline approaches, even under realistic heterogeneous workload conditions.

5 Conclusion

This work presented a novel framework designed to enhance the efficiency and flexibility of multi-tenant FPGA utilization across the edge-to-cloud continuum. By integrating Pareto-optimal design approximation and intelligent runtime orchestration, the proposed approach effectively balances performance and resource usage while minimizing QoS violations. Unlike conventional static or monolithic deployment strategies, our framework dynamically adapts to workload and resource variability, leveraging partial reconfiguration to ensure efficient sharing of FPGA resources among concurrent applications. Experimental evaluation shows an average 8.1× reduction in QoS violations across diverse workloads and baseline mechanisms, confirming the effectiveness of the proposed framework in satisfying user requirements under dynamic operating conditions.

References

1 Lamees M Al Qassem, Thanos Stouraitis, Ernesto Damiani, and Ibrahim M Elfadel. Fpgaaas: A survey of infrastructures and systems. *IEEE Transactions on Services Computing*, 15(2):1143–1156, 2020.

2 Fadi Al-Turjman, Hadi Zahmatkesh, and Ramiz Shahroze. An overview of security and privacy in smart cities' iot communications. *Transactions on Emerging Telecommunications Technologies*, 33(3):e3677, 2022. doi:10.1002/ETT.3677.

3 Amazon Web Services. Ec2 instances (f1) with programmable hardware. https://aws.amazon.com/blogs/aws/developer-preview-ec2-instances-f1-with-programmable-hardware/, 2016. Accessed: 2025-10-25.

4 AMD Vitis Unified Software Platform. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis.html, 2025. Accessed: 2025-11-10.

5 Ebubekir Buber and DIRI Banu. Performance analysis and cpu vs gpu comparison for deep learning. In *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*, pages 1–6. IEEE, 2018.

6 Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling fpgas in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, pages 1–10, 2014. doi:10.1145/2597917.2597929.

7 Hongzheng Chen, Jiahao Zhang, Yixiao Du, Shaojie Xiang, Zichao Yue, Niansong Zhang, Yaohui Cai, and Zhiru Zhang. Understanding the potential of fpga-based spatial acceleration for large language model inference. *ACM Transactions on Reconfigurable Technology and Systems*, 18(1):1–29, 2024. doi:10.1145/3656177.

8 Yuze Chi, Weikang Qiao, Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. Democratizing domain-specific computing. *Communications of the ACM*, 66(1):74–85, 2022. doi:10.1145/3524108.

9 Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011. doi:10.1109/TCAD.2011.2110592.

10 Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline FY Young, and Zhiru Zhang. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 129–132. IEEE, 2018.

11 Wei Dai and Daniel Berleant. Benchmarking contemporary deep learning hardware and frameworks: A survey of qualitative metrics. In *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*, pages 148–155. IEEE, 2019. doi:10.1109/COGMI48466.2019.00029.

12 Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. Virtualized fpga accelerators for efficient cloud computing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 430–435. IEEE, 2015. doi:10.1109/CLOUDCOM.2015.60.

13 Aggelos Ferikoglou. FPGAScheduler. Software, swhId: swh:1:dir:2941cf80f4a5b396622da5ebb3f1fb76ffae7b8b (visited on 2026-02-26). URL: https://github.com/aferikoglou/FPGAScheduler, doi:10.4230/artifacts.25581.

14 Aggelos Ferikoglou, Andreas Kakolyris, Vasilis Kypriotis, Dimosthenis Masouros, Dimitrios Soudris, and Sotirios Xydis. Data-driven hls optimization for reconfigurable accelerators. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6, 2024.

15 Aggelos Ferikoglou, Andreas Kakolyris, Dimosthenis Masouros, Dimitrios Soudris, and Sotirios Xydis. Collectivehls: A collaborative approach to high-level synthesis design optimization. *ACM Transactions on Reconfigurable Technology and Systems*, 18(1):1–32, 2024. doi:10.1145/3702005.

16 Fabrizio Ferrandi, Pier Luca Lanzi, Daniele Loiacono, Christian Pilato, and Donatella Sciuto. A multi-objective genetic algorithm for design space exploration in high-level synthesis. In *2008 IEEE Computer Society Annual Symposium on VLSI*, pages 417–422. IEEE, 2008. doi:10.1109/ISVLSI.2008.73.

17 GenHLSOptimizer GitHub. https://github.com/aferikoglou/GenHLSOptimizer, 2025. Accessed: 2025-11-01.

18 GNWSIS Dataset Hugging Face. https://huggingface.co/datasets/aferikoglou/GNWSIS, 2025. Accessed: 2025-11-01.

19 Oliver Knodel and Rainer G Spallek. Rc3e: provision and management of reconfigurable hardware accelerators in a cloud environment. *arXiv preprint arXiv:1508.06843*, 2015. arXiv:1508.06843.

20 Xhafer Krasniqi and Edmond Hajrizi. Use of iot technology to drive the automotive industry from connected to full autonomous vehicles. *IFAC-PapersOnLine*, 49(29):269–274, 2016.

21 Zhe Lin, Jieru Zhao, Sharad Sinha, and Wei Zhang. Hl-pow: A learning-based power modeling framework for high-level synthesis. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 574–580. IEEE, 2020. doi:10.1109/ASP-DAC47756.2020.9045442.

22 Anushree Mahapatra and Benjamin Carrion Schafer. Machine-learning based simulated annealer method for high level synthesis design space exploration. In *Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn)*, pages 1–6. IEEE, 2014.

23 Hosein Mohammadi Makrani, Hossein Sayadi, Tinoosh Mohsenin, Setareh Rafatirad, Avesta Sasan, and Houman Homayoun. Xppe: cross-platform performance estimation of hardware accelerators using machine learning. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 727–732, 2019. doi:10.1145/3287624.3288756.

24 Dimosthenis Masouros, Aggelos Ferikoglou, Georgios Zervakis, Sotirios Xydis, and Dimitrios Soudris. Late breaking results: Language-level qor modeling for high-level synthesis. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–2, 2024. doi:10.1145/3649329.3663500.

25 Microsoft Research. Project brainwave: A deep learning platform for real-time ai inference in the cloud and on the edge. https://www.microsoft.com/en-us/research/project/project-brainwave/, 2018. Accessed: 2025-10-25.

26 Sparsh Mittal and Jeffrey S Vetter. A survey of methods for analyzing and improving gpu energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):1–23, 2014. doi:10.1145/2636342.

27 Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

28 Benjamin Carrion Schafer and Zi Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2628–2639, 2019. doi:10.1109/TCAD.2019.2943570.

29 Anirban Sengupta and Saumya Bhadauria. User power-delay budget driven pso based design space exploration of optimal k-cycle transient fault secured datapath during high level synthesis. In *Sixteenth International Symposium on Quality Electronic Design*, pages 289–292. IEEE, 2015. doi:10.1109/ISQED.2015.7085441.

30 Ahmad Shawahna, Sadiq M Sait, and Aiman El-Maleh. Fpga-based accelerators of deep learning networks for learning and classification: A review. *ieee Access*, 7:7823–7859, 2018. doi:10.1109/ACCESS.2018.2890150.

31 Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016. doi:10.1109/JIOT.2016.2579198.

32 Fadi Shrouf, Joaquin Ordieres, and Giovanni Miragliotta. Smart factories in industry 4.0: A review of the concept and of energy management approached in production based on the internet of things paradigm. In *2014 IEEE international conference on industrial engineering and engineering management*, pages 697–701. IEEE, 2014.

33 Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. Robust gnn-based representation learning for hls. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2023. doi:10.1109/ICCAD57390.2023.10323853.

34 Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. Autodse: Enabling software programmers to design efficient fpga accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(4):1–27, 2022. doi:10.1145/3494534.

35 Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. Accurate operation delay prediction for fpga hls using graph neural networks. In *Proceedings of the 39th international conference on computer-aided design*, pages 1–9, 2020. doi:10.1145/3400302.3415657.

36 Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. A survey on fpga virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 131–1317. IEEE, 2018.

37 Sotirios Xydis, Gianluca Palermo, Vittorio Zaccaria, and Cristina Silvano. SPIRIT: spectral-aware pareto iterative refinement optimization for supervised high-level synthesis. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 34(1):155–159, 2015. doi:10.1109/TCAD.2014.2363392.

38 Feng Yan, Andreas Koch, and Oliver Sinnen. A survey on fpga-based accelerator for ml models. *arXiv preprint arXiv:2412.15666*, 2024. doi:10.48550/arXiv.2412.15666.

39 Lingyun Yang, Yongchen Wang, Yinghao Yu, Qizhen Weng, Jianbo Dong, Kan Liu, Chi Zhang, Yanyi Zi, Hao Li, Zechao Zhang, Nan Wang, Yu Dong, Menglei Zheng, Lanlan Xi, Xiaowei Lu, Liang Ye, Guodong Yang, Binzhang Fu, Tao Lan, Liping Zhang, Lin Qu, and Wei Wang. Gpu-disaggregated serving for deep learning recommendation models at scale. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, USENIX NSDI '25. USENIX Association, 2025. URL: https://www.usenix.org/conference/nsdi25/presentation/yang.

40 Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 741–755. IEEE, 2022. doi:10.1109/HPCA53966.2022.00060.

41 Qian Zhao, Motoki Amagasaki, Masahiro Iida, Morihiro Kuga, Toshinori Sueyoshi, et al. hcode 2.0: An open-source toolkit for building efficient fpga-enabled clouds. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 267–270. IEEE, 2017.

Performance Modeling & Mapping of LLM Inference on Heterogeneous Vectorized CGRAs

Dionysios Kefallinos
National Technical University of Athens, Greece

Georgios Alexandris
National Technical University of Athens, Greece

Alexis Maras
National Technical University of Athens, Greece

Panagiotis Chaidos
National Technical University of Athens, Greece

Manil Dev Gomony
Eindhoven University of Technology, Netherlands

Henk Corporaal
Eindhoven University of Technology, Netherlands

Dimitrios Soudris
National Technical University of Athens, Greece

Sotirios Xydis
National Technical University of Athens, Greece

Abstract

Since the emergence of transformer-based models, the computational demands for Large Language Model (LLM) inference have been increasing exponentially, primarily due to their compounding parameter sizes, their structural complexity, and the use of non-linear functions. This tendency leads to the necessity of deploying them on low-power edge devices and DNN accelerators, to fuel next-generation agentic AI systems. Coarse-Grained Reconfigurable Architectures (CGRAs) have proven to be a compelling paradigm for edge acceleration, combining the programmability of general-purpose platforms with the high performance and energy efficiency associated with ASICs. In this work, we introduce an end-to-end performance modeling and mapping framework for LLM inference on heterogeneous CGRAs. Our methodology enables rapid exploration of the micro-architectural design space parameters, i.e., the number of processing elements, vector sizes, and memory configurations, by providing an accurate, explainable, and analytical CGRA performance modeling methodology, with an average cycle error of 0.9%. Architecturally, we build upon R-Blocks, a heterogeneous CGRA platform, and extend it to support floating-point arithmetic operations as well as a full-stack compilation and mapping flow for both full (FP32) and quantized (INT8) Llama2 models. The proposed methodology, evaluated on a 22nm technology node, achieves superior peak performance per Watt compared to related works such as REVAMP and CFEACT (1.8× and 2.8× respectively).

2012 ACM Subject Classification Computer systems organization → Reconfigurable computing; Hardware → Modeling and parameter extraction

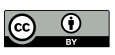
Keywords and phrases Edge AI, LLM, CGRA, Heterogeneous Architectures, Performance Modeling, Hardware Acceleration, Low Power Computing

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2026.8

Funding This work is funded in part by the Convolve project evaluated by the EU Horizon Europe research and innovation program under grant agreement No. 101070374.

1 Introduction

Over the last few years, AI and more specifically Large Language Models (LLMs), have emerged as a transformative technology and have become a focal point of research at both the architectural [16, 14, 23] and algorithmic levels [19, 35]. The need for agentic AI deployment on the edge has fueled a new wave of hardware research, targeting power efficiency when handling the intense linear and non-linear [3] operations of these models, without sacrificing performance.



© Dionysios Kefallinos, Georgios Alexandris, Alexis Maras, Panagiotis Chaidos, Manil Dev Gomony, Henk Corporaal, Dimitrios Soudris, and Sotirios Xydis;
licensed under Creative Commons License CC-BY 4.0

17th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 15th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2026).
Editors: Davide Baroffio, Paola Busia, Lev Denisov, and Nitin Shukla; Article No. 8; pp. 8:1–8:14
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Introducing flexibility, in order to adapt to the dynamic computational patterns of LLMs, is key when targeting performance on a low power budget. Prior approaches have revolved around software/hardware co-optimization methodologies [10, 9, 4], others have explored FPGAs as a more flexible acceleration platform [11, 37], while some have developed ASIC solutions [13, 27, 36].

Trying to balance this trade-off, Coarse-Grained Reconfigurable Architectures (CGRAs) have re-emerged as a promising alternative. CGRAs are composed of an array of programmable Processing Elements (PEs), organized in an n-dimensional grid, interconnected via a routing network that can be reconfigured dynamically or per application. This means that they can incorporate highly optimized, pre-designed hardware units with close-to-ASIC performance as their coarse-grained PEs, while keeping interconnection overheads low compared to conventional fine-grained reconfigurable platforms. This property positions them as a dominant middle ground between fixed-function hardware and fully programmable platforms [34][20], as well as a promising candidate for edge LLM acceleration.

Typical CGRAs, thus far, map standard operation kernels into a grid of homogeneous PEs; i.e., offering limited customization capabilities to the targeted application domain. In this paper, we investigate the viability of a heterogeneous CGRA approach by presenting an end-to-end flow, capable of modeling, mapping, simulation, synthesis, and evaluation of LLM inference across multiple CGRA architectural configurations. Our architecture and code mapping flow build upon the low-power, heterogeneous R-Blocks CGRA [7] paradigm, and our evaluation is performed using the standard Llama2 architecture [32], producing broadly generalizable results. More specifically, the key contributions of this work are as follows:

- We extend the R-Blocks CGRA with full hardware and software support for floating-point arithmetic. This includes integrating a new single-precision Floating-Point Unit (FPU) as a Processing Element, and providing ISA support for compilation and scheduling. The resulting CGRA is capable of executing both linear and non-linear operations, while also leveraging parallelization, making it suitable for LLM acceleration.
- We introduce a high-level performance modeling framework for our architecture, capable of accurately estimating inference latency with minimal error. Through this analytical approach, we enable rapid design space exploration of different CGRA architectures without resorting to full hardware synthesis and RTL simulations for each micro-architectural configuration.
- We effectively map and evaluate end-to-end LLM inference of Llama2 models (the forward pass of base (FP32) and a quantized (INT8)) on several optimized CGRA instances at the post-synthesis level. We extract key insights regarding the effect of parallelization on performance and measure the power and area efficiency of our designs, synthesized at a 22nm technology node.

Through extensive experimental evaluation, we verify the accuracy of our performance modeling framework with a 0.9% error introduction.

2 Related Work

Heterogeneous CGRAs. One of the earliest works presenting a heterogeneous approach was the REVAMP [1] CGRA, a DSE framework that, relying on the application characteristics, produces CGRA architectures based on the HyCUBE [15] paradigm, with its own compilation and mapping scheme.

Table 1 Qualitative comparison of prior work.

CGRA	PE Structure	Application Mapping	Performance Modeling	LLM Inference
REVAMP [1]	Heterogeneous	DFG-Based	×	×
R-Blocks [7]	Heterogeneous	Custom C Compiler	×	×
DRIPS [30]	Homogeneous	MLIR-Based	×	×
CFEACT [22]	Homogeneous	MLIR-Based	×	BERT-models
ML-CGRA [21]	Homogeneous	MLIR-Based	×	BERT-models
PICACHU [26]	Heterogeneous	MLIR-Based	TimeLoop	Non-Linear Ops.
Our Work	Heterogeneous	Custom C Compiler	✓	Llama2

Carrying the torch, R-Blocks CGRA [7] suggests an operation-level PE disaggregation approach, where memory and logic tiles can be placed on the grid independently, with a custom NoC supporting the data and instruction interconnections between them. Application mapping on R-Blocks is performed from high-level code, using a custom compilation flow, while a high-level simulation tool is also provided.

Notably, none of these platforms has been used to map and evaluate LLM benchmarks on a heterogeneous PE grid, nor do they offer a dedicated performance modeling tool. This is precisely the domain in which our research seeks to provide new insights.

LLMs on CGRAs. ML-CGRA [21] is one of the earliest CGRA-related works utilizing MLIR to provide support for application mapping from popular ML front-ends such as TensorFlow and PyTorch directly to the reprogrammable, homogeneous PE fabric. This is achieved by producing the Data Flow Graph (DFG) of the target application and implementing optimization passes to map it on OpenCGRA [31] or CGRA-ME [5] generated architectures. With this tool, mapping LLMs such as BERT [8] on a CGRA becomes possible for the first time.

Other early publications such as DRIPS [30] and CFEACT [22], also utilize MLIR for compilation, although, again, on homogeneous PEs. DRIPS achieves dynamic load rebalancing for Graph Convolutional Network applications where kernels can vary in features and computational demands, whereas CFEACT further optimizes the produced DFG and allows for BERT models and even floating-point operations to be mapped more efficiently.

The benchmarks evaluated in these works, however, are simpler language representation models (BERT) and, therefore, the performance insights gained cannot be generalized when scaling up to the requirements of modern generative LLMs, such as GPT [28] or Llama [32]. Our work aims to provide more widely applicable conclusions by mapping and evaluating larger models of the Llama2 series.

Finally, the more recently published work of PICACHU [26] takes a different approach, utilizing a CGRA as a co-accelerator exclusively for non-linear operations. The main bulk of linear computations is handled by a systolic array, which communicates with the CGRA through a shared SRAM buffer. This work also makes use of MLIR [17] for mapping efficient code and scheduling data transfers between the systolic array and the CGRA, but only for the non-linear part of the models. It additionally utilizes TimeLoop [25] as a modeling tool to estimate area, power, and latency.

While accelerating non-linear functions using a CGRA seems promising, our work aims to map the entire token generation step on the reconfigurable fabric, taking advantage of the heterogeneous nature of the PEs. At the same time, we provide an analytical and explainable

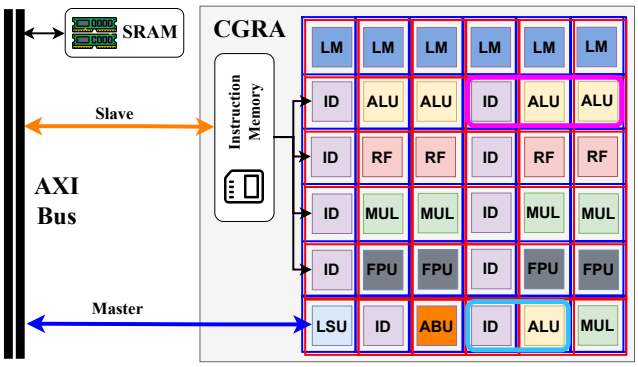


Figure 1 Architectural Overview of the Proposed CGRA.

(therefore interpretable) LLM performance modeling tool, tailored to our specific architecture, something missing from most prior works. Table 1 summarizes the above discussion in a qualitative manner.

3 Proposed CGRA Architecture

3.1 Architecture Overview

The first distinction of our approach, compared to the previously discussed CGRAs, is its support for heterogeneous architectural configurations regarding the Processing Elements. Most proposed CGRAs consist of identical PEs adhering to a predetermined dataflow pattern. By contrast, our work introduces several types of heterogeneous PEs, each with a specific Instruction Set Architecture (ISA). These Functional Units (FUs) or tiles, as commonly referred to in our work, bear a close resemblance to the core building blocks of CPUs, such as Arithmetic-Logical Units (ALU), Register Files (RF), Floating-Point Units (FPU), Local Memories (LM) and Load-Store Units (LSU), among others.

Disaggregating the ISA into simpler and smaller operation sets (PE heterogeneity) comes with multiple benefits:

- Narrower instruction interconnect network, since opcode reuse is allowed among different types of FUs.
- More efficient PEs, through increased specificity in their respective functionality, and more specialized pre-designed hardware.
- Better utilization of resources by effectively mapping operations to the appropriate PEs.
- Designer flexibility, regarding the number of PEs of each type to be used in a specific application domain.

Figure 1 depicts the proposed CGRA scheme. The architecture is built upon the R-Blocks [7] framework, extended with additional support for Floating-Point Unit PE, to allow for the efficient execution of non-linear functions required for LLM inference. The proposed CGRA consists of a grid of programmable FU tiles communicating through two distinct Network-on-Chip (NoC) interconnect structures, as depicted with blue and red colors in Figure 1. The first NoC handles instruction transfers between tiles, while the second manages all data transfers that take place during the program execution. The two NoCs are structured in a 2D mesh formation, utilizing Wilton switchboxes [29] for reduced reconfiguration overhead and wire congestion.

The programming of each FU is handled by an Instruction Decoder (ID) tile, which stores its respective instruction memory segment, and every cycle dispatches the instruction word to be executed by its associated FU.

To enhance parallelism and improve computational throughput, we extend our architecture with SIMD (Single Instruction, Multiple Data) capabilities through vectorization. In this configuration, a single instruction decoder (ID) can be configured to control multiple function units (FUs) of the same type simultaneously, thereby enabling efficient parallel execution and reducing control overhead across the reconfigurable fabric.

Figure 1 shows an exemplary Vectorized CGRA physical grid layout, i.e., the ID and ALUs highlighted in magenta form a Vector Unit, while the ones in light blue are in a scalar layout.

In terms of memory hierarchy, our accelerator makes use of Local Memories (LM), facilitating smaller but faster transfers between the micro-architectural components. Communication with the external, scratchpad memory is facilitated by the AXI interface protocol, which allows the CGRA to be flexibly connected to any compatible host. In our configuration, a Master Interface accesses the external memory address space, while a Slave Interface is used to program the Instruction Memory, which in turn programs the ID of every FU, utilizing the NoC.

3.2 Compilation Framework

The process of mapping high-level code to a CGRA instance is facilitated by a fully custom compilation flow. Compilation occurs for a specific architecture and a specific grid formation of FUs defined by the programmer. An architecture can easily be created and modified by the designer according to the requirements of the application domain and may contain any number of FUs of each type, and also vector units for parallel computation.

Starting from a C-language specification, the code is compiled using the LLVM compilation back-end and operations are lowered to simple instructions supported by the hardware. Subsequently, given an architectural description, these instructions are bound to the ISA of the CGRA's FUs and scheduled in an executable manner, utilizing the OpenASIP [12] compiler and scheduler. OpenASIP tools target Transport Triggered Architectures [6] (TTA), a processor design paradigm where computations revolve around data transfers on the buses. As such, the entire CGRA can easily be modeled as a TTA machine in order to take advantage of these tools. With this compilation scheme, macros and custom intrinsics allow the programmer to explicitly control the code mapping (e.g., by assigning computations to standard PEs) before producing the final executable.

3.3 HW/SW Extensions for FP Arithmetic Support

Running full LLM inference on the edge requires execution of non-linear functions, which rely on floating-point arithmetic and its properties. To compute them effectively without suffering from complicated emulation algorithm overheads or large accuracy loss, we employ a low-power Floating-Point Unit (FPU). The OpenCores FPU [24] and its accompanying Compare Unit are packed into a compatible tile and integrated within the CGRA environment as a new type of FU. This low-power FPU is fully compliant with the IEEE 754 single-precision floating-point standard (FP32), supports the basic arithmetic operations, as well as comparisons and conversions between the INT32 and floating-point (FP32) representations.

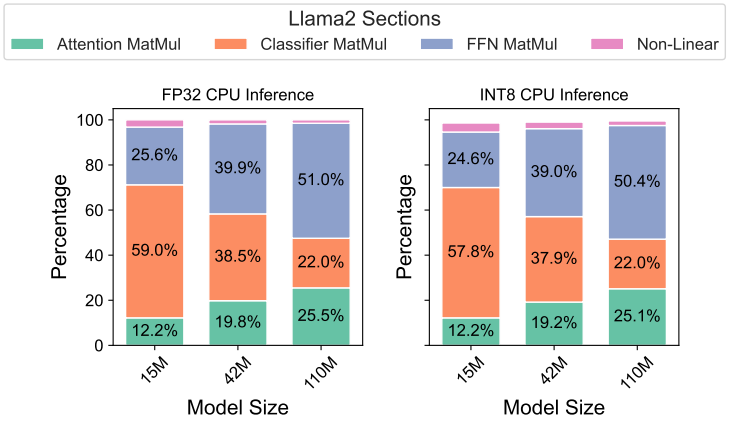


Figure 2 Llama2 CPU Profiling Overview.

To support full functionality of the new FU, we bound the lowered operations from the LLVM compilation back-end to the corresponding TTA instructions, and those, in turn, to the new ISA created for the FPU tile. This allows the token generation step of LLMs to be mapped entirely on the reconfigurable fabric of a heterogeneous CGRA (evaluated for the first time in CGRAs, to the best of our knowledge).

4 LLM Mapping and Modelling on the CGRA

4.1 LLMs on the Edge

LLMs, just like many traditional neural networks, combine linear layers such as matrix multiplications and non-linear operations like normalization or softmax. The linear layers are good at infusing the activation vectors with the stored information in the model's weights, while the non-linearities enforce stability and ensure non-uniform node activation, giving the models their neuron-like morphology and capabilities. LLMs, additionally, utilize the Attention mechanism to encode long-range dependencies between the tokens of a sequence [33].

In this work, we make use of the Llama2 [32] architecture as a representative sample of the different and rapidly evolving transformer implementations for Language Generation. Llama2 employs architectural patterns commonly found in most state-of-the-art LLM models; inference consists of a series of layers, each containing an Attention and a Feed Forward Network (FFN) block. The bulk of the model's weights are used in matrix multiplications in these two blocks. The non-linear functions included between the linear blocks are Softmax, RMS Normalization, SwiGLU (a ReLU variation), and Llama's signature Rotary Positional Encoding (RoPE)[2]. After the token embedding passes through multiple layers, another matrix multiplication (Classifier) produces the final probability distribution of the forward pass from which the output token is selected.

Trying to understand the distribution of the computational load in LLMs, we perform time-based profiling on 3 differently sized base (FP32) Llama2 models, ranging from 15M to 110M parameters, and their symmetrically quantized (INT8) versions (Figure 2). For inference, we use an AMD Ryzen 3 5300U CPU on single-thread execution. As expected, the non-linear operations, while complex in form –containing inverse square root, exponential, and trigonometric functions– are used sparingly between linear operations and in most cases

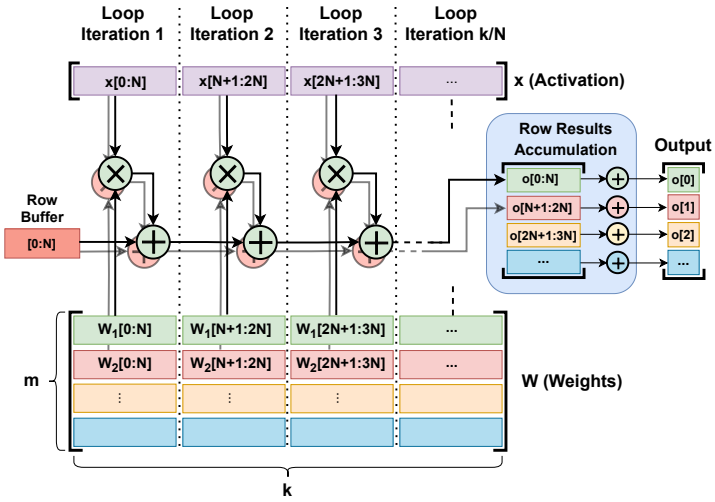


Figure 3 Vectorization of the MatMul Kernel.

don't cause a computational bottleneck, especially as model sizes increase, where they account for an even smaller percentage of runtime. On the other hand, matrix multiplication is a much more computationally intensive operation.

Since matrix multiplication operations are usually free of dependencies and parallelizable to a huge extent, hardware with parallel execution capabilities becomes invaluable for LLM inference and training, e.g., modern GPUs utilizing multiple tensor cores. In the case of GPU execution, as showcased in [26], the percentage of runtime taken up by matrix multiplications can drop as low as 53% for 7B parameter Llama2 models, especially as the context window increases.

Our particular approach aims to achieve some degree of parallelization while still operating within the strict power constraints of Edge Computing. As mentioned before, instead of focusing on a section of inference and using the CGRA as an operation-specific accelerator [26], we utilize the compilation toolchain in conjunction with LLM-specific optimizations to map the entire forward pass of the Llama2 inference model to a single CGRA instance.

4.2 Mapping Methodology

Since matrix multiplications benefit tremendously from Instruction Level Parallelism in the execution model, and no spatial dependencies exist between the operands, multiple elements of the matrix's rows or columns can be loaded for parallel execution as long as the hardware, the memory structure, and the mapping methodology support it.

In our proposed CGRA, FUs of the same type are aggregated into Vector Units, connected to the same ID, thus performing operations supported by the particular FU type in an SIMD fashion. To enable this functionality, the operands are stored in Register Files (RF) or Local Memory units, so that they can be simultaneously accessed.

The matrix-vector multiplication ($W[m,k] \cdot x[k]$) was vectorized using per-row operand grouping. Elements are grouped by N (Vector Unit size), both in the weight matrix rows and the activation vector. As illustrated in Figure 3, in every loop iteration, the corresponding operand vectors are multiplied and accumulated on a vector buffer. This reduces the number of loop iterations per row from k to k/N, effectively introducing a degree of parallelization N. The elements of the buffer vectors are then accumulated to produce the final result.

This method of vectorization takes advantage of the temporal locality of operations by storing the activation vector in the Local Memory for the entire duration of the computation. It also utilizes the spatial locality of data by fetching entire rows to the Local Memories, decreasing the total number of required Global Memory accesses, and, ultimately, effectively distributing the most time-consuming operation of the inference among our computational resources.

The rest of the model's sections, consisting mostly of non-linear functions, are scheduled and mapped to the scalar FUs of the defined architecture by the compiler, concluding the mapping process.

4.3 Analytical Performance Modeling

The need for an analytical modeling framework, capable of producing realistic performance estimates for Llama2 inference on various architectures, stems from the time-consuming nature of simulating the entire execution at the RTL level. Full model inference for multiple benchmarks on different architectural configurations can be very time-intensive, consuming tens of minutes per token generation. Even the cycle-accurate simulation tool of the OpenASIP environment, which models bus transfers at a high level, fails to account for physical memory access times, while still imposing significant simulation time overheads (several minutes per token).

What we propose is an even higher-level performance estimation model, specific for LLM inference execution on heterogeneous CGRA architectures. The framework is both hardware and software-aware, meaning it requires a high-level description of the architectural configuration, as well as model parameters and mapping decisions, in order to produce a performance estimation with minimal error.

More specifically, we begin with modeling the lower-level functions of the inference model, i.e., the matrix multiplications and the non-linearities that account for, practically, the entire runtime, as well as data-flow overheads. We first use a combination of RTL cycle-accurate measurements and TTA simulator runs on microbenchmarks to estimate the influence of certain loop kernels on the overall performance, while accounting for memory access times.

Using the TTA simulator and the LLM parameters, we come up with analytical equations for entire functions that make use of these kernels.

As an example, we present the generalized cost estimation equations for matrix-vector multiplication execution ($W[m,k] \cdot x[k]$) with the mapping methodology outlined in 4.2.

$$C = m\left((k+1)\,M + \frac{k}{N}\cdot G + N\cdot E\right) \tag{1}$$

$$C_q = m\left(k\cdot M + \frac{k}{N}\cdot G + N\cdot\frac{k}{S}\cdot E\right) + Q \tag{2}$$

Here, N stands for the degree of vectorization, while M, G and E refer to the SRAM access cost, the vector multiply-accumulate cost, and the element extraction and accumulation cost from the final row buffer, respectively. These parameters depend on the operating frequency of the CGRA, among other configuration settings, and are determined in the lower-level analysis. Since k/N multiplications-accumulations take place in each of the m rows, this is the coefficient of G. The coefficients of M and E correspond to the numbers of memory accesses and accumulated vector elements, respectively, which also relate to loop iterations.

Table 2 CGRA Architectures for Evaluation.

Architecture	Vector Size	Grid Size	VFPU	VMUL	Llama2 Model
Vec4	4	6x9	✓	×	FP32
Vec8	8	10x8	✓	×	FP32
Vec16	16	18x8	✓	×	FP32
Qvec4	4	6x9	×	✓	INT8
Qvec8	8	10x8	×	✓	INT8
Qvec16	16	18x8	×	✓	INT8

In Eq. 2, which models the same cost for the INT8 quantized Llama2, S symbolizes the quantization group size (meaning the number of elements that share the same scaling factor) and Q denotes the quantization overhead cost, which is calculated by a different equation and depends on the input dimensions.

To complete the Llama2 modeling validation, we account for all matrix multiplication blocks on the mapped inference code, their operand dimensions, and the flow of data between them, through non-linear functions and memory accesses. In an extensive validation study of the proposed modeling framework, which included both microbenchmarks and full model inference runs on simulated hardware, we achieved an average error of 0.9% and a maximum error of 2.5% against cycle-accurate measurements.

This modeling methodology enables a major part of the following analysis by accelerating the exploration of CGRA architectures and software configurations. Most importantly, it allows us to extract critical insights about the effect of vectorization on performance, the impact of design decisions, and the cost of scaling model parameters.

5 Results & Discussion

5.1 Experimental Setup

Initially, we map the entire token generation step of two different versions of the 42M parameter Llama2 model: the base FP32 inference model and a (post-training) quantized version with INT8 operands for better power performance and a smaller memory footprint. We explore the design space of vectorized architectures with vector sizes of 4, 8, and 16, as shown in Table 2, to measure the impact of vectorization on performance.

All of the evaluated architectures build upon a simple `Baseline` architecture consisting of only the computational FUs necessary for basic CGRA functionality, meaning only an ALU, a MUL, an RF and an FPU. `Vec4`, `Vec8`, and `Vec16`, add to that vectorized FPUs and ALUs for executing the FP32 version of Llama2 and achieve an ILP factor of 4, 8, and 16, respectively. `Qvec4`, `Qvec8`, and `Qvec16`, where the quantized version of the Llama2 will be mapped, are designed in the same way as their FP counterparts, but utilize vectorized MUL units instead of vectorized FPUs. Every architecture evaluated also has two vectorized Register Files (VRF), with the same vector size.

5.2 Performance Evaluation

Figure 4 compares the estimations of our performance modeling to the measured RTL cycles for the entire token generation step. Our modeling methodology achieves a maximum error of 1.3% on the full model executions and 1.5% on the quantized versions.

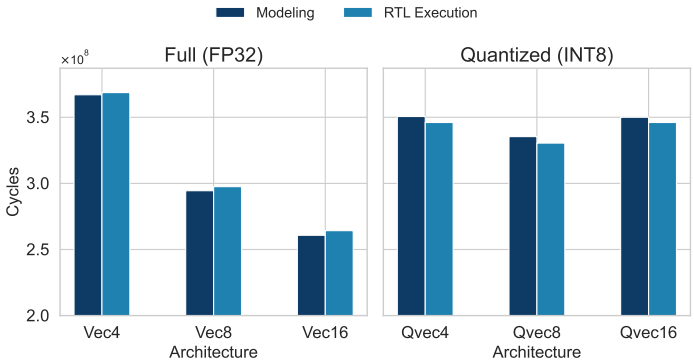


Figure 4 Performance Modeling vs Actual Cycles.

Table 3 Comparison with State-of-the-Art Designs. Performance metrics are reported from the respective publications.

		REVAMP [1]	R-Blocks [7]	DRIPS [30]	CFEACT [22]	PICACHU [26]	This Work (V4/V8)
Arch.	Technology (nm)	28	22	45	40	45	22
	Clock Freq. (MHz)	100	100	800	650	1000	200
	# of PEs	36	77	25	60	16	54 / 80
	Area (mm^2)	0.125	0.486	2.07	2.411	1	0.311 / 0.439
Perf.	Power (mW)	5	3.99	564.8	839	64.2	36.24 / 50
	Peak GOPs	0.9	461	N/A	78	N/A	11.8 / 16
	Peak GOPs/W	180	115	N/A	92.9	N/A	325.6 / 320

Following the methodology of R-Blocks [7], we compare all of the presented architectures against `Baseline` to assess the impact of parallelization. `Vec4`, `Vec8` and `Vec16` achieve 1.38×, 1.72× and 1.93× speedup respectively. We observe diminishing returns in performance gains as the degree of parallelization increases. This is caused by the result accumulation step at the end of each row calculation, during the execution of vectorized matrix multiplication, described in Section 4. This makes sense because weight and activation matrices in the 42M parameter Llama2 model are quite small in size (usually 512 elements), and can not take full advantage of large degrees of parallelization.

Even more notably, INT8-quantized runs of the forward pass achieve marginal performance gains as the vector size increases. `Qvec4` and `Qvec8` achieve 1.55× and 1.62× speedup, respectively, compared to `Baseline` (not depicted), and further increasing the vector size to 16 causes the group accumulation overheads to surpass the performance gains. This is because the Llama2 quantization approach of grouped operand scaling, induces an accumulation overhead inversely proportional to the group size S, since scaled elements have to be accumulated in their groups. However, quantized model execution requires lower energy per token generation and significantly less memory usage [18], therefore, it is worth exploring as an alternative mapping in this analysis.

5.3 Post-Synthesis PPA Analysis

Taking into account all these observations, we opt to synthesize the more performant `Vec4`, `Vec8`, `Qvec4` and `Qvec8` architectures and extract power and area measurements. We use the Synopsys Design Compiler at GF 22nm node and a 200MHz operating frequency. Power measurements are obtained using Synopsys PrimeTime from the switching activity data generated from gate-level simulations.

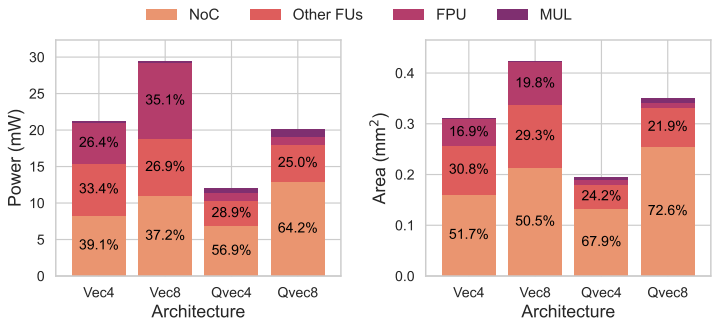


Figure 5 Power and Area Breakdown.

Figure 5 presents the power consumption and area utilization per architecture, both of which, naturally, increase as the vector units get larger. As expected from a computationally intensive tile, the inclusion of more FPU tiles influences both metrics much more heavily than adding MUL units. However, increasing the vector size, in general, also has the side-effect of requiring a larger interconnect network, which also draws significant power in every configuration. The effects of INT8 quantization also become clear: `Qvec4` and `Qvec8` report lower power and area, due to the lighter INT8 computations.

5.4 Comparative Study with SotA

Most of the previous approaches, as presented in section 2, are not directly comparable to our work. They either haven't evaluated their platform on LLM inference or differ architecturally. Even PICACHU [26], one of the most recent (and comparable) works in the field, approaches LLM inference differently, by only computing non-linear operations on the CGRA. Despite these, to give the reader an idea of where our work stands in terms of performance, we present a qualitative comparison table for key area, power and performance metrics.

6 Conclusions

The key insights from this work revolve around the efficacy of deploying LLMs on heterogeneous CGRAs. We show that it is not only possible to map the entire token generation step on a low-power edge device, but also provide an analytical framework to estimate its performance at design time, allowing for fast architectural DSE. We believe the merit of this initial contribution lies in its extendability and how it enables future CGRA designers to optimize LLM execution.

References

1 Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. REVAMP: a systematic framework for heterogeneous CGRA realization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 918–932, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503222.3507772.

2 Federico Barbero, Alex Vitvitskyi, Christos Perivolaropoulos, Razvan Pascanu, and Petar Veličković. Round and Round We Go! What makes Rotary Positional Encodings useful?, 2025. doi:10.48550/arXiv.2410.06205.

3 Fenglong Cai, Dong Yuan, Zhe Yang, and Lizhen Cui. Edge-LLM: A Collaborative Framework for Large Language Model Serving in Edge Computing. In *2024 IEEE International Conference on Web Services (ICWS)*, pages 799–809, 2024. doi:10.1109/ICWS62655.2024.00099.

4 Lvcheng Chen, Ying Wu, Chenyi Wen, Shizhang Wang, Li Zhang, Bei Yu, Qi Sun, and Cheng Zhuo. An Agile Framework for Efficient LLM Accelerator Development and Model Inference, 2024.

5 S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. CGRA-ME: A unified framework for CGRA modelling and exploration. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 184–189, 2017. doi:10.1109/ASAP.2017.7995277.

6 Henk Corporaal and Reinoud Lamberts. TTA processor synthesis. In *First Annual Conf. of ASCI*, pages 18–27. Citeseer, 1995.

7 Barry de Bruin, Kanishkan Vadivel, Mark Wijtvliet, Pekka Jääskeläinen, and Henk Corporaal. R-Blocks: An Energy-Efficient, Flexible, and Programmable CGRA. *ACM Trans. Reconfigurable Technol. Syst.*, 17(2), May 2024. doi:10.1145/3656642.

8 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019. arXiv:1810.04805.

9 Lidong Guo, Zhenhua Zhu, Tengxuan Liu, Xuefei Ning, Shiyao Li, Guohao Dai, Huazhong Yang, Wangyang Fu, and Yu Wang. Towards Floating Point-Based Attention-Free LLM: Hybrid PIM with Non-Uniform Data Format and Reduced Multiplications. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '24, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3676536.3676776.

10 Yingbing Huang, Lily Jiaxin Wan, Hanchen Ye, Manvi Jha, Jinghua Wang, Yuhong Li, Xiaofan Zhang, and Deming Chen. Invited: New Solutions on LLM Acceleration, Optimization, and Application. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3649329.3663517.

11 Suyeon Hur, Seongmin Na, Dongup Kwon, Joonsung Kim, Andrew Boutros, Eriko Nurvitadhi, and Jangwoo Kim. A Fast and Flexible FPGA-based Accelerator for Natural Language Processing Neural Networks. *ACM Trans. Archit. Code Optim.*, 20(1), February 2023. doi:10.1145/3564606.

12 Pekka Jääskeläinen, Timo Viitanen, Jarmo Takala, and Heikki Berg. *HW/SW Co-design Toolset for Customization of Exposed Datapath Processors*, pages 147–164. Springer International Publishing, 2017. doi:10.1007/978-3-319-49679-5_8.

13 Jaeyong Jang, Yulhwa Kim, Juheun Lee, and Jae-Joon Kim. FIGNA: Integer Unit-Based Accelerator Design for FP-INT GEMM Preserving Numerical Accuracy. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 760–773, 2024. doi:10.1109/HPCA57654.2024.00064.

14 Roman Kaplan. Intel Gaudi 3 AI Accelerator: Architected for Gen AI Training and Inference. In *2024 IEEE Hot Chips 36 Symposium (HCS)*, pages 1–16, 2024. doi:10.1109/HCS61935.2024.10665178.

15 Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017. doi:10.1145/3061639.3062262.

16 Hanjoon Kim, Younggeun Choi, Junyoung Park, Byeongwook Bae, and Hyunmin et al. Jeong. TCP: A Tensor Contraction Processor for AI Workloads Industrial Product*. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 890–902, 2024. doi:10.1109/ISCA59077.2024.00069.

17 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law, 2020. arXiv:2002.11054.

18 Wenjie Li, Aokun Hu, Ningyi Xu, and Guanghui He. Quantization and Hardware Architecture Co-Design for Matrix-Vector Multiplications of Large Language Models. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 71(6):2858–2871, 2024. doi:10.1109/TCSI.2024.3350661.

19 Jun Liu, Zhenglun Kong, Peiyan Dong, Changdi Yang, Xuan Shen, Pu Zhao, Hao Tang, Geng Yuan, Wei Niu, Wenbin Zhang, Xue Lin, Dong Huang, and Yanzhi Wang. RoRA: Efficient Fine-Tuning of LLM with Reliability Optimization for Rank Adaptation, 2025. doi:10.48550/arXiv.2501.04315.

20 Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *ACM Comput. Surv.*, 52(6), October 2019. doi:10.1145/3357375.

21 Yixuan Luo, Cheng Tan, Nicolas Bohm Agostini, Ang Li, Antonino Tumeo, Nirav Dave, and Tong Geng. ML-CGRA: An Integrated Compilation Framework to Enable Efficient Machine Learning Acceleration on CGRAs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023. doi:10.1109/DAC56929.2023.10247873.

22 Yiqing Mao, Xuchen Gao, Jiahang Lou, Yunhui Qiu, Wenbo Yin, Wai-Shing Luk, and Lingli Wang. CFEACT: A CGRA-based Framework Enabling Agile CNN and Transformer Accelerator Design. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 213–219, 2024. doi:10.1109/FPL64840.2024.00037.

23 Seungjae Moon, Jung-Hoon Kim, Junsoo Kim, Seongmin Hong, and Junseo et al. Cha. A Latency Processing Unit: A Latency-Optimized and Highly Scalable Processor for Large Language Model Inference. *IEEE Micro*, 44(6):17–33, 2024. doi:10.1109/MM.2024.3420728.

24 OpenCores. OpenCores Floating Point Unit. https://opencores.org/projects/fpu.

25 Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019. doi:10.1109/ISPASS.2019.00042.

26 Jiajun Qin, Tianhua Xia, Cheng Tan, Jeff Zhang, and Sai Qian Zhang. PICACHU: Plug-In CGRA Handling Upcoming Nonlinear Operations in LLMs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 845–861, 2025. doi:10.1145/3676641.3716013.

27 Yubin Qin, Yang Wang, Zhiren Zhao, Xiaolong Yang, Yang Zhou, Shaojun Wei, Yang Hu, and Shouyi Yin. MECLA: Memory-Compute-Efficient LLM Accelerator with Scaling Sub-matrix Partition. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 1032–1047, 2024. doi:10.1109/ISCA59077.2024.00079.

28 Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI Blog*, 1(8), 2019. URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

29 Seyyed Ahmad Razavi, Morteza Saheb Zamani, and Kia Bazargan. A tileable switch module architecture for homogeneous 3D FPGAs. In *2009 IEEE International Conference on 3D System Integration*, pages 1–4, 2009. doi:10.1109/3DIC.2009.5306586.

30 Cheng Tan, Nicolas Bohm Agostini, Tong Geng, Chenhao Xie, Jiajia Li, Ang Li, Kevin J. Barker, and Antonino Tumeo. DRIPS: Dynamic Rebalancing of Pipelined Streaming Applications on CGRAs. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 304–316, 2022. doi:10.1109/HPCA53966.2022.00030.

31 Cheng Tan, Chenhao Xie, Ang Li, Kevin J. Barker, and Antonino Tumeo. OpenCGRA: An Open-Source Unified Framework for Modeling, Testing, and Evaluating CGRAs. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 381–388, 2020. doi:10.1109/ICCD50377.2020.00070.

32 Hugo Touvron, Louis Martin, and Kevin Stone et al. Llama 2: Open Foundation and Fine-Tuned Chat Models, 2023. doi:10.48550/arXiv.2307.09288.

33 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, 2023. arXiv:1706.03762.

34 Mark Wijtvliet, Luc Waeijen, and Henk Corporaal. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 235–244, 2016. doi:10.1109/SAMOS.2016.7818353.

35 Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR, 23–29 July 2023. URL: https://proceedings.mlr.press/v202/xiao23c.html.

36 Eunji Yoo, Gunho Park, Jung Gyu Min, Se Jung Kwon, Baeseong Park, Dongsoo Lee, and Youngjoo Lee. TF-MVP: Novel Sparsity-Aware Transformer Accelerator with Mixed-Length Vector Pruning. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023. doi:10.1109/DAC56929.2023.10247799.

37 Shulin Zeng, Jun Liu, Guohao Dai, Xinhao Yang, Tianyu Fu, Hongyi Wang, Wenheng Ma, Hanbo Sun, Shiyao Li, Zixiao Huang, Yadong Dai, Jintao Li, Zehao Wang, Ruoyu Zhang, Kairui Wen, Xuefei Ning, and Yu Wang. FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '24, pages 223–234, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3626202.3637562.