Proceedings of

16TH INTERNATIONAL SYMPOSIUM ON HIGH-LEVEL PARALLEL PROGRAMMING AND APPLICATIONS
HLPP 2023

June 29–30, 2023

Cluj-Napoca, Romania

Hosted by the Babeș-Bolyai University

Faculty of Mathematics and Computer Science

Editors: Virginia Niculescu, Adrian Sterca and Darius Bufnea

DRAFT VERSION

Foreword

Welcome to the 16th International Symposium on ***High-level Parallel Programming and Applications, HLPP 2023.***

As processor and system manufacturers adjust their roadmaps towards increasing levels of both inter and intra-chip parallelism, so the urgency of reorienting the mainstream software industry towards these architectures grows. At present, popular parallel and distributed programming methodologies are dominated by low-level techniques such as send/receive message passing, or equivalently unstructured shared memory mechanisms. Higher-level, structured approaches offer many possible advantages and have a key role to play in the scalable exploitation of ubiquitous parallelism.

HLPP 2023 is the sixteen in a series of symposiums seeking to provide a forum for discussion and research about such high-level approaches to parallel and distributed programming.

The call for papers generated 13 submissions, from Austria, Sweden, Italy, France, Portugal, Germany, Spain, Israel and USA. The Program Committee selected 8 papers for inclusion in the final program, 6 of these being proposed to be published in a special issue of *the International Journal of Parallel Programming* (IJPP), dedicated to HLPP 2023 symposium. In addition to the refereed papers, the final program includes one invited keynote, by Dana Petcu (West University of Timișoara, Romania).

We would like to thank the authors and speakers for providing the content of the program. We also would like to express our gratitude to the program committee, who worked very hard in reviewing the papers and providing suggestions for their improvements. A special thank you goes to the sponsors of the event: HUAWEI Technologies France SASU, Robert Bosch SRL and Babeş-Bolyai University. We are also grateful to the maintainers of the EasyChair conference system for hosting the refereeing process.

HLPP 2023 Program Chairs:

Virginia Niculescu
Babeş-Bolyai University, Cluj-Napoca, Romania
Adrian Sterca
Babeş-Bolyai University, Cluj-Napoca, Romania
Darius Bufnea
Babeş-Bolyai University, Cluj-Napoca, Romania

HLPP 2023 Symposium Organization

Chairs

- Virginia Niculescu, Babeș-Bolyai University of Cluj-Napoca, Romania
- Adrian Sterca, Babeș-Bolyai University of Cluj-Napoca, Romania
- Darius Bufnea, Babeș-Bolyai University of Cluj-Napoca, Romania

Program Committee Members

- Miguel Areias, University of Porto
- Jost Berthold, Runtime-Verification
- Murray Cole, The University of Edinburgh
- Iacopo Colonnelli, University of Torino
- Frédéric Dabrowski, LIFO – Université d'Orléans
- Marco Danelutto, Dept. Computer Science – Univ. Pisa – Italy
- Inês Dutra, University of Porto
- Jose Daniel Garcia, University Carlos III of Madrid
- Frédéric Gava, University of Paris-East
- Alex Gerbessiotis, NJIT
- Sergei Gorlatch, University of Muenster
- Clemens Grelck, University of Amsterdam
- Dalvan Griebler, PUCRS/SETREM
- Gaétan Hains, University of Paris-Est
- Christoph Kessler, Linköping University
- Peter Kilpatrick, Queen's University Belfast
- Herbert Kuchen, University of Muenster
- Kiminori Matsuzaki, Kochi University of Technology
- Gianluca Mittone, University of Torino
- Virginia Niculescu, Babes-Bolyai University
- Dana Petcu, West University of Timisoara
- Aleksandar Prokopec, Ecole Polytechnique Fédérale de Lausanne
- Kostis Sagonas, Uppsala University
- Massimo Torquati, University of Pisa
- Jesper-Larsson Träff, TU Wien Informatics
- Ana Lucia Varbanescu, University of Amsterdam

Steering Committee

- Gaétan Hains (Université Paris-Est Créteil, France)
- Clemens Grelck (Universiteit van Amsterdam, Netherlands)
- Kiminori Matsuzaki (Kochi University of Technology, Japan)
- Christoph W. Kessler (Linköping University, Sweden)
- Herbert Kuchen (University of Münster, Germany)
- Marco Danelutto (University of Pisa, Italy)
- Ines Dutra (University of Porto, Portugal)
- Arturo Gonzalez-Escribano (Universidad de Valladolid, Spain)
- Frédéric Dabrowski (Université d'Orléans, France)
- Virginia Niculescu (Babeș-Bolyai University, Romania)

Local Organization Committee

- Alexandru Kiraly
- Alexandrina Colț
- Claudia Coste

Sponsors of the Symposium:

HUAWEI Technologies France SASU BOSCH Robert Bosch SRL Babeș-Bolyai University

Invited Keynote

Dana Petcu

Professor at the West University of Timișoara, Romania
Dean of the Faculty of Mathematics and Computer Science

Title: ***Extreme Data Processing in Exascale Systems***

Abstract: Extreme data refers to massive amounts of data that must be queried, communicated and analyzed in near real-time. Analyzing massive scientific data gathered in each second, mining huge sets of images in crisis management, or dealing with millions of social data posts for are few examples. Processing them on exascale systems requires new programming models. Data-aware basic operations for data-intensive applications supporting the scale use of a massive number of processing elements were introduced recently (a bird-view of these proposals will be presented in the talk). As follow ups, solutions for anomaly and event detections and monitoring in data processing were proposed (a focused view of these proposals will be provided in the talk). However, moving part of the data (pre-)processing out of the exascale system towards the edge of the network is another solution and techniques like transprecision computation or machine learning to adapt to the device capabilities are able to reduce the load for the exascale system.

Table of contents

Symposium on High-Level Parallel Programming & Applications
HLPP 2023

Automatic Discovery of Collective Communication Patterns in Parallelized Task Graphs

**Fabian Knorr · Philip Salzmann ·
Peter Thoman · Thomas Fahringer**

Abstract Collective communication APIs equip MPI vendors with the necessary context to optimize cluster-wide operations on the basis of theoretical complexity models and characteristics of the involved interconnects.

Modern HPC runtime systems with a programmability focus can perform dependency analysis to eliminate the need for manual communication entirely. Profiting from optimized collective routines in this context often requires global analysis of the implicit point-to-point communication pattern or tight constrains on the data access patterns allowed inside kernels.

The Celerity API provides a high degree of freedom for both runtime implementors and application developers by tieing transparent work assignment to data access patterns through user-defined range-mapper functions. Canonically, data dependencies are resolved through an intra-node coherence model and inter-node point-to-point communication.

This paper presents *Collective Pattern Discovery* (CPD), an online, fully distributed, coordination-free method for detecting collective communication patterns on parallelized task graphs. Through extensive scheduling and communication microbenchmarks as well as a strong scaling experiment on a compute-intensive application, we demonstrate that CPD can achieve substantial performance gains in the Celerity model.

Keywords Task Graph · Scheduling · MPI · Collective Communication

1 Introduction

As we enter the Exascale era with ever increasing parallelism and heterogeneity in clusters, a growing number of HPC applications become bound primarily by memory and communication bottlenecks. Efficiently managing communication

Fabian Knorr, Philip Salzmann, Peter Thoman, Thomas Fahringer
University of Innsbruck, Austria
E-mail: firstname.lastname@uibk.ac.at

between memory hierarchies is now of the utmost importance for scaling any application beyond a small number of compute nodes.

With traditional HPC software stacks – i.e. MPI+X – these hardware developments necessitate an increasing level of expertise in parallelization and distributed software optimization on part of the application programmer. However, as the actual domain of the computations performed on HPC systems is generally some other physical science, such expertise is only available to large projects consortia, or by leveraging existing domain-specific software packages.

This state of the art hampers the development of new algorithms and science, as there is a clear trade-off: experiment with new algorithmic and scientific approaches while restricted to smaller-scale or less efficient computation; or accept the limits of existing software packages, but scale more easily to larger systems and problem sizes.

One approach towards bridging this gap between a focus on allowing relatively straightforward implementation of domain science on the one hand and the complexities of large heterogeneous distributed memory clusters on the other hand are *HPC runtime systems* which seek to automate aspects like data distribution. While systems like Celerity [14] can greatly reduce the burden on the application programmer, meeting the high degree of freedom necessary to target the vast cosmos of data access patterns found in scientific computing requires a communication model built around point-to-point primitives in the general case.

For communication patterns involving a large number of cluster nodes however, collective communication primitives as found in MPI can outperform point-to-point cascades in network latency and throughput while also reducing tracking overhead in the runtime. In this paper, we suggest that the conflict in requirements between API expressiveness, programmability and communication efficiency can best overcome by automated pattern detection and optimization on an existing point-to-point model.

To substantiate this claim, we present *Collective Pattern Discovery* for the Celerity model, a method which automates detection of data access patterns that map to collective communication steps and inserts eager collective communication steps where possible. Our approach is deterministic and fully distributed without coordination between participating nodes and exhibits low overhead. It neither requires training, observation of previous communication nor guidance from the application developer.

1.1 MPI Collectives

The MPI Standard [10] defines five categories of non-mutating collective operations that can replace equivalent, hand-rolled point-to-point communication cascades for improved latency and throughput.

These collectives are either symmetric or revolve around one *root* node; and transmitted data is either *personalized* (nodes receive disjoint buffer subranges) or *non-personalized* (every node receives the full buffer range).

collective	operation	MPI function
broadcast	non-personalized one-to-all	`MPI_Bcast`, `MPI_Ibcast`
scatter	personalized one-to-all	`MPI_Scatter[v]`, `MPI_Iscatter[v]`
gather	all-to-one	`MPI_Gather[v]`, `MPI_Igather[v]`
all-gather	non-personalized all-to-all	`MPI_Allgather[v]`, `MPI_Iallgather[v]`
all-to-all	personalized all-to-all	`MPI_Alltoall[vw]`, `MPI_Ialltoall[vw]`

The significance of efficient collectives for MPI application performance becomes apparent in the extensive library of research on optimizing these operations in popular implementations [13,9]. Accurate theoretical models allow latency- and throughput-optimized implementations to select optimal communication patterns depending on cluster topology [5] and problem size [11].

1.2 Celerity

Celerity is a high-level C++ runtime system for accelerator clusters, focusing on programmability in the complex environment of distributed-memory accelerator computing [14]. It provides developers with a dataflow-based parallelism model reminiscent of single-GPU programming while transparently distributing computation across compute nodes. In order to ease adoption and leverage existing standards as far as possible, its programming interface is closely related to the established SYCL API, with minimal extensions required for operation on distributed memory [6].

Celerity is built around fully distributed and asynchronous task and command graph generation, which has previously been shown to scale up to 128 GPUs for compute-intensive algorithms [12]. However, prior to this work, Celerity's implicit communication model was exclusively implemented through asynchronous MPI point-to-point operations.

1.3 Case Study: Direct N-Body Simulation

To familiarize the reader with the Celerity model and demonstrate the performance impact of collective communication later in this paper, we showcase the implementation of a direct gravitational N-body simulation as defined by

$$v_{i,t+1} := v_{i,t} + \sum_{j \neq i} \frac{G m_j (p_j - p_i)}{\|p_j - p_i\|^3} \Delta t, \qquad p_{i,t+1} := p_{i,t} + v_{i,t+1} \Delta t, \tag{1}$$

where p are 3-dimensional body positions, v their velocities, m their masses, G the gravitational constant and t are time steps of length Δt.

The abbreviated Celerity program in listing 1 represents this system in two virtualized buffers P and V. In a loop, it submits two kernels per time step: `time_step` computes $v_{i,t+1}$ from $v_{i,t}$ by integrating over the entirety of P for each work item i; then `update_p` updates $p_{i,t+1}$ in-place from $p_{i,t+1}$ and $v_{i,t+1}$.

```
 1  using namespace celerity;
 2
 3  buffer<double3, 1> P(N);
 4  buffer<double3, 1> V(N);
 5  const double M = 1.0 /* kg */;
 6
 7  distr_queue q;
 8  for (double t = 0.0f; t < T; t += dt) {
 9      q.submit([&](handler &cgh) {
10          accessor p(P, cgh, access::all(), read_only);
11          accessor v(V, cgh, access::one_to_one(), read_write);
12          cgh.parallel_for<class time_step>(range<1>(N), [=](item<1> i) {
13              double F = 0.0;
14              for (size_t j = 0; j < N; ++j) { F += gravity(p[i], p[j]); }
15              v[j] += M * F * dt;
16          });
17      });
18      q.submit([&](handler &cgh) {
19          accessor v(V, cgh, access::one_to_one(), read_only);
20          accessor p(P, cgh, access::one_to_one(), read_write);
21          cgh.parallel_for<class update_p>(range<1>(N),
22              [=](item<1> i) { p[i] += v[i] * dt; });
23      });
24  }
```

Listing 1 Simplified implementation of direct N-body simulation in Celerity.

Each kernel is submitted as part of an asynchronous *command group*, which ties the kernel function to an *execution geometry* (lines 12 and 21) and any number of *buffer accessors* (lines 10–11 and 19–20).

The execution geometry describes parallelization through a dimensionality (here 1), an execution range (here N), a work item offset (implicitly 0 here) and a work-group size (implicit and implementation-defined here).

Through lambda captures, accessors inject device-buffer pointers into the kernel while providing the scheduler with metadata in the form of an *access mode* (here `read_only`, `read_write`) and a *range mapper* (here `all` and `one_to_one`).

1.4 Range Mappers

Range mappers are an essential concept of the Celerity model, mapping sub-ranges of the execution range to sub-ranges of the buffer in an accessor. This enables the discovery of data requirements after arbitrary work assignment.

Given an execution range E, a **range mapper** $r : \mathcal{P}(E) \to \mathcal{P}(E)$ is any pure function that forms a homomorphism over the union of execution sub-ranges:

$$r(E_1 \cup E_2) \;=\; r(E_1) \cup r(E_2) \qquad \forall E_1, E_2 \subset E \tag{2}$$

Any range mapper r that is used in a writing access is further required to be **non-overlapping** to allow tracking of the unique producer for any buffer item:

$$E_1 \cap E_2 = \emptyset \;\Rightarrow\; r(E_1) \cap r(E_2) \;=\; \emptyset \qquad \forall E_1, E_2 \subset E \tag{3}$$

Celerity ships a selection of built-in range mapper functions. Relevant to the following discussion are `one_to_one` (the identity function, requires equal kernel and buffer dimensions), `all` (constant, accessing the entire buffer range) and `transposed` (an isomorphic shuffling of dimensions). Out of these, `one_to_one` and `transposed` exhibit the non-overlapping property, while `all` does not.

1.5 Graph-Based Scheduling

Celerity's parallel schedule is derived from the flow of command group submissions in two steps: The high level *task graph*, constructed synchronously on all participating nodes, describes execution on a cluster-wide level. From this task graph, each rank generates an individual *command graph* that models the kernel launches and communication steps performed within the node.

Work is assigned to accelerators by splitting the global execution range into near-equally-sized sub-ranges while observing any constraints imposed by the execution geometry. As one Celerity process usually drives all accelerators of a cluster node, scheduling will produce multiple execution sub-ranges locally. The graph generation process itself does not involve communication.

State-of-the-art Celerity resolves data-flow dependencies between nodes to point-to-point transfers. In this approach, each node tracks the buffer sub-ranges produced by kernels within its address space through a combination of R-trees [3], from which inbound communication sub-ranges (*await-push commands*) and outbound communication targets (*push commands*) are derived. Lowered to MPI point-to-point primitives, these commands satisfy any data access pattern that can be described by the range-mapper model.

Figure 1 shows an excerpt of the task and command graphs resulting from Listing 1. Here, as Celerity decides to assign the same execution sub-ranges to the same nodes across kernels, only the `all`-read requirement of `time_step` necessitate communication. The corresponding command graph contains $M-1$ push commands and one await-push command on every node out of M.

1.6 Multi-Device Execution and Memory Coherence

Each Celerity process generates and streams its command graph to its *executor* thread, which drives all accelerators addressable by the node. The executor dynamically establishes memory coherence between host and device memories by tracking buffer writes and replications in separate R-trees, issuing memory transfers before passing kernels to the SYCL backend.

While this lazy-update approach effectively balances irregular workloads, missing context about the higher-level operation each sequence of commands is part of can leads to sup-optimal execution patterns at times. This holds especially true for the all-gather pattern found in our N-body simulation, for which the executor will issue a coherence update for every incoming transfer ($M-1$ for M nodes) instead of coalescing them into a single transfer.

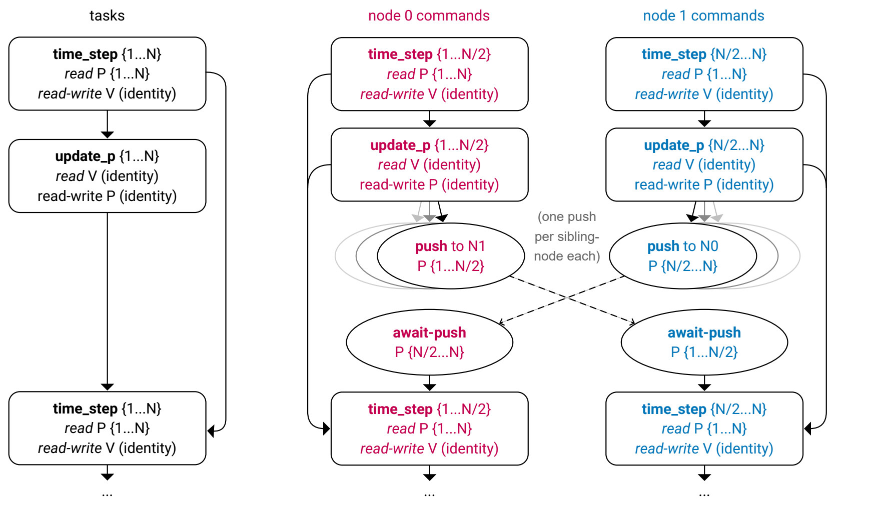


Fig. 1 Task graph (left) and command graphs (right) of a point-to-point communication schedule for direct N-body simulation from listing 1 on $M = 2$ nodes. We show tasks up to the second `time_step` kernel submission and hint at the additional push commands (grey) that would be required for a command graph on $M > 2$ nodes.

2 Related Work

Uncovering and exploiting opportunities for collective communication in user programs has been examined from different angles in recent literature.

These approaches can be broadly categorized into *bottom-up* schemes discovering collective patterns through centralized analysis of existing point-to-point programs, and *top-down* methods which derive these patterns from high-level cluster-wide representations and can frequently be coordination-free.

Knüpfer et al. [7] perform post-hoc, bottom-up analysis of application traces with MPI point-to-point communication, hinting potential sites for collective communication to the application developer help manual refactoring.

Hoefler et al. [4] use compiler transformations to replace point-to-point operations with library function calls that build a communication DAG at runtime. In a centralized bottom-up analysis pass, this approach reliably detects all regular (i.e. non-`MPI_*[vw]`) collective patterns. By re-using optimized schedules across program iterations, the authors are able to amortize the overhead of their optimization.

libWater [2] is an OpenCL-based runtime that dynamically offloads work from a designated *root node* to devices attached to other MPI processes. In a bottom-up scheme, it detects gather, scatter and broadcast patterns among the point-to-point commands generated as part of data redistribution pass and inserts MPI collective operations accordingly.

Denis et al. [1] extend the PaRSEC runtime to opportunistically discover broadcast patterns bottom-up during task graph build time. To avoid the synchronization penalty from orchestrating a call to `MPI_Bcast` from otherwise independent schedulers, the sending node initiates a binomial-tree broadcast through point-to-point messages which are forwarded by intermediate nodes.

In a top-down approach, the cluster backend of SkePU [8] leverages MPI collectives to exchange data between operations where applicable. The rigid skeleton model significantly eases the modelling of global data movement and computational patterns when compared to Celerity, which must allow near-arbitrary non-overlapping writes based on range mappers.

Collective Pattern Discovery as presented in the remainder of this document falls into the *top-down* category, analyzing data requirements of a parallelized task-graph through a distributed and coordination-free algorithm.

3 Collective Pattern Discovery

Collective Pattern Discovery (CPD) is a novel, deterministic, synchronization- and coordination-free method for detecting instances of all five collective data exchange patterns found in section 1.1. In two phases, CPD transforms both the replicated task graph and the per-node individual command graph to identify dataflow edges that can profit from eager collective communication.

By guaranteeing that all nodes generate collective commands in identical order regardless of individual work assignment, it satisfies the MPI requirement that all ranks in a communicator participate in every collective operation.

3.1 Forward Task Generation

The first step in Collective Pattern Discovery (CPD) locates *potential* edges in task graph, where an eager collective operation may preempt later point-to-point buffer updates that would be inserted lazily on command generation.

Although the task graph is oblivious to communication and fully independent of the underlying cluster configuration (including the number of participating nodes), it must still keep track of collectives to guarantee that all nodes participate in the same operations. This also avoids inadvertently exchanging buffer ranges multiple times, as the task graph will reveal whether a dataflow dependency terminates at the original data producer or whether there are intermediate tasks for which the data has potentially been exchanged before.

CPD thus inserts a **forward task** whenever a read-requirement of task c (the *consumer*) would introduce the first task-level dependency on the original writer task p (the *producer*) for the accessed region (algorithm 1).

To maximize the number of forward tasks that result in non-trivial collective communication after work assignment, CPD ignores any task edges it deems to be *communication-free* by assuming that tasks which share an execution geometry will receive identical work assignment in the scheduler.

```
$R_{t,B}$ := read-set of task $t$ on buffer $B$
$W_{t,B}$ := write-set of task $t$ on buffer $B$
$W^*_{t,B}$ := subset of $W_{t,B}$ not overwritten by any subsequent task
$A_t(F) := \{r \mid r \text{ is a range mapper in } t \wedge r(E_t) \cap F \neq \emptyset\}$

procedure IsCommunicationFree($p$, $c$, $F$)
    {Assume tasks of identical geometry will have identical work assignment}
    if $F = \emptyset$ then return True
    else if either $p$ or $c$ is a forward task then return False
    else if $p$ and $c$ have different execution geometry then return False
    else if $A_p(F, \text{write}) = A_c(F, \text{read})$ then return True
    else return False

procedure GenerateForwardTasks($t$)
    for each buffer $B$ with $R_{t,B} \neq \emptyset$ do
        for each previous task $p \neq t$ with $W^*_{p,B} \cap R_{t,B} \neq \emptyset$ do
            $F \leftarrow W^*_{t,B} \cap R_{t,B}$
            for each task $c \notin \{t, p\}$ dependent on $p$ do
                if not IsCommunicationFree($p$, $c$, $W^*_{p,B} \cap R_{c,B}$) then
                    $F \leftarrow F \setminus R_{c,B}$
            if not IsCommunicationFree($p$, $t$, $F$) then
                insert forward-task $f$ with dependencies $p \rightarrow f \rightarrow t$
                $R_{f,B} \leftarrow W_{f,B} \leftarrow F$
```

Algorithm 1 Forward-task generation for a command-group task t

3.2 Eager Collective Command Generation

In the Celerity model, work assignment and thus the number of nodes participating in a task is a function of the execution geometry and the number of nodes and accelerators in the system. This ensures that command graph generation, while distributed, agrees on a single global schedule. Our implementation guarantees this through fully-static scheduling, although dynamic methods and still apply even with CPD, provided that they remain deterministic and reproducible around forward tasks.

After work assignment, the second step of CPD materializes forwards between producer and consumer tasks as **collective commands** if they match one of the patterns found in table 1. Any non-matching forward task is dropped, and communication will proceed through the general point-to-point algorithm.

The pattern matching approach is independent of the exact buffer regions each node accesses, rather, the collective operation is determined in constant

collective	producer nodes	consumer nodes	producer range mappers	consumer range mappers
gather	M	1	non-overlapping	any
all-gather	M	M	non-overlapping	constant
broadcast	1	M	non-overlapping	constant
scatter	1	M	non-overlapping	non-overlapping
all-to-all	M	M	— non-trivial transposition —	

Table 1 Discovery patterns for collective operations on $M > 1$ nodes

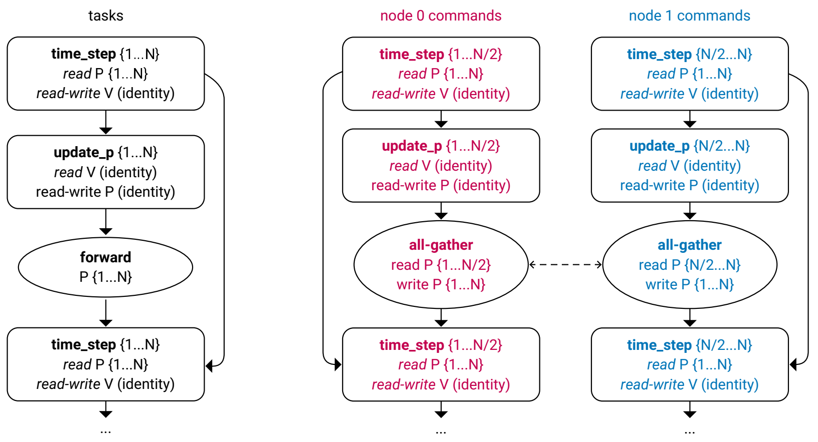


Fig. 2 Task graph (left) and command graphs (right) of a direct N-body simulation with Collective Pattern Discovery. The forward task on P materializes as a all-gather operation, replacing the push-await cascade seen in figure 1.

time from the number producer and consumer commands and range-mapper metadata. The non-overlapping property of producer (writer) range mappers is assumed to hold by definition (see section 1.4). Our implementation detects *constant* and *non-overlapping* consumer range mappers as well as *transpositions* through meta-programming on the range-mapper functions.

The common *gather*, *all-gather*, *scatter* and *broadcast* patterns are identified by analyzing read- and write range mappers in separation.

The *all-to-all* communication pattern is identified through a consumer access that forms a *non-trivial transposition* of the corresponding producer, i.e. one that is not communication-free after work assignment:

1. Producer task p has exactly one write range mapper w; consumer task c exactly one read range mapper r participating in the forwarded region F
2. It holds that $w(E_p) = r(E_c) = F$
3. For any dimension d, all mappings of nodes i to produced buffer ranges $w_d(E_{p,i})$ and $r_d(E_{c,i})$ are either constant or the identity function
4. There exists d such that $w_d(E_{p,i})$ is constant while $r_d(E_{c,i})$ is the identity
5. There exists d such that $w_d(E_{p,i})$ is the identity while $r_d(E_{c,i})$ is constant.

Figure 2 visualizes the effects of Collective Pattern Discovery on command-graph generation for the N-body simulation in listing 1.

Collective Pattern Discovery first analyzes the data flow between the initial `time_step` and `update_p` tasks. Since producer and consumer both access buffer V through the same identity range mapper and the tasks have identical execution geometry, the edge is considered to be communication-free and no forward task is generated.

The read of $P\{1 \dots N\}$ by the second `time_step` kernel however applies a different range mapper than the producer `update_p`. As the buffer has not been read by any task since, CPD inserts a forward task on $P\{1 \dots N\}$.

After work assignment, the producer–consumer relationship around P connects an M-node non-overlapping producer to a M-node constant consumer, matching the all-gather pattern of table 1. Celerity thus inserts an *all-gather* command on each node, which becomes the new writer of $P\{1 \ldots N\}$.

Since all data requirements of the second `time_step` are now fulfilled, no additional push-await pairs are generated during dependency analysis.

3.3 Collective Command Execution

Celerity lowers all collective commands to their non-blocking MPI counterparts (e.g. `MPI_Iallgatherv`). As required by the standard, these operations are initiated in-order, but can overlap for the remainder of their execution time.

Since each process potentially drives multiple accelerators, the runtime compiles larger device-to-device collectives from the host-to-host MPI operations by issuing local memory transfers before and after the MPI invocation.

Knowledge about the cluster-wide collective operation provides optimization potential beyond the lazy coherence update mechanism (section 1.6) employed for point-to-point transfers: Celerity will issue a parallel *device broadcast* to update all accelerator memories after completing an MPI collective operation with receiver-broadcast semantics (*broadcast* and *all-gather* patterns).

4 Evaluation

To assess the performance characteristics of Collective Pattern Discovery in isolation, we implement a set of synthetic benchmarking applications that require communication between device memories (table 2).

Where applicable, one-to-all communication is paired with an all-to-one operation to maintain meaningful dataflow throughout the programs. As control we study the overhead of CPD on a stencil-like program with a neighborhood exchange pattern that does not benefit from collective communication.

benchmark	step	kernel	reads	writes
all-gather		N	$B \leftarrow \{1 \ldots N\}$	$B' \leftarrow$ identity
gather-scatter	1.	1	$B \leftarrow \{1 \ldots N\}$	$B \leftarrow \{1 \ldots N\}$
	2.	N	$B \leftarrow$ identity	$B' \leftarrow$ identity
gather-bcast	1.	1	$B \leftarrow \{1 \ldots N\}$	$B \leftarrow \{1 \ldots N\}$
	2.	N	$B \leftarrow \{1 \ldots N\}$	$B' \leftarrow$ identity
all-to-all		$N \times N$	$B \leftarrow \text{transpose}(0, 1)$	$B' \leftarrow$ identity
stencil (control)		$N \times N$	$B \leftarrow \text{neighborhood}(1, 1)$	$B' \leftarrow$ identity

Table 2 Access patterns of the synthetic benchmarks examined in this section. Executing the steps of each program in a loop generates detectable collective communication patterns (except *stencil*). After each iteration, buffers B and B' are swapped.

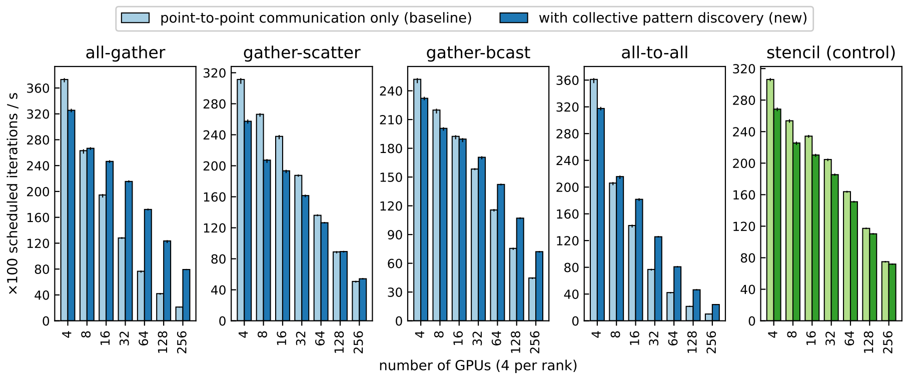


Fig. 3 Scheduler throughput for each program listed in table 2 (higher is better). Reported is median of 100 benchmarks together with minima and maxima.

All benchmarks in this section were run on the Marconi-100 supercomputer in Bologna, Italy, rank 26 of the TOP500 list as of June 2023[1]. It is a cluster of 980 IBM Power AC922 nodes with four Nvidia Volta V100 GPUs each, intra-node NVLink 2.0, and dual Infiniband EDR system interconnect.

Celerity was built using Clang 12.0.1 and OpenSYCL 0.9.2[2] with `-O3` optimization, linking against CUDA 11.7 and IBM Spectrum MPI 10.4.0. All binaries were executed with mimalloc 2.0.9[3] replacing the system allocator.

4.1 Scheduling Microbenchmarks

Celerity generates task- and command graphs concurrently with kernel execution and data transfers. Scheduling latency can thus usually be hidden after startup, but applications with very short device kernels may become throughput-limited.

By isolating the scheduling process, we can analyze scheduler throughput as a function of node count. Each node must compute the work assignment of every other node in the system to detect potential non-collective data requirements. The number of communication commands tracked however remains constant with CPD while increasing linearly with point-to-point communication.

Figure 3 demonstrates that all patterns except *gather-scatter* greatly profit from CPD's reduction in tracking complexity, with *all-gather* achieving a more than 3× throughput increase for 256 nodes. For small node counts, the constant-time overhead of forward-task generation yields a visible drop in scheduler performance, both for collective and non-collective patterns. As we will show in section 4.2, this reduction in throughput is negligible for large-scale runs.

[1] https://www.top500.org/lists/top500/list/2023/06/
[2] https://github.com/OpenSYCL/OpenSYCL/releases/tag/v0.9.2
[3] https://github.com/microsoft/mimalloc/releases/tag/v2.0.9

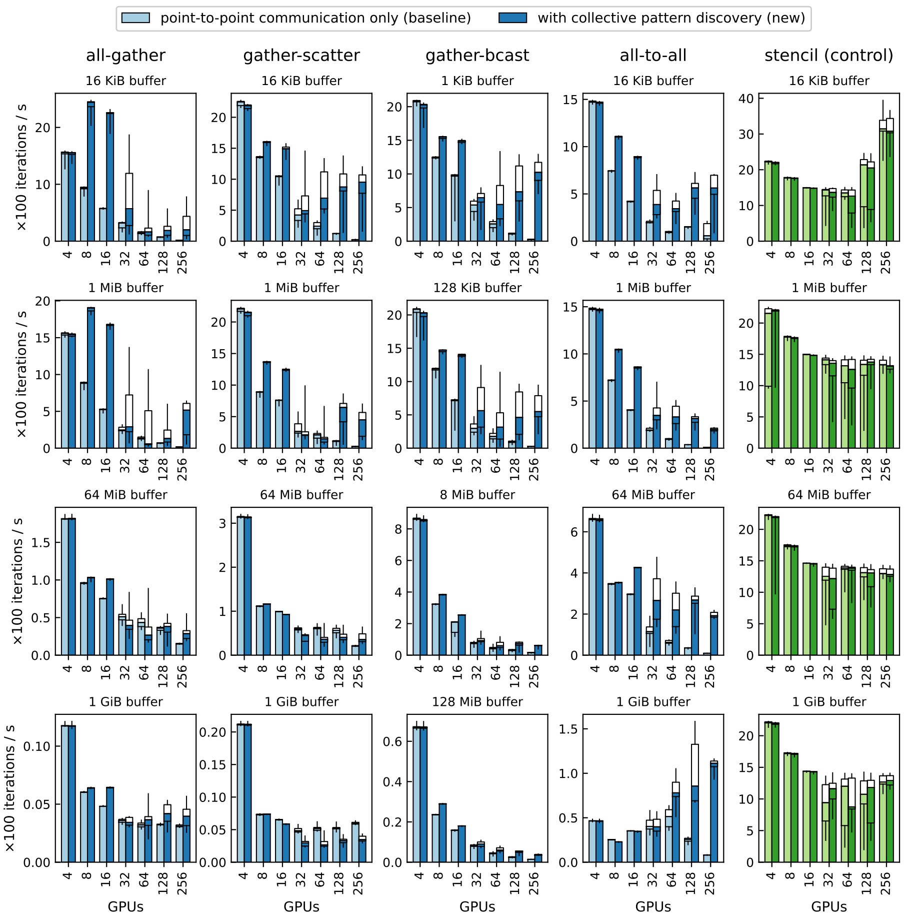


Fig. 4 Throughput of communication-only system benchmarks from table 2 with kernel execution disabled (higher is better). Shown is a mixed bar-box plot containing the median, center quartiles and minima / maxima over 20 runs on varying node configurations. Each measurement is the mean over 20 iterations.

4.2 Communication-Only System Benchmarks

As Celerity is structured around accelerator computation, we benchmark device-to-device transfer performance specifically by executing the synthetic benchmarks from table 2 with and without CPD while disabling kernel execution.

Figure 4 visualizes the communication throughput achieved as benchmark iterations per second. All collective patterns profit massively from reduced overheads on small buffer sizes, and all except *gather-scatter* can consistently take advantage of reduced bandwidth requirements on larger-sized buffers.

For large node counts, we can observe a high variance in the performance of MPI collective communication, which is caused by process scheduling differences on part of the SLURM workload manager.

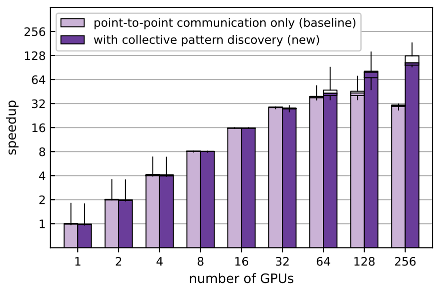


Fig. 5 Strong-scaling speedup of 20 time steps of the direct N-body simulation for $N = 1{,}048{,}576$ in double precision. We report the median, center quartiles and minima / maxima over 20 benchmark runs allocated to varying node configurations by the workload manager.

The non-collective stencil program shows no difference in communication times between enabling or disabling CPD, demonstrating that the increase in scheduler latency seen in figure 3 can be fully hidden.

4.3 Strong Scaling Experiment: Direct N-Body Simulation

To evaluate the efficacy of CPD on a full application, we implement and optimize the direct N-body simulation from section 1.3 as a Celerity application. Compared to the simplified listing 1, we use an array-of-struct (AOS) to struct-of-array (SOA) transformation on P and V, increase parallelism in `time_step` by writing one item in V per 32 threads and reduce the required global-memory bandwidth in the same kernel by shared-memory tiling the read of V.

We choose a strong-scaling experiment specifically to showcase the effects of transitioning from a compute-bound to a communication-bound problem as the node count increases. Figure 5 shows the speedup attained from a varying number of GPUs participating in the simulation of $N = 1{,}048{,}576$ bodies.

Up to 64 GPUs (16 nodes), both point-to-point and collective communication scale equally. Increasing beyond 128 GPUs yields no additional speedup for the point-to-point configuration, but does so significantly when Collective Pattern Discovery is enabled.

Profiling reveals that scalability in this case is limited primarily by latency of small host-to-device copies for every incoming message, which CPD can effectively reduce through the use of a *device broadcast* (section 3.3).

5 Conclusion

This work introduces Collective Pattern Discovery (CPD), a novel, distributed and coordination-free method for reliably identifying opportunities for collective communication in the parallelized task graphs of the Celerity model.

In a two-stage approach, CPD identifies task graph edges suitable for eager communication in the form of *forward tasks* and matches the concrete data exchange pattern after work assignment to generate per-node *collective commands*. This transforms a large class of distributed-memory interactions into collective operations while reliably avoiding duplicated communication.

Through synthetic scheduling and communication benchmarks, we demonstrated how CPD reduces tracking overhead of large runs in the runtime system by replacing a linear number of point-to-point communication pairs with singular collective operations. On large transfers, this transformation allow us to profit from decades of research on MPI collective optimization.

In a strong-scaling experiment, we were able to prove sizable gains in scalability over the point-to-point model, effectively scaling a direct N-body simulation implemented in Celerity to 256 GPUs for the first time.

5.1 Limitations and Future Work

While demonstrably highly efficient in common settings, the graph transformations performed by Collective Pattern Discovery (CPD) cannot claim algorithmic optimality in the general case. For example, the eager generation of forward tasks masks the original producer task of the forwarded buffer subregion: if the forward is not materialized, or later tasks would benefit from a superset of the generated collective (e.g. a logical *all-gather* access following a simple *gather*), an opportunity for collective communication will be missed. Future work could be able to improve CPD in these situations through a lookahead scheme analyzing longer sequences of tasks at once.

Acknowledgements

This research is supported by the European High-Performance Computing Joint Undertaking (JU) project LIGATE under grant agreement No 956137.

References

1. Denis, A., Jeannot, E., Swartvagher, P., Thibault, S.: Using dynamic broadcasts to improve task-based runtime performances. In: Euro-Par 2020, Warsaw, Poland, August 24–28, 2020, Proceedings 26. pp. 443–457. Springer (2020)
2. Grasso, I., Pellegrini, S., Cosenza, B., Fahringer, T.: libWater: Heterogeneous distributed computing made easy. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. p. 161–172. ICS '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2464996.2465008
3. Guttman, A.: R-trees: a dynamic index structure for spatial searching. ACM SIGMOD Record **14**(2), 47–57 (Jun 1984). https://doi.org/10.1145/971697.602266
4. Hoefler, T., Schneider, T.: Runtime detection and optimization of collective communication patterns. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques. p. 263–272. PACT '12, ACM/doi, New York, NY, USA (2012). https://doi.org/10.1145/2370816.2370856
5. Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., Bhoedjang, R.A.F.: MagPIe: MPI's collective communication operations for clustered wide area systems. In: Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 131–140. PPoPP '99, ACM, New York, NY, USA (1999). https://doi.org/10.1145/301104.301116
6. Knorr, F., Thoman, P., Fahringer, T.: Declarative data flow in a graph-based distributed memory runtime system. International Journal of Parallel Programming pp. 1–22 (2022)

7. Knüpfer, A., Kranzlmüller, D., Nagel, W.E.: Detection of collective MPI operation patterns. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11. pp. 259–267. Springer (2004)
8. Majeed, M., Dastgeer, U., Kessler, C.: Cluster-SkePU: A multi-backend skeleton programming library for GPU clusters. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). p. 468. Citeseer (2013)
9. Mamidala, A.R., Kumar, R., De, D., Panda, D.K.: MPI colblectives on modern multicore clusters: Performance optimizations and communication characteristics. In: 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID). pp. 130–137. IEEE (2008)
10. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 4.0, https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
11. Pjesivac-Grbovic, J., Angskun, T., Bosilca, G., Fagg, G.E., Gabriel, E., Dongarra, J.J.: Performance analysis of MPI collective operations. In: 19th IEEE International Parallel and Distributed Processing Symposium. pp. 8–pp. IEEE (2005)
12. Salzmann, P., Knorr, F., Thoman, P., Gschwandtner, P., Cosenza, B., Fahringer, T.: An asynchronous dataflow-driven execution model for distributed accelerator computing. In: 2023 23rd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid). p. (to appear). IEEE (2023)
13. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. The International Journal of High Performance Computing Applications **19**(1), 49–66 (2005)
14. Thoman, P., Salzmann, P., Cosenza, B., Fahringer, T.: Celerity: High-level C++ for accelerator clusters. In: Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings 25. pp. 291–303. Springer (2019)

Symposium on High-Level Parallel Programming & Applications
HLPP 2023

High-Level Programming of FPGA-Accelerated Systems with Parallel Patterns

Björn Birath · August Ernstsson · John Tinnerholm · Christoph Kessler

Abstract As a result of frequency and power limitations, multi-core processors and accelerators are becoming more and more prevalent in today's systems. To fully utilize such systems, heterogeneous parallel programming is needed, but this introduces new complexities to the development. High-level frameworks such as SkePU have been introduced to help alleviate these complexities. SkePU is a skeleton programming framework based on a set of programming constructs implementing computational parallel patterns, while presenting a sequential interface to the programmer. Using the various skeleton backends, SkePU programs can execute, without source code modification, on multiple types of hardware such as CPUs, GPUs, and clusters. This paper presents the design and implementation of a new backend for SkePU, adding support for FPGAs. We also evaluate the effect of FPGA-specific optimizations in the new backend. For simple examples, we find that the FPGA-backend's performance is similar to that of the existing backend for GPUs, while it falls behind in more complex tasks. Finally, some shortcomings in the backend are highlighted and discussed, along with potential solutions.

Keywords algorithmic skeletons · reconfigurable computing · FPGA · single-source heterogeneous programming

1 Introduction

For a long time, the trend in computer architecture has been the move to multi-core processors. Additionally, the use of accelerators such as massively parallel GPUs has increased, leading to many of today's systems being heterogeneous.

An alternative accelerator to GPUs is the field-programmable gate array (FPGA). The strength of FPGAs is that they can be reconfigured and adapted

PELAB, Dept. of Computer and Information Science
Linköping University, Linköping, Sweden
E-mail: bjorn@birath.dev, <firstname>.<lastname>@liu.se

for the type of algorithms to execute, mapping an algorithm one-to-one to the FPGA hardware. This involves "programming" the FPGA using a Hardware Description Language (HDL) such as VHDL or Verilog [6]. The HDLs are used to generate a circuit description which is loaded onto the FPGA.

Historically, this process of programming FPGAs has required specialized training since HDLs lack many high-level constructs found in conventional programming languages and use a parallel data flow model rather than a sequential one. There are also differences between FPGA platforms, leading to difficulties of reusing existing designs [12]. To alleviate these issues, there have been many attempts to create tools that utilize higher level languages, such as C or C++, to automatically produce a circuit specification in a HDL. These high-level synthesis (HLS) tools allow developers to program FPGAs faster and without hardware expertise [18]. More recently, both Intel[1] and Xilinx[2] introduced HLS toolchains based on *OpenCL*, a framework for creating portable parallel programs targeting multiple types of platforms.

While these tools make it easier to program FPGAs, developers still need to handle the challenges of programming against heterogeneous processors. This includes communication, memory management and synchronization. Here, skeleton programming frameworks can provide a more high-level interface for the developer [11] by abstracting from some of the more complex interactions and specifics of a multiprocessor system. One such framework is SkePU[3], an open-source skeleton programming framework for heterogeneous parallel systems. Today SkePU supports multi-core CPUs, GPUs and clusters. By adding support for FPGAs, it would allow developers to program FPGAs without the need for hardware expertise or deep knowledge of OpenCL.

To this end, we design and implement a new backend in SkePU targeting reconfigurable architectures by integrating an existing OpenCL HLS toolchain. This will allow SkePU to further accelerate the types of problems that FPGAs are particularly suitable for, such as problems that can take advantage of the high-throughput pipelines that FPGAs can create, while keeping full source-code portability with multicore CPU, GPU and cluster execution.

Overall, this paper makes the following main contributions:

- We present the design and implementation of a new OpenCL-based backend for FPGA, including FPGA-specific optimizations, atop Intel OpenCL SDK for FPGA, for the SkePU skeletons Map, Reduce, MapReduce, Scan, and MapOverlap.
- We evaluate the FPGA backend on an Intel Programmable Acceleration Card with an Arria 10 GX FPGA. We demonstrate the great performance benefit of applying FPGA-specific optimizations such as loop unrolling, register pipelining and skeleton fusion in the backend. We also identify challenges for future improvements of the FPGA backend, such as performance issues with complex kernels.

[1] `https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html`

[2] `https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/ehb1504034292718.html`

[3] `https://skepu.github.io/`

```cpp
#include <skepu>

float mult(float a, float b) { return a * b; }
float add(float a, float b)  { return a + b; }

int main(int argc, char *argv[])
{
    size_t const size = 100;
    skepu::Vector<float> a(size), b(size);
    auto dot_product = skepu::MapReduce(mult<float>, add<float>);
    float res = dot_product(a, b);
}
```

Listing 1: SkePU program that calculates the dot product of two vectors.

The remainder of this paper is organized as follows: Section 2 presents background on algorithmic skeletons and SkePU, and Section 3 on FPGAs. Section 4 discusses related work. Section 5 presents the implementation of the FPGA backend, Section 6 experimental results and discussion. Section 7 concludes and proposes future work.

2 Skeleton programming and SkePU

Skeleton programming is a programming model offering pre-built *skeletons*, generic programming constructs derived from higher-order functions that match different computation patterns for which parallel implementations are provided by the framework. Problem-specific user code is inserted as function arguments (*user functions*) into a skeleton to instantiate a complete algorithm [5].

SkePU [9] is a C++ open-source skeleton programming framework that provides an interface to create parallel computations with support for different backends: sequential, multi-core CPU (OpenMP), GPU (CUDA or OpenCL), cluster (StarPU-MPI), and combinations of those. SkePU programs define user functions that the skeletons use as operators. SkePU contains a source-to-source compiler that translates user functions for each backend and a runtime library that handles the scheduling, communication and memory management between the host and backend [10]. Listing 1 shows an example SkePU program that calculates the dot product of two vectors, using the `MapReduce` skeleton and two user functions: `mult` and `add`.

SkePU implements a set of data-parallel skeleton patterns. The basic `Map` is a fundamental building block in SkePU programs, as it provides a flexible interface, e.g., with variadic input and output arity and optional use of non-trivial memory access patterns. `MapOverlap` and `MapPairs` are optimized extensions of `Map` for stencil computations and Cartesian-product patterns, respectively. Similarly, `Reduce` and `Scan` are specialized patterns for *reductions* and *prefix sums*. SkePU offers efficient combinations when a reduction is used after a map-based pattern through `MapReduce` and `MapPairsReduce`. Each skeleton is instantiated with one or more *user functions*, which contain the program-specific code, executed in parallel according to the respective pattern semantics.

SkePU provides so-called *smart data-containers* [7] that manage memory and coherency between host and backends automatically. In the latest version of SkePU there are four different types of smart containers for four different dimensionalities: `Vector` (1D), `Matrix` (2D), `Tensor3D` and `Tensor4D`.

Smart containers are C++ objects in main memory and can therefore not be used directly in user functions. While this is not needed for element-wise access, some computations require access to all elements in a container. SkePU therefore provides *proxy containers* which can be used to access any element in a container inside a user function, along with other types for partial container access.

3 Field-programmable gate arrays (FPGAs)

*FPGA*s are computer chips that can be programmed to implement different digital circuits. The term comes from the fact that FPGAs are programmable in-field even after deployment. They consist of an array of configurable logic and I/O blocks that are connected through a network with programmable switches. Modern FPGAs often also have *hard blocks*, blocks that cannot be programmed but instead implement a specific functionality such as multipliers or Ethernet interfaces. Another common hard block is RAM, since implementing RAM using the configurable logic is much less area efficient [4].

Programming an FPGA is divided into three stages. First the desired hardware circuit is described in an HDL. This is then translated to logic gates via synthesis, which generates the physical design. Later, the place-and-route stage maps the physical design to a device. In both these steps, constraint checks ensure that the design will fit on the chosen device and has no timing errors. Finally, a hardware configuration file, a *bitstream*, is generated which can be loaded onto the FPGA. The process can take hours or days to complete [2].

Hardware description languages (HDLs) describe the hardware circuits that are programmed to FPGAs. While HDLs might look procedural, they are very different. For example, rather than a sequential control flow model, HDLs use a data flow model where statements can run in parallel whenever the input is changed.

Since HDL code represents the hardware it synthesizes to, an understanding of circuit design is required to get the best use out of an FPGA. To make FPGA programming easier, there has been focused attention in the industry to create *high-level synthesis* (HLS) tools for converting code written in a high-level language to HDL code [2]. The generated HDL code can then be synthesized using the normal FPGA programming flow.

Since synthesis takes multiple hours, the standard OpenCL kernel just-in-time compilation cannot be used. Instead, the SDK compiles OpenCL into Verilog, which is passed to a synthesis program. The compiler is an extension of the LLVM compiler which first produces a LLVM intermediate representation

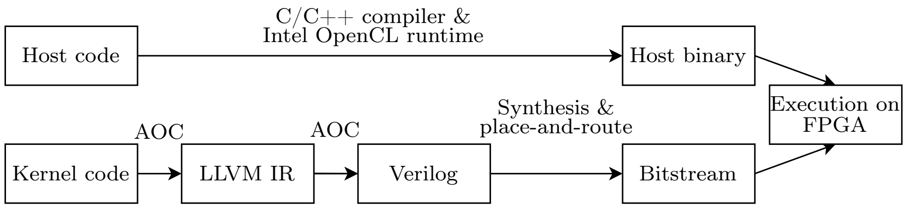


Fig. 1 Intel OpenCL SDK compilation flow. AOC is the OpenCL compiler in the SDK.

of the kernel, from which Verilog code is produced, followed by the normal FPGA programming flow of synthesis and place-and-route [6]. Figure 1 shows the compilation flow when using the SDK.

The SDK also includes libraries for the host application to program the FPGA with the generated bitstream and communicate with the kernel using the OpenCL API. These libraries do not need to be referenced explicitly in the host application: as long as they are included when compiling, they will automatically be called when using the OpenCL API. To use the SDK, a board support package (BSP) is required. The BSP includes the drivers to communicate between the FPGA, the host and the hardware configuration for generating interfaces between the OpenCL kernel and the FPGA, such as the external memory or the PCI-E. The BSP is provided by the FPGA's manufacturer and is unique to each FPGA platform. They allow the manufacturer to provide interfaces to hard blocks specific to their FPGA, like on-board networking or signal processors.

Czajkowski et al. [6] state that the reason for choosing OpenCL over a high-level language is the separation between the host and kernel. The kernel can be implemented as a highly performant hardware circuit while the host can handle the communication and programming of the FPGA. This means that the entire system can be implemented as opposed to other HLS tools that only generate HDL code for synthesis.

The SDK generates *hardware pipelines* based on the OpenCL code in the kernel. A pipeline's size is measured in terms of its *depth*, which is how many stages there are before the final output, and its *width* which is how many operations are done in parallel in each stage. The SDK supports two execution models: NDRange and Single Work-Item. Both use pipelining to achieve parallelism, but differ in how they issue new data into the pipeline.

NDRange model When using the NDRange model, the kernel is executed once for each work-item. This is the model most commonly used on GPUs, as multiple work-items can execute in parallel. As FPGAs do not have multiple processing elements, pipelining is used instead. Each region between subsequent barrier calls will generate an independent pipeline that is flushed at the barrier. The work-items are issued into the pipeline iteratively by a run-time scheduler generated in hardware. If the kernel references any of the indices in the NDRange or uses a barrier, the kernel is interpreted as a NDRange kernel [23].

```
__attribute((num_simd_work_items(2)))
__kernel void sum( __global float* a, __global float* b,
                   __global float* result) {
    int gid = get_global_id(0);
    result[gid] = a[gid] + b[gid];
}
```

Listing 2: Example of kernel vectorization.

```
__kernel void sum( __global float* a, __global float* b,
                   __global float* result) {
    int gid = get_global_id(0);
    result[gid * 2 + 0] = a[gid * 2 + 0] + b[gid * 2 + 0];
    result[gid * 2 + 1] = a[gid * 2 + 1] + b[gid * 2 + 1];
}
```

Listing 3: Manual kernel vectorization.

The SDK's programming and best practice guides recommend two techniques for better performance: Kernel vectorization and compute unit replication. *Kernel vectorization* is achieved by using the `num_simd_work_items(N)` attribute (as seen in Listing 2), which instructs the compiler to translate each scalar operator to a SIMD operation. This allows the programmer to increase the throughput of the kernel without any modifications to the kernel code or NDRange used in the invocation of the kernel [20]. The vectorization fails if the kernel contains code that the compiler deems "SIMD-unfriendly", such as thread-dependent branching. Vectorization can be done manually, like in Listing 3, but then the NDRange must be manually changed to match the number of items each kernel handles.

For *compute unit replication*, using attribute `num_compute_units(N)` instructs the compiler to replicate the full pipeline `N` times. Work-groups are split across all compute units, scheduling is handled by a hardware scheduler.

Both methods increase throughput by increasing the amount of hardware generated. It is recommended to first use kernel vectorization, as this generates coalesced memory accesses and less total hardware. The methods can also be combined, which can give better throughput depending on what type of work the kernel performs [16].

Single work-item model In the Single Work-Item model, the kernel is executed only once as a single work-item. High performance is achieved by pipelining loop iterations, mapping each outer loop to a separate pipeline. This allows multiple loop iterations to be computed in parallel, allowing a pipelined loop to finish faster than a non-pipelined loop. No run-time scheduler is used, instead the scheduling is determined at compile-time. The *initiation interval* II [1] is the number of clock cycles between two subsequent loop iterations being issued into the pipeline. For a pipelined loop with an input size of L and a depth of P, the total amount of clock cycles to complete is

$$T_{cycles} = P + II * (L - 1) \tag{1}$$

which can be converted to time in seconds by $T_{seconds} = T_{cycles}/f_{max}$, where f_{max} is the operating frequency of the FPGA. As f_{max} is often fixed for each FPGA, P not contributing much to the execution time and L being application dependent, II is the one parameter the developer can change with the largest impact on the performance of a single work-item kernel. The compiler always tries to pipeline loops so that the II is 1, but loop-carried dependencies like data dependencies or memory dependencies can cause the II to increase. The run-time II can also differ from the II determined at compile time due to stallable load and store operations or nested loops [22], so the actual execution time can be longer than the calculated $T_{seconds}$.

Parallelism similar to vectorization can be added to single work-item kernels by unrolling loops, increasing the width and depth of the pipeline at the cost of more hardware being used. If the length of the loop is known at compile time, the compiler can fully unroll the loop, otherwise it can be unrolled by a user-specified factor. On top of increasing parallelism, loop pipelining also allows the compiler to coalesce memory operations, reducing the amount of global memory accesses [21]. Equation 1 can be extended with loop unrolling:

$$T_{cycles} = P' + II * \frac{(L - N_p)}{N_p} \tag{2}$$

where P' is the new depth of the pipeline and N_p is the unroll factor. Assuming $L >> P'$, unrolling should result in a theoretical performance improvement of almost N_p times. This does, however, not take the run-time II into account, so in practice the performance improvement will not be as large.

Shift registers is a technique that can be used to relax some loop-carried dependencies in pipelined loops. Shift registers are implemented using the FPGA's registers, which have an access latency of one clock cycle [22], meaning they can be accessed without increasing the loop's II. An array in OpenCL will be implemented as a shift register if (1) the array size is known at compile time, (2) all accesses to the array are made with addresses known at compile time, and (3) all content in the array is shifted by a compile-time known amount in each loop iteration (see Listing 4).

The primary way to use a shift register is to increase the dependency distance for a variable, by writing values to one end of the shift register and operating on the other end of the shift register. This way operations that take more than one clock cycle to perform can be used without increasing the loop's II as different loop iterations will operate on different parts of the shift register. Shift registers can also be used for data sharing across loop iterations by reusing values across multiple iterations, creating a *sliding window*[15]. This is especially applicable to computations where a stencil is used, which otherwise requires multiple redundant memory accesses per loop iteration to read the input elements for the stencil. If the elements are instead stored in a shift register, one element being read per iteration, the entire stencil is accessible in a single clock cycle.

```
float reg[SIZE + 1] = {0.0};

for (int i = 0; i < N; ++i) {
    #pragma unroll
    for (int j = 0; j < SIZE; j++)
        // Shifts the content one step per iteration
        reg[j] = reg[j + 1];
    reg[SIZE] = reg[0] + input[i]
}
```

Listing 4: Creating and using a shift register in OpenCL.

4 Related work

We review three high-level frameworks similar to SkePU that target FPGAs, of which two use OpenCL and one is fully implemented in an HDL.

Melia [21] is a MapReduce framework for FPGAs that uses the Intel FPGA OpenCL SDK. The framework lets the user define a *map* and *reduce* function in the OpenCL C language which are then compiled, synthesized and programmed to an FPGA. Melia includes memory optimizations such as coalescing and "private memory optimization" and applies some FPGA-specific optimizations: Converting nested loops to a single loop, loop unrolling, and pipeline replication. Loop unrolling is the only optimization that is performed automatically on the user functions. The user is responsible for applying memory optimizations and must pass parameters to Melia for the pipeline replication and loop unrolling before synthesis. Since synthesis is a long process, Wang et al. [21] developed a cost model using the resource estimation tool included in the Intel FPGA OpenCL SDK. The cost model estimates the execution time of a given OpenCL kernel by multiplying the estimated hardware frequency and estimated amount of clock cycles needed to execute the kernel. Through testing they found that their model could closely predict the hardware frequency of a kernel, and generally capture the trend of the required clock cycles. Using the model, a user can experiment with different parameters in a matter of minutes instead of the hours it would take to complete a full synthesis. Applying the FPGA-specific optimizations to seven common MapReduce applications led to speedups of 1.4× to 43.6×. The Melia implementations demonstrated high energy efficiency compared to CPU and GPU implementations and were not much slower than the GPU implementations.

OpenACC-to-FPGA [16] is a framework for translating OpenACC C programs to a hardware configuration file for running on FPGAs. It is an extension of the Open Accelerator Research Compiler (OpenARC) [17], an open-source compiler for OpenACC which supports CUDA and OpenCL as backend programming models using source-to-source translation. OpenACC-to-FPGA uses the OpenCL backend to generate OpenCL code which is passed to the Intel FPGA OpenCL SDK to generate the hardware configuration file. To generate efficient OpenCL code for FPGAs, OpenACC-to-FPGA adds to OpenARC boundary check elimination and directives for controlling loop unrolling, kernel vectorization and compute unit replication (pipeline replication). The OpenACC-to-FPGA runtime performs dynamic memory-transfer alignment of memory that will be transferred to maximize throughput and lowering latency when transferring data between the host and FPGA memory. Correctly aligned memory on both the host and device allows the Intel FPGA OpenCL runtime to use direct memory access (DMA) between host and FPGA, speeding up data transfers. By this method, Lee et al. [16] achieved a 100-fold speed-up in data transfers between the host and device, in both directions.

In follow-up work, Lambert et al. [15] extend OpenACC-to-FPGA by optimizations for single work-item kernels. First, they added FPGA-specific loop collapsing, changing the existing OpenARC loop collapsing to calculate the indexes of the collapsed loops without modulo and divisions operations, which are relatively expensive on FPGAs. A reduce-specific optimization was also added. It generates OpenCL code for reduce loops that uses shift registers to relax the data dependency that can occur in reduce loops when using instructions that take more than one clock cycle. It also adds a new directive `window` which generates OpenCL code for creating a sliding window for stencil operations. Based on offsets used to access the stencil, it automatically generates a shift register large enough to store the stencil and the offsets to be used to access the window inside the shift register. Unlike Melia, OpenACC-to-FPGA does not provide any tools or models to alleviate the long synthesizing process.

FPMR is a MapReduce framework for FPGAs [19], though unlike Melia and OpenACC-to-FPGA it does not use any HLS tools. Instead, the framework is implemented in an HDL, providing data synchronization, scheduling and handling communication between the map and reduce tasks. The user implements the *map* and *reduce* user functions by designing a *mapper* and *reducer* processor using the corresponding interfaces in FPMR. During execution, a processor scheduler, which is implemented on the FPGA, is used to dynamically utilize the mapper and reducer processors using a set of queues for idle processors and tasks for both types of processors.

The framework uses three levels of storage: Global memory, local memory and register files inside each processor. Global memory is implemented using SDRAM modules, providing large capacity and high bandwidth. It is managed by a data controller responsible for transferring data between the host and device memory, dispatching requested data to the mappers, and storing output data from the reducers. An important feature of the data controller is the *common data path*, which allows the controller to overlap data transfer to multiple mappers at once. This is useful for applications where some data are the same for all mappers. The local memory stores the mappers' intermediate results before being passed to a reducer. It is implemented with on-chip RAM giving it low access latency, and if multiple RAMs are implemented, it can also be accessed simultaneously by mappers and reducers.

In a case study of the RankBoost algorithm using FPMR, Shan et al. [19] obtained 31.8$\times$ speedup over their CPU reference implementation, which was comparable to a manually designed FPGA implementation with 33.5$\times$ speedup.

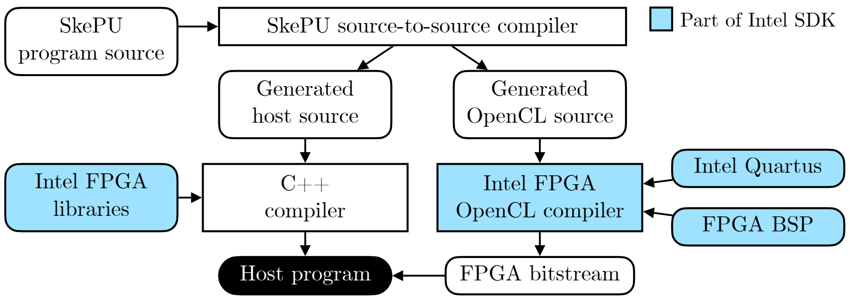


Fig. 2 FPGA backend compilation flow.

5 SkePU FPGA Backend

The first part of the implementation was to integrate the Intel FPGA OpenCL SDK into SkePUs backend code generation. SkePU provides OpenCL files to the SDK and accepts generated bitstream files in return. The user is responsible for ensuring that an installation of the SDK and the relevant FPGA board packages are available. The runtime was also extended to recognize Intel FPGAs and the Intel FPGA emulator as valid OpenCL devices, see Figure 2. While this initial implementation worked, it used the OpenCL backend designed for GPUs, which resulted in subpar performance on FPGAs. Therefore, a new FPGA-specific backend was created, similar to the OpenCL backend for GPUs with specialized kernel code generation for the respective skeletons. Functionality such as device detection and memory management is shared with the OpenCL GPU backend.

An initial design decision was whether to generate *single work-item* (SWI) or *NDRange* kernels. Using NDRange kernels would have meant that the existing OpenCL code generation could have been reused. However, those kernels use branches that depend on the global id of the work-items, meaning that automatic vectorization cannot be applied. This is a key optimization for NDRange kernels, so new code generation would have had to be done either way. Furthermore, Intel's general recommendation is to use SWI kernels, and they are faster for many types of problems [23,22]. It was therefore decided to implement code generation of SWI kernels. The main goal for the skeleton implementations was to reach an II of 1 for the main loop and not reducing the maximum frequency, while still pipelining the main loop.

The optimization techniques used are taken from Intel's programming documentation[4,5] and papers that applied the techniques [12,22,23]. The compiler referenced in the following sections is the Intel FPGA SDK OpenCL Offline Compiler. The Intel documentation recommends some general optimizations that can be applied to all kernels, mostly focusing on helping the compiler:

[4] https://intel.com/content/www/us/en/docs/programmable/683521/21-4/introduction-to-pro-edition-best-practices.html
[5] https://intel.com/content/www/us/en/docs/programmable/683846/21-4

– Using the `restrict` keyword on pointer arguments which never alias to other pointers. This can prevent the compiler from assuming memory dependencies between read and write operations.
– Using the `const` keyword on any read-only buffers. This allows the compiler to perform more optimizations on load operations.
– Applying the `uses_global_work_offset(0)` attribute. Applying this attribute allows the compiler to not generate hardware for supporting kernel invocations with a non-zero `global_work_offset` (which is never not zero in the FPGA backend), reducing area usage.

SkePU's memory allocation code was modified to allocate 64-byte aligned memory for all host buffers. The dynamic memory-transfer alignment logic used in OpenACC-To-FPGA was also implemented to handle cases when the buffer is not aligned on the FPGA. This guarantees that both the host and device buffer will be aligned when transferring memory between them, allowing all data transfers larger than 64 bytes to use direct memory access.

We now present the implementations of the Map, Reduce, MapReduce, Scan and MapOverlap skeletons in the FPGA backend.

Map The Map kernel was replaced by a for loop with a controllable unroll factor. As the loop contains no loop-carried dependencies, it did not require the use of a shift register or any other techniques to attain an II of 1.

Reduce Firstly, the generated kernel was changed to a single work-item kernel instead of a NDRange kernel. This meant the kernel could be reduced to a single for loop, which the compiler can pipeline. However, such reduce loops often result in a loop-carried data dependency on the reduce variable. If the user-function is complicated or uses instructions that are expensive to execute on FPGAs, this cannot be done in a single clock cycle. This will increase the II of the loop to the number of clock cycles the user function takes.

The reduce kernel tries to relax such data dependencies by increasing the number of variables that store the intermediate result of the reduction using a shift register. In each iteration, the head of the shift register is read, and the partial result is written to the tail of the register. If the size of the shift register is equal to or greater than the number of cycles that the user-function takes to execute, the data dependency can be eliminated.

Finally, parallelism is increased by adding partial loop unrolling. This could be applied to the main loop, but doing so acts as a multiplier to the latency of the loop [15], increasing the II. There are two ways to solve this: increasing the size of the shift register by the unroll count or performing manual loop unrolling as done by Zohouri [22]. Increasing the size of the shift register also increases the total area consumption, so for large unroll factors this can become impractical. Therefore manual loop unrolling was used in the reduce kernel. An example of a generated reduce kernel can be seen in Listing 5, where lines 17-22 show the manual loop unroll.

```
__kernel void reduce(__global float const* restrict input,
                     __global float* restrict output,
                     unsigned long size)
float shift_reg[LATENCY + 1];
#pragma unroll
for (int i = 0; i < LATENCY; i++) {
    shift_reg[i] = input[i];
}
int exit = (size % UNROLL == 0) ?
    (size / UNROLL) :
    (size / UNROLL) + 1;
for (int i = 0; i < exit; i++) {
    float partial_result = (
        LATENCY <= i * UNROLL && i * UNROLL < size
    ) ? input[i * UNROLL] : (float) {0};
    #pragma unroll
    for (int j = 1; j < UNROLL; j++) {
        int index = i * UNROLL + j;
        partial_result = (index < size) ?
            user_func(partial_result, input[index]) :
            partial_result;
    }
    shift_reg[LATENCY] = user_func(shift_reg[0], partial_result);
    #pragma unroll
    for (int j = 0; j < LATENCY; j++) {
        shift_reg[j] = shift_reg[j+1];
    }
}
float result = shift_reg[0];
#pragma unroll
for (int i = 1; i < LATENCY; i++)
    result = user_func(shift_reg[i], result);
output[0] = result;
```

Listing 5: Example of a generated reduce kernel.

MapReduce The MapReduce kernel used the same approach as the Reduce kernel, with an added call to the Map user function in the manual loop unroll and in the ramp-up phase. In the normal OpenCL backend, MapReduce is a two-phase kernel where the second phase performs a final reduction. This is not needed in the FPGA version, as the full reduction is performed in a single kernel execution, making the call to the reduce-only kernel unnecessary.

Scan The Scan kernel uses a shift register and loop unrolling. It begins with a prelude to populate the shift register, and after that it reads a single element from the input each iteration and applies it to the user function together with an element from the shift register, with the result being stored in the end of the shift register. The shift register is used to avoid the memory dependency that would otherwise be created when immediately accessing the previously read element in the next loop iteration. To also avoid a data dependency if the user function latency is larger than one, the shift register is accessed using an offset `OFFSET` that should be equal to or larger than the latency. This means that there will be `OFFSET` loop iterations between an element being read from main memory and that element being accessed from the shift register. This relaxes both the memory dependency and potential data dependency, assuming the offset is large enough, and allows the outer loop to attain an II of 1.

The results stored in the shift register are not complete when initially written, since the offset also needs to be taken into account. To compensate for this, a scan is performed on the first `OFFSET` elements in the shift register before writing the result to the output. This starts after the shift register has shifted the first input enough times, which depends on the size of the shift register, offset and if the scan is inclusive, according to $delay = size - offset + inclusive$. The outer loop will need extra iterations to write all elements to the output, but assuming the input size is large, it will not affect the performance.

As found by Lambert et al. [15], to allow the outer loop to be unrolled while not increasing the II, the shift register size should be $newsize = size * unrollfactor$. This does lead to a high area usage since the user function will be unrolled both by the outer loop and the final scan loop. As a result, the size of the unroll factor is limited compared to the other skeletons when using floating-point types, since these multiple unrolls of the user function quickly consume all hard-blocks used for floating-point arithmetic.

MapOverlap MapOverlap was the most complicated skeleton implemented on the FPGA backend. Therefore only the one-dimensional version was implemented for now, supporting the `Vector` variant and both the row- and column-wise `Matrix` variants. All edge handling modes are supported.

The implementation uses a shift register for all three variants, while loop unrolling is only applied to the main loop in the `Vector` variant due to too high resource usage in the other variants. The edge handling modes are implemented in a single kernel for each variant, which is one of the culprits for the larger hardware usage, as each unroll needs to include the logic needed to handle the different modes.

6 Experimental Results and Discussion

The FPGA evaluations were performed on Intel Devcloud[6] using a Programmable Acceleration Card (PAC) with an Arria 10 GX FPGA, with 1150000 logical elements and 1518 DSP blocks, connected via PCIe. Version 19.4.0 of the Intel FPGA OpenCL SDK was used to compile the FPGA kernels. The GPU benchmarks were run on a NVIDIA Tesla V100 SXM2 32GB GPU connected via PCIe. Both devices have a similar release date and price point and were therefore deemed to be a fair comparison.

6.1 Single skeleton performance

The FPGA backend was first evaluated for single skeleton calls. A program was created for each skeleton type supported by the FPGA backend, with a simple, typical user function for that skeleton type. Each program was compiled and run with 3 different unroll factors: 1, 8, and 16, and all used the `float`

[6] `https://intel.com/content/www/us/en/developer/tools/devcloud/overview.html`

Table 1 Single-skeleton test programs with user functions used in the evaluation.

Test Program	Skeleton	User Function
Adding squares	Map	$f(a,b) = a^2 + b^2$
Global sum	Reduce	$f(a,b) = a + b$
Dot product	MapReduce	$f_M(a,b) = a * b, f_R(a,b) = a + b$
Prefix sum	Scan	$f(a,b) = a + b$
Overlap average (1D stencil)	MapOverlap	$f(region) = average(region)$

data type. The execution time of each variation was recorded with different input sizes from 10^6 to 10^7 elements, in increments of 250000, and with 10 runs for each input size. Memory transfer time in each direction was included in the execution time. Each skeleton was invoked once before the measuring invocations to remove the time to program the FPGA. Table 1 lists the skeletons and user functions that were evaluated. The skeletons were also evaluated with the OpenCL backend on both an FPGA and on a GPU. All benchmarks used a one-dimensional `Vector` as input container. The FPGA kernels were compiled using the `-fast-compile` flag, which significantly speeds up the compilation speed by reducing the compiler's optimization efforts.

Figure 3 shows the performance of the single skeleton calls. The *No Unroll* line was the FPGA backend run with the unroll factor set to 1 for each skeleton, effectively disabling the unrolling, while *Unroll 8* and *Unroll 16* set the unroll factor to 8 and 16. The *Baseline* and *GPU* lines show the execution time of using kernels generated by the OpenCL backend where *Baseline* is the execution time on the FPGA and *GPU* is the execution time on the GPU. Scan is missing *Unroll 16* because the design generated by this unroll factor did not fit on the FPGA. It ran out of DSP blocks, which are used for float operations. For MapOverlap, a baseline is missing due to a bug in the SDK's library, preventing the benchmark from running. Finally, we omitted *No Unroll* for MapOverlap due to taking too long time: 420ms for the smallest size and 4.2s for the largest. These results show that the kernels generated by the FPGA backend are faster than the ones generated by the OpenCL backend when run on the FPGA, even with no unrolling. With unrolling, they are close to the GPU execution time and faster in the Reduce kernel case.

Figure 3 shows that the new kernel generation with FPGA-specific optimizations in the FPGA backend gives a noticeable performance gain over the OpenCL backend. The largest performance gain is observed for Scan, where the baseline implementation performed poorly. One cause of the poor performance could be the many barriers used in the baseline kernels, which force the pipeline to be flushed before moving to the next section. Map is the skeleton with the least performance gain. This is likely because the FPGA Map kernel does not use any FPGA-specific techniques besides loop unrolling, meaning both implementations are similar and are largely memory-bound. The Map kernel would likely gain from being implemented as an NDRange kernel instead, as the hardware scheduler can help alleviate memory bottlenecks.

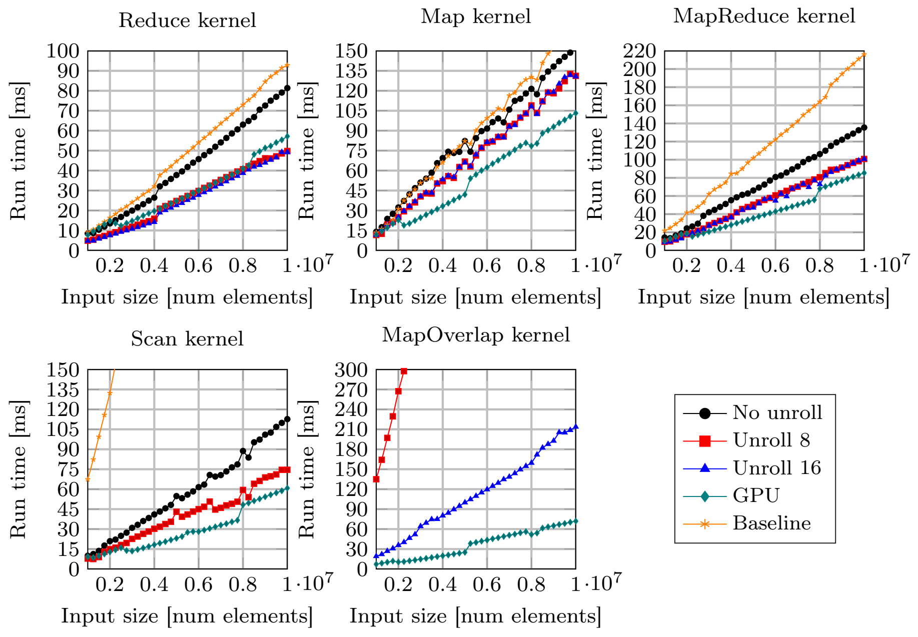


Fig. 3 The median execution times for the single skeleton evaluation (lower is better).

The experiments also show that increasing the unroll factor does not improve performance past a certain point, as almost all benchmarks have similar execution times. The one outlier is the MapOverlap kernel, where the execution time for the largest input size using unroll factor 16 is 6 times faster than that with an unroll factor of 8.

Finally, we see that the fastest FPGA execution time for all but MapOverlap is close to the GPU execution time. These results are promising, given the maturity of the skeleton implementations used by the OpenCL backend.

6.2 FPGA-specific optimizations performance

The effect of the two major FPGA-specific optimizations used in the skeleton implementations, loop unrolling, and shift registers, were evaluated by executing the same skeleton kernel compiled with different parameters. The Reduce skeleton was used with the same user function as in the single skeleton evaluation. Four combinations of parameters (No unrolling / Unroll by 8, and No shift register / Shift register of size 8) were used to compile four variants of the skeleton kernel. The variant with both unrolling and shift register was evaluated once with the memory alignment and dynamic DMA transfers turned off to evaluate its impact. The total execution time was evaluated for each variant with an input size of 10^7. Each variant was run 10 times, and the median execution time was recorded. Figure 4 shows the impact of FPGA-specific op-

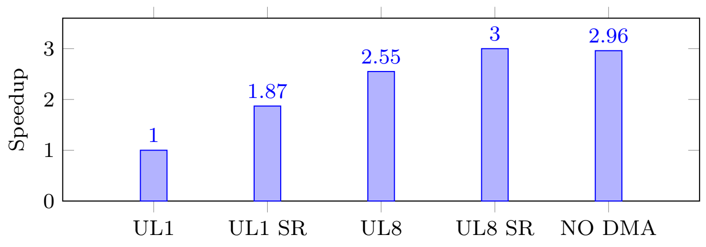


Fig. 4 Result of FPGA-specific techniques (ULN means an unroll factor of N).

timizations. ULN means the kernel was compiled with an unroll factor of N and *SR* means it used a shift register. It shows that both loop unrolling and using a shift register gives performance benefits. The shift-register had a larger impact when not using unrolling, giving an 87% speedup, but when combined with unrolling the performance boost is only 17% compared to just using loop unrolling. Lastly, turning off the dynamic memory-transfer alignment logic had a small impact on the execution time.

The evaluation of the FPGA-specific optimizations makes it clear that they are worth implementing as they give a large speedup without the user needing to do anything. The compiler reports for the four variants show that II is 3, and the maximum frequency is 98 MHz without the shift register compared to 1 and 240 MHz with the shift register. This is the case both with and without loop unrolling. According to Equations 1 and 2, decreasing the II from 3 to 1 and more than doubling the maximum frequency should give close to six times better performance. Nevertheless, we see that this does not hold in practice, as the evaluated speedup is close to half the theoretical speedup. Similarly, an unrolling factor of 8 gives a theoretical speedup of almost eight, but the evaluated speedup is far from that. Even so, the equations are still useful in guiding what parameters to change to get performance benefits.

Furthermore, the finding that turning off the memory alignment and transfer logic did not impact performance was surprising, given that Lee et al. [16] reported over 100 times faster transfer times when using this method. Profiling the kernel to get the exact memory transfer times shows that the two variants only differ by a few nanoseconds. This can mean 3 different things: DMA is not used in either variant, DMA is used in both variants, or DMA does not affect the memory transfer time.

6.3 Multiple skeletons performance

To evaluate the performance of calling multiple different skeleton instances in a single SkePU program two programs were created: One chaining two `Map` calls, and one calling a single `Map` instance explicitly fusing the two user functions, as shown in Listing 6. The total execution time of the program was recorded for an input size of 10^6 elements. Moreover, the time to reprogram the FPGA for each

```
float add(float a, float b) { return a + b; }
float multiply(float a, float s) { return a * s; }
float combined(float a, float b, float s) { return (a + b) * s; }
```

Listing 6: The user functions used for evaluating multiple skeleton calls.

Table 2 Running times of the two variants evaluated.

Variant	**Total Time** (s)	**FPGA Programming Time** (s)	**Execution Time** (s)
Chained	7.253	7.241	0.012
Merged	0.018	0	0.018

skeleton call was measured using Intercept Layer for OpenCL Applications[7], a tool for profiling OpenCL applications. Both benchmarks were run 10 times to reduce the effect of timing variations, with the median execution time and time spent reprogramming the FPGA being recorded. The results are presented in Table 2. *Total time* is the median total amount of time spent to run the benchmark, *Programming time* is the time spent on programming the FPGA before each kernel invocation, and *Execution time* is the time spent to run the computations, including memory allocation and transfers.

The difference in total time between the variants is stark: *Chained* spends 99% of the total time just programming the FPGA. The reason why the *Merged* variant does not need to reprogram the FPGA during the benchmark is that this is done when instantiating the merged skeleton, which is not part of the benchmark. This is of course also the case for other variants, but since two skeletons are instantiated, they “overwrite” each other, forcing the FPGA to be reprogrammed for the first skeleton call in the benchmark every time.

The results of running multiple skeletons show the large overhead reprogramming the FPGA between each skeleton invocation adds. Hence, any SkePU program that calls multiple skeleton instances will suffer large performance penalties when running on the FPGA backend, which is needed for many computations. Therefore, as many skeletons as possible should be fused when targeting the FPGA backend. In the current version of SkePU, there is no automatic fusion of skeletons, even for cases such as a chained Map and Reduce [8], so manual fusion by the user must be applied.

Still, even if all skeleton instances are fused, the FPGA must be programmed at least once. Until this operation is optimized, all SkePU programs that target the FPGA backend will take significantly longer every time it needs to program the FPGA. The FPGA will cache the kernel between program executions, so if the same program with a single kernel is run multiple times, only the first execution will require the FPGA to be programmed.

[7] https://github.com/intel/opencl-intercept-layer

```
T mmmult(const skepu::MatRow<int> ar, const skepu::MatCol<int> bc)
{
    T res = 0;
    for (size_t k = 0; k < ar.cols; ++k)
        res += ar(k) * bc(k);
    return res;
}
```

Listing 7: Matrix multiplication user function in SkePU.

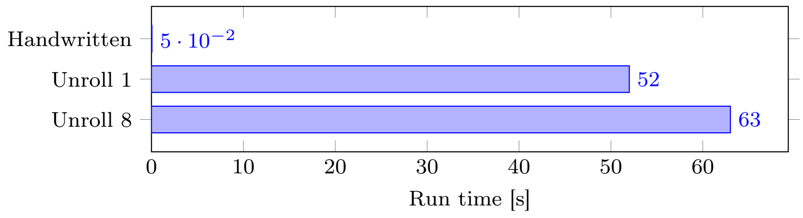


Fig. 5 Result of the complex user function evaluation.

6.4 Complex user function performance

To test more complex user functions, such as functions with loops, we used the matrix multiplication code from SkePUs set of example programs. It uses the `Map` skeleton with the user function shown in Listing 7. Two versions were evaluated: One with an unroll factor of 1 and one with an unroll factor of 8.

For a comparison with the performance achievable on an FPGA, a handwritten Matrix Multiplication OpenCL kernel from Boyi's [14] collection of OpenCL FPGA kernels[8] was also benchmarked. The version used was the NDRange kernel with 64 as an unroll factor, a SIMD factor of 8 and 2 kernel replications (`ul64_simd8_cu2`). All kernels were compiled with the same flags[9] and executed with 2048×2048 matrices. Like the previous evaluations, all kernels were run 10 times, and the median execution time was recorded.

The results in Figure 5 clearly show that both SkePU variants run on the FPGA (*Unroll 1* and *Unroll 8*) are much slower than the handwritten variant. Furthermore, the SkePU variant with a higher unroll factor is slower than with the lower one, in contrast to our results in Section 6.1. The reason appears to be a failure to pipeline the *Unroll 8* variant's main loop because of the inner loop in the user function. Without the unroll, this failure does not occur.

While the results in Figure 5 are not surprising, as kernels specifically created for a task should always be faster than a SkePU implementation, the difference in performance is still an issue if the FPGA backend is to be used for any computation with complex user functions. The main issue is loops in the user function, as these often require a lot of modifications not to cause

[8] `https://github.com/jjiantong/Boyi/tree/fpga20`

[9] The handwritten kernel used an extra compilation flag (`-no-interleaving=default`) to store the two matrix buffers in different areas of the FPGA's memory. The FPGA backend does not use this optimization technique, so the flag was not used for those kernels.

the outer loop's II to increase. According to Intel's documentation [13] such nested loops should preferably be fully unrolled or collapsed, but that requires that the user function is modified by SkePU, which is currently not possible.

7 Conclusion and Future Work

We presented a new backend targeting reconfigurable architectures, specifically FPGAs, for the skeleton programming framework SkePU. The new backend implements many features also supported in other backends in SkePU and implements FPGA-specific optimizations for better performance. We evaluated the new FPGA backend and compared it to one of the GPU backends. Our results show that performance is close for simpler tasks and highlight the importance of using FPGA-specific optimizations. However, the backend falls behind for more complicated tasks.

While the speedup by the FPGA-specific optimizations varied depending on the task, all skeletons saw performance benefits. Changing the kernels to be single work items with shift registers gave a speedup compared to running the OpenCL code by the existing OpenCL backend, from 1.06× for Map to 6.10× for Scan. Adding more optimizations improved speedup in all cases, ranging from 1.25× for Map to 9.23× for Scan.

Further details about the implementation and results can be found in the first author's recent master thesis [3]. A fork of SkePU with the FPGA backend implementation is available at `https://github.com/Birath/skepu/`.

A main issue for future work is the performance when invoking multiple skeleton instances. The backend should not need to reprogram the FPGA every time another skeleton is invoked. A potential solution to this is to generate a single large kernel that only needs to be programmed once. Though this could lead to issues with resource usage on the FPGA if large loop unrolling factors are used or the user functions are non-trivial.

Acknowledgments

Ernstsson and Kessler acknowledge partial funding by ELLIIT, project GPAI. We thank NSC for providing access to the Sigma cluster (LiU-gpu-2021-1).

References

1. V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys (CSUR)*, 27(3):367–432, 1995.
2. D. F. Bacon, R. Rabbah, and S. Shukla. FPGA programming for the masses. *Communications of the ACM*, 56(4):56–63, Apr 2013.
3. B. Birath. Skeleton computing for reconfigurable architectures. Master thesis LIU-IDA/LITH-EX-A–23/005–SE, Department of Computer and Information Science, Linköping University, Sweden, Mar. 2023. To appear.
4. A. Boutros and V. Betz. FPGA architecture: Principles and progression. *IEEE Circuits and Systems Magazine*, 21(2):4–29, 2021.

5. M. I. Cole. *Algorithmic skeletons: structured management of parallel computation.* Pitman London, 1989.
6. T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From OpenCL to high-performance hardware on FPGAs. In *22nd Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 531–534. IEEE, Aug 2012.
7. U. Dastgeer and C. Kessler. Smart containers and skeleton programming for GPU-based systems. *International Journal of Parallel Programming*, 44(3):506–530, 2016.
8. A. Ernstsson. *Pattern-based Programming Abstractions for Heterogeneous Parallel Computing.* PhD thesis, Linköping University, 2022.
9. A. Ernstsson, J. Ahlqvist, S. Zouzoula, and C. Kessler. SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters. *International Journal of Parallel Programming*, 49(6):846–866, 2021.
10. A. Ernstsson, L. Li, and C. Kessler. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, 46(1):62–80, 2018.
11. H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.
12. K. Hill, S. Craciun, A. George, and H. Lam. Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA. In *26th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, pages 189–193, Jul 2015.
13. Intel. *Intel FPGA SDK for OpenCL Pro Edition: Programming Guide.*
14. J. Jiang, Z. Wang, X. Liu, J. Gómez-Luna, N. Guan, Q. Deng, W. Zhang, and O. Mutlu. Boyi: A systematic framework for automatically deciding the right execution model of OpenCL applications on FPGAs. In *Proc. Int. Symposium on Field-Programmable Gate Arrays*, FPGA'20, page 299–309. ACM, Feb 2020.
15. J. Lambert, S. Lee, J. Kim, J. S. Vetter, and A. D. Malony. Directive-based, high-level programming and optimizations for high-performance computing with FPGAs. In *Proc. Int. Conf. on Supercomputing*, ICS'18, page 160–171. ACM, Jun 2018.
16. S. Lee, J. Kim, and J. S. Vetter. OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing. In *Int. Parallel and Distrib. Processing Symposium (IPDPS)*, pages 544–554, May 2016.
17. S. Lee and J. S. Vetter. OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proc. int. symp. on high-performance parallel and distributed computing*, HPDC '14, pages 115–120. ACM, Jun 2014.
18. R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016.
19. Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. FPMR: MapReduce framework on FPGA. In *Proc. int. symposium on Field programmable gate arrays*, FPGA '10, pages 93–102. ACM, Feb 2010.
20. Z. Wang, B. He, W. Zhang, and S. Jiang. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, page 114–125, Mar 2016.
21. Z. Wang, S. Zhang, B. He, and W. Zhang. Melia: A MapReduce framework on OpenCL-based FPGAs. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3547–3560, Dec 2016.
22. H. R. Zohouri. *High Performance Computing with FPGAs and OpenCL.* PhD thesis, Tokyo Institute of Technology, 2018.
23. H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In *Proc. SC16 Conference*, pages 409–420. IEEE, Nov 2016.

Symposium on High-Level Parallel Programming & Applications
HLPP 2023

A Single-source Algorithmic Skeleton Programming Framework for Deep Learning on Heterogeneous Parallel Systems

**August Ernstsson · Sehrish Qummar ·
Christoph Kessler · Oleg Sysoev**

Abstract Machine learning is rapidly becoming a common class of application workload for high-performance computer systems. In particular, deep neural networks (DNNs) is a popular approach for machine learning that comes with large computational demands and relies on systems with high computational throughput. Such neural networks are therefore usually implemented in domain-specific programming models and frameworks, where individual components have been hand-tuned for one or several target hardware architectures. In this work, we investigate to which extent the algorithmic skeleton model as implemented in the SkePU framework can accommodate deep neural networks, in particular, convolutional neural networks (CNNs). We extend SkePU with new functionality aimed at improving the interface for computational- and data-access patterns seen in CNNs, and demonstrate the design and implementation with concrete practical scenarios.

Keywords algorithmic skeletons · deep learning · convolutional neural networks · heterogeneous computing · high-level parallel programming

1 Introduction

Machine learning has developed at a remarkable pace and is attracting a large number of researchers and practitioners. It has become one of the most popular research directions and has many applications such as text mining, spam detection, image classification, and video recommendation. Deep learning, i.e., machine learning using deep neural networks (DNN), has achieved good performance in a variety of application domains, such as audio and speech recognition, natural language processing, and visual data processing. The high computational requirements of DNN call for the use of GPUs and other hardware

Dept. of Computer and Information Science
Linköping University, Linköping, Sweden
E-mail: <firstname>.<lastname>@liu.se

accelerators. Hence, the resulting code needs to be able to run on heterogeneous systems.

Except for specific DNN benchmarks, DNN computations are usually embedded in an application context that also requires substantial pre- and/or postprocessing of input and output data. In addition, DNN computations are increasingly being integrated as subcomputations into more complex applications in various domains, thus leading to mixed (DNN+X) applications.

While there exist well-known high-level domain-specific languages and frameworks for deep learning, such as TensorFlow, Keras and PyTorch, these frameworks lack expressivity and code generation support for other (non-DNN) domains. However, interoperability across multiple DSL frameworks is difficult because the generated procedural (and possibly, parallel) code from a declarative DNN specification is not exposed to the end programmer. Hence, program analysis and optimizations across framework boundaries are difficult.

In this work, we propose and prototype a procedural, high-level, multi-domain programming framework based on algorithmic skeletons [5,4] that seamlessly integrates both DNN-specific and traditional (non-DNN) computation patterns, combining the conciseness and high abstraction level of declarative DNN specifications with the tighter execution flow and resource control in procedural code, providing seamless interoperability with procedural skeleton-based computations in general-purpose and other domains.

This paper makes the following contributions:

– We study the fundamental computational patterns in convolutional neural networks and analyze the requirements of conveniently expressing mixed applications containing both CNN, image processing and linear algebra subcomputations, using patterns in a C++ based high-level programming framework for heterogeneous parallel systems.
– Based on this analysis, we extend the SkePU high-level programming framework with several new language features that are particularly suitable for programming deep learning applications with convolutional neural networks, including strided access for all map-based SkePU skeletons (in particular, the stencil skeleton `MapOverlap`), a new skeleton for the `MapPool` pattern, and a new generic proxy data-container `Pool<T>` for expressing CNN-typical data access patterns, which was not possible before in SkePU.
– We also introduce a higher-level interface to DNN patterns in SkePU, similar to that of domain-specific languages. For this purpose, we introduce method chaining to allow for more concise code for operation sequences in SkePU, thereby enabling a programming style that today's data scientists are more familiar with.

The remainder of this paper is organized as follows: Section 2 introduces background about SkePU and DNN computations. Section 3 presents the design and implementation of the SkePU-DNN extensions. Early results are reported in Section 4. Section 5 discusses related work. Section 6 concludes and proposes future work.

2 Background

2.1 Algorithmic Skeletons

Algorithmic skeletons is a programming model introduced by Cole [5] that builds upon concepts from functional programming and forms a high-level interface to parallel programming. The model is centered around parallel patterns such as *map*, *reduce*, and *prefix-sums* which can be efficiently parallelized as their internal dependency structure is well-studied. The idea has subsequently been implemented in many academic high-level parallel programming frameworks [2,12,11,25,10,29,14], and similar abstractions also show up in programming languages and standard libraries and in industry tools and frameworks, such as Intel TBB, Nvidia Thrust [3], Google MapReduce [9], Apache Spark [33], SYCL, and Intel OneAPI.

Some skeleton programming APIs are focused on data parallelism, others on task- or stream-parallel patterns. They also differ in their intended target platforms. In particular, skeleton programming frameworks often target heterogeneous platforms such as systems equipped with several CPU cores and one or more GPU accelerators.

2.2 SkePU

SkePU [14] is an open-source[1] framework for C++-based, single-source high-level programming of heterogeneous parallel systems using algorithmic skeletons. SkePU was first introduced in 2010 [11] and went through several major redesign efforts, the most recent version being SkePU 3 [14].

Since version 2, the SkePU programming interface is an embedded language extension of C++11$^+$. Each SkePU program is thus a valid C++11$^+$ program which, if compiled with an ordinary C++ compiler, yields sequential code with the same semantics. If instead compiled with the SkePU *precompiler*, the SkePU-specific program constructs (skeleton objects, user functions etc.) are translated in a specific way and platform-specific source code variants for use with the parallel SkePU back-ends (e.g., OpenMP for multicore CPUs, OpenCL and CUDA for GPUs) is generated, which is further compiled with a platform-specific compiler such as `nvcc` for CUDA code.

SkePU focuses on skeletons that implement data-parallel computation patterns which operate on in-memory multi-dimensional arrays of one to four dimensions, represented as generic *data-container* objects with a STL-like interface: `Vector`, `Matrix`, `Tensor3D` and `Tensor4D`.

The current SkePU version provides a set of different skeletons: `Map` (elementwise application of an operator to each element in one or multiple input containers in 1D to 4D), `MapOverlap` (generic stencil computations in 1D to 4D), `MapPairs` (generic outer product of vectors), `Reduce` (reductions in 1D to 4D), `MapReduce` (fused combination of `Map` and `Reduce`), `MapPairsReduce`,

[1] `https://skepu.github.io`

and `Scan` (generic prefix sums computations). The skeletons can be configured in various ways, for example `MapOverlap` in the stencil overlap size (i.e., filter width) in each dimension and in the boundary handling.

A skeleton only models the generic internal data dependence and access structure of the computation pattern that it implements. The missing problem-specific operators are specified by the programmer as so-called *user functions*. These are side-effect-free ordinary C/C++ functions, with which a skeleton can be customized and instantiated, thereby creating a *skeleton instance* object which then can be called like any hand-written C++ function.

SkePU skeletons are *polymorphic* in both number and shape of their operands, depending basically only on the signature of the user function that a skeleton is instantiated with. Hence, an instance of a skeleton (e.g., `Map`) can accept any number and dimensionality of element-wise accessed container operands of any dimensionality, any number of multi-element accessible container operands of any dimensionality, plus any number of uniform arguments (e.g., constant values), in this order. With element-wise access to input data-containers being the default access pattern, multi-element data access patterns are specified by using *proxy container* objects in user function parameters, such as `Mat` for random access to any element or `MatRow` for access within a matrix row, or `Region`X`D` for access within a limited-distance X-dimensional neighborhood in a X-dimensional data-container, where the latter is used for stencil computations in $X = 1, ..., 4$ dimensions. For more details, we refer to [14].

With the different backends provided for each SkePU skeleton, a skeleton instance can execute either in sequential or parallel on CPU or on one or several GPUs using OpenCL or CUDA; also a hybrid CPU-GPU backend exists. The backend selection for a skeleton instance's execution can be either explicitly specified by the programmer (also at runtime) or done automatically by SkePU. The latter uses either the default setting (namely, the most parallel backend available for the target system) or the SkePU autotuner which builds an internal performance model for backend selection from previous training executions of the skeleton instance. SkePU programs can also transparently execute on multiple cluster nodes without any changes in the source code [14].

SkePU data containers transparently manage memory and communication of elements at runtime, depending on where computations on them are executed. They apply a number of optimizations, such as lazy transfers [7] and lazy evaluation with global tiling [15].

Recent work on SkePU has added integrated, portable pseudo-random number generation [16] that internally leverages available parallelism but behaves *deterministically* across the different backends, regardless of the number of processors or the type or number of accelerators used.

Listing 1 shows a basic example SkePU program: a two-dimensional cellular automation. The `MapOverlap` skeleton is used to perform the simulation, along with matrix smart data-containers to model the world. A set of evolution rules is also stored in a matrix, which is passed to the user function in its entirety and used as a look-up table.

Listing 1: SkePU example program: a 2D cellular automaton.

```cpp
char automaton_update(skepu::Region2D<char> r, skepu::Mat<char> rules)
{
  float count = 0;
  for (int i = -r.oi; i <= r.oi; ++i)
    for (int j = -r.oj; j <= r.oj; ++j)
      if (!(i == 0 && j == 0)) count += r(i, j) ? 1 : 0;
  return rules(r(0, 0), count) ? 1 : 0;
}
auto update = skepu::MapOverlap(automaton_update);
update.setOverlap(1, 1); // set overlap radii
update.setEdgeMode(skepu::Edge::Pad);
update.setPad(0); // set padding value
skepu::Matrix<char> domainA(size, size, 0), domainB(size, size);
skepu::Matrix<char> updateRules(2, 8);
// Each empty cell with three neighbors becomes populated.
// Each populated cell with two or three neighbors survives.
updateRules(0, 3) = true;
updateRules(1, 2) = true;
updateRules(1, 3) = true;

domainA.randomize(0, 2);
for (size_t i = 0; i < iters; ++i)
{
  update(domainB, domainA, updateRules); // read from A and write to B
  update(domainA, domainB, updateRules); // read from B and write to A
}
```

2.3 Deep Learning and Convolutional Neural Networks

Machine learning incorporates various kinds of learning, while supervised and unsupervised learning are probably the most widely known types of learning. Supervised learning techniques learn from data sets that contain features and labels or targets (y). Among various supervised learning methods, deep learning approaches have gained a lot of interest in recent decades. Various kinds of neural architectures have been proposed, including Recurrent Neural Networks (RNN), Convolutional Neural Networks (CNN), and Long Short Term Memory neworks (LSTM) [22]. In unsupervised learning, the data set is unlabeled, and the model learns the features to recognize unidentified structures or relationships, for example clusters.

A starting point of our work is CNN architectures as shown in Figure 1. This architecture is commonly applied in deep learning for computer vision applications. The advantage of CNN architectures is that they automatically extract the features that are important for the prediction task and thus can replace the resource consuming manual feature engineering. The first CNN architecture proposed by LeCun [23] in 1998 motivated more advanced architectures such as AlexNet [21] which marked a turning point in the field of computer vision.

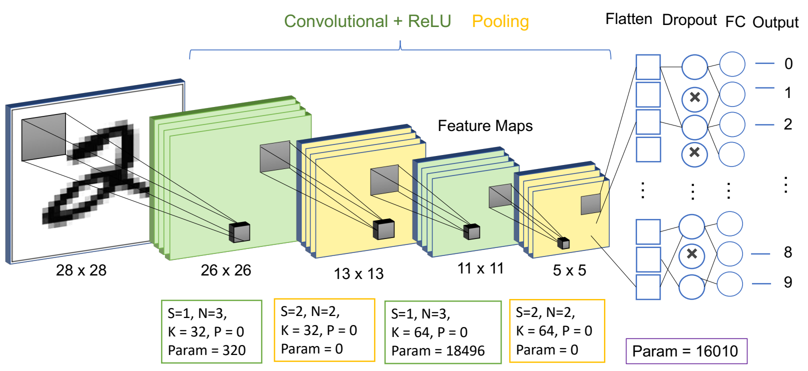

Convolutional + ReLU
Pooling
Flatten
Dropout
FC
Output
Feature Maps
28 x 28
26 x 26
13 x 13
11 x 11
5 x 5
0
1
2
8
9
S=1, N=3, K = 32, P = 0 Param = 320
S=2, N=2, K = 32, P = 0 Param = 0
S=1, N=3, K = 64, P = 0 Param = 18496
S=2, N=2, K = 64, P = 0 Param = 0
Param = 16010

Fig. 1: Example of a basic CNN architecture (from the Keras examples `https://keras.io/examples/vision/mnist_convnet/`). It takes a 28×28 image as input, which is processed by convolutional layers (C1, C2) with activation function ReLU, pooling layers (P1, P2), and a flattening, a dropout, and a fully connected layer, with 10 outputs. The hyperparameters of the convolution and pooling layers are S (stride), N (filter size), K (number of kernels), and P (padding). *Param* gives the total number of weight and bias parameter elements for each layer. The fully connected layer at the end contains only 10 neurons with softmax activation functions, which also act as an output layer.

2.3.1 Forward propagation: Predictions

To learn the parameters of neural networks, optimization algorithms are employed. These algorithms typically require an estimate of the gradient of the cost function of the network with respect to the model parameters, at each optimization iteration. To estimate this gradient, back-propagation [30] is typically used in deep learning. Back-propagation is illustrated in Figure 2: (1) forward pass and (2) backward pass, also known as backpropagation.

During the forward pass, the network parameters, such as the weights and biases are fixed and used to process the input. A forward pass is used to calculate a predicted output during training for each vector $x_i, i = 1, \ldots, m$ having a known output y_i. In Figure 2, $q^{(0)} = x$ is the input, which is processed by layers $(1, \ldots, \lambda)$. Each layer has parameters such as weights and bias where $W^{(1)}$ and $b^{(1)}$ belong to layer 1, $W^{(2)}$ and $b^{(2)}$ belong to layer 2, and so on, where the dot product is stored in $z^1, z^2, \ldots, z^\lambda$. The result of each layer is represented as $q^{(1)}, q^{(2)}, \ldots, q^{\lambda-1}$, so, $q^{(1)}$ is the output of layer 1, and input for layer 2 until last layer (λ).

2.3.2 Backpropagation Algorithm

Backpropagation is the process of computing the gradients of the cost function with respect to the network parameters (input, weights, and biases) $\theta = \{W^1, b^1, q^1, W^2, q^2, ...\}$. To compute the optimal values of parameters

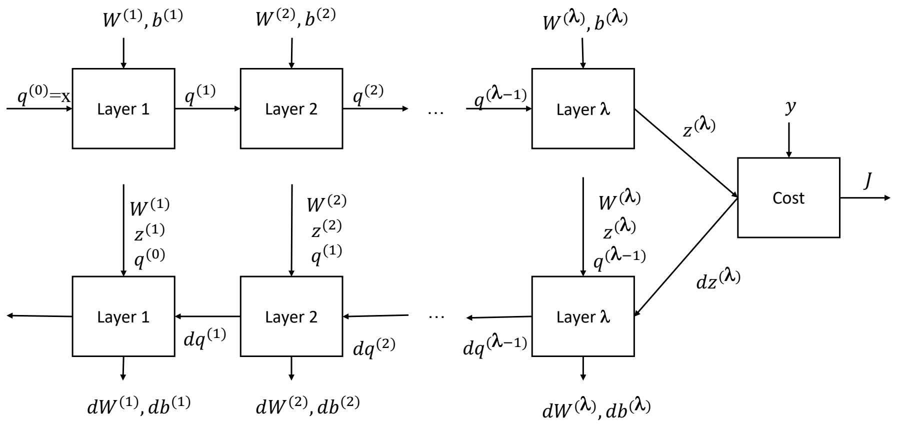


Fig. 2: A computational graph of the forward and backward propagation algorithm for training a multi-layer feed-forward network (adapted from [24]). The model input (upper left corner) is propagated forward from layer (1 to λ) to calculate the outputs q and z of each layer. After evaluating the cost function, the learning algorithm calculates the gradient of each layer w.r.t input dq, weights dW and bias db, and propagates backward in the network.

$\theta = \arg\min J(\theta)$ where $J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(x_i, y_i, \theta)$. Here $J(\theta)$ is a cost function and $\mathcal{L}(x_i, y_i, \theta)$ is a loss function. The backpropagation algorithm works by propagating the evaluated gradients back through the layers of the network, using the chain rule of calculus to calculate the gradients of the loss function with respect to the inputs (dq), weights (dW), and biases (db) of each layer. These gradients can be used to update the parameters through an optimization algorithm such as stochastic gradient descent (SGD). In Figure 2, a gradient is computed for each layer $(1, \ldots, \lambda)$ w.r.t input (dq), weights (dW), and bias (db). Where $dW^{(1)}$ is the weight gradient of layer 1, $dW^{(2)}$ of layer 2, and so on. The bias gradient of each layer is presented as $db^{(1)}, db^{(2)}, \ldots, db^{(\lambda)}$. And the input gradient of each layer is computed as $dq^{(1)}, dq^{(2)}, \ldots, dq^{(\lambda)}$.

2.4 CNN Layer Types

A basic convolutional neural network architecture is commonly described as a sequence of *layers*. A layer typically models either a computation with trainable weights/parameters (such as a *fully-connected* or *convolutional* layer) or a transformation of the structure of the layer input (such as *flatten* or *dropout* layers).

A simple CNN architecture has been implemented `https://keras.io/examples/vision/mnist_convnet/`) for the classification of handwritten digits. This architecture has three types of layers: (1) convolutional layer, (2) subsampling (pooling) layer, (3) the dropout layer and (4) fully connected

Listing 2: Example in Keras

```
model = keras.Sequential([
    keras.Input(shape=(28, 28, 1)),
    layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dropout(0.5),
    layers.Dense(10, activation="softmax")
])
```

layer, see Figure 1. The input (X) to the CNN model at each layer is organized into three dimensions, namely height (m), width (m), and depth (d), where height and width are equal in size, and depth is the number of channels in the input image (RGB or grayscale input). There are several *kernels* (*filters*) in the convolutional layer that convolve on the image. The dimensions for each kernel are also the same as the input height (n), width (n), and depth (q), where $n < m$ and $q = d$. Each kernel k has parameters such as *weights* (W^k) and *bias* (b^k) that make a connection with input while doing the convolutional operation (see Sect. 2.4.1), to generate feature maps. A feature map is an output of each layer. The kernel weights are randomly initialized at the beginning of the model training process and updated during the training process.

2.4.1 Convolutional Layer

The purpose of a convolutional layer is to learn high-level features of the input image by employing the spatial information in the neighborhood of each pixel. The *convolutional layer* (CL) is the first layer of the CNN model. It first computes

$$z = conv(X, W) + b, \tag{1}$$

where $conv(X, W)$ is a convolution operator between the input X and the weights W (the kernel), i.e.

$$conv(X, W)_{ij} = \sum_{\mu=1}^{n} \sum_{\nu=1}^{n} W_{\mu,\nu} X_{i+\mu,j+\nu}, \text{ for all } i, j = 1, ..., m - n, \tag{2}$$

and then applies a non-linear function (called *activation function*) to z values to generate the feature map. A feature map is an output of a convolutional layer, where each element of the feature map corresponds to a specific location in the input image. By stacking multiple feature maps together, a CNN model can learn to recognize more complex features. There are many possible activation functions that can be used (see Sect. 2.5).

There are several hyperparameters in CL, such as the number of kernels, size of the kernel filter, strides, padding, in addition to the input. The strides

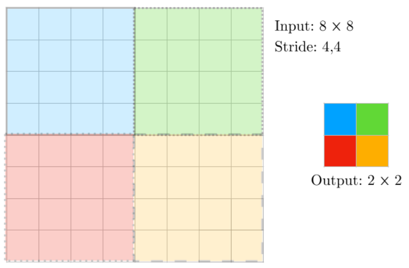


Fig. 3: Pooling. Output is smaller than input by a factor of the pool size.

and padding determine the *size* of the feature map; the size is $m - n - 1$. The *stride* is a selected step size in the horizontal and vertical positions of the image. The selected step size value is usually equal to one but can be set as a larger value. *Padding* inserts additional (normally zero-valued) pixels on the border of the image to enable computing convolutions even for the pixels that are located on the boundary. If padding is applied to the input, it ultimately increases the size of the feature maps.

Backpropagation of the convolutional layer employs Equations 1–2 and the chain rule to compute the gradients: the gradient w.r.t. the input (X), the gradient w.r.t. the kernel (W) (filter), and the gradient w.r.t. the bias (b). Eq. (1) is used to compute the partial derivatives of z.

The gradient of the loss function with respect to parameters W and inputs X can be computed as

$$\frac{\partial \mathcal{L}}{\partial W} = conv(X, \frac{\partial \mathcal{L}}{\partial z}) \tag{3}$$

Similarly, by employing Eq. (1), the gradient with respect to the bias and input terms becomes

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{i,j=1}^{m} \frac{\partial \mathcal{L}}{\partial z_{i,j}} \tag{4}$$

$$\frac{\partial \mathcal{L}}{\partial X_{\mu,\nu}} = conv(\ padded\left(\frac{\partial \mathcal{L}}{\partial z}\right), \bar{W}) \quad \forall \mu, \nu = 1, ..., m \tag{5}$$

where $\frac{\partial \mathcal{L}}{\partial z}$ is a gradient from the next layer that, after padding with 0 at the boundary (operator *padded*) convolves with $\bar{W}$ which is the inverted kernel (W rotated by 180 degrees), to get gradients of $\frac{\partial \mathcal{L}}{\partial X}$.

There is no weight sharing between any two neighboring neurons in a CL. Also, there are relatively few weights (sparse connectivity) between two adjacent layers. These two main advantages of CL make the CNN more effective in training, and memory efficient.

2.4.2 Aggregation Layers

The second layer type of the CNN architecture is the *pooling layer* (PL). The computations of the pooling layer are similar to the convolution layer using stride and padding. PL kernels compute a block-wise partial reduction to reduce the size of the feature map, see Figure 3. The primary purpose of this layer is to aggregate inputs into a more compact representation which may speed up the training process and reduce the noise and redundancies in the input. A sliding kernel is applied to an adjacent region of size $a \times a$ in the feature map, where a is the size of the pool, and a pooling function is applied to the elements of that region. Commonly used pooling functions are the maximum, minimum, average maximum, and global average maximum, but maximum and average pooling are probably the most popular in the literature.

We have used max-pooling in our implementation. During a forward pass, the pooling layer calculates the maximum element and its index in the region, and this information is used for the gradient calculation. The pooling layer has no parameters that would require learning.

During backpropagation of the pooling layer, the gradient is only computed for the recorded index of the maximum element. The maximum element is obtained after applying the pooling $a \times a$ is 1. So, in backpropagation, we only need to compute input (X) values that come from the next layer, thus calculate $\frac{\partial \mathcal{L}}{\partial X}$. Taking the partial derivative on $a \times a$ we only have 1 value because the other elements did not participate.

2.4.3 Dense (fully connected) Layers

The last layer of the CNN model is a *fully connected layer*, also known as the *classifier* of the model. A dense layer consists of a set of neurons each being connected to every neuron of the previous layer, parameterized by weights and biases. A dense layer computes a dot product between the input and weights, with the bias as an offset:

$$z = \sum_{i=1}^{m} X_i W_i + b \tag{6}$$

which is then processed by the activation function g: $g' = g(z)$.

During the backpropagation, the gradients are calculated w.r.t. the weights W, bias b, and input X:

$$\frac{\partial \mathcal{L}}{\partial W} = X^T * \frac{\partial g}{\partial z} \qquad \frac{\partial \mathcal{L}}{\partial b} = \sum_{i=1}^{m} \frac{\partial g_i}{\partial z_i} \qquad \frac{\partial \mathcal{L}}{\partial X} = \frac{\partial g}{\partial z} * W^T \tag{7}$$

2.4.4 Flattening Layers

The last feature map from the convolution and pooling layers is converted to a flat one-column vector. This layer simply reshapes the matrix input into a vector, in order to be compatible to the next layer input (feed-forward layer).

During the backpropagation, the flattening layer takes the input from the following layer and passes it to the previous layer by re-indexing.

2.4.5 Dropout Layers

Dropout is a regularization technique described in [31]. During training, the weights associated with certain nodes are temporarily forced to zero. The neurons in a dropout layer are selected at random with a fixed probability p, and the associated weights are set to zero. p is a hyperparameter set at model architecture design time. In high-level APIs such as Keras[2] and PyTorch[3], dropout is modeled by a separate layer, although this type of layer does not contain trainable parameters. The purpose of random dropout is to prevent overfitting of the trained model by preventing co-adaption of neurons [20]. Random dropout during the training phase forces the model to train individual neurons to a greater degree.

2.5 Activation Function

An activation function in the network performs a (typically non-linear) transformation of the output of a neuron. There are many activation functions such as sigmoid, tanh, ReLU, Leaky ReLU, PReLU, and softmax . Some activations such as Tanh, ReLU, Leaky ReLU, and PReLU are typically used in hidden layers. Activations in the output layer depend on the problem: for example, for classification problems, softmax activation is used. In our paper, we use ReLU in hidden layers and softmax in the output layer.

ReLU (Rectified Linear Unit) deactivates a neuron if the output of its linear transformation is less than 0. Mathematically it can be represented as: $f(x) = \max(0, x)$. During the backpropagation, the derivative with respect to x in the case $x < 0$ is $f'(x) = 0$; for the case $x > 0$, $f'(x) = 1$.

Softmax is used for classification; it transforms the inputs z_i into numbers between zero and one that are interpreted as probabilities of the respective classes: $\textit{softmax}(z_i) = \exp(z_i) \ / \ \sum_j \exp(z_j)$.

2.6 Optimizers

Various optimizers have been proposed to estimate the optimal parameter values of the loss function [17]. Most of these approaches are based on a gradient descent approach that reaches the optimal values of the parameters sequentially by moving from one set of parameters to the next set in the direction of the negative gradient. In our paper, we focused on stochastic gradient descent (SGD) approach which benefits from the knowledge that the gradient of the loss function is a sum over individual data observations. This sum can be approximated by summing over a subsample of all observations and scaling the result up appropriately. The SGD algorithm splits the entire dataset into small subsamples (mini-batches) and then each minibatch is used sequentially

[2] `https://keras.io`
[3] `https://pytorch.org/`

in the optimization iterations to estimate the gradient via back-propagation. Although SGD was chosen for parameter optimization in our work, similar programming constructions can be used to implement other known optimization algorithms for deep learning.

3 Design and Implementation

This section explores how components and certain functionality of CNN architectures can be expressed in a high-level data-parallel pattern-based programming model. We use SkePU in this work, but the insights can be applied to other similar programming frameworks as well. In the design and implementation work we use many features already existing in SkePU; some that have been recently introduced, such as the 4D tensor data-structures and corresponding data-parallel patterns [14], are fundamentally required for CNN workloads. In the interest of presenting new contributions, however, this section focuses in particular on the ways SkePU has been extended with new functionality specifically for this work. This includes strided skeletons in Section 3.1 and the `MapPool` skeleton in Section 3.2. We also introduce a domain-specific API extension to SkePU for (mixed-)DNN applications in Section 3.4.

3.1 Strided Skeletons

SkePU is flexible in how it can accommodate multiple input and output operands (”variadic arity”). However, up until now, invocations of a skeleton pattern on a data set has largely followed a simple rule-of-thumb: the user function is invoked *once per element in the output container(s)*. The output operands are enforced to all have the same size, as are the *element-wise* input operands. Some exceptions have existed before, for example, the user can supply iterators instead of entire smart data containers, in effect applying a skeleton pattern only to a certain window in a data container.

Now, SkePU is extended with an API and implementation for specifying *strides* in skeleton invocations. A stride is the step length between subsequent user function invocations for indexing into smart data container structures.

An example is shown in Listing 3, and visualized in Figure 4, illustrating how strides are mapped to skeleton container arguments. Strides are variadic in the same way skeletons have variable arity: each output and (element-wise) container is assigned its own stride. For `MapOverlap` specifically, the strides work a bit differently. Only the input container can be stride-accessed, but the strides here apply multi-dimensionally, e.g., each dimension in a tensor can have a separate stride length.

Listing 3: Strided skeletons in SkePU.

```
int f(int a, int b) { /* … */ };
auto strideMapper = skepu::Map(f);
skepu::Vector<int> out(16), in_a(1000), in_b(1000);
strideMapper.setStride(2, 4, 3);
strideMapper(out, in_a, in_b);
```

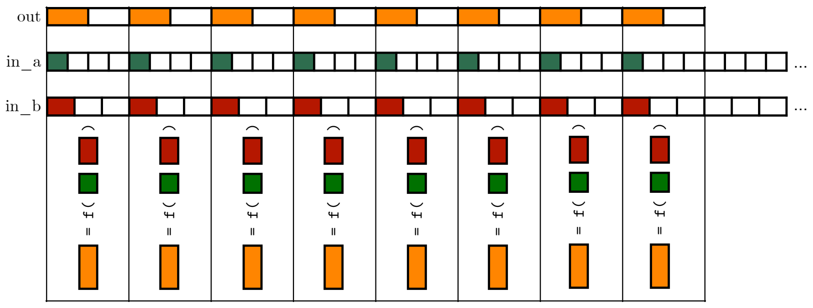


Fig. 4: Reference illustration of variadicly-strided `Map` skeleton.

3.2 `MapPool` Skeleton

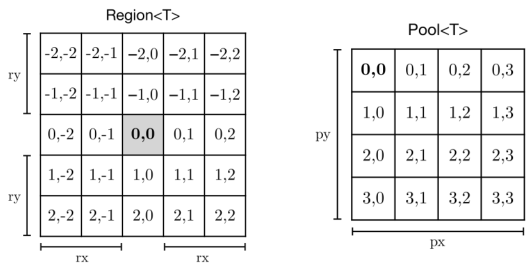
Data-parallel map transformations are by default done element-wise, as is the case for SkePU's `Map`. By adding *proxy containers* to the user function signature, like `Vec` (whole-vector random-access[4]), `Mat` (whole-matrix random-access), or `MatRow` (single-row random access); the user function can be extended to handle a more diverse set of tasks.

For stencil computations, SkePU applies the same general idea. But since this special case is so important for a wide variety of application workloads, for performance-optimization reasons this is integrated in its own separate skeleton pattern: `MapOverlap`. The container proxy called `Region` represents a center element and the hyper-rectangle that is the neighborhood region of elements as determined by overlap radii. The term ”overlap” is used as neighboring elements will have their respective regions overlapping each other. Each dimension can be configured with its own overlap radius. `Region` can only be used in user functions instantiating a `MapOverlap` pattern.

This model of representing regions or neighborhoods in input data is useful in several application domains, not least image processing and iterative differential equation solvers. But it extends poorly to the pooling layers used in convolutional neural networks:

– Pooling is not applied once per element in the input data; rather, the input data is split into a number of regions. The shape of processed data

[4] Random access here means that no access pattern is enforced, and that the user can read whichever elements in whatever order they want. It does not mean that the accesses are randomized.



(a) Region indexing (b) Pool indexing

Fig. 5: Interface and element indexing of the `Region<T>` and `Pool<T>` container proxy types.

is changed, and the existing skeleton set of SkePU primarily operates on input and output data of similar shape.

- The aforementioned regions are *disjoint*, directly contrary to the ”overlap” as used in the `MapOverlap` skeleton name.
- The sizes of pooling regions is typically selected to have even side lengths. `Region` sizes in SkePU are always odd.

For these reasons, SkePU has been extended with a new skeleton pattern: `MapPool`. The programming interface for `MapPool` is very similar to that of `MapOverlap`, but the semantics are different in the aforementioned aspects. See Figure 5 for a visualization how the indexing differs between `Region` and `Pool`. The internal implementation is partially shared between the skeletons. Listing 4 shows `MapPool` in use for implementing a *2D max-pooling* layer of a CNN. Although the pooling itself is two-dimensional, the skeleton is operating on 4D tensor data: the pool size is set to 1 in two of the dimensions.

While max-pooling is an obvious application scenario of the new skeleton pattern, it also has other uses. A pooling operation is in effect a *partial reduction*, and while SkePU has a dedicated `Reduce` skeleton for reductions, it has some limitations. `Reduce` implements partial reductions in just two special cases, *row-wise* and *column-wise* on matrix data. It also assumes that the reduction is performance-critical and applies parallelizations and optimizations in the reduction dimension. For partial reductions where we instead have a large number of small reductions, `MapPool` provides an alternative that can be adapted to fit any reduction direction in 4D space. An example of this is the *softmax* activation layer in CNN classifiers. Softmax scales values with the sum across all candidate classes, as described in Section 2.5. In batch processing modes, there may be a large number of such operations over the same tensor. Listing 5 illustrates how this can be done with `MapPool`.

Listing 4: Max-pooling computation with `MapPool` skeleton.

```cpp
float max_pooling_uf(skepu::Pool4D<float> pool)
{
  float maxval = 10e-10;
  for (size_t j = 0; j < pool.sj; ++j)
    for (size_t k = 0; k < pool.sk; ++k)
    {
      float val = pool(0, j, k, 0);
      maxval = (maxval > val) ? maxval : val;
    }
  return maxval;
}
auto skel_pool_max = skepu::MapPool(max_pooling_uf);

// input, output are of type skepu::Tensor4<float>
skel_pool_max.setPoolSize(1, 2, 2, 1);
skel_pool_max.setStride(1, 2, 2, 1);
skel_pool_max(output, input);
```

Listing 5: Batched softmax with `MapPool` skeleton.

```cpp
float softmax_1(skepu::Pool4D<float> pool)
{
  float sum = 0;
  for (size_t l = 0; l < pool.sl; l++)
    sum += exp(pool(0, 0, 0, l)); // reduction over last dimension
  return sum;
}
float softmax_2(skepu::Index4D idx, float x, skepu::Ten4<float> sums)
{
  return exp(x) / sums(idx.i, idx.j, idx.k, 0);
}
auto skel_softmax_1 = skepu::MapPool(softmax_1);
auto skel_softmax_2 = skepu::Map(softmax_2);

// input, output, temp are of type skepu::Tensor4<float>
skel_softmax_1.setPoolSize(1, 1, 1, input.size_l());
skel_softmax_1.setStride  (1, 1, 1, input.size_l());
skel_softmax_1(temp, input);
skel_softmax_2(output, input, temp);
```

Listing 6: Dropout layer using deterministic parallel PRNG.

```cpp
template<typename T>
T dropout_uf(skepu::Random<1> &rand, T el, float rate)
{
  float p = rand.getNormalized();
  return (p > rate) ? (el * 1.0/(1.0 - rate)) : 0;
}
auto skel_dropout = skepu::Map<1>(dropout_uf<float>);
```

Listing 7: Example in SkePU-DNN

```cpp
#include <skepu-lib/ml.hpp>
size_t BATCH_SIZE = 100;
auto input_shape = skepu::ml::Dimensions{BATCH_SIZE, 28, 28, 1};
auto model = skepu::ml::SequentialModel(input_shape);
model
  << skepu::ml::Conv2D(32).kernel(3, 3)
    .activation(skepu::ml::Activate::ReLU)
  << skepu::ml::MaxPooling2D().kernel(2, 2)
  << skepu::ml::Conv2D(64).kernel(3, 3)
    .activation(skepu::ml::Activate::ReLU)
  << skepu::ml::MaxPooling2D().kernel(2, 2)
  << skepu::ml::Flatten()
  << skepu::ml::Dropout(0.5)
  << skepu::ml::Dense(10)
    .activation(skepu::ml::Activate::SoftMax);
```

3.3 Weight Initialization and Dropout Layer

Deterministic parallel pseudo-random number generators as introduced in [16] are used in SkePU-based CNN models and ensures that a correct PRNG stream is used for *weight initialization* of trainable parameters and also for implementing the *dropout* layer, as shown in Listing 6.

3.4 Domain-specific DNN Interface

So far, this section has shown how to implement DNN functionality in SkePU on a function-by-function basis. SkePU skeletons and smart data-containers form a programming model that is high-level in the sense of abstraction from parallelization and platform-specific code. However, machine learning programmers expect an even higher level of abstraction: domain-specific and declarative interfaces for describing CNN architectures where the key programming constructs are layers and hyperparameters. For this purpose, in this work we also propose a SkePU API specifically for CNN computations. This contribution extends the SkePU standard library [13].

The result is that the user can describe DNN models as in Listing 7 and mix these constructs freely with standard SkePU. Internally, the SkePU-DNN components use skeletons and smart data-containers in line with the explanations earlier in this section.

The model is built lazily, e.g., the operations in Listing 7 are all constant-time and require no system resources. Once the model is initialized, explicitly or implicitly by using it for prediction or learning, all layers have their input and output shapes computed and internal smart data-containers are allocated for parameter storage and intermediate data needed during computation. Batch processing is supported (and recommended for best parallelization opportunities). Both forward and backwards-propagation (the latter used for

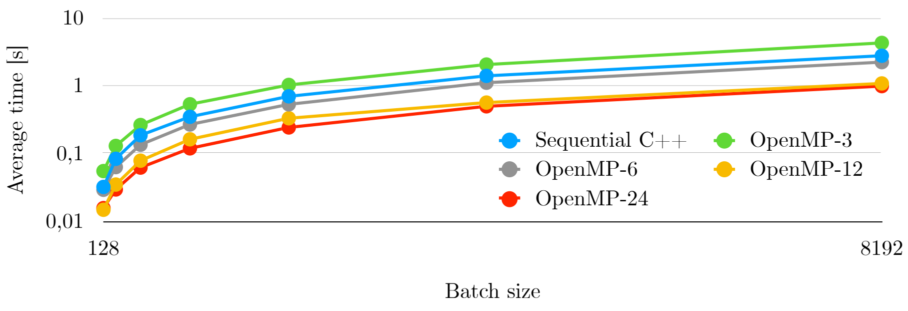


Fig. 6: Parallel speedup of batch predictions using the model in Figure 1, using sequential C++ and OpenMP with varying thread count.

model fitting) is handled by the model abstraction using the internal skeleton patterns and data-containers, using a mix of static polymorphism (e.g., SkePU precompiler and template meta-programming) and dynamic polymorphism (for communication between model layers).

The interface for model fitting is yet to be finalized; we aim for a similar level of abstraction as Keras here as well. There will also be an option for initializing model weights from serialized parameter data, with the data obtained either from earlier training in SkePU-DNN or from external tools such as Keras. This will be part of a future revision of this paper.

4 Evaluation and Discussion

We present preliminary performance evaluation of the parallelization speedup when computing batch predictions with the model in Figure 1 and Listing 7. The model is evaluated on the MNIST handwritten digit dataset (28×28 images), here using the training set of 60000 images, split into batches of powers of two from 128 to 8192. The results from the prototype implementation on a local server with 12 physical cores (24 logical cores) split between two sockets, using g++-10 with -O3 optimization settings, can be seen in Figure 6.

These early results indicate that there is a parallelization bottleneck when using the simple Keras model, even with large batch sizes. Using only a few threads incurs a slowdown compared to sequential execution. Reducing the ratio of writes in the tensor accesses, e.g. by modifying model hyperparameters with larger convolution kernels, eliminates this slowdown. But even with the simple model, speedup is eventually observed when utilizing more cores.

From these results we see that larger and more complex model architectures may be required for future evaluation. We are also looking into identifying the source of memory thrashing behavior in the `MapOverlap` skeleton when applied to tensors of arbitrary dimensions. We also plan to extend the evaluation to include model fitting for a future revision of the paper.

5 Related work

Stencil-pattern computations are at the core of the convolutional layers in CNN, but are also central in other, more traditional application domains such as image processing and iterative linear equation system solving. Accordingly, over the last decades, many domain-specific languages (DSLs) for stencil computations have been developed in order to support generating efficient memory-hierarchy aware and parallel code, such as Halide [28] and HiPAcc [26], as well as stencil patterns in skeleton-programming frameworks, as in SkePU, SkelCL [32], MueSLi [19], FastFlow [1], LiFT [18] and EPSILOD [8]. Tuning meta-parameters in SkePU (v1) convolutions for efficient execution on CUDA GPUs is described by Dastgeer [6].

TinyDNN [5] is a C++14-based header-only library for high-level programming of DNNs, where the specification corresponds to abstract but procedural code, as also in the work presented in this paper. From a specification (i.e., front-end) perspective, we consider TinyDNN closest to our work, with the difference that SkePU also supports general-purpose patterns outside the DNN domain. Unfortunately, TinyDNN is no longer maintained.

DSLs for deep learning include the well-known high-level frameworks TensorFlow, Keras and PyTorch. A common property is that they expect *declarative* specifications of the DNN model, i.e., accept a graph-like description of the neural network with its layers, parameters, training method and hyperparameters, from which the framework internally generates procedural (and possibly accelerator-based) target code for inference and training that is *not* exposed to the end programmer. In this respect, SkePU with its new DNN support features represents a relatively high-level yet procedural middle-ground between declarative DSL frameworks and domain-specific libraries. The advantage of a framework for procedural high-level programming is that it can likewise be used for seamlessly specifying the data pre-/postprocessing computations or other non-DNN code parts of a mixed AI+X application. This provides the option for global optimizations (such as global tiling, kernel fusion, data layout transformations etc.) across domain boundaries, escaping the expressivity or interoperability limitations of separate declarative DSL frameworks.

Method chaining, which we introduced in SkePU as part of this work, is a common programming pattern in Python (and more recently, also C++) libraries such as Spark [33] or PiCo [27] for big-data analytics and machine learning, where computations are commonly expressed as sequences of multiple data-dependent sweeps of operations over an input stream or data-container that is updated in-place, such that textual order matches operation order. While being merely syntactic sugar enabling concise code, it could be suitably combined with (static) identification of operation lineages for cross-operation data locality optimizations, which would otherwise have to be done at runtime, as e.g. in Spark or in SkePU since version 2 [15].

[5] `https://github.com/tiny-dnn/tiny-dnn`

6 Conclusions and Future Work

We have presented a design and implementation of convolutional neural network operations in the high-level skeleton programming framework SkePU. Through a study of computational patterns present in CNN models, we have identified necessary extensions to SkePU, such as a new pooling pattern. Based on this and earlier work, we provide a domain-specific interface for designing and executing DNN workloads in SkePU—which internally uses skeletons and smart data-containers—allowing it to seamlessly mix with other SkePU components. Both the extensions to the core of SkePU as well as the domain-specific API will be included in a forthcoming open-source version of the framework.

For upcoming revisions of this paper, we plan to perform more extensive performance evaluation and comparison, in particular on GPUs. We also aim to use more, and larger, CNN models. In future work, we hope to include more types of deep neural network layers and features, such as RNNs.

Acknowledgments

This work has been partly funded by ELLIIT, project C05 GPAI, and by CUGS. We thank SNIC and NSC for providing access to HPC resources Tetralith (SNIC 2021/22-971, SNIC 2022/22-1248) and Sigma (LiU-gpu-2021-1) that were used for this work.

References

1. M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, G. P. Pezzi, and M. Torquati. The loop-of-stencil-reduce paradigm. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 3*, pages 172–177. IEEE, 2015.
2. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: High-level and efficient streaming on multicore. In *Programming multi-core and many-core computing systems*, chapter 13, pages 261–280. Wiley, 2017.
3. N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems, Jade Edition*, 2011.
4. M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.
5. M. I. Cole. *Algorithmic skeletons: Structured management of parallel computation.* Pitman and MIT Press, Cambridge, Mass., 1989.
6. U. Dastgeer and C. Kessler. A performance-portable generic component for 2D convolution computations on GPU-based systems, Jan. 2012.
7. U. Dastgeer and C. Kessler. Smart containers and skeleton programming for GPU-based systems. *International Journal of Parallel Programming*, 44(3):506–530, 2016.
8. M. de Castro, I. Santamaria-Valenzuela, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. EPSILOD: efficient parallel skeleton for generic iterative stencil computations in distributed GPUs. *J. Supercomput.*, 79(9), 2023.
9. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
10. D. del Rio Astorga, M. F. Dolz, J. Fernández, and J. D. García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, 29(24):e4175, 2017.

11. J. Enmyren and C. W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proc. Fourth Int. Workshop on High-Level Parallel Programming and Applications*, HLPP'10, page 5–14. ACM, 2010.
12. S. Ernsting and H. Kuchen. Algorithmic skeletons for multi-core, multi-gpu systems and clusters. *Int. J. of High Perf. Computing and Networking*, 7(2):129–138, apr 2012.
13. A. Ernstsson. Pattern-based Programming Abstractions for Heterogeneous Parallel Computing. PhD thesis, Linköping University Electronic Press, 2022.
14. A. Ernstsson, J. Ahlqvist, S. Zouzoula, and C. Kessler. SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters. *International Journal of Parallel Programming*, 49:846–866, 2021.
15. A. Ernstsson and C. Kessler. Extending smart containers for data locality-aware skeleton programming. *Concurrency and Comput.: Pract. Exp.*, 31(5):e5003, 2019.
16. A. Ernstsson, N. Vandenbergen, J. Keller, and C. Kessler. A deterministic portable parallel pseudo-random number generator for pattern-based programming of heterogeneous parallel systems. *Int. J. of Parallel Programming*, 50:319–340, Aug. 2022.
17. I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.
18. B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach. High performance stencil code generation with LIFT. In *Proc. Int. Symposium on Code Generation and Optimization (CGO'18)*, pages 100–112. ACM, Feb. 2018.
19. N. Herrmann, B. A. De Melo Menezes, and H. Kuchen. Stencil calculations with algorithmic skeletons for heterogeneous computing environments. *Int. J. Parallel Program.*, 50(5-6):433–453, 2022.
20. G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. arXiv:1207.0580, 2012.
21. A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.
22. Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
23. Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
24. A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön. *Machine learning: a first course for engineers and scientists*. Cambridge University Press, 2022.
25. K. Matsuzaki and K. Emoto. Implementing fusion-equipped parallel skeletons by expression templates. In M. T. Morazán and S.-B. Scholz, editors, *Implementation and Application of Functional Languages*, pages 72–89. Springer, 2010.
26. R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. HIPAcc: A domain-specific language and compiler for image processing pipelines. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):210–224, 2016.
27. C. Misale, M. Drocco, G. Tremblay, and M. Aldinucci. PiCo: A novel approach to stream data analytics. In *Euro-Par 2017: Par. Proc. Worksh.*, pages 118–128. Springer, 2018.
28. J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proc. PLDI*, 2013.
29. C. Rieger, F. Wrede, and H. Kuchen. Musket: A domain-specific language for high-level parallel programming with algorithmic skeletons. In *Proc. ACM Symp. on Appl. Computing*, SAC'19, page 1534–1543. ACM, 2019.
30. D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.
31. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, Jan. 2014.
32. M. Steuwer, M. Haidl, S. Breuer, and S. Gorlatch. High-level programming of stencil computations on multi-GPU systems using the SkelCL library. *Parallel Process. Lett.*, 24(3), 2014.
33. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.

Symposium on High-Level Parallel Programming & Applications
HLPP 2023

LSH SimilarityJoin pattern in *FastFlow*

**Nicolò Tonci · Sébastien Rivault ·
Mostafa Bamha · Sophie Robert ·
Sébastien Limet · Massimo Torquati**

Abstract Similarity joins are recognized to be among the most used data processing and analysis operations. In this work we introduce a C++-based high-level parallel pattern implemented on top of FastFlow Building Blocks to provide the programmer with ready-to-use similarity joins computations. The *SimilarityJoin* pattern is implemented according to the MapReduce paradigm enriched with Locality Sensitive Hashing (LSH) to optimize the whole computation. The new parallel pattern can be used with any C++ serializable data structure and executed on shared- and distributed-memory machines. We present some experimental validation of the proposed solution on two different clusters using the original hand-tuned Hadoop implementation of the LSH-based similarity join algorithms as a reference baseline.

Keywords LSH Similarity Join · High-Level Parallel Programming · Distributed Programming · Parallel Patterns · Big Data MapReduce

1 Introduction

The Similarity join is a fundamental operation in data analysis that consists in finding pairs of close tuples according to a given distance metric. This operation is often used in various applications, including data cleaning [1], entity resolution [2], and collaborative filtering [3,4].

Naively, similarity join computations can be performed by comparing all the data pairs, thus requiring the computation of the entire Cartesian product. This has a fatal effect on performance and limits the scalability of processing

N. Tonci E-mail: nicolo.tonci@phd.unipi.it, M. Torquati E-mail: massimo.torquati@unipi.it
Computer Science Department, University of Pisa, Italy
S. Rivault E-mail: sebastien.rivault@univ-orleans.fr, M. Bamha E-mail: mostafa.bamha@univ-orleans.fr, S. Robert E-mail: sophie.robert@univ-orleans.fr, S. Limet E-mail: sebastien.limet@univ-orleans.fr Université d'Orléans, France

large datasets. A cluster of machines and scalable distributed algorithms are required to perform similarity joins for huge datasets. Consequently, it is challenging for scientists unfamiliar with parallel and distributed computing to employ similarity join algorithms in their applications.

Parallelization patterns, such as MapReduce [5], have gained popularity easing the life of scientists in many application domains, accelerating code prototyping and time-to-solution. This is achieved by hiding all complex low-level mechanisms and ignoring the tuning of a considerable number of parameters to reach the maximum performance of the targeted parallel system.

In [6], we proposed *MRS-join* (MapReduce Similarity Join) a scalable similarity join computation using MapReduce and Locality Sensitive Hashing (LSH). Besides being easy to use for scientists, the proposed approach significantly reduces the number of comparisons needed and the communication costs while guaranteeing perfect computation balancing. The *MRS-join* algorithm was implemented in Hadoop. Apache Hadoop, with the Hadoop Distributed File System (HDFS), is the reference framework of the MapReduce paradigm and the de facto standard in industry and academia due to its ease of use, horizontal scalability, and failover properties. However, Hadoop can be overkill for small-medium datasets in terms of performance when the dataset fits in the aggregated memory of the cluster nodes used or when the MapReduce computation spans multiple chained jobs, like in the similarity join algorithm (i.e., histogram calculation and similarity join computation).

In this work, we propose the *SimilarityJoin* high-level pattern on top of the *FastFlow* [7] library, a parallel programming library providing the programmer with both high-level parallel patterns and a lower-level software layer of nestable and composable data-flow components called Building Blocks (BBs) [8]. As a result, *FastFlow*'s BBs can be used for implementing efficient and scalable data processing with a single source for high-end multi-core servers and a cluster of multi-core nodes. The *SimilarityJoin* pattern interface is back-end agnostic, offering all the benefits of the *FastFlow* library yet hiding all its complex parameters tuning of the *FastFlow* run-time system. Furthermore, the Input/Output data of the pattern is based on standard POSIX files, whereas all intermediate results are computed and stored in the main memory.

We validated the *SimilarityJoin* parallel pattern through a set of experiments based on the *trajectories similarity* use case, showing its scalability on a cluster of 16 server nodes varying the datasets size and the number of nodes employed. Moreover, we made an initial performance comparison with an already existing hand-tuned implementation of the same use case in Hadoop [6], which has highlighted the strengths and current limitations of the proposed *SimilarityJoin* pattern.

The outline of the paper is as follows. Section 2 presents an overview of the LSH-based similarity join algorithm and the *FastFlow* library. Section 3 introduces the *SimilarityJoin* pattern, and its *FastFlow* implementation. Section 4 presents the experimental evaluation conducted. Section 5 provides a discussion of related works and Section 6 draws the conclusions of this paper.

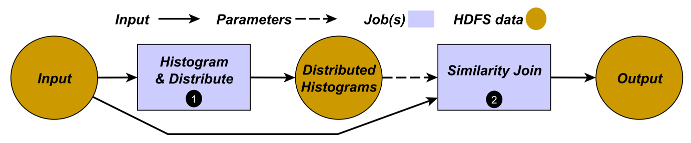


Fig. 1 MapReduce similarity join computation steps in Hadoop.

2 Background

This section presents the principle of the similarity join algorithm and the basics of the *FastFlow* parallel library.

2.1 Similarity join algorithm

Formally, a similarity join for two collections of data R and S is $R \bowtie_\lambda S = \{(u,v) \in R \times S \mid Dist(u,v) \leq \lambda\}$ where Dist(u,v) is a distance between u and v, and λ is the threshold parameter. The algorithm *MRS-join* [6] is based on MapReduce patterns to compute similarity join. To avoid comparing all data pairs, which requires a Cartesian product computation, the search space is reduced using a random hashing framework called Locality Sensitive Hashing (LSH) [9,10]. It is based on a hashing scheme that ensures that nearby data points are more likely to collide than distant ones. The random hashing function depends on the type of data and the chosen distance. To produce a good fraction of all pairs of similar records, several independent iterations are required. The value obtained for each iteration is used as a join attribute. We refer the reader to [11] for more details on LSH.

In large skewed datasets, the computation of the similarity join may be inefficiently distributed, i.e. few computation nodes are used for the distance computations. To avoid these effects, *MRS-join* uses distributed histograms and randomised communication patterns to ensure perfect balancing properties during all the steps of similarity join computations while reducing communication costs to only relevant data [12,6,13]. The histogram of a join is defined as the mapping between a join attribute value and its frequencies. Only join attribute values which might appear in the join result are retained to reduce communication costs to relevant data. For large datasets, we expect that the corresponding histogram does not fit in memory. Therefore, the histogram is distributed according to the occurrence of join attribute values in different portion of input data. It is then used to generate communication templates, allowing to transmit only relevant data fairly during the join processing step. We refer the reader to [12,6,13] for further details about distributed histograms and randomised communication templates.

MRS-join has been implemented in the Hadoop framework. It proceeds in two steps as illustrated in Figure 1 where:

❶ The histogram of the join is computed and distributed to reduce computation to only relevant data while guaranteeing balanced communication patterns.

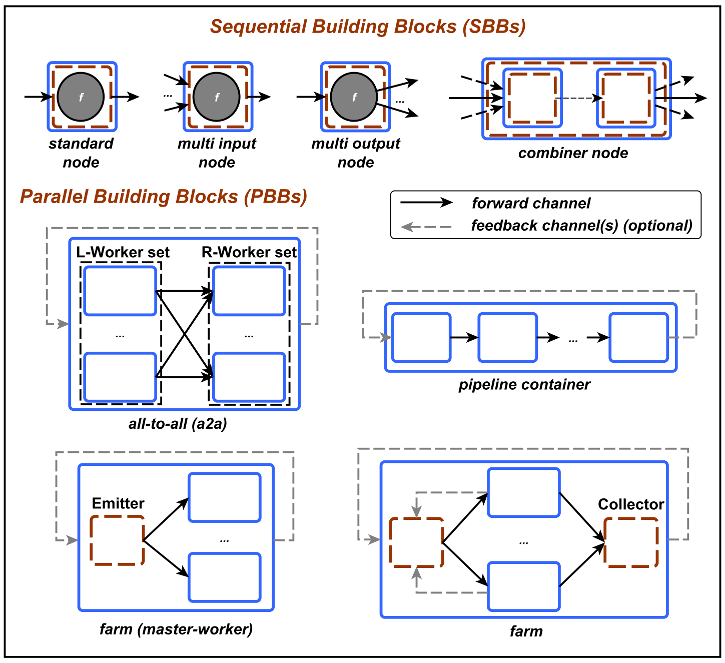


Fig. 2 *FastFlow* shared-memory Building Blocks (`BBs`)

❷ Using distributed histograms, efficient and scalable communication templates are generated to balance the load of distance computations between pairs identified as similar.

At the beginning of each step, the join attribute values are computed using LSH for each record in the input. The step ❶ is composed of two MapReduce jobs, the first one is used to compute the histogram of the join and the second to distribute it. Using distributed histograms, the step ❷ computes the similarity join output from an additional MapReduce job.

2.2 The *FastFlow* parallel library

The C++ header-only *FastFlow* library [7] is the result of a research effort started in 2010 intending to provide application designers with essential features for parallel programming via suitable abstractions and a carefully designed run-time system (RTS). At the lower software layer of the library, there are the so-called *Building Blocks* (`BBs`), i.e., recurrent data-flow compositions of concurrent activities working in a streaming fashion, which are used as the primary components for building *FastFlow* parallel patterns (e.g., Pipeline, ordered Task-Farm, Divide&Conquer, Parallel-For-Reduce, Macro Data-Flow) and, more generally *FastFlow* streaming topologies [14,15].

Following the principles of the structured parallel programming methodology, a parallel application (or one of its components) is conceived by selecting and adequately assembling a small set of well-defined `BBs` modeling data and control flows. The set of *FastFlow* `BBs` are sketched in Figure 2. They comprise both parallel `BBs`, i.e. the pipeline composition (`ff_pipeline`), task-farm (`ff_farm`), and all-to-all (`ff_a2a`), and sequential `BBs`, i.e., standard node (`ff_node`), multi-input/output node (`ff_minode/monode`), and the node combiner (`ff_comb`) implementing the sequential compositions of *FastFlow* nodes.

FastFlow's `BBs` can be combined and nested in different ways forming either acyclic or cyclic concurrency graphs, where nodes are *FastFlow* concurrent entities and edges are communication channels carrying heap-allocated pointers. They have either bounded or unbounded capacity (*feedback* channels always have unbounded capacity). The concurrency control can be either blocking or non-blocking (default). `BBs` mainly target system programmers who want to build new frameworks, patterns or RTSs. All high-level parallel patterns offered by the *FastFlow* library have been implemented using `BBs`.

Initially, *FastFlow* was designed to target multi/many-cores. Recently, its run-time system has been extended to deploy *FastFlow* programs in distributed-memory environments [8]. The distributed RTS has been implemented by leveraging `BBs` and extending them with the objective of preserving the original data-flow streaming programming model. By introducing a small number of edits to programs already written using *FastFlow*'s `BBs`, the programmer may port its shared-memory parallel application to a hybrid implementation (shared-memory plus distributed-memory) in which parts of the concurrency graph will be executed in parallel on different machines according to the well-known SPMD model. Such minimal refactoring involves the introduction of *Distributed Groups* (called *dgroups*) concept, i.e., the identification of logical partitions of the `BBs` composing the application streaming graph according to a small set of graph-splitting rules [8]. A simple example of a *FastFlow* shared-memory streaming application (left-hand side) partitioned into k distributed groups (right-hand side) is given in Figure 3. The *dgroups* have been created by $k-1$ horizontal graph cuts. In a nutshell, a graph cut is valid if the resulting sub-graph can be expressed with the composition of `BBs` (i.e., is a valid *FastFlow* shared-memory application). Currently, inter-*dgroup* (i.e., inter-process) communications leverage raw TCP/IP or MPI, whereas intra-*dgroup* communications use highly efficient lock-free shared-memory communication channels [16].

In the distributed *FastFlow* RTS, data serialization can be carried out in two different ways. The programmer may select the best approach, between the two, for each data type flowing into the inter-group channels (i.e., the data types produced/received by the edge nodes of a *dgroup*). The first approach employs the *Cereal* serialization library [17]. It can automatically serialize base C++ types and compositions of C++ standard-library types; it just requires the implementation of simple mapping functions for custom or user-defined types. The second approach lets the user specify its serialization and deserialization function pair. This might be useful, when feasible, to avoid any extra copies needed by the serialization process itself. In this work, we always

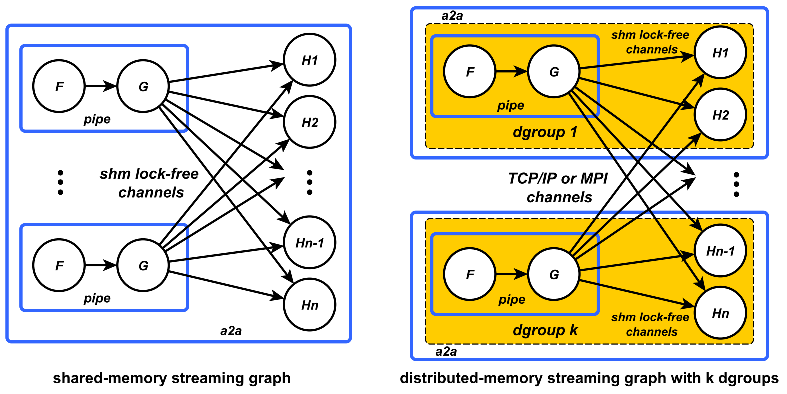


Fig. 3 *FastFlow* streaming graph example. The communication topology is defined as a composition of `BBs`s in a data-flow fashion. The concurrency graph can be partitioned into distributed groups (`dgroups`) each implemented by a dedicated multi-threaded process. In this example, the k `dgroups` have been obtained by cutting the `a2a` `BB` horizontally

use *Cereal* based serialization, which guarantees portable representations over different platforms.

3 The *SimilarityJoin* pattern

3.1 Description

The similarity join pattern is exposed to users through a C++ templated callable object, `SimilarityJoin`, in which inputs and outputs are based on files. Once the template parameter T is fixed at the current datatype of the application, the constructor requires the pattern configuration file's path, the input file's path, a few functions, and, optionally, the size of the application batching feature. The first function defines the parsing of a dataset's line. It returns a template-based `struct` (i.e., `sj_item<T>`) holding the parsed data and some other related metadata. Specifically, it contains three fields: *content* representing the actual data, the *dataset* describing from which dataset the item comes, and a unique identifier, *id*. We expect to have in the input dataset a unique identifier and a tag to get the side (R or S). The next set of functions is used to implement the locality-sensitive hashing (LSH) for a given item. Since usually, the LSH functions are numerous (i.e., 8 or 16), they can be passed as a list of functions (through an iterable `std` container or using the curly brackets notation). Finally, the last function implements the similarity algorithm. It returns `true` if the two passed items are similar, `false` otherwise. In this proposed interface, all the functions may be provided using either lambdas (i.e., anonymous functions), `std::functions`, or C++ functors. The object must be called using the blocking `()` operator to start the execution without

```
struct sj_item<T> {
    size_t id;   // unique identifier
    int dataset; // R or S
    T content;
};
SimilarityJoin<T> instance( configFilePath, inputFilePath,
    [](const string& line) -> sj_item<T> {  /* parse function */ },
    {
      [](const T& o) -> long { /* LSH function i*/ },
      ...,
      [](const T& o) -> long { /* LSH function n*/ }
    },
    [](const T& o1, const T& o2) -> bool { /* similarity function */ },
    batchSize
);
instance();
```

Fig. 4 Similarity Join pattern's interface for a generic data type `T`. It requires: a pattern configuration file, a input dataset file path, a line parsing function, a set of LSH functions, a similarity predicate, and the batching size (optional).

any parameter. An overview of the interface and its data types is sketched in Figure 4.

The configuration file is required to specify the number of processes/hosts, the process-host mapping and the quantity of mappers and reducers for each server node. Thus, it is possible to run and tune the execution of a cluster of heterogeneous machines (i.e., with different number of cores and main-memory capacity). If the file is left blank or not found, the pattern assume the execution with just one computing node (shared-memory execution) in which the mapper and reducer parameters are set to the number of physical cores.

Concerning the input dataset, the *SimilarityJoin* pattern manages it as follows: if the execution is in a single server node (i.e., shared-memory execution), the dataset is expected to be provided as a single monolithic file. Conversely, if the execution spans multiple server nodes, the dataset must be split to assign a partition to each server node (for instance, using three server nodes, the input file must be split/organized into three partitions). In addition, each part must be tagged with a suffix in the form "000", "001", etc.[1]. The final output, similarly to what Hadoop does, is always written into the disk using a separate file for each reducer.

3.2 Use cases

With the interface described in the previous section, it is possible to express several scientific applications leveraging the LSH-based similarity join algorithm and using different data types. This section presents two prominent case studies: *trajectories* and *sets*.

[1] For the case of equal sized partitions and k server nodes, the following `split` Linux command can be used: `split -d -n l/k --suffix-length=3 inputDatasetPath outputDatasetPath`

```
using data_t = std::vector<std::tuple<double, double>>;
sj_item<data_t> parseDatasetLineF(const string& line) { ... }
long lsHash(const data_t& o, size_t id){ ... }
SimilarityJoin<data_t> trajExample( configFilePath, inputFilePath,
    parseDatasetLineF,       // parse function
    {
      std::bind(lsHash, 1), //  1st LSH function
      ...,
      std::bind(lsHash, n), // n-th LSH function
    },
    [](const data_t& o1, const data_t& o2) -> bool { // similarity function
        return Frechet(o1,o2) < THRESHOLD;
    }
);
```

Fig. 5 Example of usage of the *SimilarityJoin* pattern for the *trajectories* use-case. The dataset parsing and LSH implementations are omitted for brevity.

Trajectories are seen as polygonal lines where each point belongs to $\mathbb{R}^d$ with d the space dimension. Thus, the datatype T of the pattern can be something like `std::vector<std::tuple<double, double>>` for a two-dimensional space. The similarity algorithm employs the discrete distance of Fréchet [18] to compare trajectories. The Fréchet distance is often explained by the following metaphor: a man holds his dog on a leash, both are walking on finite trajectories. Man and dog can vary their speed but cannot turn back. The Fréchet continuous distance is the minimum length of the leash to connect the man to his dog during the entire journey. To speed up the distance computations, the algorithm uses several heuristics to test in near-linear time if two trajectories have a distance less than a given threshold λ [19,20]. The Fréchet LSH function family [21,22] uses a random grid of dimension d defined from a resolution σ and an origin randomly chosen in the half-open hypercube $[0, \sigma[^d$. Each trajectory is transformed into a sequence of grid nodes. The resulting sequence is universally hashed to be used as a join attribute. The same procedure is iterated several times to produce a good fraction of all pairs of similar trajectories. Therefore, the Fréchet distance is only computed for trajectories with the same sequence of grid nodes. The resolution parameter of the grid is set to $\sigma = 4 \times d \times \lambda$ as it was done in the experiments of [6,22]. Trajectories application is used for the experimental evaluation in Section 4.

A second well-known similarity join application in the literature concerns sets. It can be applied to different data-structure representing a group of objects. This can be easily done by varying the template parameter and adapting LSH and similarity functions. Similarity join on sets has a large number of applications, including similar text detection [23,24], collaborative filtering [3], and clustering of large malware dataset [25]. The pruning power of the set similarity join is also used to reduce the number of candidate pairs for edit-based string similarity joins [26]. A popular distance in the literature is the Jaccard distance defined as $\text{Jaccard}(u, v) = 1 - \|u \cap v\| / \|u \cup v\|$ where $\|\cdot\|$ is the cardinality of a set. MinHash is a family of LSH functions that estimates the Jaccard distance. It takes advantage of a random permutation to retrieve

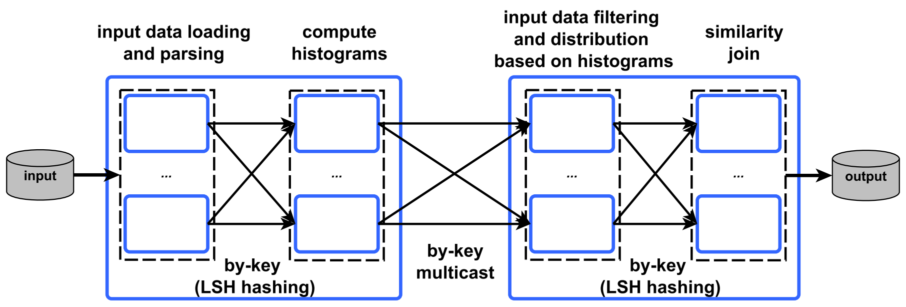


Fig. 6 `BBs`-based implementation of the *SimilarityJoin* pattern: a pipeline of two all-to-alls. The left-hand Workers are the Mappers. The right-hand side ones are the Reducers.

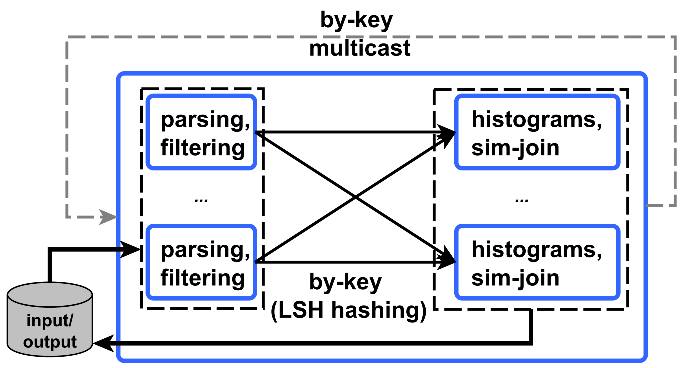


Fig. 7 Resource-optimized implementation of the *SimilarityJoin* pattern with *FastFlow*BBs: a single all-to-all with feedback channels.

similar sets. Each set is hashed by its element with the smallest position in the random permutation. The procedure is repeated several times to produce almost all pairs of similar sets. We refer the reader to [27,28] for more detailed information on MinHash and to [13] for details on the implementation of *MRS-join* for set similarity join.

3.3 *FastFlow*-based implementation

This section describes the implementation of the *SimilarityJoin* pattern using *FastFlow* as a back-end, allowing us to execute the application both in shared-memory and distributed-memory environments using a single-source.

The algorithm can be implemented as a pipeline composition of two map-reduce phases: the computation of histograms and the effective similarity join. In *FastFlow*, we can implement it using a `pipeline` of two stages, each implementing a map-reduce. The map-reduce paradigm can be easily implemented using the `a2a` `BB` (i.e., all-to-all) as sketched in Figure 6. The mappers are those *FastFlow* sequential `BBs` in the left-hand set of Workers of the all-to-all, whereas the reducers are those sequential `BBs` in the right-hand set of

Workers. However, since the two map-reduce phases are computed in batch, i.e., computational phases do not overlap, the implementation can be realized using a single all-to-all `BB` iterated two times by leveraging the `feedback` modifier of the `a2a` `BB` In this way, mappers and reducers must contain the business logic of both phases. The resulting graph topology is shown in Figure 7. Such reduced configuration optimizes resources halving the number of *FastFlow* sequential nodes and increases flexibility when moving from shared to distributed memory. In fact, exploiting a single all-to-all as a root `BB` allows the resulting graph to be cut both vertically as well as horizontally (and also in a mixed fashion) to create sub-graphs composing the distributed groups that are still valid all-to-all `BBs`. Unlike Hadoop MapReduce, the implementation does not store nor sort intermediate results on disks, preferring to store all data in the main memory instead. However, this can also be seen as a weak point of the current implementation in the case of huge datasets. Large datasets cannot be executed on memory-constrained systems. In the *FastFlow*-based implementation, mappers send directly to reducers the key-value pairs in a streaming fashion using the shuffle communication pattern of the all-to-all `BB`. The burden of sorting data is left to reducers that perform this step while receiving data, partially amortizing its cost by overlapping the computation time with those of mappers. The same kind of streaming-like computation is also exploited in the next phase through feedback channel communications to distribute the histograms back to mappers who merge to a local and private data structure each received histogram partition.

We now briefly go through the main steps of the algorithm, giving some details of how those are implemented in *FastFlow*. The first phase is the computation of histograms. Mappers, in parallel, seek their part and start parsing the memory-mapped dataset line by line calling the provided parsing function. For each item, they invoke all the given LSH functions and emit the results to the corresponding reducers along with the dataset tag (R/S). Reducers count the frequency of hash values based on the dataset tag and store the sender id of the mapper. The latter is used to send back, through the feedback channels, only the relevant frequencies to each mapper. The reducers trigger the distribution of histograms once they have received all data from all mappers, indicating that no more `key-value` pairs will be emitted in this phase. This kind of end-of-phase signal is implemented at the pattern run-time with a custom `<key, value>` pair detected by the *FastFlow* `BB` implementing the `a2a` reducers. Afterward, as soon as all histograms produced by the reducers are received and merged by the mappers, the next phase (i.e., similarity join) can start. Now, for each hashed value of all elements of the dataset partition, the mapper knows the frequency of that value for both sides (R/S) and thus may decide to discard or multicast the computed content to a proper subset of reducers. A hashed value is discarded if it comes from a side and the frequency of the other side is equal to zero. This pre-filter means that there are no items of the other set that is possibly similar to the one being processed according to the LSH family of functions provided. The number of reducers required to balance computation for that hash value is retrieved by computing the maximum

```
1: function RandomUnicast($o, R_0, R$)
2:     ff_send_out_to(o, $(R_0 + (rand() \bmod R)) \bmod \#totalReducers$);
3: function Multicast($o, R_0, R$)
4:     for $i \leftarrow 1$ to $R$ do
5:         ff_send_out_to(o, $(i + R_0) \bmod \#totalReducers$);
6: function Process($o, lsh$)
7:     if $min\{freq_R(lsh), freq_S(lsh)\} = 0$ then return;
8:     $\#reducers_{req} \leftarrow min\{\lceil \frac{max\{freq_R(lsh), freq_S(lsh)\}}{f_{max}} \rceil, \#totalReducers\}$;
9:     $index_{R0} \leftarrow lsh \bmod \#totalReducers$;
10:    if $freq_R(lsh) > freq_S(lsh)$ then
11:        if o comes from R then RandomUnicast($o, index_{R0}, \#reducers_{req}$);
12:        else  Multicast($o, index_{R0}, \#reducers_{req}$);
13:    else
14:        if o comes from S then RandomUnicast($o, index_{R0}, \#reducers_{req}$);
15:        else  Multicast($o, index_{R0}, \#reducers_{req}$);
```

Fig. 8 Algorithm followed by the mappers during similarity join phase to implement the randomised communication schema in *FastFlow* to ensure load balancing among reducers for skewed datasets. The process function is the one called by the mapper for all LSH for each object. Random Unicast and Broadcast are commodity procedure to wrap the *FastFlow*'s `ff_send_out_to` method.

frequency between side R and S, divided by the parameter f_{max}. Such (f_{max}) value denotes the number of per-key records that a reducer should store and process during the similarity join step. The quantity of instantiated reducers always bounds the computed number of required reducers. Once the number of reducers is defined, we need to determine which ones will be used among all we have available. To do so, we choose the first one and then all the subsequent ones up to the amount needed. Then, the first reducer is chosen by computing the modulo operation between the LSH value and the number of all instantiated reducers. Then, if a hash comes from a side X element (between R and S) and the maximum frequency is from the same side X, it will be sent to one of the selected reducers. Otherwise, it will be sent to all selected reducers. This way, we minimize the quantity of replicated data, broadcasting the elements from the smaller sub-dataset (R or S) for a given key. This procedure is summarized in Fig. 8 and is implemented exploiting the `ff_send_out_to` method of the multi-output sequential `BB`.

Once all the `<key,value>` have been generated during the second phase, mappers send the *FastFlow end-of-stream* (`EOS`) message and terminate. Reducers collect all the items and sort directly while receiving. As soon as all `EOS` messages were received, the mapper of the similarity join procedure, is triggered. Similarity join consists of testing the provided predicate over all the possible combinations of the elements with the same locality-sensitive hash key. For similar items, the reducer prints to its output file the pair of related IDs.

In both map-reduce phases, data is sent from mappers to reducers in batches. This feature has been added to reduce the number of exchanged messages and, most of all, to optimize memory allocation for shared-memory executions. The batching size can be set in the pattern constructor, its default value is set to 256.

Concerning the distributed-memory execution of the *SimilarityJoin* pattern, the user needs to properly define the configuration file (the first parameter of the pattern instance). Based on the number of lines contained in the configuration file, the corresponding number of horizontal cuts will be automatically applied to the `a2a` `BB`, which defines the implementation skeleton of the pattern. Each resulting `dgroup` has a unique identifier (e.g., `G0`..`Gn` where `n` is the number of lines of the configuration file). These identifiers are used to define the `dgroup` to server host mapping and to tag inter-group communications by the *FastFlow* RTS. The `Gk` group is created with the number of Mappers and Reducer specified by the user in the line-`k` of the configuration file. Communications between local group Mappers and Reducers happen in the shared-memory domain (i.e., communication channels are implemented with lock-free shared-memory queues), whereas communications from a Mapper in a group and a Reducer in a different group happen in the distributed-memory domain (i.e., inter-node communications). Data is serialized if the communication is inter-`dgroups`. Data serialization is wholly entrusted to *Cereal* library, which requires the user to provide a serialization function for type T specified in the pattern template, if and only if T is a custom data type (i.e., user-defined `class` or `struct`).

4 Experimental evaluation

In this section, we assess both the quality of the new C++ *SimilarityJoin* pattern based on the *FastFlow* framework and its parallel performance on two different clusters with different interconnections, main memory availability, and number of cores per node. We consider the hand-tuned Hadoop-based implementation of the *MRS-join* computation presented in [6] as the baseline.

4.1 Platforms

The experiments were carried out on two different clusters. The first one called *Mirev*, is hosted by the University of Orléans, and it is composed of two servers connected by a switched 1Gbit/s *Ethernet* network. Each server has 256GB of memory and two Intel Xeon Gold 6248R CPUs running at 3.0GHz for a total of 48 physical cores (96 hardware threads) and a fast dedicated NVMe disk for local storage. The second cluster called *Openhpc4* is hosted by the University of Pisa's Green Datacenter. It includes 16 nodes connected by a 100Gbit/s $Infiniband$ network. Each server is a diskless node with two Intel Xeon Silver 4114 CPUs running at 3.0GHz for a total of 20 physical cores (40 hardware threads). About 128GB of memory is reserved for running applications on one *Openhpc4* cluster node. All distributed tests presented in this section were executed using the TCP/IP transport protocol for the *FastFlow*-based implementation.

Dataset size	**Similar items in the dataset (M=1,000,000)**	**Similarities found**	
		Hadoop version	***FastFlow* version**
5GB	50M	99.36%	99.51%
50GB	100M	99.34%	99.51%
100GB	1000M	99.35%	99.51%

Table 1 Percentage of similarities discovered by the two implementations for the three distinct datasets considered. The Hadoop-based version is configured with 48 Mappers and 24 Reducers with 2GB and 4GB of memory, respectively. The *FastFlow*-based version is configured with 24 Mappers and 24 Reducers and no memory limitations. The number of LSH functions used is 8, the threshold value is set to $\lambda = 10$.

4.2 Datasets

To use different-sized datasets for the experiments, we employed a synthetic data generator that allows us to specify the number of similarity join outputs. To this end, the generator takes as parameters the number of clusters and their sizes in the datasets R and S and a threshold value. For each cluster, the algorithm generates a random trajectory used as a template. The cluster trajectories take this template and alter it according to the given threshold. The dataset is supplemented with noise, i.e., several random trajectories that will not produce any similarity join output. For example, in the smallest dataset (i.e., 5GB), there are 5,000 clusters of size 200 equally distributed in R and S, with 2,000,000 additional random trajectories. For larger datasets, we increase the number of clusters and the number of random trajectories by the same factor so that the number of results produced follows the same scaling. All generated trajectories are 2-dimensional and comprise an average of 50 points spaced according to the given threshold. In our tests, we used three distinct datasets, namely: a “small” dataset of 5GB with 50 million of similar trajectories, a “medium” dataset of 50GB with 500 million of similar trajectories, and a “large” dataset of 100GB with 1 billion similar trajectories.

4.3 *SimilarityJoin* pattern validation

To validate the *FastFlow*-based *SimilarityJoin* pattern implementation, we compared the number of similar trajectories found for the three datasets considered with those found by the Hadoop-based implementation proposed in [6]. The results obtained running the two versions on the *Mirev* cluster are reported in Table 1. The number of similar trajectories found (reported in percentage in the table) for all datasets is almost the same. The *FastFlow*-based version is capable of obtaining a small extra-fraction of similar trajectories. We did not investigate deeply such a small difference. Reasonably, it could be due to different random seeds. For the validation tests and for all performance tests presented in the following, we used 8 LSH functions and a threshold value of 10 (i.e., $n = 8$ and $\lambda = 10$ in the code in Figure 5). It is worth noting that by using 8 LSH functions, the memory requirement for the largest dataset is relevant (there is a factor on the input size of about $2.4X$ with 8 functions). Specifically, the amount of memory needed to run the *SimilarityJoin* pattern on a single node with the 100GB dataset is a bit more than 240GB. Therefore, the *FastFlow*-based version cannot be executed either on a single *Openhpc4* node or on a single *Mirev* node. In both server nodes, the virtual memory size is equal to the physical memory available (that is equal to the nominal memory minus the space used by the OS and the running services). For this reason, the validation for the 100GB dataset with the *FastFlow*-based version has been conducted on two *Mirev* nodes. Finally, the number of similarities found in the dataset strongly depends on the number of LSH functions used. For example, for the 5GB dataset, with 2 LSH functions, the similarity score falls to 73.7% whereas with 16 LSH functions, the similarity score reaches 99.99%

4.4 Performance evaluation

The first test aims to evaluate the `batchSize` parameter of the *SimilarityJoin* pattern (cf. Figure 4). This internal pattern parameter aims to reduce both the number of messages exchanged between Mappers and Reducers in each iteration phase and to intensify contiguous main memory allocation for messages in the *FastFlow* run-time. The results of the test conducted on one node of both clusters are shown in Figure 9 (top left-hand side). The dataset is the $5GB$ one. The optimal value of the `batchSize` for both platforms falls in the range $32 - 512$. With smaller values, the overhead of memory management is higher in the *FastFlow* run-time. Higher values of `batchSize` might reduce the pipeline parallelism between Mapper and Reducers.

The second test aims to estimate a good value for the number of Mappers and Reducers for a single node. Again we used for the test the smallest dataset (i.e., 5GB) and one single cluster node. The `batchSize` was fixed to 256. Figure 9 shows the results obtained on one *Mirev* node (top right-hand side) and on one *Openhpc4* node (bottom left-hand side). In these tests, we did not verify all possible configurations for the number of Mappers and Reducers. Our aim was to estimate a "close-to-optimal" value for the per-node number of Mapper and Reducer parameters of the *SimilarityJoin* pattern and to verify that using either the total number of physical cores (assigning half of them for the Mappers and the other half for the Reducers) or the total number of logical cores (yet half and half) may be a reasonable choice for those parameters. Specifically, in the *Mirev* node the best value among those tested are $24 - 24$, whereas on the *Openhpc4* node the best values are $20 - 20$ (thus filling all logical cores for a node).

The next test was to estimate a good value of the *distributed batching size* (called `dFF_batch`) of the *FastFlow* run-time system to optimize the network bandwidth in distributed communications transparently. For this test, we employed two nodes of the two clusters. Unlike the `batchSize` parameter, the distributed batching configuration parameter is used only for distributed communications between non-local-node Mappers and Reducers. Such

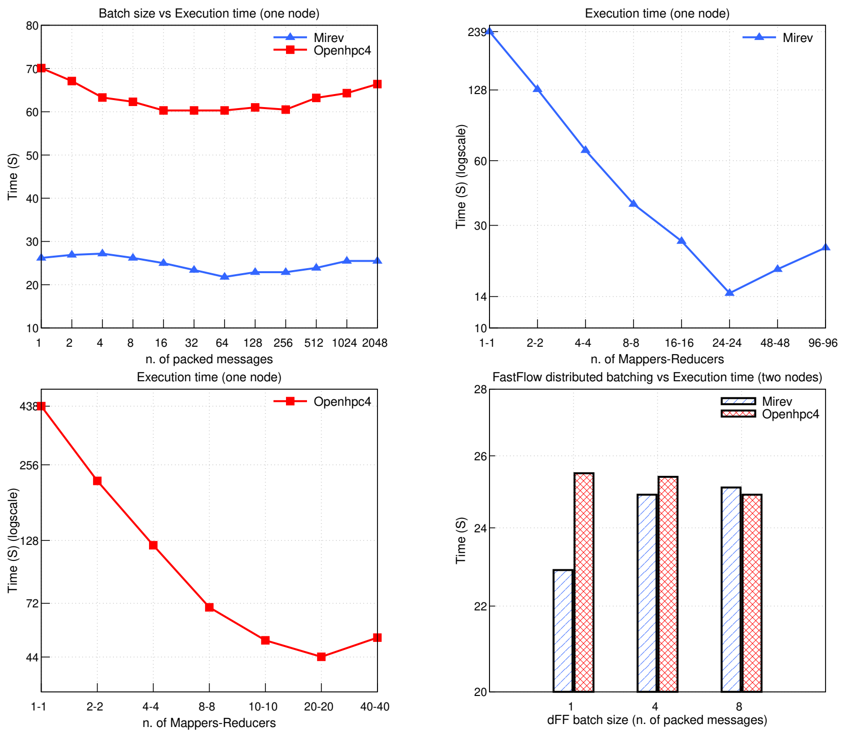


Fig. 9 Dataset size: 5GB. `Top left`: Impact on the execution time of the `batchSize` parameter. `Top right`: Execution time varying the number of Mapper and Reducer on a single *Mirev* node (`batchSize`=256). `Bottom left`: Execution time varying the number of Mapper and Reducer on a single *Openhpc4* node (`batchSize`=256). `Bottom right`: Impact on the execution time of the *FastFlow* distributed batching on two cluster nodes (`batchSize`=256).

value depends on the amount of data transmitted (thus also depends on the value of the `batchSize`) and the kind of network and protocol used. However, as it can be seen from Figure 9 (bottom right-hand side), with large enough application batching (i.e., `batchSize`=256), there is no significant difference between the case of `dFF_batch`=1 (i.e., no distributed batching) and the case of `dFF_batch`=8, in particular for the fastest network (i.e., *Infiniband* in the *Openhpc4* cluster). Given the results of this test, we decided to set the `dFF_batch` to 4 for the *SimilarityJoin* pattern implementation. All subsequent tests were executed with such fixed value regardless of the cluster used.

Once all pattern parameters had been studied (i.e., `batchSize`, number of per-node Mappers and Reducers, and the `dFF_batch`) we tested the *SimilarityJoin* pattern scalability by increasing both the dataset size as well as the number of cluster nodes. Table 2 summarizes the results obtained on the *Openhpc4* cluster for the three datasets considered varying the number of

Dataset size	Execution time						
	Openhpc4					Mirev	
	1	2	4	8	16	1	2
5GB	46s	23s	14.5s	11.7s	7.7s	17s	25s
50GB	*no mem*	299s	143s	88s	52s	283s	228s
100GB	*no mem*	*no mem*	298s	154s	82s	*no mem*	450s

Table 2 Execution times (in seconds) varying the number of machines up to 16 nodes on the Openhpc4 cluster and up to 2 nodes on the Mirev cluster for the three datasets considered. Configuration: 20 Mappers and 20 Reducers per node on *Openhpc4*, 24 Mappers and 24 Reducers per node on *Mirev*, `batchSize`=512, `dFF_batch`=4. The 50GB and 100GB datasets cannot be executed in all configurations due to the lack of main memory.

Dataset size	Hadoop version (Exec. time)		*FastFlow* version Improvement	
	1	2	1	2
5GB	133s	140s	7.8X	5.6X
50GB	507s	537s	1.8X	2.4X
100GB	1040s	775s	none	1.7X

Table 3 Execution times (in seconds) of the Hadoop version when using 1 and 2 nodes on the *Mirev* cluster. Hadoop configuration: 48 Mappers and 24 Reducers (2GB and 4GB of memory each, respectively). Improvement factor of the *FastFlow*-based version (cf. Table 2) vs. the Hadoop one on the same cluster. The 100GB dataset, cannot be executed on one single node by the *FastFlow*-based version due to the lack of available main memory.

nodes from 1 to 16. Given the relatively small amount of main memory available on each node of the *Openhpc4* cluster, the 50GB and 100GB datasets cannot be executed on 1 and 2 cluster nodes, respectively. However, the execution time scales reasonably well with the number of nodes for the most extensive dataset (100GB). On the *Mirev* cluster, given the additional number of cores per processor and the slower network, there is no performance improvement when moving from 1 node to 2 nodes for the 5GB dataset. The single-node shared-memory version of the pattern exploiting all physical cores of the machine (i.e., 24+24) is already relatively fast (about 17s). For the 50GB dataset, the execution time improvement is marginal, from 283 to 228 seconds. Instead, the *SimilarityJoin* pattern cannot be executed for the largest dataset on a single node because of insufficient main memory. Still, it can be run on two nodes giving an execution time of about 450s. We executed the Hadoop version on the *Mirev* cluster for all datasets to have a performance comparison. The Hadoop framework is configured to use the HDFS filesystem with 1 name node and 2 data node. The results obtained are shown in Table 3. As expected, the overhead introduced by the Hadoop framework is not negligible for small datasets. On the other hand, for big datasets, Hadoop can execute on a single node with constrained memory (192GB) and obtains a relevant reduction of the execution time of about 300s on two cluster nodes. The execution time differences between the Hadoop version and the *SimilarityJoin* pattern implemented in *FastFlow*, are primarily due to: i) the extensive use of the HDFS filesystems in Hadoop, which introduces overheads but enables running with

"Larger-than-Memory" datasets; ii) the storing of intermediate phase results into files; iii) the serialization of all messages between Mappers and Reducers. Finally, Hadoop provides the user with fault-tolerance capability (though not enabled in our tests), a feature currently missing in the *SimilarityJoin* pattern implementation. Reasonably, we may expect the larger the input dataset, the better the performance of the Hadoop version.

5 Related Work

Exact similarity joins have received considerable attention. Filtering and verification techniques use filters to eliminate comparisons that cannot reach the threshold distance. However, such techniques are metric-space-dependent and cannot be applied to the general case. On the other hand, the metric-space-partitioning technique permits handling the similarity join for any metric space. Nevertheless, the experimental survey [29] on exact set similarity joins shows that these methods often fail to compute the join on small datasets. Recently in [30], "Bloom filters" and "Fuzzy filters" were introduced for fuzzy join operations to eliminate most non-joining elements in the input dataset before sending the data to the join processing step. Thus, it reduces intermediate data and unnecessary comparisons. However, these approaches may face scalability problems when the probabilistic data structures used to store the filters cannot fit in the main memory of the processing nodes, especially for massive datasets. In an approximate context, i.e., algorithms that do not produce the total results, similarity join algorithms are usually based on Locality Sensitive Hashing (LSH) to generate candidate pairs by hashing similar input records into the same "buckets" with high probability. This drastically reduces the number of pair comparisons and generates almost all similarity join results. In the context of massively parallel computations, some recent works [31,32] present an algorithm relying on LSH that achieves guarantees on the result completeness and good utilization of the processing nodes. However, in the case of large and skewed datasets, the workload balance among computing nodes is not ensured. Recently, [33] extented the analysis and improved the algorithm using sketching and deduplication.

Concerning programmability, offering easy-to-use yet sophisticated parallel pattern for specific application domains to end-users, who are not necessarily familiar with parallel programming, is a relevant research topic in the fields of high performance distributed computing. Notably, several parallel programming libraries or domain-specific languages (DSLs) have been proposed in the context of structured parallel programming [34], e.g., Muesli [35], SkePU [36], SkeTo [37], GrPPI [38], SkelCL [39], Musket [40] and SPar [41]. FastFlow [7] owns to this category but, in addition to some high-level parallel patterns, it also provides a lower-level software layer to the parallel programmers (i.e., *Building Blocks*) to enable the easy development of new patterns and run-times system yet following the structured parallel programming methodology [14]. Several examples of domain-specific parallel patterns can be found in the literature, ranging from exact combinatorial search [42] in the distributed-memory domain to image filtering for visual data restoration [43] based for heterogeneous many-cores equipped with GPU accelerators, to Window-based stateful data-streaming operators for multi-core systems [44]. In the field of similarity joins, some works targeted different methodologies for different architectures: *HySet* [45] and *fgssjoin* [46], offering set similarity exploiting both CPU and GPU accelerators but not clusters, whereas in [47] the authors proposed an online streaming approach using distributed systems. However, all previous works focus mainly on set similarity, while the *SimilarityJoin* pattern we proposed in this work aims to target all similarity join operations.

6 Conclusions and Future Work

We proposed *SimilarityJoin*, a C++ high-level parallel pattern for computing similarity joins that relieves the user from many hidden pitfalls related to parallel programming. The implementation, based on *FastFlow*'s `BBs`, follows the MapReduce computation paradigm, enabling efficient execution on a single multi-core server and a cluster of multi-cores. The proposed solution has been validated using different-sized datasets, and its scalability has been studied on a 16-node cluster. Additionally, an initial performance comparison with a hand-tuned Hadoop implementation of the same use case has yielded interesting qualitative and quantitative results.

In future work, we intend to improve the memory management of the *SimilarityJoin* pattern to enable the execution of "larger-than-memory" datasets and tune its absolute performance by comparing it with other Big-Data frameworks such as Spark [48], optimized for in-memory computing.

References

1. Chaudhuri, S., Ganti, V., Kaushik, R.: A Primitive Operator for Similarity Joins in Data Cleaning. In: 22nd International Conference on Data Engineering (2006)
2. Dey, D., Sarkar, S., De, P.: A distance-based approach to entity reconciliation in heterogeneous databases. IEEE Transactions on Knowledge and Data Engineering **14**(3), 567–582 (2002)
3. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: Proceedings of the 16th International Conference on World Wide Web, pp. 131–140 (2007)
4. Shang, Y., Li, Z., Qu, W., Xu, Y., Song, Z., Zhou, X.: Scalable collaborative filtering recommendation algorithm with mapreduce. In: 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, pp. 103–108 (2014)
5. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM **51**(1), 107–113 (2008)
6. Rivault, S., Bamha, M., Limet, S., Robert, S.: A scalable similarity join algorithm based on MapReduce and LSH. International Journal of Parallel Programming **50**(3-4), 360–380 (2022). DOI 10.1007/s10766-022-00733-6
7. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. Programming multi-core and many-core computing systems, parallel and distributed computing (2017). DOI 10.1002/9781119332015.ch13
8. Tonci, N., Torquati, M., Mencagli, G., Danelutto, M.: Distributed-memory fastflow building blocks. Intl. Journal of Parallel Programming (2022)

9. Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, pp. 604–613 (1998). DOI 10.1145/276698.276876
10. Gionis, A., Indyk, P., Motwani, R.: Similarity Search in High Dimensions via Hashing. In: Proceedings of the 25th International Conference on Very Large Data Bases, pp. 518–529 (1999)
11. Wang, J., Shen, H.T., Song, J., Ji, J.: Hashing for similarity search: A survey (2014)
12. Hassan, M.A.H., Bamha, M., Loulergue, F.: Handling data-skew effects in join operations using mapreduce. Procedia Computer Science **29**, 145–158 (2014). DOI 10.1016/j.procs.2014.05.014. 2014 International Conference on Computational Science
13. Rivault, S., Bamha, M., Limet, S., Robert, S.: Towards a scalable set similarity join using mapreduce and lsh. In: Computational Science – ICCS 2022: 22nd International Conference, London, UK, June 21–23, 2022, Proceedings, Part I, p. 569–583. Springer-Verlag, Berlin, Heidelberg (2022). DOI 10.1007/978-3-031-08751-6_41
14. Torquati, M.: Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns. Ph.D. thesis, University of Pisa (2019)
15. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Design patterns percolating to parallel programming framework implementation. Intl. Journal of Parallel Programming **42**(6), 1012–1031 (2014). DOI 10.1007/s10766-013-0273-6
16. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An efficient unbounded lock-free queue for multi-core systems. In: Euro-Par 2012 Parallel Processing, pp. 662–673. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-32820-6_65
17. Grant, W.S., Voorhies, R.: Cereal a C++11 library for serialization
18. Alt, H., Godau, M.: Computing the fréchet distance between two polygonal curves. Int. J. Comput. Geom. Appl. **5**, 75–91 (1995)
19. Werner, M., Oliver, D.: Acm sigspatial gis cup 2017: Range queries under fréchet distance. SIGSPATIAL Special **10**(1), 24–27 (2018)
20. Driemel, A., Har-Peled, S., Wenk, C.: Approximating the fréchet distance for realistic curves in near linear time. CoRR **abs/1003.0460** (2010)
21. Driemel, A., Silvestri, F.: Locality-Sensitive Hashing of Curves. In: B. Aronov, M.J. Katz (eds.) 33rd International Symposium on Computational Geometry (SoCG 2017), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 77, pp. 37:1–37:16. Dagstuhl, Germany (2017). DOI 10.4230/LIPIcs.SoCG.2017.37
22. Ceccarello, M., Driemel, A., Silvestri, F.: Fresh: Fréchet similarity with hashing. In: Z. Friggstad, J.R. Sack, M.R. Salavatipour (eds.) Algorithms and Data Structures, pp. 254–268. Springer International Publishing, Cham (2019)
23. Theobald, M., Siddharth, J., Paepcke, A.: Spotsigs: Robust and efficient near duplicate detection in large web collections. In: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval (2008). DOI 10.1145/1390334.1390431
24. Wandelt, S., Deng, D., Gerdjikov, S., Mishra, S., Mitankin, P., Patil, M., Siragusa, E., Tiskin, A., Wang, W., Wang, J., Leser, U.: State-of-the-art in string similarity search and join. SIGMOD Record **43**(1), 64–76 (2014). DOI 10.1145/2627692.2627706
25. Oprişa, C., Checicheş, M., Năndrean, A.: Locality-sensitive hashing optimizations for fast malware clustering. In: 2014 IEEE 10th International Conference on Intelligent Computer Communication and Processing (ICCP), pp. 97–104 (2014). DOI 10.1109/ICCP.2014.6936960
26. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 918–929 (2006)
27. Broder, A.Z., Glassman, S.C., Manasse, M.S., Zweig, G.: Syntactic clustering of the Web. Computer Networks and ISDN Systems **29**(8), 1157–1166 (1997). DOI 10/br259g
28. Shrivastava, A., Li, P.: Densifying One Permutation Hashing via Rotation for Fast Near Neighbor Search. In: Proceedings of the 31st International Conference on Machine Learning, pp. 557–565 (2014)
29. Fier, F., Augsten, N., Bouros, P., Leser, U., Freytag, J.C.: Set Similarity Joins on Mapreduce: An Experimental Survey. Proceedings of the VLDB Endowment **11**(10), 1110–1122 (2018)

30. Tran, T.T.Q.: Filters based fuzzy big joins. Ph.D. thesis (2020). Thèse de doctorat dirigée par D'Orazio, Laurent et Laurent, Anne Informatique Rennes 1 2020
31. Hu, X., Tao, Y., Yi, K.: Output-optimal Parallel Algorithms for Similarity Joins. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, pp. 79–90. ACM (2017). DOI 10.1145/3034786.3056110
32. Hu, X., Yi, K., Tao, Y.: Output-Optimal Massively Parallel Algorithms for Similarity Joins. ACM Transactions on Database Systems **44**(2), 1–36 (2019). DOI 10/gnr4r4
33. Aumüller, M., Ceccarello, M.: Implementing distributed similarity joins using locality sensitive hashing. p. 13. OpenProceedings.org (2022)
34. Cole, M.I.: Algorithmic skeletons: structured management of parallel computation. Pitman London (1989). DOI 10.5555/128874
35. Ciechanowicz, P., Poldner, M., Kuchen, H.: The Münster Skeleton Library Muesli: A comprehensive overview. Ercis working papers, University of Münster, European Research Center for Information Systems (ERCIS) (2009)
36. Ernstsson, A., Ahlqvist, J., Zouzoula, S., Kessler, C.: Skepu 3: Portable high-level programming of heterogeneous systems and hpc clusters. International Journal of Parallel Programming **49**(6), 846–866 (2021)
37. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z.: A library of constructive skeletons for sequential style of parallel programming. In: Proceedings of the 1st international conference on Scalable information systems, pp. 13–es (2006)
38. del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A generic parallel pattern interface for stream and data processing. Concurrency and Computation: Practice and Experience **29**(24), e4175 (2017)
39. Steuwer, M., Kegel, P., Gorlatch, S.: Skelcl-a portable skeleton library for high-level gpu programming. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pp. 1176–1182. IEEE (2011). DOI 10.1109/IPDPS.2011.269
40. Rieger, C., Wrede, F., Kuchen, H.: Musket: A domain-specific language for high-level parallel programming with algorithmic skeletons. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19, p. 1534–1543. ACM, New York, NY, USA (2019). DOI 10.1145/3297280.3297434
41. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.G.: Spar: A dsl for high-level and productive stream parallelism. Parallel Processing Letters **27**(01), 1740005 (2017). DOI 10.1142/S0129626417400059
42. Archibald, B.: Algorithmic skeletons for exact combinatorial search at scale. Ph.D. thesis, University of Glasgow (2018)
43. Aldinucci, M., Pezzi, G.P., Drocco, M., Spampinato, C., Torquati, M.: Parallel visual data restoration on multi-gpgpus using stencil-reduce pattern. The International Journal of High Performance Computing Applications **29**(4), 461–472 (2015)
44. De Matteis, T., Mencagli, G.: Parallel patterns for window-based stateful operators on data streams: An algorithmic skeleton approach. International Journal of Parallel Programming **45**(2), 382–401 (2017). DOI 10.1007/s10766-016-0413-x
45. Bellas, C., Gounaris, A.: Hyset: A hybrid framework for exact set similarity join using a gpu. Parallel Computing **104**, 102790 (2021)
46. Quirino, R.D., Ribeiro-Júnior, S., Ribeiro, L.A., Martins, W.S.: fgssjoin: A gpu-based algorithm for set similarity joins. In: ICEIS (1), pp. 152–161 (2017)
47. Yang, J., Zhang, W., Wang, X., Zhang, Y., Lin, X.: Distributed streaming set similarity join. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 565–576. IEEE (2020)
48. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, p. 2. USENIX Association, USA (2012)

Symposium on High-Level Parallel Programming & Applications
HLPP 2023

On the Implementation of a Lock-Free Atom Table in a Prolog System

Pedro Moreno · Miguel Areias · Ricardo Rocha · Vítor Santos Costa

Abstract Prolog systems rely on an atom table for symbol management, which is usually implemented as a dynamically resizeable hash table. This is ideal for single threaded execution, but can become a bottleneck in a multi-threaded scenario. In this work, we replace the original atom table implemen-tation in the Yet Another Prolog (YAP) system with a lock-free hash-based data structure, named Lock-free Hash Tries (LFHT), in order to provide effi-cient and scalable symbol management. Being lock-free, the new implementa-tion also provides better guarantees, namely, immunity to priority inversion, to deadlocks and to livelocks. Performance results show that the new lock-free LFHT implementation has better results in single threaded execution and much better scalability than the original lock based dynamically resizing hash table.

Keywords Prolog · Concurrency · Hash Tries · Lock-Freedom · Performance

1 Introduction

The initial programming languages were designed to abstract the computer hardware where, to achieve reasonable performance, a developer would have to learn first how to express the algorithmic problems in machine-oriented terms. Higher-level languages were created to allow developers to program algorithmic resolutions in terms closer to the problem's conceptualization. It is believed that higher-level languages are particularly helpful in developing

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020. Pedro Moreno is funded by the FCT grant SFRH/BD/143261/2019.

Pedro Moreno · Miguel Areias · Ricardo Rocha · Vítor Santos Costa
CRACS/INESC TEC and Dept. of Computer Science, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
E-mail: {pmoreno, miguel-areias, ricroc, vsc}@dcc.fc.up.pt

succinct and correct programs that are easy to write and also easy to understand. Logic programming languages, together with functional programming languages, form a major class of such languages, called *declarative languages*, and because logic programming languages are based on the predicate calculus, they have a strong mathematical basis.

Prolog is the most popular and powerful logic programming language. Prolog gained its popularity mostly because of the success of the sophisticated compilation technique and abstract machine known as the Warren's Abstract Machine (WAM) presented by David H.D. Warren in 1983 [24]. Nowadays, it is widely used in multiple domains, such as, machine learning [16], program analysis [7], natural language analysis [15], bioinformatics [14] and semantic web [8]. Prolog systems represent data as terms, that can be number, strings, or atoms, or a composition of terms. Prolog atoms are particularly important, as they are both used as symbols and as a convenient representation of strings. In this work, we focus on the *Atom Table* used for atom management and we investigate whether the traditional design can still be a good solution for recent challenges Prolog systems face.

One such challenge is to take best advantage of multi-core/multi-threaded architectures, arguably one of the most popular and impactful recent hardware developments. This type of architectures allow greater performance, but resources must be properly managed and exploited. Many languages and systems were not originally designed for multi-processing, which required them to be later extended to support this type of architectures, and Prolog systems were no exception.

Multi-threading in Prolog is the ability to perform concurrent computations, in which each thread runs independently but shares the program clauses [13]. Almost all Prolog systems support some sort of multi-threading. In particular, the Yet Another Prolog (YAP) multi-threading library [18] can be seen as a high-level interface to the POSIX threads library, where each thread runs on a separate data area but shares access to the global data structures (code area, atom table and predicate table). As each thread operates its own execution stack, it is natural to associate each thread with an independent computation that can run in parallel as threads already include all the machinery to support shared access and updates to the global data structures and input/output structures.

In this paper, we replace the original atom table implementation in the YAP system with a lock-free hash-based data structure, named Lock-free Hash Tries (LFHT), in order to investigate whether an efficient and scalable symbol management, can make a difference in a multi-threaded environment. Performance results show that the new implementation shows better results both in single threaded execution and much better scalability than the original atom table.

The remainder of the paper is organized as follows. First, we introduce relevant background and present the main ideas of our design. Next, we describe in detail the points required to easily reproduce our implementation. Then, we present a set of experiments comparing the new atom table against the original one. At the end, we present conclusions and draw further work directions.

2 Background

In this section, we describe the context of our work with particular focus on the YAP system, the concurrent access to the atom table, and the LFHT design.

2.1 The YAP System

Yet Another Prolog (YAP) is a Prolog system originally developed in the mid-eighties and that has been under almost constant development since then [19].

Figure 1 presents a high-level picture of the YAP system. The system is written in C and Prolog. Interaction with the system always starts through the top-level Prolog library. Eventually, the top-level refers to the core C libraries. The main functionality of the core C libraries includes starting the Prolog engine, calling the Prolog clause compiler, and maintaining the Prolog internal database. The engine may also call the just-in-time indexer (JITI) [20]. Both the compiler and the JITI rely on an assembler to generate code that is stored in the internal database. The C-core libraries further include the parser and several built-ins (not shown in Fig. 1). An SWI-Prolog compatible threads library [25] provides support to thread creation and termination, and access to locking.

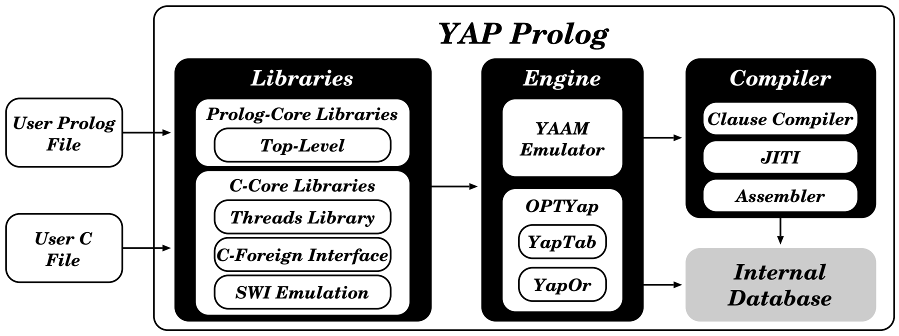


Fig. 1 The YAP system

YAP includes two main components, the *Engine* and the *Database*. The Engine maintains the abstract machine internal state, such as abstract registers, stack pointers, and active exceptions. The Database maintains the root pointers to the internal database, which includes the *Atom Table* and the *Predicate Table*. In order to support multi-threading, YAP's data structures are organized as follows:

– the `GLOBAL` structure is available to all threads and references the global data structures; locks should protect access to these data structures.
– the `LOCAL` structure is a per-thread array referencing the thread's local data structures, e.g., the engine abstract registers, internal exceptions, and thread specific predicates. The data is accessible through the thread's `LOCAL` structure, whose address is available from thread-local storage.

Figure 2 presents in more detail YAP's internal data structures with particular emphasis on the atom table. It assumes support for two threads, hence it requires two `LOCAL` structures, each containing a copy of the corresponding WAM registers.

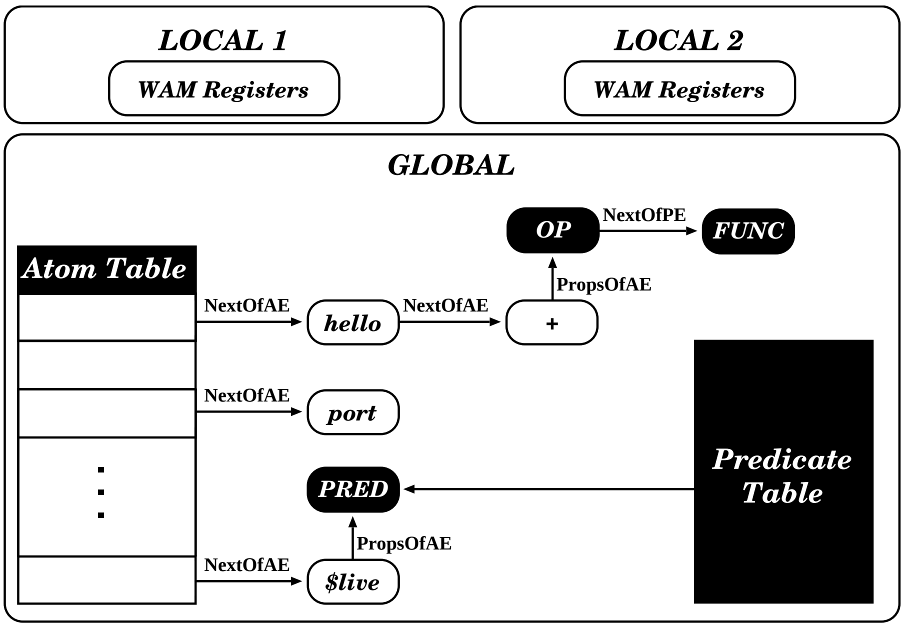


Fig. 2 YAP's internal data structures

The main structure inside `GLOBAL` is the *Atom Table*, which contains objects of the abstract type `Atom`. As discussed above, atoms are used to represent symbols and text. The latter usage stems because the same text can appear in different parts of a program. Storing text as atoms can save both space and time, once to compare two segments one just has to compare atoms, e.g., two text segments match if and only if they are the same atom, that is, if they have the same entry in the atom table. At the implementation level, the atoms are stored in a linked-list and each node within that linked-list has a reference to a secondary linked-list, that holds the properties of the atoms. Predicates with atoms as name are also stored in the atom table. Predicates are also often present in a Prolog program and there might exist several predicates with the same name (but with a different arity or belonging to different modules), and in such situations, there is a direct hash-table for them.

The abstract type `Atom` has a single concrete type, `AtomEntry`. Thus, the atom table is implemented as a single-level bucket array hash table with a separate chaining mechanism, implemented as linked lists, to support collisions among `AtomEntry` objects. Once the bucket array data structure is saturated, the hash table duplicates its size, and the `AtomEntry` objects are placed in the newly created data structure. Each `AtomEntry` contains

1. `StrOfAE`: a `C` representation of the atom's string;
2. `NextOfAE`: a pointer to the next atom in the linked list for this hash entry;
3. `PropsOfAE`: a pointer to a linked list of atom properties;
4. `ARWLock`: a reader-writer lock that serializes access to the atom.

The `Prop` type abstracts objects that we refer to by the atom's name. Example subtypes of `Prop` include functors, modules, operators, global variables, blackboard entries, and predicates. All of them are available by looking up an atom and following the linked list of `Prop` objects.

Figure 2 shows an atom table with four atoms: `hello`, `+`, `port`, and `$live`. Notice that only `+` and `$live` have associated properties. In practice, most atoms do not have properties. Every concrete type of `Prop` implements two fields:

1. `KindOfPE` gives the type of property;
2. `NextOfPE` allows organizing properties for the same atom as a linked list.

Each property extends the abstract property in its own way. As an example, *functors* add three extra fields: a back pointer to the atom, the functor's arity, and a list of predicates that share the same name and arity, but belong to different modules.

This design is based on LISP implementations, and has been remarkably stable throughout the history of the system. Main optimizations and extensions include:

1. Older versions of YAP support two atom tables: one groups all ISO-Latin-1 atoms, where each character code c is such that $0 < c < 255$, and the other stores atoms that need to be represented as wide strings. Recent versions of YAP use UTF-8 internally.
2. As discussed above, functors have their own `Prop` objects, namely, predicates and internal database keys with that functor. This was implemented to improve performance of meta-calls.
3. The case where we have predicates with the same functor but belonging to different modules is addressed by a *predicate hash-table*, which allows direct access to a predicate from a functor-module key. A typical example is the TildeCRF machine learning algorithm, implemented in YAP by Guttman and Kersting [10]. TildeCRF uses learning from interpretations, where each example is a small program containing different combinations of the same concepts, such that, each example is associated to a module and each concept to a predicate.

In Fig. 2, the atom + has two properties: one of the type `op` and another of type `functor`. The atom `$live` has a property of type `predicate`.

2.2 Lock-Free Hash Tries

YAP's atom table uses single-level hash buckets that doubles size once they are saturated. Concurrent accesses to the atom table are serialized by the use of reader-writer locks.

Lock-freedom is an alternative to lock based data structures that allows individual threads to starve but guarantees system-wide throughput. Lock-free data structures offer several advantages over their lock-based counterparts, such as, being immune to deadlocks, lock convoying and priority inversion, and being preemption tolerant, which ensures similar performance regardless of the thread scheduling policy. Lock-freedom takes advantage of the *Compare-And-Swap (CAS)* atomic primitive that nowadays is widely found on many common architectures. CAS reduces the granularity of the synchronization when threads access concurrent areas, but still suffers from contention points where synchronized operations are done on the same memory locations, leading to problems such as false sharing or cache memory ping pong effects.

Hash tries [6] minimize these problems by dispersing the concurrent areas as much as possible. Hash tries (or hash array mapped tries) are a trie-based data structure with nearly ideal characteristics for the implementation of hash tables. An essential property of the trie data structure is that common prefixes are stored only once [9], which in the context of hash tables allows us to efficiently solve the problems of setting the size of the initial hash table and of dynamically resizing it in order to deal with hash collisions. Several approaches exist in the literature for the implementation of lock-free hash tables, such as Shalev and Shavit split-ordered lists [21], Triplett *et al.* relativistic hash tables [23] or Prokopec *et al.* CTries [17].

The Lock-Free Hash Tries (LFHT) design, as originally proposed by Areias and Rocha [1,2], is a tree based data structure implementing two types of nodes: *hash nodes*, used to represent the hierarchy of hash levels where keys are indexed; and *leaf nodes*, used to store the key-value pairs. Figure 3 shows the general architecture of the LFHT design.

Each key is used to compute a hash h, which is then used to map the corresponding key-value pair in the LFHT hierarchy. For that, it uses chunks of w bits from h to index the entry in the appropriate hash level, i.e., for each hash level H_i, it uses the i^{th} group of w bits of h to index the entry in the appropriate bucket array of H_i. All bucket entries in a hash node are initialized with a reference to the hash node itself. During execution, each bucket entry stores either a reference to a hash node (itself or a deeper hash node) or a reference to a separate chaining mechanism of leaf nodes, that deals with the hash collisions for that entry. Intermediate leaf nodes hold a reference to the next-on-chain leaf node.

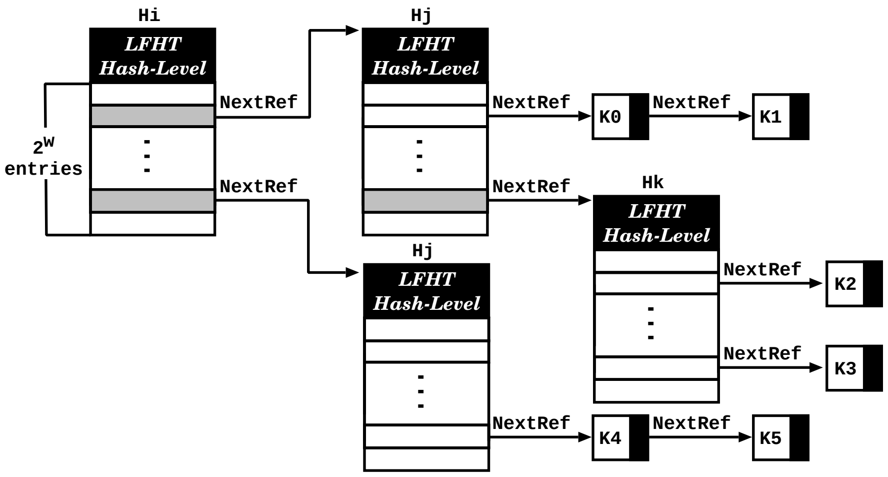


Fig. 3 General architecture of the LFHT design

To find the value associated with a given key, it begins by computing the corresponding hash value. Then, nodes are searched in the LFHT structure by following the path given by the hash value. If the key exists, it will be found in a leaf node and the corresponding associated value is returned.

The original LFHT design was implemented in `C` and proposed in the context of YAP's concurrent tabling engine [1]. In a nutshell, tabling is a refinement of Prolog's standard resolution that stems from one simple idea: save intermediate answers for current computations, in a specific data area called the *table space*, so that they can be reused when a *similar computation* appears during the resolution process. This means that in a traditional tabling environment, only concurrent search and insert operations are executed. The authors of LFHT took advantage of this fact to create a table space design that would be as efficient as possible in these two operations. Since no remove operations were executed concurrently, no emphasis was given to memory reclamation. All memory used to represent the table space would remain valid during the execution of a concurrent tabled logic program. Only at the end, when running in single-threaded mode, could memory resources be released to the operating system.

As LFHT obtained interesting results, the authors consider the possibility of extending it to support the remove operation in order to make LFHT available as a standalone data structure. However, supporting removals implied that the design would have to support some sort of memory reclamation or garbage collection mechanism or, alternatively, to be implemented on top of a framework that would do that by default. The authors decided to exploit the advantages of the *Java Virtual Machine (JVM)* and re-implemented the design from scratch in `Java`, adding the support for the remove operation [3].

To maintain LFHT's lock-freedom property, the remove operation was implemented in three stages. On the first stage, the memory is logically removed, i.e., the block of memory *m* being removed is marked with some sort of tag in such a way that all other threads know that the information in *m* is no longer valid. On the second stage, all the memory references to *m* stored in other structures are deleted, meaning that, from a given instant of time, threads entering the LFHT data structure no longer see or reach the memory *m*. Finally, on the third stage, the memory is physically released, i.e., the memory *m* can be reused in other context or freed to the host operating system. In this three stages scenario, JVM's garbage collector is very useful as it already implements the third stage by default, leaving the focus on the implementation of the first and second stages. In 2021, the LFHT design was dully formalized to prove its correctness, in particular the expand operation that handles with key collisions [4], and more recently, the design evolved as a standalone Java application with new features and operations, such as, the compress operation that is able to free unused hash levels [5].

At the same time, and starting from the ideas in the Java implementation with the remove operation, the LFHT implementation in C was adapted and extended to support a memory reclamation scheme that could fully support the three stages described above without losing the lock-freedom property [12], meaning that the design could finally meet the goal of being used as a standalone data structure and application. Experimental results showed that such a design is very competitive and scalable, when compared against the *Concurrent Hash-Map* implementation used in the Intel's *Thread Building Blocks (TBB)* library. More recently, the LFHT design was improved even further with a compression based design that would improve throughput [11].

3 Our Proposal

This section describes our proposal to improve the performance of YAP's atom table in concurrent environments. For that, we replaced the original version of the atom table, based in single level hashing, by the LFHT design in such a way that, instead of having a specialized version of a concurrent hash table implementing the atom table, we can simply use the general purpose LFHT design and allow it to manage everything, which goes from managing the concurrent accesses, to indexing the atoms for a faster access and handling atom collisions through a highly efficient chaining mechanisms. Moreover, to free memory from the atom table, we also take advantage of LFHT's memory reclamation mechanism, which will automatically handle the physical removal of atoms and corresponding internal data structures.

In what follows, we show in more detail how the LFHT data structure was integrated into the YAP system. To make the integration as smooth as possible, we need to understand all the details regarding YAP's internal database and how it is accessible from all internal and external libraries and data structures. Figure 4 presents the new organization of YAP's internal data structures based in the LFHT design (for comparison with Fig. 2, we left in gray the parts that were not changed from the original design). For the sake of presentation, the

LFHT hash levels shown at the left of the figure are presented in a compact way as a single level, representing the initial configuration, which will be expanded during executing to multiples levels as described in the previous section.

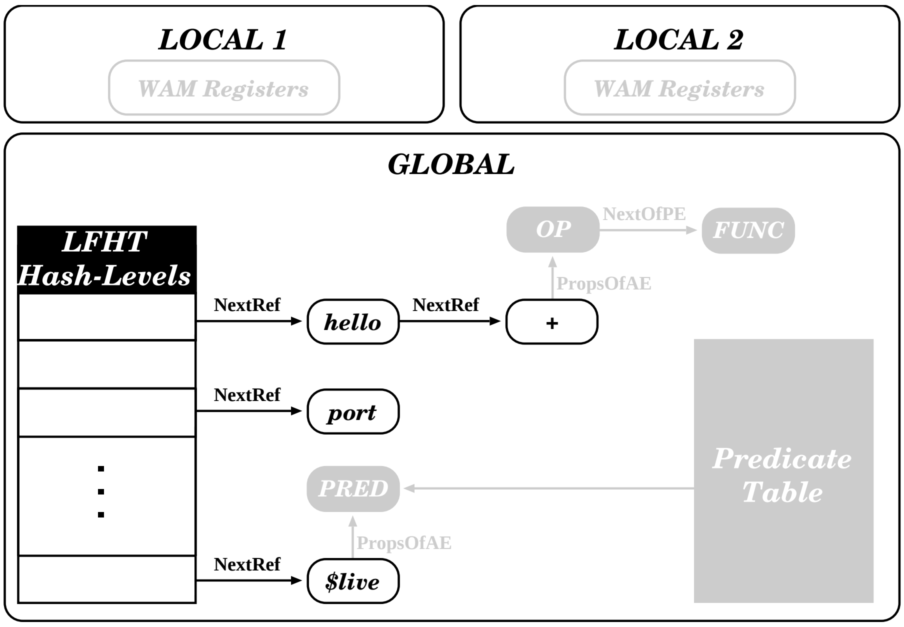


Fig. 4 The new organization of YAP's internal data structures

When comparing the new organization in Fig. 4 with the previous one in Fig. 2, one can observe two main modifications. The original `NextOfAE` field was removed, since the chaining mechanism will be managed by LFHT's design, and the read-writer lock `ARWLock`, used to serialize the access to the atoms in the original version of the atom table, was also removed, since now the LFHT design only uses CAS operations.

Using CAS operations instead of read-writer locks has some advantages. It can potentially reduce significantly the number of write operations done in memory during the execution of a program. At the implementation level, a read-write lock, requires writing operations even in when threads are only reading information from a protected memory region. This happens because read-write locks need to keep track of the number of threads that are in a protected memory region and, to do so, they use standard atomic counters. Moreover, these writing operations require also memory barriers to ensure the consistency of memory operations. These memory barriers have a considerable cost in the performance of a system, since they apply an ordering constraint between all memory operations that occur before and after the memory barrier, affecting this way all running threads.

Note that LFHT does not completely avoid memory barriers, as the CAS operation also uses them when executing a write operation. The gain comes from the fact that the design is lock-free, which means that reading operations do not require any write operations.

The remaining data structures and references are unchanged. This is the case of the `PropsOfAE` pointer to the atom's *properties* and the `StrOfAE` representation of the atom's string, therefore allowing the other YAP's data structures, such as the Predicate Table, to still access the atoms' information as they do in the original design.

In order to fully replace YAP's atom table with LFHT's design, some additional extensions were required to ensure full compatibility with the original design. These extensions include: (i) support for arbitrary keys and full-hashing collisions; and (ii) an iteration mechanism capable of traversing all keys stored in the atom table in a given instant of time. In the following subsections, we discuss how these extensions were implemented.

3.1 Arbitrary Keys

By default, the LFHT implementation assumes that the hash function is good enough to avoid key collisions, meaning that it relies only on the generated hash value to find a key, thus not considering the case of two keys generating the same hash value. To also consider this situation, when searching for a key K, we still use the hash value h to move through the hash levels but, when a node N corresponding to h is found, we need to confirm that N holds K. And, if this is not the case, we keep searching for the next node corresponding to h that may hold K.

YAP's atom table uses strings as keys, and although we could add support for strings to LFHT's design, we decided to implement a more general solution independently of the type of the key. During LFHT's initialization, now we must give the following parameters: (i) a key comparison function; (ii) a hash function; and (iii) a key destructor function. The key comparison function should implement the comparison of keys to be used in the hash value searching mechanism. The hash function allows to simplify the API, since now we only need the key as argument to the LFHT operations instead of both the key and the hash value. The key destructor functions allows to free memory used by the key when we remove a node. We also allow for any of these parameters to be undefined, and in such case we disable the associated feature. For example, if no hash function is defined, we assume that the given key is the hash itself, if no key comparison function is passed, we assume that the user knows that hash values will not collide, and if no key destructor is passed, we assume that the key will never be deleted during the execution. Figure 5 shows the new `C` language high-level API of the LFHT data structure.

```
// Initializes the data structure and returns a handler
struct lfht_head *init_lfht(size_t (*hash_func)(void *),
       int (*key_cmp)(void *, void *), void (*key_free)(void *));

// Returns the value associated with the key if it exists
void *lfht_search(struct lfht_head *head, void *key);

// Returns the value associated with the key if it exists,
// otherwise inserts the key with the provided value
void *lfht_insert(struct lfht_head *head, void *key, void *value);

// Removes the key and returns the associated value
void *lfht_remove(struct lfht_head *head, void *key);

// Returns the next key in hash/key order
void *lfht_next_key(struct lfht_head *head, void *key);
```

Fig. 5 C language high-level API of the LFHT data structure

3.2 The Iteration Procedure

During the execution of a program, a Prolog system might be required to iterate over all atoms present in the atom table. YAP is no exception, thus LFHT data structure was extended to support this additional operation. In a nutshell, the iterator of LFHT data structure presents atoms by the natural order that their hash value appears in the data structure for collision free atoms, otherwise, the LFHT data structure consumes the atoms by the natural order of their keys.

At the implementation level, the iterator begins by presenting the atom with the lowest hash value. And then, to present the next atom it uses the previously presented atom, and the process continues until there are no more atoms to be presented. If there are atoms with the same hash value, it presents the next smallest key with the same hash value. Otherwise, returns the smallest key of the next available smallest hash. By iterating this way, it ensures that iteration is done over all keys that were present when the iteration began and that were not removed during the iteration process. Keys that are inserted concurrently during an iteration might not be presented, this will happen if the iterator is iterating over a hash value which is higher than the hash value of the key that was inserted.

Algorithm 1 shows how the iteration process is done over the hash nodes, in order to find the next key. Note that we use the hash value from the most significant bits to the least significant bits from the first level to the last level, so that we can have the property that nodes in a bucket $B[i]$ always have smaller hash values than nodes in a bucket $B[k]$ in the same hash node (for $i < k$). To find the first key we pass the *Null* key to the *Iterator* function which lets us start at the bucket entry corresponding to the hash with value 0. Otherwise, we compute the hash value from the key and start iterating from the corresponding bucket. We begin in the root hash node and, if in the corresponding bucket we find a new hash node, we try to recursively find a next key in such hash node. If the bucket contains leaf nodes we call the *IterateChain()* function described in Algorithm 2 in order to find a next key in the chain. In both situations, if we find such a key we return it, otherwise we continue searching in the next bucket. If we reach the end of the hash node without finding a key, we return *Null* in order to indicate no key was found.

Algorithm 1 *Iterate*(*Key K, Node Hn*)

```
1: if $K = Null$ then
2:     $H \Leftarrow 0$
3: else
4:     $H \Leftarrow Hash(K)$
5: for $i \Leftarrow Index(Hn, H)$ to $Hn.size$ do
6:     if $Hn.array[i].type = HASHNODE$ and $Hn.array[i] \neq Hn$ then
7:         $R \Leftarrow Iterate(K, Hn.array[i])$
8:     else if $Hn.array[i].type = LEAFNODE$ then
9:         $R \Leftarrow IterateChain(K, H, Hn.array[i])$
10:    if $R \neq Null$ then
11:        return $R$
12: return $Null$
```

Algorithm 2 shows how we find the next node in a chain. We need to iterate over the whole chain as the nodes are unordered in the chain. We start by filtering the nodes that are actually ordered after the key provided, then we start by assigning the 1st node to *N* and replace it if we find a node that is ordered before it[1].

Algorithm 2 *IterateChain*(*Key K, Hash H, Node Ln*)

```
1: $N \Leftarrow Null$
2: while $Ln.type = LEAFNODE$ do
3:     if $Ln.hash > H$ or ($Ln.hash = H$ and ($K = Null$ or $Ln.key > K$)) then
4:         if $N = Null$ or $Ln.hash < N.hash$ or ($Ln.hash = N.hash$ and
           $Ln.key < N.key$) then
5:             $N \Leftarrow Ln$
6:     $Ln \Leftarrow Ln.next$
7: return $N$
```

4 Experimental Results

In order to evaluate the impact of our proposal, we next show experimental results comparing the original and new versions of YAP's atom table. To put

[1] Note that, for the sake of simplicity, we are omitting how the iterator proceeds when a concurrent expansion of hash nodes occurs.

the results in perspective, we also compare both YAP's implementations with SWI-Prolog, a well-known and popular Prolog system that also implements concurrent support for the atom table in a lock-free fashion [26]. SWI-Prolog uses a single-level hash design to implement the atom table with lock-free operations, except for the resizing of the hash table, which is not lock-free because it uses a standard read-writer locking scheme. This happens because while the resize is in progress, the next pointers linking atoms in the same bucket are generally incorrect, and dealing with this incorrectness is not a trivial task, which is solved with a standard read-writer lock.

The hardware used was a machine with 4 AMD Opteron(TM) Processor 8425 HE with 6 cores each, 64KiB of L1 cache per core, 512KiB of L2 cache per core and 5MiB of usable shared L3 cache per CPU. It had a total of 128GiB of DDR3 memory. The machine was running the Ubuntu 22.04 operating system with Linux kernel version 5.15.0-69.

4.1 Benchmark

We describe next the benchmark used to evaluate the performance of our implementation. In a nutshell, the benchmark will generate a huge stress over the Prolog's atom table, by inserting an enormous amount of atoms in a multi-threaded fashion. Although it is an artificial benchmark, it is designed to expose all the potential bottlenecks in the atom table, allowing a deeper study about using the LFHT design in YAP. Next, we show the pipeline of predicates used in the benchmark.

```
% compile the generation sequences
:- compile('seq.pl').

% top query call
benchmark(WO, T):-
   atom_dataset(DS),
   % mark the inital time
   statistics(walltime,[InitTime,_]),
   % create and join threads
   findall(Id, (between(1, T,_),
                thread_create(worker(DS, WO),Id)), Ids),
   forall(member(I,Ids), thread_join(I,_)),
   % mark the final time
   statistics(walltime,[EndTime,_]),
   Time is EndTime - InitTime,
   % show the execution time
   write('Time: '), write(Time).
```

Fig. 6 Initial setup and top query call

We begin with Fig. 6 showing the Prolog code for the initial setup of the benchmark and the *benchmark/2* predicate, which is the top predicate to be called. We start by compiling an initial set (file `seq.pl`) of 240, 000 different sequences that will be used as base sequences to generate a combination of multiple atoms to be inserted in the atom table. The *benchmark/2* predicate is then used to mark the initial and final times, create and join threads, and to show the execution time. It receives two arguments, the worker offset *WO*, used to batch a set of sequences from the initial set that will be used to create the combination of atoms, and the total number of threads *T* to be executed. For this benchmark, we used a batch of 2, 000 sequences of work to be done.

```
% setup scheduler
:- dynamic qsize/1.
:- mutex_create(qlock).
qsize(0).

% manage the working queue
worker(DS, WO) :-
   mutex_lock(qlock),
   % get work from queue
   qsize(I),
   (I =< DS -> % thread got work W
      % setup next work
      retract(qsize(I)),
      IL is I + WO, assert(qsize(IL)),
      mutex_unlock(qlock),
      % compute work W
      compute(I, IL),
      % get more work
      worker(DS, WO)
   ; % no more work to be done
      mutex_unlock(qlock)).
```

Fig. 7 The naive parallel scheduler

The second stage of the pipeline is the scheduler. Figure 7 shows the code that implements the naive parallel scheduler used in the benchmark. It uses a dynamic predicate *qsize/1* to mark the number of the next sequence from the initial set that is available to be used for the generation of atoms and a standard lock named *qlock* to synchronize threads when they are getting work. To get work, a thread *T* begins by gaining access to the lock, then it reads the next sequence *I* stored in *qsize/1* and, if there is work to be done, *T* prepares the queue with the next available sequence *IL*, releases the lock and goes to executing work. Otherwise, there is no more work to be done, thus *T* keeps *qsize/1* in the same state, releases the lock, and proceeds to the thread join predicate.

The third and final stage of the pipeline implements the process of generating atoms to be inserted and stored in the atom table. Figure 8 shows both *compute/2* and *combine_atoms/2* predicates. For each batch of work, a thread uses the *compute/2* predicate to get the corresponding sequences from the

```
% compute the sequences
compute(I, I) :- !.
compute(I, IL) :-
   atom_seq(I, AS),
   (combine_atoms(AS, _), fail; true),
   I1 is I + 1,
   compute(I1, IL).

% generation of atoms
combine_atoms(AS, R) :-
   atom_concat(A1, A2, AS),
   atom_concat(A2, A1, R).
```

Fig. 8 Generation of the atoms to be inserted in the atom table

initial set, and, for each sequence, it calls the *combine_atoms/2* predicate to generate all possible combination of atoms from the sequence. Each generated atom is then automatically inserted by the Prolog system in the atom table.

4.2 Results

The results shown in the following figures were obtained by taking the mean of 10 benchmark runs. Figure 9 shows the speedup obtained by YAP with the atom table replaced by the LFHT data structure against YAP's original implementation for every combination of 1 to 24 threads. The results show that, on average, we can achieve a minimum speedup of 1.8 with a single thread and a maximum speedup around 3.4 with 23 threads. The speedup for 24 threads is slightly worse than for 23 threads because, as the LFHT version has better CPU utilization, it is more affected by background/operation system processes when all cores are in use.

These results show that we can achieve not only better overall performance, but also much better scalability. In particular, the readers-writer locks present in the original atom table can be a significant bottleneck that the LFHT data structure is able to avoid.

To put the results in perspective, we also compared the YAP results with SWI-Prolog. Figure 10 shows the throughput of sequences that are computed per second in both the YAP (original and LFHT-based atom tables) and SWI-Prolog implementations. As one can observe, the original YAP implementation already provides much better performance and scalability than SWI-Prolog, and the LFHT-based atom table is able to provide a considerable improvement on top of it. For example, with 24 threads, our LFHT-based implementation is able to achieve 24.6 times the throughput of SWI-Prolog.

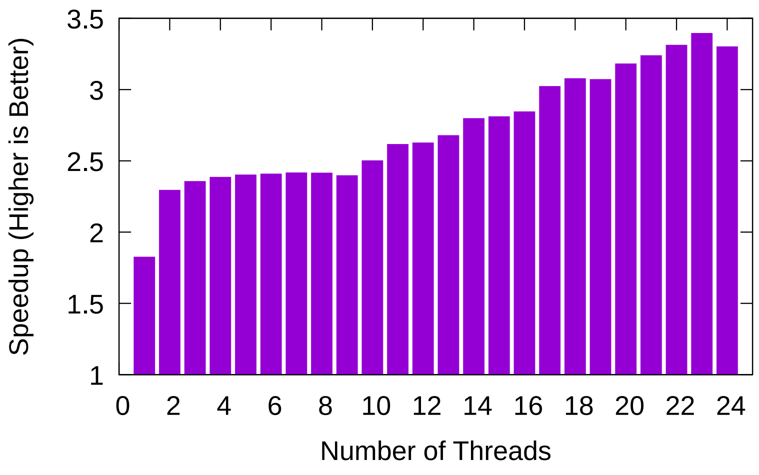


Fig. 9 Speedup of YAP's LFHT version against YAP's original implementation

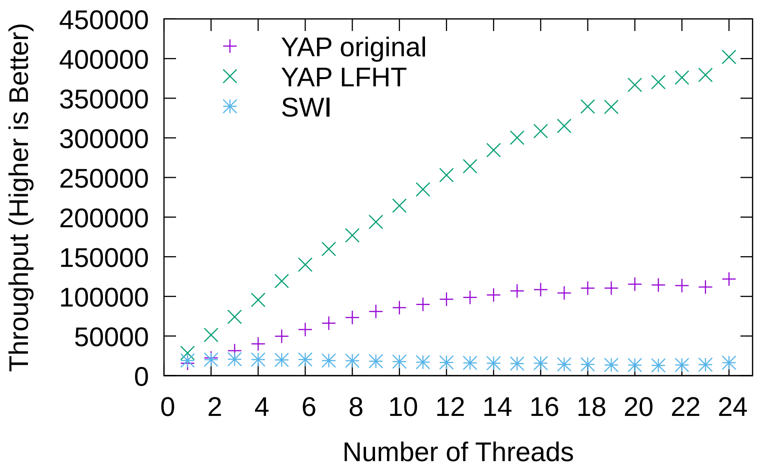


Fig. 10 Throughput for YAP and SWI-Prolog

5 Conclusions and Future Work

We have presented an approach to replace the original atom table implementation in the YAP system with a lock-free hash-based data structure, named LFHT. Our main motivation was to refine the previous atom table design in order to be as effective as possible in the concurrent search and insert operations over the atom table. We discussed the relevant details of the approach and described the main algorithms. We based our discussion on YAP's concurrent atom table data structure, but our approach can be applied to other

Prolog systems or to other generic systems that need to use similar concurrent atom tables.

A key design decision in our approach was to adapt the LFHT design to work as a fully standalone `C` application, allowing the hash function to be defined by the user, and implementing a new iterate operator. This facilitated the migration from the old lock-based atom table to the new lock-free atom table, where threads do not block when accessing the data structure. Experimental results showed that our approach can effectively reduce the execution time and scale better than the previous design.

As future work, we plan to test our approach on real world Prolog applications widely-used in the community, such as, the Aleph Machine Learning system [22] and the ClioPatria Semantic Web system[2].

References

1. Areias, M., Rocha, R.: On the Correctness and Efficiency of Lock-Free Expandable Tries for Tabled Logic Programs. In: International Symposium on Practical Aspects of Declarative Languages, no. 8324 in LNCS, pp. 168–183. Springer (2014)
2. Areias, M., Rocha, R.: A lock-free hash trie design for concurrent tabled logic programs. International Journal of Parallel Programming **44**(3), 386–406 (2016)
3. Areias, M., Rocha, R.: Towards a Lock-Free, Fixed Size and Persistent Hash Map Design. In: M. Valero, A. Melo (eds.) Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2017), pp. 145–152. IEEE Computer Society, Campinas, Brazil (2017)
4. Areias, M., Rocha, R.: On the Correctness and Efficiency of a Novel Lock-Free Hash Trie Map Design. Journal of Parallel and Distributed Computing **150**, 184–195 (2021). DOI https://doi.org/10.1016/j.jpdc.2021.01.001
5. Areias, M., Rocha, R.: On the Correctness of a Lock-Free Compression-based Elastic Mechanism for a Hash Trie Design. Computing (2022). DOI https://doi.org/10.1007/s00607-022-01085-2
6. Bagwell, P.: Ideal Hash Trees. Es Grands Champs **1195** (2001)
7. Benton, W.C., Fischer, C.N.: Interactive, scalable, declarative program analysis: from prototype to implementation. In: M. Leuschel, A. Podelski (eds.) Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 14-16, 2007, Wroclaw, Poland, pp. 13–24. ACM (2007)
8. Devitt, S., Roo, J.D., Chen, H.: Desirable features of rule based systems for medical knowledge. In: W3C Workshop on Rule Languages for Interoperability, 27-28 April 2005, Washington, DC, USA. W3C (2005)
9. Fredkin, E.: Trie Memory. Communications of the ACM **3**, 490–499 (1962)
10. Gutmann, B., Kersting, K.: Tildecrf: Conditional random fields for logical sequences. In: J. Fürnkranz, T. Scheffer, M. Spiliopoulou (eds.) Machine Learning: ECML 2006, pp. 174–185. Springer Berlin Heidelberg (2006)
11. Moreno, P., Areias, M., Rocha, R.: A Compression-Based Design for Higher Throughput in a Lock-Free Hash Map. In: M. Malawski, K. Rzadca (eds.) Proceedings of the 26th International European Conference on Parallel and Distributed Computing (Euro-Par 2020), LNCS, pp. 458–473. Springer International Publishing, Warsaw, Poland (2020)
12. Moreno, P., Areias, M., Rocha, R.: On the Implementation of Memory Reclamation Methods in a Lock-Free Hash Trie Design. Journal of Parallel and Distributed Computing **155**, 1–13 (2021). DOI https://doi.org/10.1016/j.jpdc.2021.04.007
13. Moura, P.: ISO/IEC DTR 13211–5:2007 Prolog Multi-threading Predicates (2008). URL `http://logtalk.org/plstd/threads.pdf`

[2] `http://cliopatria.swi-prolog.org`

14. Mungall, C.: Experiences using logic programming in bioinformatics. In: P.M. Hill, D.S. Warren (eds.) Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5649, pp. 1–21. Springer (2009)
15. Nugues, P.M.: An Introduction to Language Processing with Perl and Prolog: An Outline of Theories, Implementation, and Application with Special Consideration of English, French, and German (Cognitive Technologies). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)
16. Page, D., Srinivasan, A.: Ilp: A short look back and a longer look forward. Journal of Machine Learning Research **4**, 415–430 (2003)
17. Prokopec, A., Bronson, N.G., Bagwell, P., Odersky, M.: Concurrent Tries with Efficient Non-Blocking Snapshots. In: ACM Symposium on Principles and Practice of Parallel Programming, pp. 151–160. ACM (2012)
18. Santos Costa, V.: On Supporting Parallelism in a Logic Programming System. In: Workshop on Declarative Aspects of Multicore Programming, pp. 77–91 (2008)
19. Santos Costa, V., Rocha, R., Damas, L.: The YAP Prolog System. Journal of Theory and Practice of Logic Programming **12**(1 & 2), 5–34 (2012)
20. Santos Costa, V., Sagonas, K., Lopes, R.: Demand-Driven Indexing of Prolog Clauses. In: V. Dahl, I. Niemelä (eds.) Proceedings of the 23rd International Conference on Logic Programming, *Lecture Notes in Computer Science*, vol. 4670, pp. 305–409. Springer (2007)
21. Shalev, O., Shavit, N.: Split-Ordered Lists: Lock-Free Extensible Hash Tables. Journal of the ACM **53**(3), 379–405 (2006)
22. Srinivasan, A.: The Aleph Manual (2004). URL `http://www.cs.ox.ac.uk/activities/machlearn/Aleph`
23. Triplett, J., McKenney, P.E., Walpole, J.: Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In: USENIX Annual Technical Conference, p. 11. USENIX Association (2011)
24. Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical Note 309, SRI International (1983)
25. Wielemaker, J.: Native Preemptive Threads in SWI-Prolog. In: International Conference on Logic Programming, no. 2916 in LNCS, pp. 331–345. Springer (2003)
26. Wielemaker, J., Harris, K.: Lock-free atom garbage collection for multithreaded prolog. Theory and Practice of Logic Programming **16**(5-6), 950–965 (2016)

Symposium on High-Level Parallel Programming & Applications
HLPP 2023

Optimizing Three-Dimensional Stencil-Operations on Heterogeneous Computing Environments

Nina Herrmann · Justus Dieckmann · Herbert Kuchen

Abstract Complex algorithms and enormous data sets require parallel execution of programs to attain results in a reasonable amount of time. Both aspects are combined in the domain of three-dimensional stencil operations, for example, computational fluid dynamics. This work contributes to the research on high-level parallel programming by discussing the generalizable implementation of a three-dimensional stencil skeleton that works in heterogeneous computing environments. Two exemplary programs, a gas simulation with the Lattice Boltzmann method, and a mean blur, are executed in a multi-node multi-graphics processing units (GPUs) environment, proving the runtime improvements in heterogeneous computing environments.

Keywords Skeleton Programming, Three-Dimensional Stencil Operations

1 Introduction

The field of High Performance Computing (HPC) is growing as algorithms become more complex and more data needs to be processed. Evaluating massive datasets, therefore, requires writing efficient parallel programs. Most HPC environments have multiple nodes equipped with multiple central processing units (CPUs) and GPUs. Creating programs that combine multiple nodes and accelerators requires knowledge of low-level frameworks such as implementations of the Message Passing Interface (MPI) MPI, OpenMP OpenMP, and CUDA CUDA. Moreover, writing a parallel program is error-prone and tedious, e.g., out of memory errors and invalid memory accesses are troublesome to identify even for skilled programmers. Additionally, choosing memory

Nina Herrmann, Justus Dieckmann, Herbert Kuchen
Practical Computer Science Group, University of Münster, Leonardo-Campus 3, 48149, Münster, Germany
Tel.: +49 251 83-38216
Fax: +49 251 83-38259
E-mail: [nina.herrmann|justus.dieckmann|kuchen]@uni-muenster.de

spaces, distributing data, and assigning tasks to threads are design decisions that significantly impact performance but require experience, which scientists usually lack.

Since experts in this field are hard to find, high-level frameworks are often used. Those frameworks commonly abstract from the distribution of data, provide portable code for different hardware architectures, are adjustable to distinct accelerators, and require less maintenance for the end user. In 1989, COLE introduced algorithmic skeletons enclosing reoccurring parallel and distributed computing patterns as one of the most common approaches to abstract from low-level details [4]. Multiple libraries [5,3], general frameworks [6, 2], and domain-specific languages (DSLs)[14] use the concept.

The paper contributes to the ongoing work by focusing on a particularly arduous operation, namely the three-dimensional stencil operation. Stencil operations calculate elements depending on neighboring elements within the data structure and therefore require communication between the computational units used. Those operations are essential for, e.g., computational fluid dynamics of gas or airflow. Efficiently updating data in a generalized way and dealing with three-dimensional data structures are currently not solved in high-level approaches.

This paper firstly lists the related work, focusing on high-level approaches abstracting from problem-specific details (Section 2). Section 3 outlines the library used (Muesli), while Section 4 explains the additional implementation of the three-dimensional skeleton and the examples used to measure the runtime. The work is evaluated in Section 5, discussing our runtime experiments on multiple hardware set-ups. Lastly, Section 6 summarizes our work.

2 Related Work

Ongoing work discussing three-dimensional stencil operations is twofold. On the one hand, generic high-level frameworks targeting the three-dimensional stencils have the advantage of parallelizing pre- and postprocessing steps, as they offer a variety of operations/skeletons. On the other hand, specialized frameworks already contain implementations for processing stencil calculations but are often inefficient as they focus on the algorithm and not on parallelization. Most related in the area of high-level skeleton programming frameworks, SkePU3 targets multi-node and multi-GPU environments for most skeletons in combination with StarPU. However, for stencil operations (MapOverlap), the data exchange between the programs is missing for multi-node programs [6].

FastFlow added GPU support but focuses on communication skeletons and misses a comparable stencil operation [2, **?**]. Lift handles n-dimensional stencils on single GPUs [8]. SkelCL implements a MapOverlap skeleton for multiple GPUs [13]. Specialized libraries such as Palabos for the Lattice Boltzmann methods (LBMs) [11], or publications discussing a single method, e.g., the

Helmholtz equation [7] focus on the algorithm and do not include accelerators like GPUs.

This work extends the mentioned work, as the presented stencil skeleton is generalizable for multiple applications, allows pre and postprocessing steps, and runs on multiple nodes and GPUs. This is proven by implementing a version of a LBM and a three-dimensional mean blur.

3 The *Mue*nster *S*keleton *Li*brary *Muesli*

Nowadays, most skeletons frameworks are implemented in C/C++ [6,2,3,12,1,9], as it offers interoperability with multiple parallel frameworks such as OpenMP, MPI, CUDA, and OpenCL and is exceptionally performant.

The used library is called Muenster Skeleton Library (Muesli) [5]. Muesli provides an object-oriented approach that offers one, two, and three-dimensional data structures (DA, DM, DC) with skeletons as member functions. The supported skeletons are, for example, multiple versions of Map and Zip (index and inplace variants), Fold, Gather, and, as discussed in this work MapStencil. Internally, MPI, OpenMP, and CUDA are used, which enables simultaneous parallelism on multiple nodes, CPUs, and GPUs. The library can be included with a simple include statement `#include<muesli.h>`. For writing a parallel program, Muesli provides abstract methods to state the number of processes and GPUs used. Apart from that, Muesli abstracts from parallel programming details by internally distributing the data structures on the available computational units, choosing the number of threads started on the corresponding low-level framework, and copying data to the correct memory spaces. This abstraction also reduces errors commonly made by inexperienced programmers, such as race conditions and inefficient data distribution.

```
1 class Sum : public Functor2<int, int, int>{
2   public: MSL_USERFUNC int operator() (int x, int y)
3   const {return x+y;}};
4 Sum sum;
5 auto product = [] (int i, int j) {return i*j;};
6 DA<int> a(3,2);                      // delivers: {2,2,2}
7 DA<int> b = a.mapIndex(sum);         // delivers: {2,3,4}
8 a.zipInPlace(b,product);             // delivers: {4,6,8}
9 int scalarproduct = a.fold(sum);     // delivers: 18
```

Listing 1: Scalar product in Muesli.

Listing 1 shows a simple program calculating the Scalar product of the distributed arrays a and b in Muesli. In line 6, a distributed array of size three with a default value of 2 is created. In the skeleton calls in lines 7-9, it can be seen that skeletons have a user function as an argument which can either be a C++ function or a C++ functor. For the `index` variant of `map`, Muesli applies the argument function of `map` to each element of the data structure (here a distributed array (DA)) and its index (line 7). For the `zip` skeleton, the second required data structure is passed as an argument. Lastly, lines 7+9 show that the same function can be used in different contexts, firstly for calculating the sum of the index and the value and secondly as a reduction operator.

4 Three-Dimensional Stencil Operations

Stencil operations are map operations that additionally require reading the surrounding elements of each considered element. Figure 1 displays a two-dimensional stencil with a radius of one. The peculiarity regarding stencil operations on multiple nodes and accelerators is that each execution of the stencil operation requires updating elements that are shared between computational units. As communication of updated elements requires synchronization between the computational nodes, it decreases the opportunity for executing tasks in parallel within the program. Muesli abstracts from all communication between the computational nodes with a MapStencil skeleton.

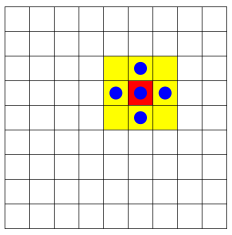


Fig. 1: Exemplary two-dimensional stencil operation.

4.1 Using the MapStencil Skeleton

The usage of the 3-dimensional `mapStencil` skeleton for the end-user is shown in Listing 2. Firstly, a function to be executed on each element is defined (l. 1-10). Merely functors of type `DCMapStencilFunctor` can be used with the `mapStencil` skeleton. Therefore, the first argument of the functor has to be of type `PLCube` (*PaddedLocalCube*), and the subsequent arguments must be integers for indexing the data structure. The class `PLCube` most importantly offers a `getter-function` taking three index arguments, relieving the end-user from index calculations (l. 8). The presented functor calculates the sum of all elements with a radius of two. It divides the sum by the number of total elements, therefore calculating a mean filter. This functor can be applied to a distributed cube by calling the mapStencil skeleton as a member function (l. 14). The skeleton takes the functor as a template argument and requires a distributed cube of the same dimension with the current data [1], the radius of the stencil[2] and the neutral value for border elements.

```
1  template <size_t radius>
2  MSL_USERFUNC float update(const PLCube<float> &plCube,
3  int x, int y, int z) {
4    float res = 0;
```

[1] A variant of the skeleton which immediately overwrites old values by new values is, however, possible and could be applied, for instance, for implementing the Gauß-Seidel method for solving systems of linear equations. However, it restricts the order in which computations can take place resulting in wave front parallelism.

[2] Other stencil shapes such as rectangular or irregular stencils can be handled by using the smallest surrounding cube, although this may introduce some overhead.

```
  for (int mx = x - radius; mx <= x + radius; mx++) {
    for (int my = y - radius; my <= y + radius; my++) {
      for (int mz = z - radius; mz <= z + radius; mz++) {
        res += plCube(mx, my, mz);}}}
  return res/(radius*radius*radius);
}
main () {
  ...
  int stencilradius = 2;
  dcp1->mapStencil<update<2>>(*dcp2, stencilradius, 0);
  ...
}
```

Listing 2: Exemplary functor for the `mapStencil` skeleton.

4.2 Implementation of the MapStencil Skeleton

Adding the MapStencil skeleton to the existing distributed cube (DC) class requires adding two additional member fields: a vector of `PLCubes` and a (maximal) supported stencil radius. As previously mentioned, the `PLCubes` class allows the end-user to abstract from the indexing of the data structure. To make access to the different memory spaces efficient, each computational unit has a separate `PLCube` storing merely the elements needed to calculate the assigned elements. This design choice makes the class flexible to be used for CPUs and for GPUs. It contains the following attributes to provide a light, minimal design:

– `int width, height, depth` - the three dimensions of the data structure,
– `int stencilRadius` radius of the stencil required to calculate the overlapping elements,
– `int neutralValue` used when the index is outside of the data structure,
– `T* data, T* topPadding, T* bottomPadding` CPU or GPU pointer for current data,
– four integers to save global indexes for the start and end of main and padding data areas.

Most importantly, the `getter-function` is implemented, taking three integers as arguments and returning the suitable value. This is either the neutral value or the corresponding element from the CPU or GPU memory.

Assuming GPUs are used as accelerators, the skeleton updates the current data structure in case the data is not up to date (Listing 3, l.6). Afterwards, it synchronizes the `PLCubes` inside one node, and the data between multiple nodes (l.7, l.9). Foreach GPU used, the `mapStencilKernelDC` kernel is called executing the functor on the appropriate part of the overall data structure. In any other case (multiple nodes and CPU), it is only necessary to synchronize the nodes (l. 21) and call the functor with the corresponding arguments (l. 29).

```cpp
template<typename T>
template<msl::DCMapStencilFunctor<T> f>
void msl::DC<T>::mapStencil(msl::DC<T> &result,
                    size_t stencilSize, T neutralValue) {
  #ifdef __CUDACC__
  this->updateDevice();
  syncPLCubes(stencilSize, neutralValue);
  msl::syncStreams();
  syncPLCubesMPI(stencilSize);
  for (int i = 0; i < this->ng; i++) {
    cudaSetDevice(i);
    dim3 dimBlock(muesli::threads_per_block);
    dim3 dimGrid((this->plans[i].size + dimBlock.x - 1)
        / dimBlock.x);
    detail::mapStencilKernelDC<T, f><<<dimGrid, dimBlock,
      0, muesli::streams[i]>>>(result.plans[i].d_Data,
   this->plCubes[i], result.plans[i].size);
  }
  msl::syncStreams();
  result.setCpuMemoryInSync(false);
  #else
  syncPLCubesMPI(stencilSize);
  #ifdef _OPENMP
  #pragma omp parallel for
  #endif
  for (int k = 0; k < this->nLocal; k++) {
    int l = (k + this->firstIndex) / (ncol*nrow);
    int j = ((k + this->firstIndex) - l*(ncol*nrow)) / ncol;
    int i = (k + this->firstIndex) % ncol;
    result.localPartition[k] = f(this->plCubes[0], i, j, l);
  }
  #endif
}
```

Listing 3: Implementation of the `mapStencil` skeleton.

4.3 Example Applications for Three-Dimensional Stencil Operations

Two examples were used to evaluate our implementation: an implementation of the Lattice Boltzmann Method (LBM) and a mean blur. The exemplary user function of the mean blur was already shown in Listing 2.

The LBM is used for fluid simulations e.g. the distribution of gas. It distinguishes between the collision and the streaming step, which alternate in continuous simulations [10, p. 61ff.]. In the streaming step, gas particles move from one cell to another. The fluid flow caused by the colliding particles is calculated in the collision step. The distribution function $f_i(x,t)$ calculates for a cell x and a timestamp t how many particles move in the next step to neighbor i. Index 0 corresponds to the cell itself. f_i^* defines the distribution after the collision of the particles (see formula (2)). Δt is the time period to be simulated.

$$f_i(x + c_i\Delta t, t + \Delta t) := f_i^*(x,t) \tag{1}$$

For the collision steps, the Bhatnagar-Gross-Krook-operator is used. τ is a constant defining the convergence of the simulation. Thus τ influences the viscosity of the gas.

$$f_i^*(x,t) := f_i(x,t) - \frac{\Delta t}{\tau}\left(f_i(x,t) - f_i^{\mathrm{eq}}(x,t)\right). \tag{2}$$

The equilibrium state is calculated by

$$f_i^{\mathrm{eq}}(x,t) := w_i \rho \left(1 + \frac{u \cdot c_i}{c_s^2} + \frac{u \cdot c_i}{2c_s^4} + \frac{u \cdot u}{2c_s^2}\right), \tag{3}$$

where w_i are the weights of the chosen grid and c_i is the position of the neighbor cells relative to the main cell. The constant number c_s is the sound velocity of the model. The mass density ρ and the puls density u are defined by

$$\rho(x,t) = \sum_i f_i(x,t), \qquad \rho u(x,t) := \sum_i c_i f_i(x,t). \tag{4}$$

For the implementation of the LBM, a D3Q19-Grid was used, D being the number of dimensions and Q the number of neighbors. Both steps (collision and streaming) are combined in one `mapStencil` call. Noteworthy, the implementation has to consider that single cells can be marked as blocked, simulating objects which are barriers to the flow of gas or as distributing constantly gas. Therefore, special cells are marked with `Not a Number` values (Listing 5 l. 5-7). To simulate this behavior without requiring additional storage, the handling of the floating point numbers is extended. According to the IEEE-754 Standard, each floating point number that has a maximal exponent with a mantissa that is not equal to zero is considered `Not a Number`. The most significant bit of the mantissa of f_0 is set so that the number is definitely understood as `NaN`. The remaining bits of the mantissa can then be used freely to store other data. In the code, bit masks and a struct with bit-fields are defined in the code to access this information as easily as possible (Listing 4).

```
const int FLAG_OBSTACLE = 1 << 0;
const int FLAG_KEEP_VELOCITY = 1 << 1;
typedef struct {
  unsigned int mantissa : 23;
  unsigned int exponent : 8;
  unsigned int sign : 1;
} floatparts ;
```

Listing 4: Handling of barriers and streaming cells.

The data stored for each cell is an `array<float, Q>`. Q is a constant number for the neighbor cells and the cell itself (l. 19). This type is abbreviated in the following listing with `cell_t`. Moreover, it is abstracted from the three-dimensional vector operations (l. 28, 29, 31, 34). The user function starts by transforming the current value of the cell into the single float parts (l. 4). In case it is a cell that distributes gas (`FLAG_KEEP_VELOCITY`), the cell remains without changes (l. 5-7). For all neighbor cells, the current amount of particles is read (l. 10-12). In the collision step, all cells which are obstacles reverse the airflow (l. 16-23). All other cells calculate the particles streaming from the next cells (l.27-34).

```
 1 MSL_USERFUNC cell_t update(const PLCube<cell_t> &plCube, int x,
 2                             int y, int z) {
 3   cell_t cell = plCube(x, y, z);
 4   auto* parts = (floatparts*) &cell[0];
 5   if (parts->exponent == 255 && parts->mantissa
 6       & FLAG_KEEP_VELOCITY) {
 7     return cell;
 8   }
 9   // Streaming.
10   for (int i = 1; i < Q; i++) {
11     cell[i] = plCube(x + (int) offsets[i].x,
12         y + (int) offsets[i].y, z + (int) offsets[i].z)[i];
13   }
14 
15   // Collision.
16   if (parts->exponent == 255 && parts->mantissa & FLAG_OBSTACLE) {
17     if (parts->mantissa & FLAG_OBSTACLE) {
18       cell_t cell2 = cell;
19       for (size_t i = 1; i < Q; i++) {
20         cell[i] = cell2[opposite[i]];
21       }
22     }
23     return cell;
24   }
25   float p = 0;
26   vec3f vp {0, 0, 0};
27   for (size_t i = 0; i < Q; i++) {
28     p += cell[i];
29     vp += offsets[i] * cellwidth * cell[i];
30   }
31   vec3f v = p == 0 ? vp : vp * (1 / p);
32 
33   for (size_t i = 0; i < Q; i++) {
34     cell[i] = cell[i] + deltaT / tau * (feq(i, p, v)
35              - cell[i]);
36   }
37   return cell;
38 }
```

Listing 5: Implementation of an exemplary LBM user function.

The second example used is a mean filter commonly used for smoothing images. Aside from images, filters are commonly used to pre-process data to reduce noise. This might also be applied to signal processing and other application contexts. This example has the advantage that the stencil size (i.e. radius) can be varied, as depending on the context different stencil sizes are reasonable. Moreover, this program does not require conditional statements that potentially slow down the program.

5 Evaluation

Our approach requires measuring the speedup achieved. For this purpose, the presented exemplary programs, a mean filter and an LBM implementation, are executed on the HPC machine Palma II[3]. Table 1 lists the hardware specification of the partitions used. Those are two GPU-partitions and two CPU-partitions. To provide meaningful results, all parallel programs are executed ten times.

		per node		per computational unit		
Identifier	Nodes	GPUs	max. CPU-threads	mem. (GB)	cores	GPU/CPU-type
normal	136	-	36	92	18	Skylake (Gold 6140)
zen2	12	-	128	496	64	Zen2 (EPYC 7742)
gpu2080	5	8	32	11	4352	GeForce RTX 2080 Ti
gpuhgx	2	8	32	80	6912	Nvidia A100 SXM

Table 1: Overview of used hardware.

For testing CPU-parallelization, the *zen2* partition is used, which is equipped with 12 nodes, each with one Zen2 (EPYC 7742) CPU with 64 cores. For running the sequential version, a single Skylake (Gold 6140) CPU is used (*normal*). It is not possible to run sequential programs on the *zen2* partition as all sequential programs have to run on the *normal* partition.

For the GPU-programs, the *gpu2080* and *gpuhgx* partitions are used. The *gpu2080* partition has five nodes, each with 8 GeForce RTX 2080 Ti GPUs. The *gpuhgx* partition is equipped with two nodes, each with 8 A100 SXM GPUs. These partitions, most importantly, vary in the maximum memory and the number of cores. The *gpu2080* partition has more nodes. However, each GPU has 11GB VRAM and 4352 cores allowing less parallelization than more powerful GPUs (such as the A100) can provide. In contrast, the *gpuhgx* partition has 80GB VRAM per GPU, allowing bigger data structures to be processed, and has more cores (6912) to speed up the program. Using different GPUs also contributes to proving the universal applicability of the MapStencil skeleton in Muesli.

Next to a sequential program, the Muesli-programs solving the LBM are compared to a native implementation. The native program is written to the best of our knowledge.

5.1 LBM

The LBM is used to simulate fluid flow in the three-dimensional space. Consequently, it is reasonable to run an experiment that does not only execute the mapStencil skeleton one time but has multiple iterations simulating multiple

[3] https://confluence.uni-muenster.de/pages/viewpage.action?pageId=27755336

dispersion steps. 200 iterations were chosen for every experiment to compare run times between different data sizes.

Data sizes were chosen to completely utilize the available storage. For the LBM, each cell requires 76 bytes, as each cell stores 19 32-bit floating point numbers. For the calculation, one data structure to read and one to write is necessary. The largest theoretically possible data structure size for a given amount of memory can be simply calculated by:

$$d(\text{gb}) := \sqrt[3]{\text{gb} \cdot \frac{2^{30}}{2 \cdot 76}}$$

This results in a maximum side length of 426 for the RTX 2080 Ti GPU and 826 for the A100 SXM. Although the CPU partition would support bigger data structures, the data size was not increased to the maximum as the speedup converged, and the runtime of the sequential program became unreasonable high (approximately 10 hours for calculating the LBM simulation for a data size of 960^3).

Table 2 shows that for the CPU-zen2 partition, a speedup of 116 can be reached with a single CPU.

Data size	Sequential	1 Node	Speedup	4 Nodes	Speedup	8 Nodes	Speedup
120^3	56.00	1.30	42.96	0.49	114.58	0.31	180.16
440^3	2994.47	30.35	98.66	16.68	179.53	8.36	358.31
800^3	18572.90	173.88	106.81	86.00	215.96	45.59	407.36
960^3	34880.80	300.44	**116.10**	149.23	**233.74**	77.11	**452.37**

Table 2: Runtimes (seconds) and speedups for the parallel implementation of the LBM gas simulation for CPU programs on the zen2 partition.

As the CPU has 64 cores, this is caused by multithreading. In total, in the optimal case, 128 threads are started on the 64 cores available. In this scenario, it should be considered that the calculations are easy to execute in parallel as all data resides on the memory of the single CPU, accessible for all threads. Using four nodes requires communicating the border values, thus requiring operations that cannot be executed in parallel. Although more elements need to be communicated with increasing data sizes, the share of operations that require communication is decreasing, thus allowing more parallelism. This can be seen as the best speedup that can be achieved with four nodes for the biggest tested data size (960^3). Eight nodes achieve a speedup of 452.

In contrast, the GeForce RTX 2080 Ti achieves a speedup of 88 with a single GPU. The maximum speed up which can be achieved by GPUs depends on the GPU used. The GeForce RTX 2080 Ti has 68 streaming multiprocessors (SMs), each capable of executing 64 threads in parallel (4352 in total). Although more threads can be scheduled, the hardware does not allow executing them in parallel. Multiple factors limit the possible speedup. Most importantly, threads with diverging execution branches cannot be executed in parallel, limiting the 64 threads executed in parallel.

The A100 has 108 SMs, allowing more threads to be executed in parallel. The maximum speedup achieved is roughly 353.

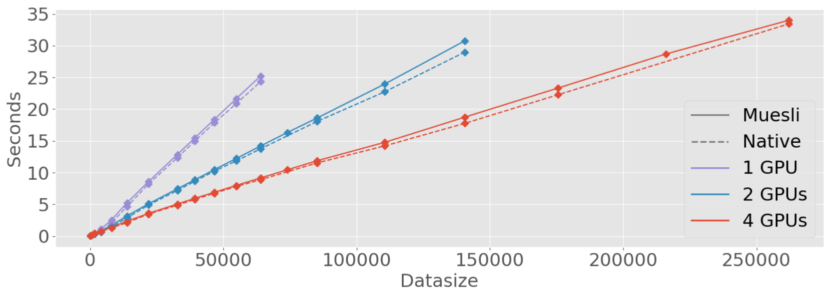


Fig. 2: Runtime comparison of a multi-GPU Muesli program and native (CUDA) implementation of the LBM on a single-node of the gpu2080 partition.

In order to check whether a low-level program is significantly faster, a native implementation was programmed to be compared against the Muesli program. As can be seen in Figure 2, the implementations are close to each other. In contrast to the native implementation, Muesli has a slight overhead. However, as it is very small, the differences are expected and neglectable. Runtimes for bigger data structures are not included for one GPU and two GPUs to increase the readability of the graph. Also worth mentioning is that the native implementation has 544 lines of code without implementing MPI inter-node communication. In contrast, the Muesli program has 246 lines of code, allowing multiple node programs.

Regarding the scalability on multiple GPUs, a speedup of 1.7 compared to a single GPU version can be reached for two GPUs, and for four GPUs, a speedup of 2.75 is achieved. To ensure that the communication causes the overhead, the time for the update function was measured separately. Without communication, a speedup of 1.94 and 3.88 was achieved, which can be attributed to the synchronization of streams. Although the speedup is limited by the communication operations, using multiple GPUs also has the advantage of being able to process bigger data structures, since there is more memory available. The total speedup can be seen in Table 3.

Considering multiple levels of parallelism, the program can also run on multiple nodes equipped with multiple GPUs. Runtimes are depicted in Figure 3 and Table 4. Most eye-catching, the runtimes for four nodes and four GPUs are not linear but show a switching pattern. This is caused by not splitting the data structure into complete slices but into incomplete slices (e.g., 640 has 40 slices per GPU while 680 has 42.5). The program can handle this. However, the communicational effort rises.

Data size	Sequential	1 GPU	Speedup	2 GPUs	Speedup	4 GPUs	Speedup	8 GPUs	Speedup
400^3	2243.63	25.23	88.93	14.20	158.06	9.16	244.92	9.06	247.51
520^3	4971.15	-	-	30.76	161.62	18.74	265.32	16.65	298.53
640^3	9261.54	-	-	-	-	33.99	272.48	28.78	321.75
800^3	18572.9	-	-	-	-	-	-	50.68	366.45

Table 3: Speedup for the parallel implementation of the LBM gas simulation on the gpu2080 partition.

Data size	GPUs	1 N	1 N-Com	4 Ns	4 Ns -Com[1]	Speedup	Speedup
280^3	1	8.59	8.59	2.74	2.16	3.14	3.97
280^3	4	3.58	2.23	2.06	0.39	1.73	5.74
400^3	1	25.23	25.23	7.21	6.32	3.50	3.99
400^3	4	9.16	6.50	5.14	1.70	1.78	3.82

Table 4: Speedup for the parallel implementation of the LBM gas simulation for multiple nodes on the gpu2080 partition.

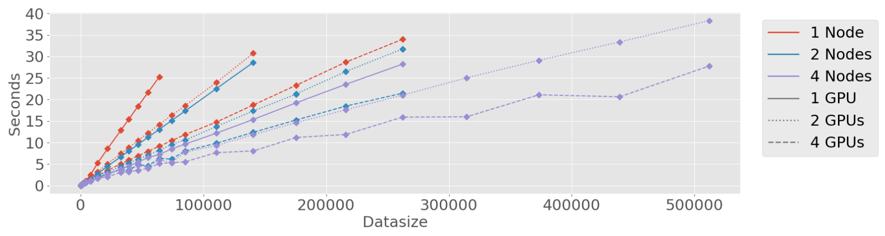


Fig. 3: Runtimes of the LBM Muesli program on multiple nodes and multiple GPUs on the gpu2080 partition.

5.2 Mean Filter

Using the LBM implementation as an exemplary program has multiple downsides. Firstly, one factor which has a major influence on the generalizability of the skeleton stays constant - namely, the stencil radius. This factor influences the number of elements that need to be communicated between the compu-

	Runtime (s)			Speedup	
Data size	Sequential	4 Nodes	8 Nodes	4 Nodes	8 Nodes
120	86.85	0.64	0.46	136.68	190.30
280	1135.88	6.82	3.95	166.43	287.84
400	3333.38	19.91	10.23	167.43	325.97
560	9188.60	49.91	26.40	184.00	348.08

Table 5: Runtimes in seconds and speedups of CPU program for the mean filter with stencil radius 2 on the gpu2080 partition.

tational nodes. Therefore, it is essential to vary this factor to analyze the performance of the skeleton. Moreover, the implementation of the LBM has multiple conditional statements like branch operations slowing down the possible parallelism. In contrast, the mean blur does not contain any if-branches.

Firstly, the CPU parallelism is discussed. Table 5 lists the speedup for one, four, and eight nodes compared to a sequential program. Noteworthy, the optimal time is not achieved by using 128 threads but by using 64 threads. As the instructions are easily executed in parallel, scheduling threads that are executed when other threads are idle is no longer beneficial. For four nodes, a speedup of 184 can be reached. As can be seen, for rising data sizes, the speedup improves as fewer communication operations are required. The same applies to programs using eight nodes reaching a speedup of 348.

Stencil radius	Data size	Runtime 1 GPU	Speedup Seq/1 GPU	Runtime 4 GPU	Speedup 1 / 4 GPUs	4 GPUs -Com[1]	Speedup 1/4-Com
2	120^3	0.16	542.83	0.07	2.16	0.05	3.51
	400^3	4.46	**747.39**	1.44	3.09	1.15	3.89
	800^3	43.65	-	12.14	3.60	10.95	3.99
10	120^3	8.24	560.50	2.64	3.12	2.50	3.29
	280^3	131.51	508.92	35.98	3.66	35.20	3.74
	800^3	3953.38	-	1022.75	3.87	1016.71	3.89

Table 6: Speedup for the parallel implementation of the mean blur for a single node of the gpu2080 partition.

Stencil radius	Data size	Runtime 1 GPU	Speedup Seq/1 GPU	Runtime 4 GPU	Speedup 1 / 4 GPUs	1 GPU -Com[1]	4 GPUs -Com[1]	Speed-up 1/4-Com
2	120^3	0.15	579.02	0.08	2.00	0.09	0.03	3.08
	400^3	2.66	**1253.15**	1.12	2.38	2.31	0.65	3.53
	800^3	24.88	-	7.41	3.36	22.09	5.81	3.80
10	120^3	5.48	842.20	1.87	2.94	4.02	1.50	2.69
	280^3	75.39	887.78	20.41	3.69	65.68	18.68	3.52
	800^3	2149.16	-	541.64	3.97	2009.69	510.79	3.93

Table 7: Speedup for the parallel implementation of the mean blur for two nodes of the gpu2080 partition.

Secondly, the speedup for the GeForce RTX 2080 Ti GPUs is measured. The program has less `if`-branches, so it is more appropriate for GPUs. The speedup for a single GeForce RTX 2080 Ti GPUs for a data size of 400^3 is 747, significantly better than for the LBM implementation. Table 6 and 7 list the runtimes and speedups for a small stencil radius of two (reading 125 elements per calculation) for one and two nodes with each one or four GPUs. Including communication operations, a speedup of 3.6 is reached. To ensure that this is really caused by the communication operations, the runtime spent on calculation is measured separately. The speedup depicted in the last column

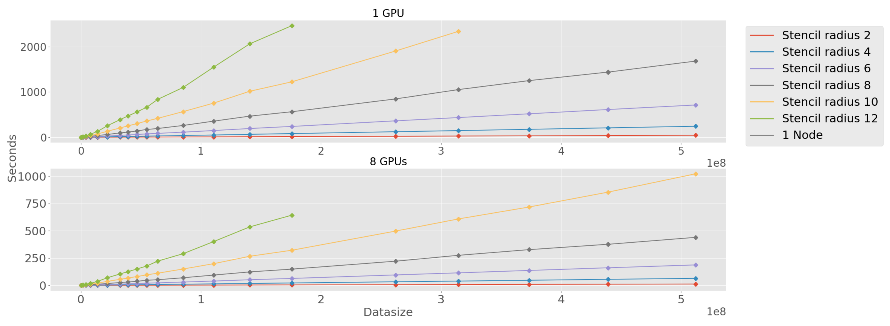


Fig. 4: Runtimes of the Blur Muesli program on the gpu2080 partition.

reaches 3.99, which is close to an optimum of 4. As communication operations require synchronization, the optimal speedup is hardly achievable. In contrast, scaling across nodes improves the speedup for a single GPU from 747 to 1253. Comparing the runtimes without communication, 43.65 seconds are nearly doubled from 24.88. Scaling from one node to two nodes is more efficient, as merely one overlapping data region needs to be communicated. The speedup for four GPUs behaves similarly to the above-explained behavior.

Regarding bigger stencil radiuses, the runtimes for a stencil radius of 10 are listed. This requires reading 9261 elements per calculation. This extreme example is chosen to observe the runtime and speedup when not all elements can be loaded in caches.

Besides changing the hardware, the stencil radius was adjusted to discuss the impact on the performance. With an increasing stencil radius, the calculation of one element requires more read and write operations. For a stencil radius of two, the sum of 125 elements is calculated, and self-explanatory, this grows cubic. Figure 4 depicts the influence on the runtime. Although the number of elements processed grows cubic, the runtime does not grow cubic.

Stencil radius	Data size	1 GPU	4 GPUs	4 GPUs -Com[1]	Speedup	Speedup	8 GPUs	8 GPUs -Com[1]	Speedup	Speedup
8	120^3	4.54	1.47	1.35	3.09	3.35	1.04	0.79	4.37	5.74
	280^3	60.58	16.44	15.85	3.68	3.82	9.71	8.47	6.24	7.15
	400^3	192.61	51.59	50.34	3.73	3.83	29.52	26.96	6.52	7.14
	560^3	563.20	148.68	146.41	3.79	3.85	83.38	78.58	6.75	7.17
12	120^3	13.59	4.20	4.03	3.23	3.37	2.53	2.17	5.38	6.27
	280^3	252.54	69.24	68.32	3.65	3.70	37.92	35.97	6.66	7.02
	400^3	836.59	222.12	220.38	3.77	3.80	118.89	115.15	7.04	7.27
	560^3	2464.76	643.33	640.16	3.83	3.85	338.46	331.65	7.28	7.43

Table 8: Speedup for the parallel implementation of the mean blur for multiple GPUs on the gpu2080 partition.

Moreover, it ascertains that the speedup improves with a growing share of calculation operations. This is detailed listed in Table 8. Similar to the

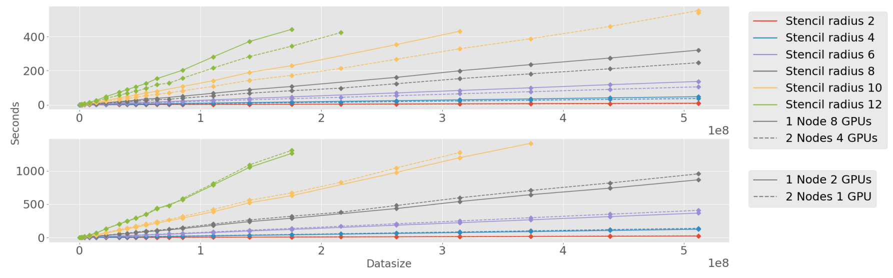


Fig. 5: Runtimes of the Blur Muesli program on the gpu2080 partitions with similar hardware.

CPU program, the speedup with and without communication is measured. For bigger stencil radiuses, the speedup comes closer to the optimum. Although more elements need to be communicated between the nodes, the intensity of the calculation requires more time which dominates the total runtime. For a stencil radius of 12 (processing 15.625 elements per thread). The operations are still very performant as elements are automatically written in GPU caches, allowing efficient data access. As multiple combinations of hardware settings are tested, it is interesting to compare having the same number of GPUs distributed on a different number of nodes. For example, having one node with two GPUs, in contrast to having two nodes with one GPU, has the downside of having one less CPU. However, it has the advantage of allowing GPU to GPU communication. Two comparisons are displayed in Figure 5. The first figure compares a one-node eight GPUs program to a two nodes four GPUs program for different stencil radiuses. As can be clearly seen, the two nodes program is faster. This is caused by parallelizing communication as some operations are executed by the CPU instead of communicating between only GPUs. In contrast, the second figure shows that when a single node program with two GPUs is compared to a two nodes program with each one GPU. The single node program is slightly faster as a communication operation between GPUs is faster than an MPI communication between two nodes.

Stencil radius	Data size	1 GPU	4 GPUs	Speedup	8 GPUs	Speedup	4 GPUs -Com[1]	Speedup	8 GPUs -Com[1]	Speedup
2	400^3	2.97	0.91	3.28	1.14	2.61	0.75	3.95	0.38	7.77
	800^3	24.00	6.57	**3.65**	4.64	**5.18**	6.02	3.99	3.03	**7.92**
8	400^3	103.60	27.16	3.81	15.19	6.82	26.56	3.90	13.35	7.76
	800^3	1016.62	258.47	**3.93**	134.69	**7.55**	256.47	3.96	129.55	**7.85**

Table 9: Runtimes in seconds and speedups for a mean filter on the gpuhgx partition

The program was also tested on the A100 partition allowing even bigger data sizes to be processed by a single GPU. In contrast to a single GeForce

RTX 2080 Ti GPUs (Speedup 747), the A100 has a speedup of 1122.34 for a data size of 400^3 elements, and 3680.54 for four GPUs (2309.08). According to the previous approach, the runtime was measured with and without communication (Table 9). Even with the communication, the speedup is close to the optimum. For four GPUs, a speedup of 3.65 and 3.93 can be reached four 400^3 and 800^3 elements. For eight GPUs, a speedup of 5.18 and 7.55 is reached.

6 Conclusion

We have presented the implementation and experimental evaluation of a three-dimensional MapStencil skeleton. The implementation was tested with two example programs: 1) an LBM implementation and 2) a mean filter. The two examples complement each other for different application contexts. The LBM has a rather complex function, with a constant stencil radius, while the mean blur has a simple user function, and the stencil radius can be varied. Both examples were tested in complex hardware environments equipped with multiple nodes, CPU cores, and accelerators. For the LBM, a speedup of 116 for one node and 452 for four nodes can be reached, confirming that for complex functions, the communication between nodes has only a minor influence on the speedup. In contrast, using multiple GPUs for complex functions does not provide the expected speedup. For a single GPU, merely a speedup of 88 can be reached. This speedup scales for two GPUs (161), but as the communication rises, the speedup does not scale according to the number of accelerators. Abstracting from the communication, the program scales across multiple nodes.

In contrast, running the mean filter example shows that non-diverging programs do not benefit from multithreading with CPUs, having a speedup of 180 for four nodes. In contrast, a GPU can reach a speedup of 747, benefiting from warps of threads that can run the same instruction in parallel. The communication between GPUs decreases the speedup from 7.7 to 5.4, taking approximately 30% of the runtime. When increasing the stencil radius, the speedup improves as the initialization of the communication and the required synchronization is more time-consuming than increasing the stencil radius, which is communicated. This finding is confirmed for two different types of GPUs an A100 partition and a GeForce RTX 2080 Ti partition.

Overall it is shown that Stencil operations are especially relevant to be included in high-level frameworks as the communication between nodes and accelerators is highly complex, and the implementation is not feasible for inexperienced programmers. The speedup achieved proves that high-level frameworks can provide the means to solve this problem.

References

1. Alba, E., Luque, G., Garcia-Nieto, J., Ordonez, G., , G.L.: Mallba: a software library to design efficient optimisation algorithms. International Journal of Innovative Computing and Applications **1**(1), 74–85 (2007)

2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. in: Programming multi-core and many-core computing systems, parallel and distributed computing pp. 261–280 (2017)
3. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible skeletal programming with eskel. In: J.C. Cunha, P.D. Medeiros (eds.) Euro-Par 2005 Parallel Processing, pp. 761–770. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
4. Cole, M.I.: Algorithmic skeletons: structured management of parallel computation. Computer science thesis. Pitman, London (1989)
5. Ernsting, S., Kuchen, H.: Data parallel algorithmic skeletons with accelerator support. International Journal of Parallel Programming **45**(2), 283–299 (2017)
6. Ernstsson, A., Ahlqvist, J., Zouzoula, S., Kessler, C.: Skepu 3: Portable high-level programming of heterogeneous systems and hpc clusters. International Journal of Parallel Programming **49**(6), 846–866 (2021)
7. Gonzales, R., Gryazin, Y., Lee, Y.T.: Parallel fft algorithms for high-order approximations on three-dimensional compact stencils. Parallel Computing **103**, 102757 (2021)
8. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with lift. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, p. 100–112. Association for Computing Machinery, New York, NY, USA (2018)
9. Karasawa, Y., Iwasaki, H.: A parallel skeleton library for multi-core clusters. In: 2009 International Conference on Parallel Processing, pp. 84–91 (2009)
10. Krüger, T., Kusumaatmaja, H., Kuzmin, A., Shardt, O., Silva, G., Viggen, E.M.: The Lattice Boltzmann Method: Principles and Practice. Graduate Texts in Physics. Springer International Publishing AG, Cham (2016)
11. Latt, J., Malaspinas, O., Kontaxakis, D., Parmigiani, A., Lagrava, D., Brogi, F., Belgacem, M.B., Thorimbert, Y., Leclaire, S., Li, S., Marson, F., Lemus, J., Kotsalos, C., Conradin, R., Coreixas, C., Petkantchin, R., Raynaud, F., Beny, J., Chopard, B.: Palabos: Parallel lattice boltzmann solver. Computers and Mathematics with Applications **81**, 334–350 (2021). Development and Application of Open-source Software for Problems with Numerical PDEs
12. Marques, R., Paulino, H., Alexandre, F., Medeiros, P.D.: Algorithmic skeleton framework for the orchestration of gpu computations. In: F. Wolf, B. Mohr, D. an Mey (eds.) Euro-Par 2013 Parallel Processing, pp. 874–885. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
13. Steuwer, M., Gorlatch, S.: Skelcl: a high-level extension of opencl for multi-gpu systems. The Journal of Supercomputing **69**(1) (2014)
14. Wrede, F., Rieger, C., Kuchen, H.: Generation of high-performance code based on a domain-specific language for algorithmic skeletons. The Journal of Supercomputing **76**(7), 5098–5116 (2020)

Symposium on High-Level Parallel Programming & Applications
HLPP 2023

Accelerating DNN Communications by Hierarchical Resharding

Hector Buffière · Pierre Leca · Chong Li

Abstract High-level computing frameworks have been proposed to simplify the deployment of deep learning models, including parallelizing the models on a given machine. Meanwhile, modern parallel machines leverage different types and levels of communication to interconnect devices in order to maximize the computing power. We formalized data redistribution costs and proposed an algorithm to efficiently schedule data redistribution on hierarchical parallel machines in order to place more expensive communication between closer devices to reduce communication cost. Our solution was implemented with the MindSpore open-source framework, and tested on a large Natural Language Processing neural network, where we observed significant speedups of up to 120% on end-to-end training time.

Keywords Parallel Programming, Cost Analysis, Hierarchical Communication, Distributed Learning, Performance Optimization.

1 Introduction

In recent years, the appearance of giant deep learning (DL) models [3,21,13] leads the dramatically heavy demand for computation power. On one hand, more and more tensor-oriented large-scale parallel computers have been designed [11,12,6]. Parallel computers are often arranged in a hierarchical manner, where the communication cost between two nodes depends on how close the nodes are topologically, and also on the type of connection between the nodes. For instance, a computing node like Huawei's Atlas or Nvidia's DGX groups 8 accelerators (NPU or GPU, resp.), organized in a twisted-torus topology, with 4 connections per device. Computing nodes are connected with each other in a fat-tree topology.

H. Buffière[*!] · P. Leca[*] · C. Li[*]
[*] Huawei Paris Research Center, 20 quai du point du jour, Boulogne-Billancourt, France
[!] Département d'Informatique de l'ENS, CNRS, PSL University, 45 rue d'Ulm, Paris, France
E-mail: hector.buffiere@ens.fr, pierre.leca@huawei.com, ch.l@huawei.com

On the other hand, in the domain of parallel programming, different kinds of high-level parallelisation techniques also emerged in order to reduce the execution time of such Deep Neural Network (DNN) models. These techniques were generalized into different Deep Learning parallelism paradigms, including data parallelism [5] (in the context of DL, this refers to splitting batches into mini-batches which are given to different devices) and operator-level model parallelism [7] (splitting parameters into different devices). These two types of parallel strategies can be again combined at the same time in a hybrid strategy for parallelizing the same operator. Modern AI frameworks like PyTorch [14], TensorFlow [1] or MindSpore [16] exhibit a modular design, which allows DNN programmers to define their networks as operator graphs acting on tensors in a high-level manner. The modularity allows better portability, as different parts of the framework can be independently modified and optimized depending on e.g. the hardware or the type of network. The top-level module is the python interface where DL engineers can concisely define various neural networks. One mid-level module of the framework can then automatically split the computation depending on the number of devices used, and a second one assigns each part of the workload to devices of the cluster and connects them with necessary logical communications. The bottom-level ones use communication libraries to write the final machine code. Our work belongs to the second mid-level module and seeks to provide new optimizations orthogonally to the already very successful parallel techniques mentioned above.

However, the choice of high-level strategies of parallelism is still a complex problem. The number of possible strategies that can be chosen depends on the number of operators and the number of devices. The best strategy for a given operator also depends on the strategy of its neighbors when redistributions are taken into account. To avoid increasing this complexity, strategies are often chosen with a simplified view of the implied redistributions. For example, by not considering the precise low-level device assignment of tensor shards, nor the precise redistribution operations that need to be performed. With DL relying mainly on linear algebra operations that are inherently highly parallelizable, and communication being much slower than computation, almost all of the extra cost induced by distributed training comes from redistributions, as showcased in [22]. As hybrid strategies are more commonly used nowadays in order to find a balance between speed and memory requirement, this causes more frequent and more common redistributions, amplifying the performance issues caused by inefficient choices of redistributions.

In this paper, we formalize the redistribution costs among different parallel strategy configurations, and propose a novel algorithm for redistributing tensor shards between two operators with different device assignments. The key idea of this algorithm is to favor performing large data transfers between devices with faster communication links and small data transfers between devices with slower communication links. In this way, we exploit the properties of the hierarchical nature of clusters.

2 Partitioning of Deep Neural Networks

2.1 Computational graph

A DNN can be defined as a computational graph $G = (V, E)$ where V is a set of vertices, either operators, parameters, input or output nodes, and E is a set of directed edges connecting vertices. A vertex $v \in V$ is defined as a pair (o, t) where o describes the type of the operator (*Add*, *MatMul*, ...) and t is the shapes of the operator's tensors, a pair of lists of tuples of integers, the first one for each input tensor and the second one for each output tensor of the operator (some might be empty, for example the input list if the node is a parameter of the model). The above defines the basic semantics of a deep learning model from a computation point of view.

2.2 Sharding

A computational graph can be executed in parallel without changing its DNN's behavior by partitioning its tensors on devices of a cluster. This way, the operators are computed in parallel on devices with different tensor *shards* by using the inherently sequential and layered nature of DNNs. We then focus on the exchange of the data resulting of parallel computations when needed (if the results need to be combined or if the data allocation changes).

The partitioning can be done by defining *tensor layouts*, which specify how each tensor is split across devices. It will be performed in 3 passes: (i) by assigning *strategies* to each operator, that specifies how to cut the tensors; (ii) the Deep Learning framework performs the assignment of tensor shards to devices; (iii) lastly the framework generates and inserts communication primitives to reallocate tensor shards between different consecutive layouts, or to synchronize data when needed by some operators (e.g. *MatMul*).

Sharding can be defined by a map $\mathcal{S} : V \to \tilde{V}$ where each vertex of $\tilde{V}$ is now a triple $\mathcal{S}((o, t)) = (o, t, s)$. The strategy s is also a pair of list of integers but specifying the sharding of each input and output tensors' dimensions in an integer number of splits. For instance, an input tensor of shape $t_{\text{in}} = (16, 1024, 1024)$ could be assigned a strategy $s_{\text{in}} = [\![4, 1, 2]\!]$ meaning that the first tensor dimension would be split 4 times, the second would be replicated and the last split twice. The tensors shards would then be of shapes $(4, 1024, 512)$, and be evenly spread into $4 \times 1 \times 2 = 8$ partitions. Note that 8 partitions could be deployed on 8 but also on 16, on 32, ... devices. An example of the distributed computation of an Add operator with two input tensors and one output tensor is shown in Figure 1.

2.3 The challenge of distributed deployment

Most DL platforms use sharding strategies (jointly with hyperparameters or automatic methods) as the unique way to define DNN partitioning, and details

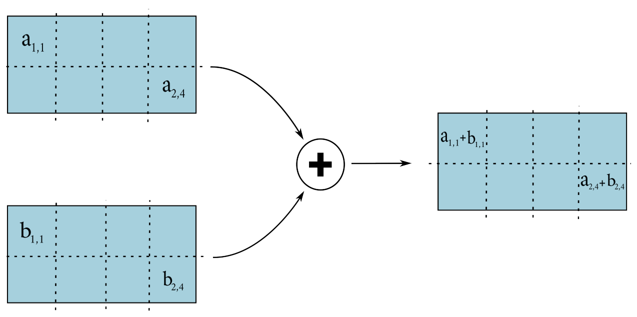


Fig. 1 Example of an Add operator computing $c = a + b$ with input and output tensors sharded 8 times with strategies $[\![2, 4]\!]$. This computation can be run on $N \geq 8$ devices, each tensor shard being replicated $N/8$ times.

are hidden from the framework user. We argue that this is an oversimplification, since the redistributions do not only depend on strategies but also on the way the computation of operators on shards is split between devices of the cluster. Moreover knowing how strategies translate into device assignments is crucial to get an idea of the communication cost of redistributions.

Example 1 As a toy example, we assume a tensor which is the output of an operator and also the input of another operator. This tensor has the strategy $[\![2, 1]\!]$, is thus split into 2 shards a_1 and a_2. When the computation is run on 4 devices $(d_i)_{1\leq i\leq 4}$, each of a_1 and a_2 will be replicated once.

If the tensor assignments are the same for both operators, for instance a_1 is processed by devices d_1 and d_2 and a_2 by d_3 and d_4, no redistribution is needed. However, it could also be that as the input of the second operator, a_1 must be processed by d_1 and d_3 and a_2 by d_2 and d_4 , in which case even if the strategies looked the same, some redistribution is needed.

Crucially, in the above case where the assignments were switched, the shard a_1 must be sent to d_3 (from d_1 or d_2), and similarly for a_2 to d_2. If the assignments were swapped, d_1 or d_3 would need to send a_1 to d_2. As in hierarchical clusters, communication between d_1 and d_2 may be significantly faster than between d_1 and d_3 (this will be made explicit in next section), we can see that what looked like straightforward sharding strategies can lead to dramatically different performances depending on the implementation.

As a result, device assignments and redistributions are often generated in ad hoc and naive ways and cannot be taken into account by DL developers when designing DNNs. It could only remain true when deploying a distributed DNN on a tiny cluster where the communication costs between devices are perfectly homogeneous. However, in real world, a DNN may be of giant size and need thousand of devices for training it. That invokes a hierarchical cluster where the communication costs are heterogeneous. We thus propose to shift the threshold of DNN design to include device assignments, whose formalism is described in the next section. This shift of perspective along with new formalism and hypotheses on DNN clusters and their hierarchical communications allows for a detailed analysis of communication performance of redistribution generation algorithms. The problem of inserting optimal communication primitives, although simplified, remains complex, and we propose an efficient algorithm to solve it.

3 Deploying of Distributed Deep Neural Networks

3.1 Hierarchical abstract machine

Inspired by BSP [17], using a high-level parallel programming model with an abstract machine will generalize hardware impacts instead of needing to handle case by case with different topologies and characteristics of cluster. The modern ML clusters, like Huawei Atlas[1] and Nvidia DGX[2], are organized in a hierarchical way: a system could be composed of one or several racks, a fully-loaded rack is composed of 8 computing nodes, a computing node is composed of 2 computing groups, and one computing group is composed of 4 accelerators. Thus we have fast intra-node communication and slow inter-node communication, which makes it particularly important to consider the topology and characteristics of a cluster network before designing device assignment and resharding algorithms. The accelerators from different computing nodes communicate between them via hierarchical switches with a fat-tree topology, and the accelerators inside a computing node communicate via HCCL or NCCL with a twisted-taurus topology. We can resume that a hierarchical and symmetric abstract machine could cover the most recent hardware architectures.

We take the SGL [8] hierarchical bridging model as our basic abstract machine, to avoid the oversimplification of communication by BSP, and refine it to a binary tree. The choice of SGL is also because it was already extended to DL and implemented in MindSpore [18]. We observed that, in both industrial and academic settings, many ML clusters typically use a number of devices that is a power of 2 to achieve best performance, so we will assume for the rest of the paper that the number of devices satisfies $N = 2^p$ for some integer p, to simplify our theoretical analysis. This assumption allows us to define an abstract hierarchical and symmetric machine using a perfect tree structure, similarly to the analysis of [18]. The leaves of the tree represent devices, and inner nodes represent the group of all their children devices, which typically correspond to a group of adjacent devices, or belonging to the same rack or server. The root node thus represents the group of all the devices of the cluster. Since we are sharding tensors evenly between $N = 2^p$ devices, the numbers appearing in strategies will always be powers of 2, and we will even assume for simplicity that they will always be 1 or 2 (this is equivalent if we allow ourselves to reshape tensors to add tensor dimensions to split along).

[1] https://e.huawei.com/en/products/computing/ascend/atlas-900-ai
[2] https://www.nvidia.com/en-us/data-center/dgx-platform/

3.2 Communication cost

By symmetry of the cluster, all groups of devices at depth level i of the tree share the same communication capacity g_i. The cost of the communication between two devices associated with leaves u and v is $q \times g_i$, where q is the amount of data transferred and i is the level of the lowest common group, or ancestor of u and v in the tree. The communication capacities are assumed to be increasing the deeper they are in the tree. We assume that $g_i > g_{i+1}$, which in practice means that so we could always consider optimal to minimize the amount of communication going through a slower connection at the expense of augmenting the amount of communication going through faster connections, even by a bigger amount.

This is the basis of our analysis on which depends the choices of our algorithm. We observed that the bigger a cluster is, the more benefit we draw from this strongly hierarchical model, as the communication links between faraway devices are orders of magnitude slower than the ones connecting close devices.

3.3 Device assignment

Once the strategies of each operators' tensors are chosen, we need to decide which shards will be processed on which devices. This is in itself a complex problem, and is critical as it ultimately defines which communication will be needed between which devices. There are theoretically $N!$ ways to assign N shards to N devices, but exploiting the symmetries of sharding and of the cluster can reduce this search space while keeping a wide range of sensible (i.e. sufficiently symmetric) assignments. The idea is to assign tensor parallelism axes to device parallelism axes, that correspond to the hierarchical division of the cluster into groups of close devices. These device axes, or device dimensions, are represented by a *device matrix* which is a vector d where d_i is the number of children of each node in the i-th level of the abstract tree defined in 3.1. Note that as a consequence $\prod_i d_i = N = 2^p$, and all d_i are powers of 2.

A tensor dimension i can then be sharded along device dimension j if they are split the same number of time, i.e. if $d_j = s_i$. In practice device matrices do not need to match the cluster hierarchy, device dimensions can split or merge several physical device groups: device matrices can be different for each tensor layout and as long as the product of their elements is equal to the device number, they can be used to specify a symmetric device assignment. Since we assume in this paper that tensor dimensions are always sharded once or twice, a vector with p elements all equal to 2 is always a valid device matrix.

The assignment of tensor parallelism dimensions to device dimensions is realized by another map $\mathcal{M} : \tilde{V} \to \hat{V}, (o, t, s) \mapsto (o, t, s, m)$ where m is a *tensor map* of the same shape of t and s such that if the tensor dimension i is not sharded (i.e. $s_i = 1$), $m_i = \varnothing$, and otherwise m_i is the index of the device dimension to which i is sharded along.

Finding an optimal $\mathcal{M}$ has a combinatorial complexity, equivalent to finding an optimal $\mathcal{S}$. Except the parameter tensors, most tensors in the computational graph are both an input and output tensor of successive operators. The impact from the choice of parallelism on operators will be quickly propagated to the whole graph in non-trivial ways. Therefore, the choice of a tensor map has implications on the optimal assignment of all connected tensors.

Redistributions are needed when the input and output layouts of a tensor differ. This task could be done with collective communications (e.g. AllGather and Slice), that can efficiently and concisely express all communication patterns needed for redistribution between all symmetric layouts expressible using device matrices and tensor maps. There are often many possibilities for arranging these collective communication operations, especially in cases where sophisticated and diverse tensor layouts are used with large numbers of devices, which is becoming the norm with automatic fine tuning of sharding strategies to improve large scale parallel training performance. Different operation sequences can have vastly different execution times and memory usage given the orders of magnitudes between close devices and distant devices link performances. We propose in the following section a topological-aware hierarchical algorithm to optimize the generation of collective communication operation sequences and analyze its performance from a symbolic point of view under the aforementioned assumptions.

4 Performance Enhancement

4.1 Preliminaries

Before formally presenting our algorithm, we clarify first some preliminaries. Our algorithm is applied separately on each tensor as the choices of communication operations do not interfere with each other. It takes as input an input layout m^{in} and the output layout m^{out} to which it must be converted. After the sharding strategy decision ($\mathcal{S}$) and the device assignment ($\mathcal{M}$), it can happen that the device matrices and tensor shapes do not match, which requires unifying them. This can be done in the general case by switching to device matrices of the form $[2, 2, 2, \ldots]$ with p elements, reshaping the tensors and modifying the tensor maps accordingly. Note that this first step does not actually change the physical distribution of the tensors but only their software representation, w.r.t. Sec 3.

Once the shapes and the device matrices match, the task can be seen uniquely as determining a sequence of redistribution operations that transforms the input tensor map into the output tensor map and minimizes the communication cost. The two operations that we will use are *AllGather* and *Slice* which are available on all hardware architectures, and will not assume any specific implementation tricks on these, which can orthogonally be applied to improve the overall performance of redistribution. Some advanced operations like *AllToAll* could also be used, but were left out to simplify the analysis. They are non primitive collective communication, and require case-by-case optimal implementations [9]. Therefore, our algorithm focus on more general cases for being usable in the most possible configurations. Taking into account a more diverse set of primitives could be a ground for future work.

AllGather is a collective communication operation that concatenates tensor slices from all devices in a device group and stores the result on each device. If applied along a tensor dimension i sharded along device dimension m_i (written AllGather(i)), the result tensor layout now has $m'_i = \varnothing$ and the communication cost of the operation is $g_{m_i} \times \prod_k \frac{t_k}{s_k}$ (the second member represents the size of the tensor initially on each device). The tensor shards stored on each device have their size multiplied by d_{m_i} at the end of the operation.

Slice is the opposite operation, it is used to split tensors in order to distribute them between devices of a group. When applied along a tensor dimension i and device dimension j (written Slice(i, j)) if $m_i = \varnothing$ and $\forall k, m_k \neq j$, the new tensor layout then has $m'_i = j$. Since it is executed locally on each device there is no communication cost, and the computation cost is again proportional to the shape of the initial tensor layout i.e. $\frac{1}{d_j} \prod_k \frac{t_k}{s_k}$. The tensor shards stored on each device have their size divided by d_j after the operation.

4.2 Redistribution chains

For most input/output layout pairs, and particularly when using simple parallelism plans like data parallelism or model parallelism, the redistributions needed will be simple and straightforward. Some tensor dimensions i can satisfy $m_i^{\text{in}} = m_i^{\text{out}}$, in which case nothing is to be done. If $m_i^{\text{in}} \neq \varnothing$ and $m_i^{\text{out}} = \varnothing$, an AllGather$(m_i^{\text{in}})$ needs to be performed so that the tensor dimension i goes from sharded to replicated. However, sometimes device dimensions can appear both in the input and in the output layout at different tensor dimensions, but we cannot split several tensor dimensions along the same device dimension, so we will need to process more than one AllGather on one tensor dimension by allowing a Slice on the other dimension. These configurations order some pairs of tensor dimensions where one must be processed before another, linking the way we treat them.

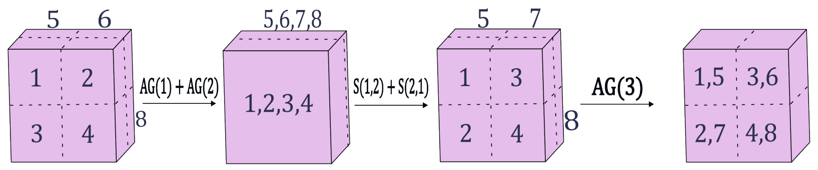


Fig. 2 Redistribution sequence on 8 devices

Example 2 Let $m^{\text{in}} = [1\ 2\ 3]$ and $m^{\text{out}} = [2\ 1\ \varnothing]$. The two first tensors dimensions are split in both layouts, but they need to be switched, while the third is only split in the input, and must be gathered across device dimension 3. Since one tensor dimension must be split across at most one device dimension, it is necessary to unsplit both dimensions 1 and 2 before splitting them again.

One thing to be noted is that even if the two first dimensions can be treated independently from the last, the tensor shards that will be sent over the network will have the same size after two AllGather and two Slice, whereas they will be twice bigger after one AllGather. Thus it is more optimal to do first the two first dimensions and then the last one. An efficient redistribution sequence is (we write AG for AllGather and S for Slice) $[1\ 2\ 3] \xrightarrow{\mathrm{AG}(1)} [\varnothing\ 2\ 3] \xrightarrow{\mathrm{AG}(2)} [\varnothing\ \varnothing\ 3] \xrightarrow{\mathrm{S}(1,2)} [2\ \varnothing\ 3] \xrightarrow{\mathrm{S}(2,1)} [2\ 1\ 3] \xrightarrow{\mathrm{AG}(3)} [2\ 1\ \varnothing]$. Steps 1, 3, 5 and 6 are shown in Figure 2.

The links defined above can create chains and even cycles of dependencies. Our algorithm starts by splitting the tensor dimensions into independent chains and cycles that can be treated in any order (it corresponds to splitting the graph of tensor dimensions and their links into its connected components). We analyze the effect of the resolution of each chain and cycle, compare their relative communication and computation cost, arrange them in an optimal way and insert their associated redistributions in this order independently.

We list the different types of chains and cycles in Table 1, representing them using two lines of rearranged and reindexed device dimensions. The device dimensions are to be thought as being extracted from the input tensor map at the top and the output tensor map at the bottom, and they are on the same column if they corresponded to the same tensor dimension initially, which are rearranged to follow the structure of the chain. Device dimensions with a variable name are assumed to be different from $\varnothing$, two device dimensions of a chain with the same name are equal, and a variable with a tilde on top means that it is not present anywhere else in the tensor maps. We don't give the exact cost but an approximation based on the assumption that the ratios g_i/g_{i+1} are very large, and consequently the cost of a sequence of communication operations along different links will be roughly equal to the cost of its communication using the slowest link. It can be shown that the table enumerates all possible chain formats. Indeed, classifying the device dimensions as empty, used twice or used only once in the two matrices, we can see that the first or last pairs of the chains shown span all the possible combinations, and the rest of the chains can be seen as a deterministic graph exploration.

4.3 Main algorithm

We first outline here the main algorithm, and then explain in more details how each redistribution chain is handled. Given the specifics of the AllGather and Slice operations explicated in 4.1, all the communication cost of resharding is carried by the AllGather operations and proportional to the size of the tensor on which they are applied. As the system is assumed to be highly hierarchical, i.e. ratios g_i/g_{i+1} are assumed to be very large, we will consider an algorithm

Table 1 The different types of chains and cycles of device dimensions.

Name	Chain format	Communication cost
Empty chain	$\begin{bmatrix}\varnothing\\\varnothing\end{bmatrix}, \begin{bmatrix}m\\m\end{bmatrix}$	0
Split	$\begin{bmatrix}\varnothing\\\widetilde{m}\end{bmatrix}$	0
Neutral chain of type 1	$\begin{bmatrix}\varnothing & m_1 & m_2 & \dots & m_l\\ m_1 & m_2 & \dots & m_l & \varnothing\end{bmatrix}$	$\max\limits_{1\leq j\leq l}(g_{m_j}) \times \prod_k \frac{t_k}{s_k}$
Neutral chain of type 2	$\begin{bmatrix}\widetilde{m}_1 & m_2 & m_3 & \dots & m_l\\ m_2 & m_3 & \dots & m_l & \widetilde{m}_{l+1}\end{bmatrix}$	$\max\left(\max\limits_{2\leq j\leq l}(g_{m_j}), g_{\widetilde{m}_1}\right) \times \prod_k \frac{t_k}{s_k}$
Split chain	$\begin{bmatrix}\varnothing & m_1 & m_2 & \dots & m_l\\ m_1 & m_2 & \dots & m_l & \widetilde{m}_{l+1}\end{bmatrix}$	$\max\limits_{1\leq j\leq l}(g_{m_j}) \times \prod_k \frac{t_k}{s_k}$
Concatenation chain	$\begin{bmatrix}\widetilde{m}_1 & m_2 & m_3 & \dots & m_l\\ m_2 & m_3 & \dots & m_l & \varnothing\end{bmatrix}$	$\max\left(\max\limits_{2\leq j\leq l}(g_{m_j}), g_{\widetilde{m}_1}\right) \times \prod_k \frac{t_k}{s_k}$
Cycle	$\begin{bmatrix}m_1 & m_2 & \dots & m_l\\ m_2 & \dots & m_l & m_1\end{bmatrix}$	$\max\limits_{1\leq j\leq l}(g_{m_j}) \times \prod_k \frac{t_k}{s_k}$

to be optimal if its AllGather operations along the slowest links are performed on the smallest possible tensors. Since it can be shown that there is never a need to do more than one AllGather accross each hierarchical level, we can reformulate this requirement by associating a resharding (a sequence of AllGather(i) and Slice(i, j)) to a tuple indexed by hierarchical levels where the element at index i is the size of the tensor at the time AllGather(i) would be performed, or 0 if there is none: an optimal algorithm should then produce a resharding such that this tuple is the lexicographically smallest possible.

In order to achieve this goal, the idea of our algorithm is to handle first the redistribution chains that reduce the sizes of the tensors, then the ones which do not change them, and lastly those that increase them. Among the chains that decrease the size of the tensors, it computes those that need an AllGather with the fastest links first, as they will be performed on larger tensors. Conversely among the chains that increase the size of the tensors those that need to use an AllGather on a slow link are done first. The pseudo code is given in Algorithm 1. The Reshard procedure is a set of handcrafted algorithms for each different type of chain that intend to minimize the communication cost of the resharding they produce without introducing unnecessary communication on unused device dimensions in order to guarantee that they can be reordered and applied independently.

We list below how Reshard handles redistribution chains of each type.

- **Empty chain**: nothing needs to be done.
- **Split** $\begin{bmatrix}\varnothing\\\widetilde{m}\end{bmatrix}$: the resharding is a single Slice($\widetilde{m}$).

Algorithm 1 Computes an optimal resharding sequence

```
Input layouts $m^{\text{in}}$ and $m^{\text{out}}$ of lengths $l$
Output ReshardingSequence
  chains ← $\{\emptyset, \emptyset, \emptyset\}$          ▷ Chains that decrease, keep, and increase the tensor size
  for $1 \leq i \leq l$ do
      if $i$ is not in any chain seen yet then
          chain, cost ← ComputeChain($i$)    ▷ cost: smallest level needed in communication
          type ← ChainType(chain)
          chains[type] ← {cost,chain}
      end if
  end for
  Sort chains[0] by decreasing first element             ▷ Chains that decrease tensor size
  Sort chains[2] by increasing first element              ▷ Chains that increase tensor size
  ReshardingSequence ← $\emptyset$
  for $0 \leq$ type $\leq 2$ do
      for all chain in chains[type] do
          ReshardingSequence ← ReshardingSequence + Reshard(chain)
      end for
  end for
```

- **Split chain** $\begin{bmatrix} \varnothing & m_1 & m_2 & \dots & m_l \\ m_1 & m_2 & \dots & m_l & \widetilde{m}_{l+1} \end{bmatrix}$: we concatenate along the input device dimension then split along the output device dimension from right to left, which gives the resharding sequence <AllGather($l+1$), Slice($l+1, \widetilde{m}_{l+1}$), AllGather(l), Slice(l, m_l), ..., AllGather(2), Slice(2,m_2), Slice(1,m_1) >.
- **Neutral chain of type 2** $\begin{bmatrix} \widetilde{m}_1 & m_2 & m_3 & \dots & m_l \\ m_2 & m_3 & \dots & m_l & \widetilde{m}_{l+1} \end{bmatrix}$: we do the same as for Split chains, except the first tensor dimension also needs to be concatenated, which gives the sequence <AllGather($l+1$), Slice($l+1, \widetilde{m}_{l+1}$), AllGather(l), Slice(l, m_l), ..., AllGather(1), Slice(1,m_1) >.
- **Neutral chains of type 1** $\begin{bmatrix} \varnothing & m_1 & m_2 & \dots & m_l \\ m_1 & m_2 & \dots & m_l & \varnothing \end{bmatrix}$: they need to be treated from left to right in order to avoid doing two consecutive AllGather, which gives <AllGather(2), Slice($1, m_1$), ..., AllGather($l+1$), Slice(l, m_l)>.
- **Concatenation chain** $\begin{bmatrix} \widetilde{m}_1 & m_2 & m_3 & \dots & m_l \\ m_2 & m_3 & \dots & m_l & \varnothing \end{bmatrix}$: Here we can only start with an AllGather, and by doing so on any dimension leaves us with two neutral chains (of types 1 on the right and 2 on the left), which we can then solve independently as previously. As the first AllGather will be done on a tensor half as big as all the others in the sequence, we should start by concatenating along the device dimension with the biggest communication capacity to gain a factor 2.
- **Cycle** $\begin{bmatrix} m_1 & m_2 & \dots & m_l \\ m_2 & \dots & m_l & m_1 \end{bmatrix}$: here again the first operation must be an AllGather, after which it becomes a split chain. Since the first AllGather will be performed on a tensor twice smaller than the others, we start with the device dimension with the slowest communication.

5 Experiments

We ran our experiments using a cloud cluster having a total of 22 Atlas computing nodes. Each node contains 8 Ascend 910 accelerator devices, described in [11]. The inter-node communication is via a 100 Gbps on-chip RoCE interface equipped by all devices, through one or more switches (depending on intra- or inter-rack). The intra-node communication is via a 240 Gbps HCCS, which is a Huawei in-house high-speed interface for Ascend 910. The bandwidth varies slightly between devices in the same node, we thus consider they are in the same level of hierarchy in our experiments.

We used the MindSpore AI computing framework on top of an Atlas cluster to perform our experiments. We evaluated our algorithm on Pangu-α [21], a large language model (LLM) based on the Transformer DNN architecture, which usually requires thousands of devices to train. We take as a starting point a pre-defined configuration of Pangu-α (13B) to have a realistic layer size with an embedding size of 5120 and 40 attention heads. We changed the number of these layers to 2 so that the network could fit into 8 to 64 devices.

We evaluate the performance of our algorithm on two different strategy generation methods. In the first method, we take sharding strategies automatically generated by the MindSpore framework using the method described in [18]. Strategies are assigned automatically to each operator with a cost model aiming to minimize communications. In the second method, we take as a starting point sharding strategies manually chosen by the DL experts who implemented the DNN, for each operator in each elementary block (encoder, linear layer, embedding...). We then modify these strategies so that redistributions are inserted between operators with different strategies.

The automatic method explores more possibilities, which leads to a bigger variety of potential sharding strategies that we can do by hand; it also optimizes strategies for performance by minimizing redistributions. However, large networks may require more consideration for strategies that reduce memory usage in order to fit a network in devices with limited memory. This trade-off means that strategies might be chosen differently and involve more costly redistributions compared to strategies that would reach better performance by avoiding most redistributions if the available memory was enough. Better performance might also be reached by performing some costly redistributions in order to choose better strategies for the operators following the redistributions. The manual strategy method allows us to specify sub-optimal strategies for some blocks and observe the potential performance gain of our algorithm when heavier and more complex redistributions are chosen.

Each test was done on different numbers of devices in order to showcase the performance of our algorithm with different cluster configurations. Specifically, since we worked with machines where the devices were grouped 8 by 8 as computing nodes on racks, there should be little to no gain given the hierarchical nature of the clusters when using 8 or less devices. Each time we double the number of devices, one hierarchical level in the device matrix is added, which increases the likelihood of redistributions needing transfers along links with communication capacities of different orders of magnitude.

We evaluated our algorithm comparatively with the non-hierarchical algorithm of MindSpore [4], and observed the speedup with respect to the parameters we varied. We increase the batch size (B) along with the number of devices (dev #) so that the workload and redistributions per device remain similar. We use step times as a performance metric, reported in milliseconds (ms). Step times correspond to the time it took for handling a batch of data. The reported results are an average of a hundred of step times, as reported by MindSpore logs when running Pangu-α.

dev #	B	automatic strategies			manually-twisted strategies		
		step time (ms)		speedup	step time (ms)		speedup
		default	hierarchical		default	hierarchical	
8	8	293.152	293.081	100.02%	329.396	329.42	99.99%
16	16	350.679	350.061	100.18%	722.646	711.084	101.63%
32	32	471.976	470.263	100.36%	1584.937	1493.349	106.13%
64	64	630.907	627.41	100.56%	2907.05	2410.023	120.62%

Table 2 Performance comparison of hierarchical redistribution algorithm on different configurations of Pangu-α with 2 layers

The left part of Table 2 shows the end-to-end step time performance of our method when paired with automatic strategies. We observe a modest speedup that increases with the number of devices and levels of topological hierarchy. This is because the automatically generated strategies are chosen for performance and avoid introducing many large redistribution for this network. On the other hand, this test case showed that our algorithm does not have a negative side effect on the performance of the light redistribution cases.

In our second test case, we manually changed the parallel strategies so that a sequence of redistribution operators was inserted between computation operators. The changes of strategy were around the Attention block (the backbone part of a Transformer DNN) computation, specifically around its two BatchMatMul operators that computes the Attention for the Head of each data item. The right part of Table 2 shows the performance of our algorithm in this configuration. We observed no performance difference with 8 devices (0.01% is a tolerable measurement error), because the communication within the same node is still homogeneous (8 devices are in the same computing node). However, we observed increasing speedups (up to +20%) as the number of devices (thus nodes) increases. This is because the bandwidth factor becomes more significant between different computing nodes.

6 Related works

While communication optimization in the setting of DL training redistributions has been long overlooked and left from scientific papers to implementation details, it is only very recently that more interest is being brought to this problem. A few other new articles investigate it and we believe a lot of exciting progress is yet to be accomplished. As such the survey [10] on automatic parallelism in DNN training identifies the development of topology-aware communication as one of the main future challenges of the domain. D-Rec is one of the most promising parallelism strategy automatic generation algorithms w.r.t. its efficiency on strategy search speed [18] and scalability on large-scale neural networks and clusters [19]. However D-Rec did not provide a solution on the device assignment.

A hierarchy-aware algorithm based on profiling the different communication speeds (as opposed to our symbolic approach with the hypothesis that g_i/g_{i+1} is large) is proposed in [20]. Using another set of primitives and a slightly different syntax for tensor sharding, they develop heuristics to generate a set of candidate redistribution sequences and rank them using profiled communication costs. Another similar profiling-based algorithm [2] generates redistributions step-by-step by greedily choosing the fastest redistribution step based on a communication cost model until the whole redistribution is achieved.

Finally, while not topologically-aware, the algorithm described in [15] is another interesting approach at optimizing redistributions through minimization of memory usage (that has to be understood as minimizing the amount of data exchanged). They provide a theoretical analysis, based on formal semantics, which allows them to prove some optimality results under the hypothesis of uniform communication speeds.

7 Conclusion

In this paper, we presented a hierarchical redistribution algorithm that decides which communications to perform in which order when redistributing tensors between multiple devices in a hierarchical cluster. This algorithm explicitly takes into account the hierarchical nature of AI clusters in order to minimize the cost of redistributing data between devices. We experimented with this algorithm and we observed a speedup in a hierarchical scenario compared to the state-of-the-art redistribution algorithm implemented in the same open-source AI framework.

References

1. Abadi, M., et al.: TensorFlow: A System for Large-Scale Machine Learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, pp. 265–283. USENIX Association (2016)
2. Bian, Z., et al.: Colossal-ai: A unified deep learning system for large-scale parallel training. preprint arXiv:2110.14883 (2021)
3. Brown, T., et al.: Language models are few-shot learners. In: Advances in Neural Information Processing Systems, vol. 33, pp. 1877–1901. Curran Associates, Inc. (2020)

4. Cai, Z., et al.: TensorOpt: Exploring the Tradeoffs in Distributed DNN Training With Auto-Parallelism. IEEE Trans. on Parallel and Dist. Syst. **33**(8), 1967–1981 (2022)
5. Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al.: Large scale distributed deep networks. Advances in neural information processing systems **25** (2012)
6. Jouppi, N.P., et al.: TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. preprint arXiv:2304.01433 (2023)
7. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Commun. ACM **60**(6), 84–90 (2017)
8. Li, C., Hains, G.: A simple bridging model for high-performance computing. In: 2011 International Conf. on High Performance Computing & Simulation, pp. 249–256 (2011)
9. Li, C., Hains, G.: GPS: Towards Simplified Communication on SGL Model. In: IEEE International Parallel & Distributed Processing Symposium Workshops, pp. 727–736 (2014)
10. Liang, P., et al.: A Survey on Auto-Parallelism of Neural Networks Training. preprint techrxiv:19522414 (2022)
11. Liao, H., Tu, J., Xia, J., Liu, H., Zhou, X., Yuan, H., Hu, Y.: Ascend: a scalable and unified architecture for ubiquitous deep neural network computing. In: International Symposium on High-Performance Computer Architecture (HPCA), pp. 789–801 (2021)
12. Nvidia: NVIDIA DGX-1 With Tesla V100 System Architecture white paper. URL https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf
13. OpenAI: GPT-4 Technical Report (2023). URL https://cdn.openai.com/papers/gpt-4.pdf
14. Paszke, A., et al.: PyTorch: An Imperative Style, High-Performance Deep Learning Library. Curran Associates Inc., Red Hook, NY, USA (2019)
15. Rink, N.A., Paszke, A., Vytiniotis, D., Schmid, G.S.: Memory-efficient array redistribution through portable collective communication. arXiv preprint (2021)
16. Tong, Z., Du, N., Song, X., Wang, X.: Study on MindSpore Deep Learning Framework. In: 2021 17th International Conference on Computational Intelligence and Security (CIS), pp. 183–186 (2021)
17. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
18. Wang, H., Li, C., Tachon, T., Wang, H., Yang, S., Limet, S., Robert, S.: Efficient and systematic partitioning of large and deep neural networks for parallelization. In: European Conference on Parallel Processing, pp. 201–216. Springer (2021)
19. Wang, H., Tachon, T., Li, C., Robert, S., Limet, S.: SMSG: Profiling-Free Parallelism Modeling for Distributed Training of DNN. International Journal of Parallel Programming pp. 1–19 (2022)
20. Xie, N., Norman, T., Grewe, D., Vytiniotis, D.: Synthesizing optimal parallelism placement and reduction strategies on hierarchical systems for deep learning. Proceedings of Machine Learning and Systems **4**, 548–566 (2022)
21. Zeng, W., et al.: Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation. preprint arXiv:2104.12369 (2021)
22. Zhang, Z., Chang, C., Lin, H., Wang, Y., Arora, R., Jin, X.: Is network the bottleneck of distributed training? In: Workshop on Network Meets AI & ML, pp. 8–13 (2020)

Symposium on High-Level Parallel Programming & Applications
HLPP 2023

From array algebra to energy efficiency on GPUs [*]

Lenore M. R. Mullin

Abstract We present a new formulation for parallel matrix multiplication (MM) to out-perform the standard row-column code design. This algorithm is formulated in the MoA formalism (A Mathematics of Arrays, [6][2]) and combines an array view of hardware (*dimension-lifting* to extend indexing to physical memory/processing units), with a contiguous data layout derived from static transformations. This view of a hardware-software model is thus a *bridging model* in the sense of Valiant's BSP. OpenACC code was derived from the MoA expression's normal form, producing optimal block sizes using the static information of types and shapes. Experiments were run on Nvidia V100 GPUs and reveal energy consumption which is quadratic in N, i.e. linear in the size of matrix. More generally this approach is proposed for formulating, optimizing, and mapping array algorithms to hardware. This work builds upon recently published results of NREL scientists.

Keywords Mathematics of arrays · compilation · performance optimization · data- vs hardware shape · memory-processor layout

1 MoA Matrix Multiplication

Programmers are seeking high level languages and tools to formally describe, and ideally verify, algorithms in their domains while compiling to accelerators without requiring an extensive engineering background. Current research addresses this problem[1], with optimizations done after the specification of algorithms in a high level language like Python or C and use pragmas. These

[*] **Acknowledgments:** This work used Nvidia V100s at XPANSE through allocation Start-UP Grant CIS210035 from the Extreme Science and Engineering Discovery Environment (XSEDE), which was supported by National Science Foundation grant number #1548562. Special Thanks go to Dima Shyshlov, XSEDE Campus Scholar, for his help in organizing and running experiments and plots and to Mohamed Zahran for his expert advice on GPUs.

College of Engineering and Applied Sciences
University at Albany, SUNY
1400 Washington Avenue, Albany, 12222, USA
E-mail: lmullin@albany.edu

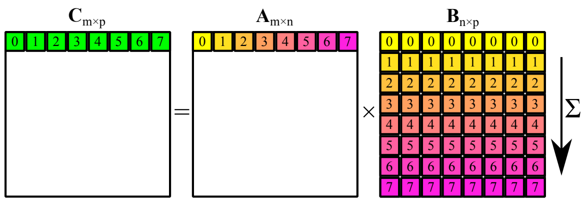


Fig. 1 Inherent parallelism of the MoA matrix-matrix multiplication (`GEMM`). The different colors and numbers on matrices **A** and **B** represent the pairings for the k index involved in the scalar-vector multiplication, e.g. the matrix element labeled 0 in **A** is multiplied with the row vector labeled 0 in **B**. A sum reduction is then performed over the resulting row vectors to yield the ith row in **C**.

incremental changes to existing theories and methods should continue. But, as suggested recently in CACM[3], there is room at the top for new ideas, and theories. We propose MoA (A Mathematics of Arrays [6][2]) to play this role of a formalism to bridge high-level functional algorithm descriptions with hardware and memory shapes and sizes. Experiments were run on Nvidia V100s while apriori theorizing about optimal block size using shapes and types of arguments.

The general matrix-matrix multiplication (`GEMM`) in MoA is a special case of the *inner product* for 2-D arrays (matrices), emphasizing that in the MoA MM **all** arrays are accessed contiguously. Define **A** as an $m \times n$ matrix, **B** as $n \times p$, and **C** as $m \times p$. In MoA notation, the shapes of $\mathbf{A}, \mathbf{B},$ and **C** are respectively $\rho\mathbf{A} = \langle m,\, n\rangle$, $\rho\mathbf{B} = \langle n,\, p\rangle$, $\rho\mathbf{C} = \langle m,\, p\rangle$ so the valid indices of the matrices that are bounded by shapes: $\forall\, i, j, k. \quad 0 \le i < m \quad 0 \le j < p \quad 0 \le k < n.$ The MoA Operational Normal Form (ONF) for `GEMM`[7,8] is:

$$\mathbf{C}[(i \times p) + j] := \sum_{k=0}^{n-1} \mathbf{A}[(i \times n) + k] \times \mathbf{B}[(k \times p) + j] \tag{1}$$

This is a "generic row-major form" whose meaning is that $\forall i : (0..m-1)$, $\forall j : (0..p-1)$, $\forall k : (0..n-1)$ the content of memory from the initial address **@C** of array C is:

$$@\mathbf{C} + (i \times p) + j := \sum_{k=0}^{n-1} (@\mathbf{A} + (i \times n) + k) \times (@\mathbf{B} + (k \times p) + j) \tag{2}$$

where **@A** (resp. **@B**) is the initial address of array A (resp. B). Let the following notation denote the 2-d Matrix Multiplication defined by MoA's **inner product** definition in (1),

$$\mathbf{C} = \mathbf{A} \bullet \mathbf{B} \tag{3}$$

In other words the theory's inner product operator on 2D arrays is equivalent to the definition of matrix multiplication.

Equation (1) is the generic code for a sequential program in MoA. Figure 1 illustrates the inherent **parallelism** of the MoA `GEMM` algorithm. In each ith row of the resultant array **C**, each scalar-vector operation involving the column

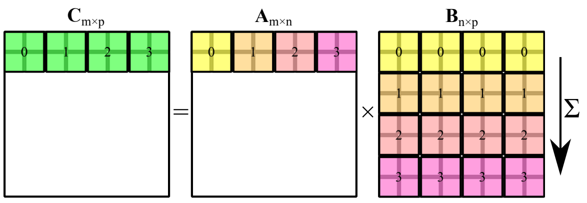


Fig. 2 Inherent parallelism of the blocked MoA general matrix-matrix multiplication (`GEMM`) algorithm. Just like the scalar `GEMM` design (Fig. 1, the block-block multiplications between different blocks in the row of matrix **A** and the corresponding rows of blocks in matrix **B** are independent of one another, and a sum reduction over the rows of blocks are performed to yield the final answer of blocks in matrix **C**.

index j is independent of each other. The ith row of **C** is contiguously filled in by the summation of scalar-vector multiplications involving each matrix element at the ith row and kth column of **A** (the scalar) with each kth row of **B** (the vector) obtained by accessing the arguments contiguously. A row-wise sum reduction is then applied over the k index to yield the final answer as the ith row in **C**.

Figure 2 shows how the *blocking* algorithm applies the inner product, Equation (3), to blocks in a round robin, row-major order, just like the scalar version in Figure 1, but this time summing blocks (subarrays) of partial sums. With each matrix block just large enough to fit in the L1 cache, each block operation is performed contiguously, round robin style, and efficiently. Our algorithm:

1. was symbolically derived from a functional specification and equivalence laws,
2. is a normal form exists and can be derived by confluent rewriting, much like parallel functional or skeleton-based programs,
3. its resulting normal form can be written in C to express parallel execution of loop-nests semi-explicit processor layout and memory-block accesses.

2 From Normal Form to C to Dimension Lifting

Dimension lifting is a way to abstract and unify an algorithm with the architecture it maps to. Through dimension lifting, all parallelism is revealed s.t. costs and optimizations become possible[5,4]. The C programs presented herein are augmented with OpenACC, noting that after parallelism is revealed, pragmas can be easily added with confidence of competitive performance with CUDA[4].

First the generic design in Equation (1) is implemented in a C program we call `ip.c` :

```
void ip(double *C, double *A, double *B,
        int sizel , int sizer , int np, int shr0)
 {int i,j,sigma;
  for (i=0; i<sizel; i++)
    {for (sigma=0; sigma<shr0; sigma++)
      {for (j=0; j<sizer; j++)
        {C[j+i*sizer]= C[j+i*sizer]
         + A[(i*shr0)+sigma] * B[(sigma*sizer)+j];      }}}}
```

Then dimension lifting over the rows of A and C, the i loop, reveals parallelism and assigns an index to processors. This was done by code we call `ip_rows.c`:

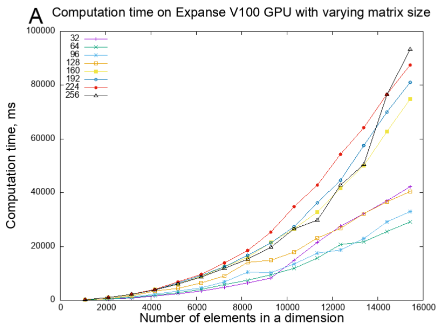
```
   void ip_rows(double *C, double *A, double *B,
            int sizel, int sizer, int np, int shr0)
 {int i,j,k,ip,sigma;
   for (k=0; k<np; k++)
    {for (ip=0; ip<(sizel/np); ip++)
     {for (sigma=0; sigma<shr0; sigma++)
      {for (j=0; j<sizer; j++)
      { C[j+(ip+(sizel/np)*k)*sizer]= C[j+(ip+(sizel/np)*k)*sizer]
        +A[((ip+((sizel/np)*k))*shr0)+sigma]
        *B[(sigma*sizer)+j];                                   }}}}}
```

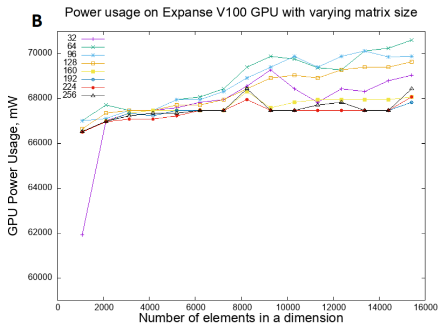
Dimension lifting over the columns of B reveals parallelism also. This would break up the j loop as illustrated in the code below. Mapping each row of B could be in groups of 8 e.g. to a vector register or a group of threads. Finally, the sigma loop is broken up creating the block. This necessitates another addition loop to add up the blocks realizing Figure 2.

```
void ip_cols(double *C, double *A, double *B,
          int sizel, int sizer, int np, int shr0,int rsize)
 {int i,j,jp,kp,sigma;
  for (i=0; i<sizel; i++)
   {for (sigma=0; sigma<shr0; sigma++)
         {for (jp=0; jp<(sizer/rsize); jp++)
           {for (kp=0; kp<rsize; kp++)
      { C[((jp*rsize)+kp)+i*sizer]= C[((jp*rsize)+kp)+i*sizer]
        + A[(i*shr0)+sigma] * B[(sigma * sizer)+((jp*rsize)+kp)];
           }}}}}
```

3 Performance measurements

We measured speed and energy consumption as a function of block sizes and matrix sizes. The best time was achieved with a 32KB by 32KB block size changing to 64KB by 64KB. Best time block size was correlated to best Energy block size. The worst Time and Energy block sizes occurred when Power and Heat were the best. A goal of this research is to identify how shapes can predict execution time as a function of the array (dimension) size and help obtain a linear relationship. Here are some meaningful time and energy measurements for a V100 GPU with L1 cache size x = 128 (KiB), L2 cache size y = 6 (MiB), Global memory size z = 32 (GiB), Number of SMs N_{SM} = 80.

A


B


In plot A, a 32 by 32 block, the purple graph, is sufficient for keeping time linear (and minimal) until size about 9000. After that point its slope becomes polynomial, and to maintain the linear performance a 64 by 64 block size (green) can be chosen. For power vs an inverse relationship is observed: the fastest, purple and green, graphs require the most power. Consequently, it is possible to adjust blocksize given the size of the memory and size of the data components.

4 Conclusion

A new matrix multiplication algorithm was presented with blocking described in terms of *dimension lifting*, a term coined for the MoA (mathematics of arrays) theory, that formalizes splitting of indices to give an index to an architectural component, thus increasing the dimension of the algorithm. Dimensions increase because MoA views the algorithm *and* architectural resources in a uniform Cartesian way. This approach internalizes the target hardware shape in the array formalism and allows formal, high-level transformations to optimize generated parallel code. Reproducible results imply that MoA's contiguous view of memory accesses outperforms the best efforts to optimize the classical design[7, 8].

References

1. Chi, Y., Qiao, W., Sohrabizadeh, A., Wang, J., Cong, J.: Democratizing domain-specific computing. CACM **50**(1), 74==85 (2023). DOI 10.1145/3524108
2. Hains, G., Mullin, L.M.R.: Parallel functional programming with arrays. The Computer Journal **36**(3), 238–245 (1993)
3. Leiserson, C.E., et. al.: There's plenty of room at the top: What will drive computer performance after Moore's law? Science **368** (2020)
4. Mullin, L., Phan, W.: A transformational approach to scientific software: The mathematics of arrays (moa) fft with openacc. https://www.openacc.org/events/openacc-summit-2021 (2021)
5. Mullin, L., Rutledge, E., Bond, R.: Monolithic compiler experiments using c++ expression templates. https://slideplayer.com/slide/8543511/ (2002)
6. Mullin, L.M.R.: A mathematics of arrays. Ph.D. thesis, Syracuse University (1988)
7. Thomas, S., Mullin, L., Swirydowicz, K.: Improving the performance of DGEMM with MoA and cache-blocking. In: Proceedings of ARRAY'21: ACM Symposium on Array Programming. ACM, NY, NY (2021)
8. Thomas, S., Mullin, L., Swirydowicz, K., Khan, R.: Threaded multi-core gemm with moa and cache-blocking. In: 19th International Conference on Scientific Computing (CSC'21). Las Vegas, Nevada (2021)