12th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures

10th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms

PARMA-DITAM 2021, January 19, 2021, Budapest, Hungary

Edited by
João Bispo
Stefano Cherubin
José Flich

Editors

João Bispo
University of Porto, Portugal
jbispo@fe.up.pt

Stefano Cherubin
Codeplay Software Ltd, London, United Kingdom
stefanix@acm.org

José Flich
Universitat Politècnica de València, Spain
jflich@disca.upv.es

ACM Classification 2012
Hardware → Reconfigurable logic and FPGAs; Software and its engineering → Compilers; Computer systems organization → Parallel architectures; Theory of computation → Concurrency

ISBN 978-3-95977-181-8

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-181-8.

Publication date
March, 2021

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

Digital Object Identifier: 10.4230/OASIcs.PARMA-DITAM.2021.0

ISBN 978-3-95977-181-8 **ISSN 1868-8969** **https://www.dagstuhl.de/oasics**

OASics – OpenAccess Series in Informatics

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

ISSN 1868-8969

https://www.dagstuhl.de/oasics

In a moment of exceptional changes,
We remember all the people we lost.
May they inspire us to look ahead
To the new opportunities we found.

Contents

12th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 10th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2021).
Editors: João Bispo, Stefano Cherubin, and José Flich
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume collects the papers presented at the 12^{th} Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures, and the 10^{th} Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2021). The workshop is co-located with the 2021 edition of the HiPEAC conference and was held on January 19, 2021. Although the workshop was originally planned to take place at Budapest, Hungary, due to the COVID-19 pandemic it switched to a virtual online event.

The current trend towards many-core and the emerging accelerator-based architecture requires a global rethinking of software and hardware design. The PARMA-DITAM workshop focuses on many-core architectures, parallel programming models, design space exploration, tools and run-time management techniques to exploit the features of such (heterogeneous) many-core processor architectures from embeddded to high performance computing platforms.

The scope of the PARMA-DITAM workshop include the following topics:

- Parallel programming models and languages, compilers and virtualization techniques
- Runtime adaptivity, runtime management, power management and memory management
- Heterogeneous and reconfigurable many-core architectures and design space exploration
- Design tools and methodologies for many-core architectures
- Parallel applications for many-core platforms
- Architectures and compiler techniques to accelerate deep neural networks

12th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 10th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2021).
Editors: João Bispo, Stefano Cherubin, and José Flich

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

Towards Adaptive Multi-Alternative Process Network

Hasna Bouraoui
Technische Universität Dresden, Germany

Chadlia Jerad
University of Manouba, Tunisia
University of Carthage, Tunis, Tunisia

Jeronimo Castrillon
Technische Universität Dresden, Germany

Abstract

With the increase of voice-controlled systems, speech based recognition applications are gaining more attention. Such applications need to adapt to hardware platforms to offer the required performance. Given the streaming nature of these applications, dataflow models are a common choice for model-based design and execution on parallel embedded platforms. However, most of today's models are built on top of classical static dataflow with adaptivity extensions to express data parallelism. In this paper, we define and describe an approach for algorithmic adaptivity to express richer sets of variants and trade-offs. For this, we introduce multi-Alternative Process Network (mAPN), a high-level abstract representation where several process networks of the same application coexist. We describe an algorithm for automatic generation of all possible alternatives. The mAPN is enriched with meta-data serving to endow the alternatives with annotations in terms of a specific metric, helping to extract the most suitable alternative depending on the available computational resources and application/user constraints. We motivate the approach by the automatic subtitling application (ASA) as use case and run the experiments on an mAPN sample consisting of 12 randomly selected possible variants.

2012 ACM Subject Classification Theory of computation → Streaming models

Keywords and phrases High level process network, algorithmic adaptivity, automatic subtitling application

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2021.1

1 Introduction

With the proliferation of digitalization and the easy access to storage capacity, large volumes of audio data including broadcasts and meetings are increasingly generated and stored. As a result, a growing need for automated processing of human language has emerged, leading to the advent of many audio processing applications. One example is Audio Indexing, enabling the search and retrieval of *who spoke what* from an audio source. Another example is the Automatic Subtitling Application (ASA)[1, 11], where Speaker Recognition (SpkR), Speech Recognition (SpR), and Speaker Diarization (SD) are all combined. For the design of such complex applications, embedded programmers need to understand algorithmic variants implementing the same functionality (i.e. algorithmic adaptivity) and how can they be deployed in parallel into possibly different many-core platforms (i.e. parallelism adaptivity). In any given situation, characterized by available hardware resources and application/user constraints, it is difficult to manually select the adequate algorithm while delivering the required performance.

These streaming applications can exploit the parallelism of embedded many-core platforms especially when described using appropriate Models of Computation (MoC [18]). For example, Synchronous Dataflow (SDF) work well for describing one particular algorithmic variant.

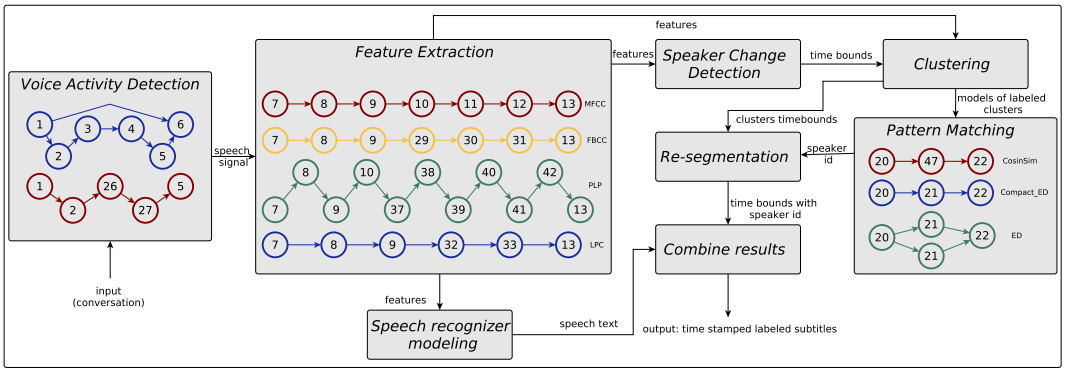


Figure 1 Representation of the Automatic Subtitling Application.

However, it is ill-suited to express the adaptivity required today. For static models (e.g., Cyclo-Static Dataflow (CSDF) [4] and Parameterized Synchronous Dataflow (PSDF) [3]) adaptivity is at the token production and consumption levels. However, Dynamic models allow for topology updates. A prominent example is Scenario-Aware Dataflow (SADF) [21] which expresses behavioral adaptivity by modifying the application graph or varying the amount of parallelism. However, this model suffers from limitations with regard to the size and complexity of the model itself [5, 12]. Otherwise, one option would be to have a different graph for every single variant of the ASA algorithms to be arbitrated at run time. Obviously, such a solution may not be practical and would lead to a combinatorial explosion on top of being prohibitively complex to manage. In this paper, we propose a novel model for compact representation and exploration of multiple algorithmic variants in one single-source specification. Our model, called mAPN, extends the well-known Kahn Process Network (KPN) [15] model. mAPN carries annotations on metrics and enables automatic exploration of the different implementations to choose a well-suited variant. The main contributions of this paper are:

- We introduce mAPN to capture multiple algorithmic variants, beyond what existing models allow, in a compact single-source specification (cf. Sec. 3.1).
- We present an algorithm for the automatic exploration of several variants based on mAPN annotations and the constraints introduced by the designer (cf. Sec. 3.2).
- We demonstrate the analysis fidelity of our approach for the ASA use case. cf. Sec. 4

2 Motivational example

In this paper we use ASA as a case study to demonstrate the need for parallelism and algorithmic adaptivity. ASA is a complex application that combines the functionality of SpkR, SD and SpR, to recognize who is speaking, when s/he is speaking, and what s/he is saying, respectively. The different functionalities share common phases (e.g., Feature Extraction (FE)), as well as common algorithms serving different phases (i.e. Classification, Pattern Matching (PM), and Speaker Change Detection (SCD)). Fig. 1 illustrates a coarse-grained representation of ASA. To better grasp the scale at which the number of possible implementations can grow, we will detail the VAD, FE, and PM phases. For FE, speaker characteristics can be categorized based on different features. Prominent algorithms for feature extraction are MFCC, FBCC, PLP, and LPC [24, 23, 17]. These possible implementations are schematically shown in Fig. 1. For each particular phase, processes with the same number represent common nodes of these algorithms. The PM phase exhibits three possible

implementations: Euclidean Distance (ED) and Cosine Similarity (CS) (i.e. compact, or expanded if we consider parallelism adaptivity). In absolute terms, no variant is better than all the others. Which variant is best depends on the application/user constraints and the desired target hardware.

Depending on these constraints, one has to specialize the phases for each functionality. Phase reuse and algorithmic choices create a large space of possible variants for ASA. Some researchers use classical techniques that are accurate and can run on embedded hardware with resource constraints. Other approaches are based on deep learning, requiring large databases. This gain in complexity makes it hard from a developer point of view to choose the right implementation of the application beforehand. This becomes even harder considering the diversity of possible target hardware, as the achieved performance of an implementation may significantly differ from one hardware to another. In any given situation, it is difficult to manually select an adequate algorithm under application/user/HW constraints. To ease development, applications such as ASA are not written from scratch, but are built from existing implementations. This is known as the Algorithm Selection Problem [19], which shifts the burden from finding the right solution to identifying and composing appropriate existing algorithms. In this paper we propose a novel model to support designers in this process, named mAPN.

3 mAPN

3.1 mAPN Formalization

A KPN is a directed graph composed of a set of concurrent processes (nodes), communicating through unidirectional unbounded First In First Out (FIFO) channels (edges) having blocking reads and non blocking writes semantics. Formally, a KPN is a tuple $G = (P, Ch)$, where P is a set of processes, and $Ch \subseteq P \times P$ a set of channels. A Multi-Alternative Process Network (mAPN) is a graph that concisely represents many different KPNs. We use colors to tag and then generate all possible alternatives. Let $\Xi = \{\xi^1, \xi^2, ..., \xi^n\}$ be the set of colors. $\xi^* \in \Xi$ denotes a particular *neutral* color and $\mathcal{P}(\Xi)$ the power set of Ξ. Similar to KPNs, an mAPN is a directed graph composed of processes and channels. However, and unlike KPNs, channels are annotated with colors indicating the local alternatives that the channel belongs to. Formally,

▶ **Definition 1** (mAPN). *A multi-Alternative Process Network is a tuple $G = (P, Ch, col)$, where P is a set of processes, Ch is a set of channels $Ch \subseteq P \times P$ and col is a function that maps each channel to a subset of colors, that is $col : Ch \rightarrow \mathcal{P}(\Xi)$.*

Let $wr, rd : Ch \rightarrow P$ be two functions that map each channel to the process that writes into it, respectively reads from it. In a similar way, we define $\widehat{wr}, \widehat{rd} : P \rightarrow \mathcal{P}(Ch)$ that map each process into a subset of Ch it writes to, respectively, it reads from (i.e., $\widehat{wr}(p) = \{ch \in Ch, wr(ch) = p\}$). We call *source processes* (Src) the subset of P that do not read from any channel. Analogously, *sink processes* (Snk) do not write to any channel. That is, $Src = \{p \in P, \widehat{rd}(p) = \emptyset\}$, and $Snk = \{p \in P, \widehat{wr}(p) = \emptyset\}$. The function $\widehat{col}_{rd}$ returns the colors a process reads from. Analogously, $\widehat{col}_{wr}$ returns the colors a process writes to (i.e., $\widehat{col}_{wr}(p) = \cup_{ch \in \widehat{wr}(p)} col(ch)$). In the sample mAPN of Fig. 2, the colors of the channel that connects process v to x (ch_v^x), has colors {■, ■}, while $col(ch_b^p) = \{$■$\}$. Note also that $Src = \{a, v\}$, $Snk = \{u\}$ and $\widehat{col}_{wr}(p) = \{$■, ■, ■$\}$.

A colored subgraph $G^\xi = (P^\xi, Ch^\xi)$ of an mAPN $G = (P, Ch)$ gathers all the channels having the same color ξ, and the processes connected to them. The neutral color ξ^* (black color in Fig. 2) marks the *nominal end-to-end* implementation of the application

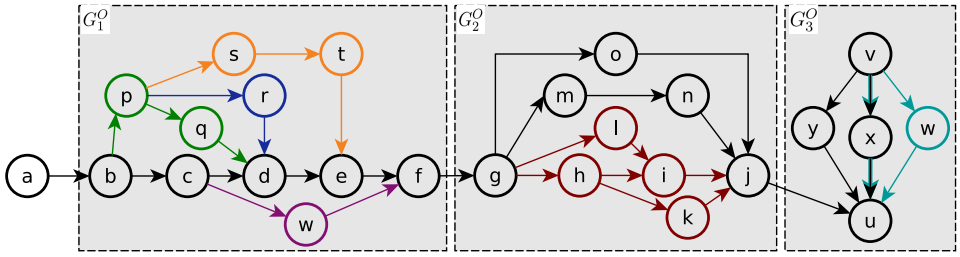


Figure 2 mAPN of a synthetic example.

(cf. Definition 2). Colored subgraphs represent local alternatives. For example, the purple subgraph connecting processes c and f is a local alternative to the black subgraph, replacing the functionality of d and e. For clarity reasons, nodes in the figure receive the colors of the local alternative they belong to. Local alternatives can be nested, as is the case of the blue and green ones. Consequently, the flow from b to d can go through c (black), through p and q (green), or through p and r (composition of green and blue).

▶ **Definition 2** (Nominal alternative). *A nominal alternative* $G^{\xi^*} = (P^{\xi^*}, Ch^{\xi^*})$ *of an mAPN* $G = (P, Ch, col)$ *is an end-to-end subgraph of* G *where* $\forall ch \in Ch_n, \xi^* \in col(Ch_n)$, *and* $Src, Snk \subset P^{\xi^*}$.

We distinguish processes from/at which local alternatives fork/join. These processes are important since they identify anchor nodes for generating alternatives. We classify them into the following subsets:

- $\mathcal{F}$ is the subset of processes of P that write different channels with different colors. Formally, $\mathcal{F} = \{p \in P, \exists ch_i, ch_j \in \widehat{wr}(p), i \neq j, col(ch_i) \neq col(ch_j)\}$.
- $\mathcal{J}$ is the subset of processes of P that read different channels with different colors. Formally, $\mathcal{J} = \{p \in P, \exists ch_i, ch_j \in \widehat{rd}(p), i \neq j, col(ch_i) \neq col(ch_j)\}$.

In the mAPN of Fig. 2, $\mathcal{F} = \{b, c, p, v, g\}$ and $\mathcal{J} = \{d, e, f, j, u\}$. Based on the nominal alternative and the collection of local alternatives forming an mAPN, one can generate all possible alternatives. Generation is based on a set of assumptions that the mAPN is a well-formed. Concretely:

▶ **Definition 3** (Well-formed mAPN). *A well-formed mAPN* $G = (P, Ch, col)$ *has the following properties: (i) existence of a nominal alternative, (ii) every sink node of a colored subgraph is a join node, (iii) every source node of a colored subgraph is a fork node, (iv) colors cannot be re-used in disjoint subgraphs, and (v) preserving KPN semantics (that is, for a fork node, the number of write channels per color must be the same).*

KPN semantics preservation does not include a condition over the number of read channels per color for a join node. Process u in Fig. 2 is a counter example, as we have two branches of alternatives coming from two distinct source nodes.

We structure an mAPN so that alternatives are created from products over the sets of generated sub-KPNs. These subsets are generated from mAPN subgraphs, and we call them *closed*. Nesting alternatives occur only within such subgraphs. Three subgraphs are marked by dashed gray boxes in Fig. 2, each one itself, a well-formed mAPN. By adding anchor nodes to every element of the product of sub-KPNs, we get the set of all possible alternatives. We define a closed mAPN as follows:

Algorithm 1 Generation of Kahn Process Networks.

param: $G : (P, Ch, col), M, V, C$
return: $KPNs : \{kpn_i\}_i$

```
1:  procedure AltGen
2:      KPNs, KPN'_s ← ∅
3:      {G_i^O}_i ← getClosedGraphs(G)
4:      (P_com, Ch_com) ← rmCommon(G, {G_i^O}_i)
5:      for every G_i^O do
6:          KPN'_s ← AltGenRecur(G_i^O)
7:          KPNs ← mixNmatch(KPNs, KPN'_s)
8:      for kpn_i = (P_i, Ch_i) ∈ KPNs do
9:          kpn_i ← (P_i ∪ P_com, Ch_i ∪ Ch_com)
10:         v = ν(kpn_i, M, V)                    ▷ Metrics evaluation of kpn_i
11:         if v ⊭ C then                         ▷ Checking constraints
12:             KPNs ← KPNs \ {kpn_i}
13:     return KPNs
```

▶ **Definition 4** (Closed mAPN). *A closed mAPN $G^O = (P^O, Ch^O, col^O)$ of a well formed mAPN $G = (P, Ch, col)$ has the following properties: (i) G^O is a well formed mAPN, $Src^O \subset \mathcal{F}$ and $Snk^O \subset \mathcal{J}$, (ii) no loop backs are allowed across closed mAPNs, (iii) a closed mAPN is minimal, that is it cannot include a composition of two or more closed sub-mAPNs, and (iv) the set of colors that fork within a closed subgraph is the same that joins.*

3.2 Exploration algorithm

Algorithm 1 describes the generation process by: (i) extracting closed graphs (line 3), (ii) generating the set of KPNs for each one and mixing and matching them (lines 5–7), (iii) adding anchor nodes for each partially constructed KPN (line 9). After adding adding anchor nodes, the KPN is complete and can then be evaluated (line 10) using the rules of aggregation, and checked against the given constraints (line 11). If the constraints are not satisfied, that particular KPN is removed from the set of alternatives to return (line 12).

The recursive procedure `AltGenRecur()` (Algorithm 2) visits source nodes (line 4) and generates a set of KPNs for each one. Given a source process p, the algorithm iterates over the local alternatives, i.e., colors of the channels p writes to (`for` in line 6). For every colored subgraph (kpn_ξ) having p as source (line 7), if kpn_ξ reaches sink processes without including any fork, then it is a complete generated local KPN (lines 8, 9), and is added to the set of generated KPNs starting from p (line 19). This is the example of the purple colored subgraph starting from the fork source c of G_1^O. Otherwise, a recursive call over the computed subgraph G' (line 17) will return alternatives, to be mixed and matched (line 18). Computing G' depends on whether there are nested alternatives, or a sink of G is not reached. In the first case, G' is the subgraph of G that starts from the immediately encountered fork nodes of the colored subgraph kpn_ξ (lines 11-14). For example, if we consider the green colored subgraph, G' will be the entire subgraph of G_1^O that has process p as source. In the second case, i.e. no sink is reached, G' is the subgraph that starts from sink processes of kpn_ξ (lines 16). This is the case of the three colored subgraphs of G_1^O starting from process p. `mixNmatch()` takes two sets of KPNs and computes all possible combinations of their elements (i.e., set product).

Algorithm 2 Recursive generation of Kahn process networks.

param: $G \ : \ (P, Ch, col)$
return: $KPNs : \{kpn_i\}_i$

procedure AltGenRecur
  $KPNs, KPN_s^p \leftarrow \emptyset$
  $Src \leftarrow \{p \in P, \widehat{rd}(p) = \emptyset\}$
  for $p \in Src$ **do**
    $KPN'_s \leftarrow \emptyset$
    for $\xi \in \widehat{col}_{wr}(p)$ **do** ▷ Iterate over colors
      $kpn_\xi = (P_\xi, Ch_\xi) \leftarrow subgraph(G, \{p\}, \xi)$
      if $(P_\xi \cap F = \{p\}) \wedge (Snk_{/kpn_\xi} \subseteq Snk)$ **then**
        $KPN'_s \leftarrow \{kpn_\xi\}$
      else
        if $(P_\xi \cap F \neq \{p\})$ **then** ▷ Nested alt.
          $\{p_j\}_j \leftarrow getClosest(p, P_\xi \setminus \{p\} \cap F)$
          $G' \leftarrow subgraph(G, \{p_j\}_j)$
          $kpn_\xi \leftarrow kpn_\xi \setminus \{kpn_\xi \cap G'\}$
        else ▷ No sink is reached
          $G' \leftarrow subgraph(G, Snk_{/kpn_\xi})$
        $KPN'_s \leftarrow AltGenRecur(G')$
        $KPN'_s \leftarrow mixNmatch(\{kpn_\xi\}, KPN'_s)$
      $KPN_s^p \leftarrow KPN_s^p \cup KPN'_s$
    $KPNs \leftarrow mixNmatch(KPNs, KPN_s^p)$
  return $KPNs$

4 Evaluation

4.1 The mAPN model of ASA

We demonstrate our approach on an ASA application. Recall the coarse-grained graph presented in Fig. 1. The figure shows an example of existing implementations and commonalities across them, characterizing the large design space of algorithmic variants for this application. Several approaches and algorithms for each phase can be found in the literature. Readers may refer to [6] for a detailed survey. In this section we explain the most common algorithms for SpkR, SR and SpR, and how they can be combined in one graph to create a different ASA implementation. Each phase in Fig. 1 is replaced by one or more possible implementations using different colors. Fig. 3 represents one possible compact graph for the ASA application. The voice activity detection (VAD) has two different variants while FE has 7. By mixing and matching these two phases, we generate 14 possible variants implementing the part ending at node 13. So far, we have excluded adaptive parallelism in the variants. To exploit Task-level parallelism (TLP), expanded/compacted versions of a Process Node (PN) can be added as an additional algorithmic variants. This is similar to the work presented in [20]. The algorithm that implements the MFCC FE can be executed by one PN (i.e. Node 28), or expanded over several PNs (5 PNs:9-13). The same process can be applied to all the local variants of the FE phase leading to 7 additional algorithmic variants just for this phase. Similarly, Data-level parallel (DLP) versions of some phases can be deployed to balance the load across application phases, as seen in [16]. For example, in the pattern matching phase (node 15), a new alternative can be created where multiple *Euclidean Distance (ED)* nodes run in parallel and then send the results to the *merge* node. Every added possibility implementing a local phase in the compact graph greatly enriches the space of implementation variants. After only adding the discussed TLP/DLP alternatives, the number of possible variants reaches 672.

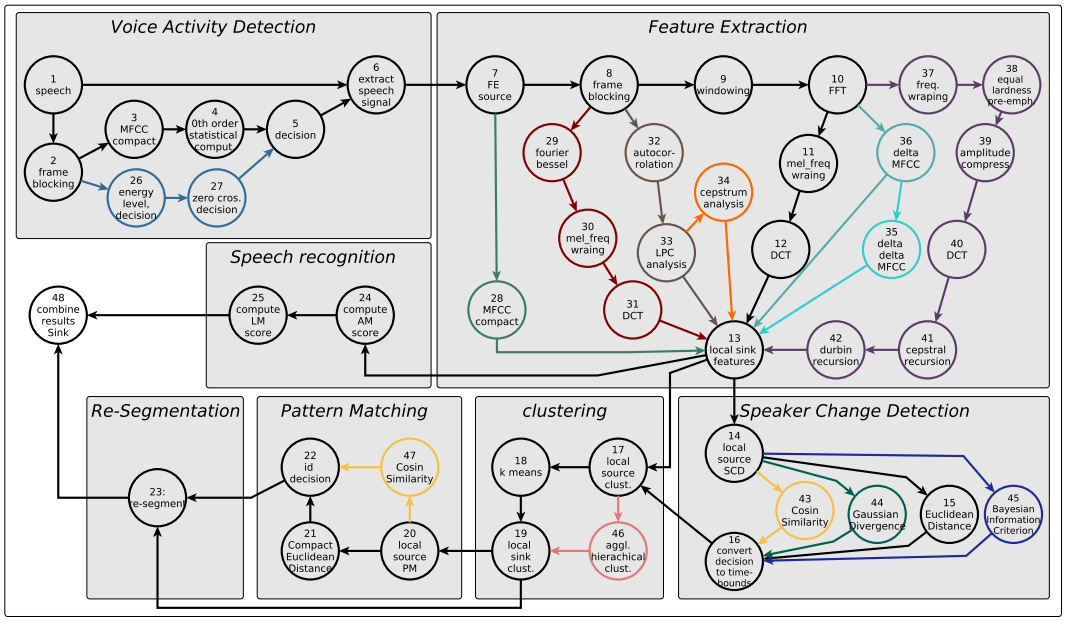


Figure 3 Multi-Alternative Process Network for ASA.

4.2 Experimental Results

For the experimentation, we implemented for the VAD the black alternatives corresponding to an MFCC-based VAD as presented in Fig. 3. In phase FE, we explore parallelism adaptivity by applying TLP to the implementation of the MFCC algorithm (compact version, 28, vs. expanded version, 8 to 13), and algorithmic adaptivity by adding an FBCC implementation (9-29-30-31). SCD phase presents algorithmic adaptivity: ED (node 15) vs. CS (43); while the k-means algorithm (18) is used for clustering. Finally, DLP is applied to the PM phase by varying the number of ED-Compact nodes (21) to be either 1 or 4. We implemented these algorithms and end up with 12 possible alternatives in total (combinations of 32 different process ids).

We explore the space of alternatives to select feasible ones and compare it to a brute force approach, where all implementations are generated and executed. As an evaluation metric, we use the execution time as a metric under a constraint of *55s*. To factor the errors introduced by estimation, we use actual measurements using the estimator of the SLX tool suite[1] on the target hardware. Based on these estimations, evaluation of alternatives are performed using the max-plus algebra traditionally used for timing analysis of dataflow graphs [13]. Experiments are made on a speech of 5 minutes of length, and considering two platforms: Odroid XU4[2] and a GPP[3] (GPP).

As shown in Table 1, column *Estimations* contains the execution time estimates on both platforms. Each assessed alternative is then considered feasible (✓), or rejected (✗) depending on whether it respects the defined constraint or not (not exceeding 55s). Under column *Real Exp.Results*, we present the experimentally measured wall-clock time for these alternatives. The column *R/F* (right/false decision) lists the correctness of the decision

[1] `https://www.silexica.com/`
[2] Exynos 5422 big.LITTLE chip: 4 Cortex-A15 and 4 Cortex-A7 cores
[3] 3.40GHz quad-core Intel(R) Core(TM) i7-6700 CPU

Table 1 Experimental results.

	Estimations				Real Exp. Results			
	Odroid		GPP		Odroid	R/F	GPP	R/F
Alt 1: {VAD-FE(Bessel)-SCD(ED)-Cl-PM(DLP-4)}	102.36	✗	26.24	✓	56.43	R	20.66	R
Alt 2: {VAD-FE(Exp.MFCC)-SCD(ED)-Cl-PM(DLP-4)}	52.11	✓	11.53	✓	24.43	R	9.94	R
Alt 3: {VAD-FE(Comp.MFCC)-SCD(ED)-Cl-PM(DLP-4)}	51.80	✓	10.56	✓	23.66	R	9.66	R
Alt 4: {VAD-FE(Bessel)-SCD(ED)-Cl-PM(DLP-1)}	115.20	✗	30.74	✓	67.69	R	25.73	R
Alt 5: {VAD-FE(Exp.MFCC)-SCD(ED)-Cl-PM(DLP-1)}	64.95	✗	16.03	✓	53.78	F	20.54	R
Alt 6: {VAD-FE(Comp.MFCC)-SCD(ED)-Cl-PM(DLP-1)}	64.64	✗	15.06	✓	52.78	F	19.51	R
Alt 7: {VAD-FE(Bessel)-SCD(CS)-Cl-PM(DLP-4)}	102.35	✗	26.23	✓	45.72	F	20.93	R
Alt 8: {VAD-FE(Exp.MFCC)-SCD(CS)-Cl-PM(DLP-4)}	52.10	✓	11.52	✓	23.10	R	10.05	R
Alt 9: {VAD-FE(Comp.MFCC)-SCD(CS)-Cl-PM(DLP-4)}	51.79	✓	10.55	✓	23.68	R	9.32	R
Alt 10: {VAD-FE(Bessel)-SCD(CS)-Cl-PM(DLP-1)}	115.19	✗	30.73	✓	56.88	R	20.42	R
Alt 11: {VAD-FE(Exp.MFCC)-SCD(CS)-Cl-PM(DLP-1)}	64.94	✗	16.02	✓	53.74	F	19.94	R
Alt 12: {VAD-FE(Comp.MFCC)-SCD(CS)-Cl-PM(DLP-1)}	64.63	✗	15.05	✓	55.47	R	20.01	R

made about the feasibility of each alternative. By using a time constraint of 55s, Table 1 shows that estimations reaches a success rate of 66.66% and 100% on Odroid and GPP respectively. Besides being able to decide on which alternative to use, we are also interested on how fast such a decision can be made. For the *brute force* approach, and considering the Odroid platform, choosing among the 12 alternatives requires an execution time of 537.37s. By extrapolating to 672 alternatives, this would correspond to around 8h. Conversely, the mAPN approach only requires to assess 32 different nodes to cover the 12 alternatives which takes 206.2s. This extrapolates to 16m considering the 672 alternatives (covered by 49 nodes as in Fig. 3). We notice that the estimations are generally slower than the real results, which can be explained by the fact that we are not taking into consideration the pipelining parallelism hidden in the KPN dataflow graph. In addition, using better assessment leads to more accurate estimation. However, this is kept out of the scope of this paper.

Further analysis is done to show the fidelity of the estimation. Fig. 4 illustrates the correlation plot between the estimated cost of an alternative and the actual execution time of the real implementation. The grey area in the plot represents the 95% confidence intervals of the results following the smoothing method. We also sort the results of the 12 alternatives and compute the rank correlation to measure the ordinal association between them. This results in 0.936 and 0.802, using the *Spearman's* ρ and *Kendall's* τ methods respectively. Recall that 1 represents a perfect alignment between the two rankings. For both methods, very high agreement levels are achieved, proving that our mAPN, ensure an acceptable ordering of the alternatives in terms of the considered metric. Even with the noticed deviation between the estimations and the measures, we are still able to say which alternative is better than the other without a time consuming evaluation of all alternatives in the target hardware. This helps the user to decide on the feasible and adequate ones in a large space of variants.

5 Related work

Many research works propose methods for parallelism adaptivity. Flextream [14] studied adaptable compilation of a running application to the changing hardware characteristics. Adaptivity is achieved by replicating stateless processes to form a more fine-grained graph. Authors in [22] went further and proposed actor merging for sequentially executing processes. Likewise, Lee et al [7] proposed an optimal method to adapt the running application to available resources. Optimality refers to memory footprint under core-count constraints to

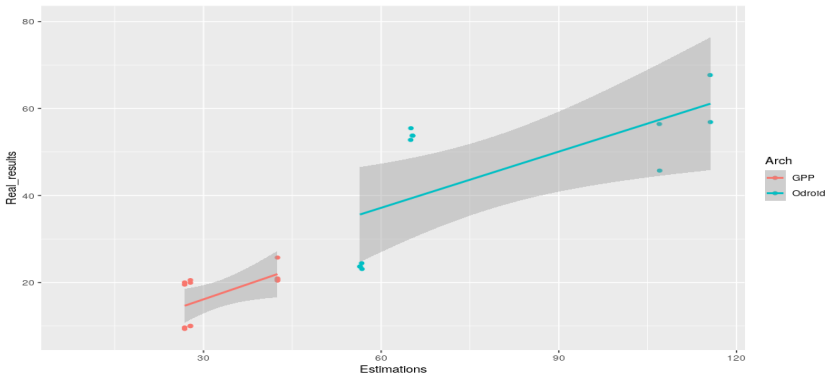


Figure 4 Fidelity analysis of the mAPN.

reduce energy consumption. [10] extended the semantics of SDFs to increase expressiveness and enable the specification of dynamic reconfigurable signal processing applications. They exploit static and adaptive task, data and pipeline parallelism. The work in [9] determines the minimal required performance of the hardware to be used for Macro Dataflow. Similarly, authors in [16], vary parallelism at run-time for KPN applications by duplicating stateless processes. Previous works considered one possible implementation of a particular application. They expressed adaptivity by breaking its tasks further down, or consolidate them into less tasks if few hardware resources are available. They do not support deeper implementation changes where different algorithms can be used to perform the same task. Authors in [8] present a methodology that allows for multiple algorithmic variants for a given actor in an application. These algorithmic variants are passed as metadata to the compiler, which selects the best implementation for a given platform. Authors in [2] introduce a new programming language to ensure algorithmic choice at the language level. They provide algorithmic choices to the compiler. However, these two approaches do not support adapting the graph topology. Our approach is more general, combining parallelism adaptivity with algorithmic adaptivity.

6 Conclusion

In this paper we introduced mAPN, a novel model where multiple algorithmic variants are concisely represented in one compact graph. The abstract high-level model we proposed is built on top of KPN, taking advantage of its formal properties. Provided with additional information (annotation in terms of chosen metric), this model enlarges the variant space and eases the process of retaining feasible variants while meeting application/user/HW constraints. Furthermore, our approach is generic, combining parallelism adaptivity with algorithmic adaptivity. We motivated the mAPN approach with variants of ASA. We evaluated the end-to-end paths of the co-existing algorithmic variants in the graph using annotation on processes in terms of execution time. Being able to reason about these metrics and algorithmic adaptivity with a concise model will be key to achieve the most suitable implementation in terms of considered application/user/HW constraints. In future work, we will investigate efficient run-time algorithmic switching mechanisms, and considering annotation over more abstract domain-specific metrics like accuracy or robustness.

References

1 C. Aliprandi, C. Scudellari, I. Gallucci, N. Piccinini, M. Raffaelli, A. del Pozo, A. Alvarez, Ha. Arzelus, R. Cassaca, T. Luis, et al. Automatic live subtitling: state of the art, expectations and current trends. In *Proceedings of NAB Broadcast Engineering Conference: Papers on Advanced Media Technologies, Las Vegas*, page 23, 2014.

2 Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. *ACM Sigplan Notices*, 44(6):38–49, 2009.

3 Bishnupriya Bhattacharya and Shuvra S Bhattacharyya. Parameterized dataflow modeling for dsp systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, 2001.

4 G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

5 Adnan Bouakaz, Pascal Fradet, and Alain Girault. A survey of parametric dataflow models of computation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(2):1–25, 2017.

6 Hasna Bouraoui, Chadlia Jerad, Anupam Chattopadhyay, and Nejib Ben Hadj-Alouane. Hardware architectures for embedded speaker recognition applications: a survey. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(3):78, 2017.

7 Dai Bui and Edward A Lee. Streamorph: a case for synthesizing energy-efficient adaptive programs using high-level abstractions. In *Proceedings of EMSOFT*, page 20. IEEE Press, 2013.

8 Jeronimo Castrillon, Stefan Schürmans, Anastasia Stulova, Weihua Sheng, Torsten Kempf, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Component-based waveform development: The nucleus tool flow for efficient and portable software defined radio. *Analog Integrated Circuits and Signal Processing*, 69(2-3):173–190, December 2011.

9 Marco Danelutto, Daniele De Sensi, and Massimo Torquati. A power-aware, self-adaptive macro data flow framework. *Parallel Processing Letters*, 27(01):1740004, 2017.

10 Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S Bhattacharyya, and Slaheddine Aridhi. Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 41–48. IEEE, 2013.

11 Brecht Desplanques, Kris Demuynck, and Jean-Pierre Martens. Adaptive speaker diarization of broadcast news based on factor analysis. *Computer Speech & Language*, 46:72–93, 2017.

12 Pascal Fradet, Alain Girault, Ruby Krishnaswamy, Xavier Nicollin, and Arash Shafiei. RDF: Reconfigurable Dataflow (extended version). Research Report RR-9227, INRIA Grenoble - Rhône-Alpes, December 2018. URL: `https://hal.inria.fr/hal-02079683`.

13 Amir Hossein Ghamarian, Marc CW Geilen, Sander Stuijk, Twan Basten, Bart D Theelen, Mohammad Reza Mousavi, Arno JM Moonen, and Marco JG Bekooij. Throughput analysis of synchronous data flow graphs. In *Sixth International Conf. on Application of Concurrency to System Design (ACSD'06)*, pages 25–36. IEEE, 2006.

14 Amir H Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of PACT*, pages 214–223. IEEE, 2009.

15 Gilles KAHN. The semantics of a simple language for parallel programming. *In Information Processing*, 74:471–475, 1974.

16 Robert Khasanov, Andrés Goens, and Jeronimo Castrillon. Implicit data-parallelism in Kahn process networks: Bridging the MacQueen Gap. In *Proceedings of PARMA-DITAM*, pages 20–25. ACM, 2018.

17 Mohaddeseh Nosratighods, Eliathamby Ambikairajah, and Julien Epps. Speaker verification using a novel set of dynamic features. In *18th International Conference on Pattern Recognition (ICPR'06)*, volume 4, pages 266–269. IEEE, 2006.

18 Claudius Ptolemaeus. *System Design, Modeling, and Simulation using Ptolemy II, 2014*. Ptolemy.org, 2014.

19 John R. Rice et al. The algorithm selection problem. *Advances in computers*, 15(65-118):5, 1976.

20 Lars Schor, Iuliana Bacivarov, Hoeseok Yang, and Lothar Thiele. Adapnet: Adapting process networks in response to resource variations. In *Proceedings of CASES*, page 22. ACM, 2014.

21 Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Proceedings of SAMOS*, pages 404–411. IEEE, 2011.

22 Anastasia Stulova, Rainer Leupers, and Gerd Ascheid. Throughput driven transformations of synchronous data flows for mapping to heterogeneous mpsocs. In *Proceedings of SAMOS*, pages 144–151. IEEE, 2012.

23 Roberto Togneri and Daniel Pullella. An overview of speaker identification: Accuracy and robustness issues. *IEEE Circuits and Systems Magazine*, 11(2):23–61, 2011.

24 OB Tuzun, M Demirekler, and KB Nakiboglu. Comparison of parametric and non-parametric representations of speech for recognition. In *Electrotechnical Conference, 1994. Proceedings., 7th Mediterranean*, pages 65–68. IEEE, 1994.

BifurKTM: Approximately Consistent Distributed Transactional Memory for GPUs

Samuel Irving
Louisiana State University, Baton Rouge, LA, USA

Lu Peng
Louisiana State University, Baton Rouge, LA, USA

Costas Busch
Augusta University, GA, USA

Jih-Kwon Peir
University of Florida, Gainesville, FL, USA

Abstract

We present BifurKTM, the first read-optimized Distributed Transactional Memory system for GPU clusters. The BifurKTM design includes: GPU KoSTM, a new software transactional memory conflict detection scheme that exploits relaxed consistency to increase throughput; and KoDTM, a Distributed Transactional Memory model that combines the Data- and Control- flow models to greatly reduce communication overheads.

Despite the allure of huge speedups, GPUs are limited in use due to their programmability and extreme sensitivity to workload characteristics. These become daunting concerns when considering a distributed GPU cluster, wherein a programmer must design algorithms to hide communication latency by exploiting data regularity, high compute intensity, etc. The BifurKTM design allows GPU programmers to exploit a new workload characteristic: the percentage of the workload that is Read-Only (e.g. reads but does not modify shared memory), even when this percentage is not known in advance. Programmers designate transactions that are suitable for Approximate Consistency, in which transactions "appear" to execute at the most convenient time for preventing conflicts. By leveraging Approximate Consistency for Read-Only transactions, the BifurKTM runtime system offers improved performance, application flexibility, and programmability without introducing any errors into shared memory.

Our experiments show that Approximate Consistency can improve BkTM performance by up to 34x in applications with moderate network communication utilization and a read-intensive workload. Using Approximate Consistency, BkTM can reduce GPU-to-GPU network communication by 99%, reduce the number of aborts by up to 100%, and achieve an average speedup of 18x over a similarly sized CPU cluster while requiring minimal effort from the programmer.

2012 ACM Subject Classification Computer systems organization → Heterogeneous (hybrid) systems

Keywords and phrases GPU, Distributed Transactional Memory, Approximate Consistency

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2021.2

1 Introduction

GPUs have become the device of choice for high performance and scientific computing due to their high computational throughput and high memory bandwidth compared to the CPU. The popularity of the GPU has ushered in the era of "General Purpose GPU Computin" in which Graphics Processors are increasingly used for applications that they were not originally designed for, including applications with recursion, irregular memory access, atomic operations, real-time communication between the GPU and its host processor [15], and even real-time communication between the GPU and remote devices using MPI [12].

12th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 10th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2021).
Editors: João Bispo, Stefano Cherubin, and José Flich; Article No. 2; pp. 2:1–2:15

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

Despite its strengths and increasing flexibility, the GPU suffers from poor programmability, as GPU programmers must meticulously balance performance, correctness, and code readability [7]. Programmers must be wary to: avoid critical warp divergence caused by misuses of the GPU's Single Instruction Multiple Data architecture; be wary of the GPUs weak memory model and requirement that programmers manually control several distinct memory spaces; and be careful to ensure data integrity for degrees of parallelism several orders-of-magnitude greater than traditional CPUs. These concerns are magnitude exponentially when scaling GPU applications to the cluster scale.

Transactional Memory (TM) [8] has emerged as a concurrency control strategy that can ensure GPU program correctness while greatly improving programmability. When using TM, programmers simply mark the beginning and end of critical sections that are then converted, at compile time, into architecture-safe implementations that guarantee correctness and dead-lock freedom regardless of application behavior. TM allows programmers to quickly develop performant versions of complex applications without complete understandings of the underlying hardware, workload behavior, or synchronization requirements. These advantages have motivated research into Hardware TM [7, 4, 5], Software TM (STM) [3], and Distributed TM (DTM) [11] for GPUs. Transactional Memory implementations are often modular and highly customizable, allowing programmers to try a variety of performance optimization techniques and hardware configurations without rewriting the program.

We present BifurKTM, the first distributed Transactional Memory system for GPU clusters that uses *Approximate Consistency* [1] to improve the performance. BifurKTM allows programmers to benefit from the GPU's high computational throughput and high memory bandwidth despite the presence of GPU-initiated remote communication and irregular memory accesses in their application. Furthermore, BifurKTM allows GPU programmers to leverage a new application property to improve performance: the percentage of the workload that reads but does not modify shared memory.

The contributions of this paper are as follows:

1. We propose *GPU KoSTM*: a read-optimized conflict detection protocol for GPUs to reduce conflicts using Approximate Consistency.
2. We propose *KoDTM*: a distributed transactional memory model using K-Opacity [1] to reduce node-to-node communication frequency using Approximate Consistency.
3. Based on GPU KoSTM and KoDTM, we design *BifurKTM*: an implementation of GPU DTM that achieves better speedups over CPU clusters using Fine-Grained locking and existing GPU DTM designs.

2 Background and Related Work

Software Transactional Memory (STM) for the GPU must have two key components 1) a strategy for detecting conflicts and 2) a method for transforming the original transaction, written by the programmer, into an implementation that ensures deadlock freedom, correctness, and forward progress despite the addition of loops unanticipated by the programmer (retry after abortion, repeat non-blocking atomic operation, etc.).

Lightweight GPU STM [10] uses a fall-through transformation, in which each transaction is placed inside a while-loop that is repeated until all threads in a warp are successful; threads are "masked off" when a conflict is detected. To allow for long-running remote operations and the pausing and resuming of transactions, a state-machine transformation is used in CUDA DTM; this transformation introduces many additional branching overheads, but allows threads within the same warp to execute the stages of the transaction completely out of sync [11]. GPU KoSTM relies on a combination of the fall-through and state-machine transformations.

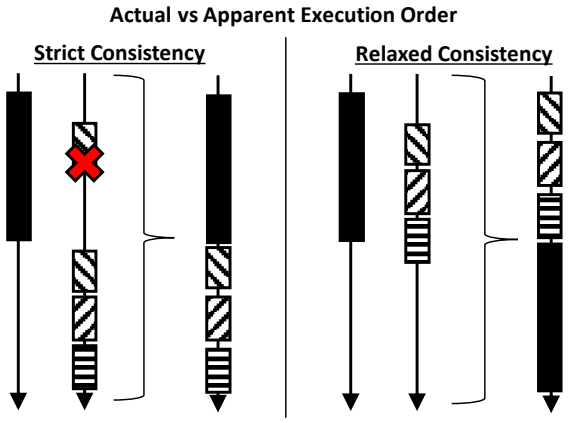


Figure 1 Conflict between Update (solid) and Read-Only transactions (striped) is prevented while maintaining serializability. When using Relaxed Consistency, the Read-Only (striped) transactions "appear" to have executed earlier than they actually did (compared to the solid Update transaction).

Distributed Transactional Memory (DTM) [9] has been implemented for traditional CPU clusters using two models: (1) the Data-Flow model [14], in which objects move between nodes but transaction execution is stationary; and (2) the Control-Flow model [16], in which shared memory is statically mapped but transactions execute across multiple nodes using Remote Procedure Calls (RPCs). DTM for GPU clusters has only been implemented using the Control-Flow model, which avoids the overheads of maintaining a coherent data cache on the GPU which can greatly impair performance [11]. DTM requires support threads that are not explicitly created by the programmer for facilitating remote communication and executing RPCs. Alongside GPU DTM research, there is ongoing research in using Transactional Memory to facilitate cooperation between the CPU and GPU [2], [17], which faces similar challenges due to the requisite communication between different memory spaces, large number of working threads, and high penalties for synchronizing both devices.

Approximate Consistency has been proposed as a method to reduce conflicts by allowing transactions to "appear" as if they executed in a "more convenient" order that would not have resulted in a conflict [1]. The term "approximate" is used in the sense that transactions appear to execute "approximately" when issued by the application; the runtime system is not producing estimates of current values. Fig. 1 shows how relaxed consistency allows three Read-Only transactions to appear to have been executed before a conflict with Update transaction, allowing all transactions to finish earlier while maintaining serializability.

3 Design

BifurKTM (BkTM) is an implementation of Distributed Transactional Memory (DTM) for GPU clusters designed for applications in which workloads typically exhibit characteristics well-suited for the GPU architecture, but irregular memory accesses and atomic operations present major challenges to the programmer. The goal of BkTM is to relax the high barrier of entry required for performant GPU programming by allowing developers to ignore the underlying GPU hardware and network topology while still ensuring that a program runs correctly and to completion. Though some knowledge of the system is required to achieve large speedups over corresponding CPU clusters, The goal of BkTM is to allow developers to create distributed GPU applications in minutes that have historically taken days or months.

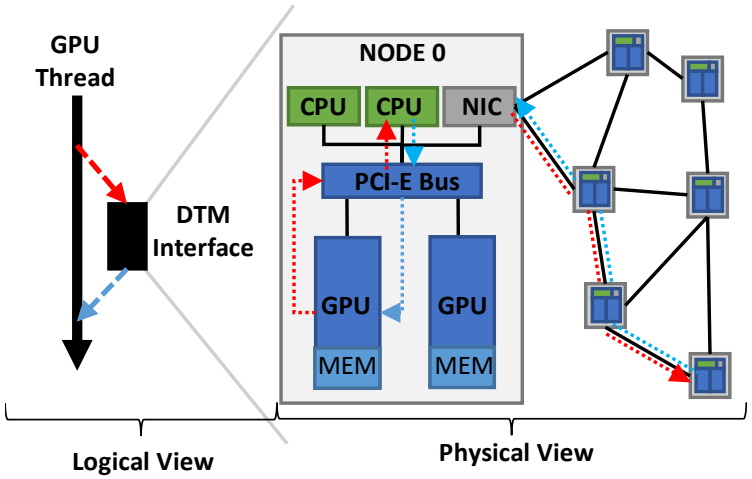


Figure 2 BkTM System Overview showing how the network topology, off-device communication, and distributed memory spaces are "hidden" behind the DTM interface (from the perspective of GPU threads).

When using BkTM, the programmer can achieve correctness and performance while being blind to the underlying GPU architecture and network topology, as shown in Fig. 2. To use BkTM: the programmer has two key responsibilities; 1) marking the beginning and end of critical sections and 2) marking which data is shared between nodes. At compile-time, BkTM will transform transactions into multiple GPU-safe implementations that can be repeated until successful, despite atomic operations and remote communications. Memory accesses within the loop are converted into TM_READs and TM_WRITEs, which direct memory accesses into the STM manager. Finally, a programmer can choose to mark a critical section as "Read-Only", allowing the BkTM runtime system an opportunity to improve performance.

The BkTM runtime system guarantees shared data integrity and strict correctness at all times. Any transaction that might modify shared memory ("Update Transaction") is not allowed to use the Approximate Consistency optimizations; the use of Approximate Consistency is limited exclusively to transactions declare in advance that they will not make changes to shared memory ("Read-Only Transactions"). Correctness and deadlock freedom is guaranteed at the device level by the KoSTM protocol, and at the cluster-level by KoDTM detailed in section 3.2.

BkTM is built on top of a custom Partitioned Global Address Space (PGAS) implementation, though the principles here described can be adapted to any Distributed Shared Memory implementation. A PGAS allows GPU threads to read and write the locally mapped partition with minimal overheads and no additional atomic operations. In BkTM: a single-copy model is used for modifiable data, which is evenly divided between devices. Attempts to modify data in a partition owned by a remote device will incur a series of network communications before the transaction can complete; network communications along the critical path for a transaction can be devastating for overall GPU performance as off-device bandwidth and latency immediately become bottlenecks for transactional throughput. BkTM prevents some remote accesses by introduces a virtual memory hierarchy between the device and the network, as shown in Fig. 3. The Read-Only cache contains copies of remote partitions, and eliminates the need for long-latency remote partition accesses for read only transactions. In

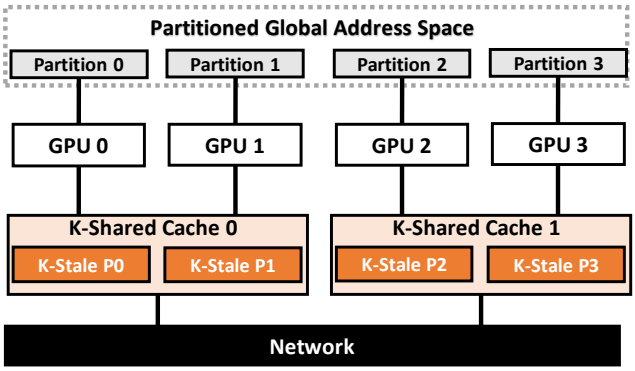


Figure 3 BkTM Virtual Memory Hierarchy showing that each GPU has 1) fast access to a local partition 2) fast read-only access to K-Stale copies of partitions for GPUs within the same sub-cluster, and 3) comparatively slow access to all partitions via the network.

this work we term Read-Only K-Opaque copies of remote partitions as "K-Shared" copies that can be used by "K-Reading" nodes. Modifiable values are statically mapped to a home node, which must track and update K-Shared copies on remote nodes.

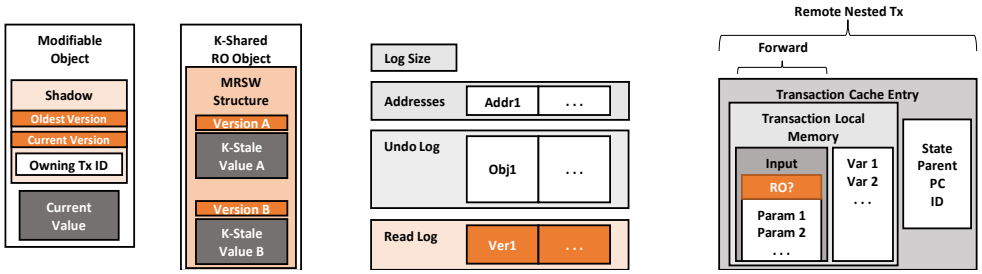
The GPU is a particularly unforgiving device for developers, where the penalties for misunderstanding application behavior can be devastating to performance and expose the system to deadlocks. While BkTM makes the usual TM guarantees for deadlock-freedom, forward progress, etc. the system cannot capitalize on any the GPU advantages (with respect to CPUs) unless they are fundamental to the workload: this can be any of 1) compute intensity, 2) high data access regularity, 3) high parallelizability, and, as introduced in this work, 4) high read intensity. The absence of all of these critical advantages ensures that GPU-to-GPU bandwidth will be the bottleneck of the system, and the high FLOPS and local memory bandwidth of the GPU become irrelevant to performance.

In BkTM, Read-Only transactions are allowed to access "stale" copies of shared data objects, even if the current version is being modified by another thread or if the only modifiable version is owned by another device, provided the stale version is within "K" versions of the current (K-Opaque). Both GPU KoSTM and KoDTM work together to guarantee K-Opacity across all devices. While this strategy cannot introduce any error to the system, whether K-Stale values are acceptable for read-only purposes is at the digression of the programmer.

3.1 GPU K-Opaque Software Transactional Memory (KoSTM)

BkTM uses novel software transactional memory model, GPU K-Opaque Software Transactional Memory (KoSTM), that prevents conflicts between Read-Only and Update transactions, while maintaining performance, data integrity, and minimizing memory overheads. GPU KoSTM allows transactions to read from old versions of shared objects as long as they are within K versions of the current version. In this work, only Read-Only transactions to use Approximately Consistency; Update transactions that might modify shared objects must use precise values.

Update Transactions detect and resolve conflicts eagerly using atomic operations and usage metadata for each shared object, shared memory access, and transaction creation. Read-Only transactions recover a version access history, use lazy conflict detection, and require no



(a) Shadow Entry. **(b)** Access Log. **(c)** Transaction meta-data.

Figure 4 GPU KoSTM Meta-Data. Optimizations for Read-Only Transactions are highlighted in orange. (a) Shadow entries store usage information for each shared memory object; (b) Access Logs are created for each in progress transaction and are used to track changes to Shadow Entries; (c) Transaction Meta-Data contains raw transaction inputs and facilitates the creation of Remote Nested Transactions.

atomic operations. GPU KoSTM guarantees strict correctness for Update transactions, as exclusive locks and strict data management guarantee data integrity. Read-Only transactions do not modify Shadow Entries or shared data values. The meta-data used for GPU KoSTM is shown in Fig. 4 as explained in the following subsections. All KoSTM meta-data is stored in GPU Global Memory. The total memory overhead of the system is the combination of all Shadow Entries, Access Logs, and Transaction Meta-Data.

3.1.1 Shadow Entries

The GPU KoSTM Shadow Entry for modifiable objects has three components: 1) the "Owning Tx ID" integer that is used to track ownership and prevent two transactions from modifying a shared object at the same time and 2) an integer representing the current version of the shared object and 3) an integer containing the version of the oldest K-Shared copy in the system. GPU KoSTM ensures that these versions never differ by more than K-1.

Read-Only transactions are allowed to access K-Shared copies of shared objects, which use a separate Shadow Entry containing a Multiple-Reader Single-Writer (MRSW) structure comprised of two version numbers and two object values. This structure allows the K-Shared object to be duplicated without the use of atomic operations, even while the writer is updating the current version. Including K-Shared objects, Shadow Entry memory overheads range from 200% if shared objects are very large up to 800% if shared objects are very small.

When an object is modified, Update transactions must update the oldest value in the K-Shared shadow and update the version number, before finalizing and releasing exclusive ownership of the shared object. Read-Only transactions with only access to K-Shared versions of the object can always retrieve a valid K-Opaque copy of the object value by 1) storing the largest version (max(a,b)), 2) copying the object value into local memory, 3) performance a CUDA threadFence, and 4) confirming that the largest version has not changed, ensuring that the copied value was not modified by any other thread.

3.1.2 Access Log

A history of all successful data accesses is stored in an undo log for Update transactions and a Read Log for Read-Only transactions. Transactions only ever use either the Undo Log OR the Read Log; the Read Log is kept separate to avoid type casting overheads.

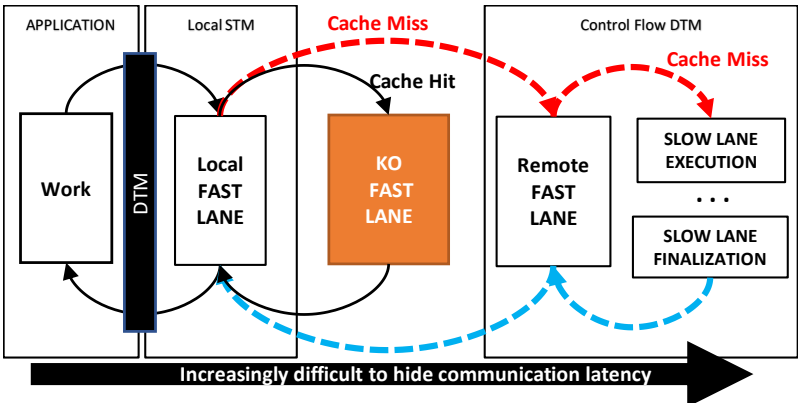


Figure 5 The Control-Flow Model is used for all Transactions. Memory accesses that resolve to the local memory partition can be completed "quickly" without remote communication ("Local Fast Lane"); local-partition misses require either a single Remoted-Nested Transaction ("Remote Fast Lane") or potentially a chain of nested transactions ("Slow Lane Execution"); BFKTM allows some Read-Only transactions to avoid created Remote-Nested Transactions by hitting the K-Shared cache ("KO Fast Lane").

At first access: Update Transactions perform an atomic compare-and-swap on the Owning Transaction ID stored in the object's Shadow Entry. If the object has no current owner, then the transaction is allowed to continue. The transaction must then check that incrementing the version counter at commit-time would not result in a KO violation by ensuring that the Shadow Entry indicates that the current version is at most K-2 versions ahead of the oldest version. If this is not the case, then the transaction is aborted.

Update transactions store the address, a copy of the current value, and increment the log size by 1. Read-Only transactions perform the same record keeping, but instead store the address and version number that was read. Depending on the size of the shared object, recording a version number can be significantly faster than creating a copy of the object.

Update transactions are eagerly aborted at access-time, while Read-Only transactions cannot be aborted until validation. This allows read-only transactions to avoid atomic operations while still guaranteeing K-Opacity.

3.1.3 Transaction Life-Cycle

The initialization and validation of Update transactions are unchanged in this work; any Update transaction still in the ACTIVE state at the time of validation has been successful. Successful update transactions increment the current version counter by 1.

Read-Only transactions must re-check all versions in the Read Log to ensure that each version used is within K versions of the current version. Accesses to the local partition are compared against the current version in the modifiable object Shadow Entry; accesses to K-Shared objects must be greater than or equal to the lowest version number in the K-Shared Shadow Entry.

3.2 Distributed TM Model: KoDTM

BkTM's DTM model uses a combination of the Control- and Data-Flow models. Control-Flow, in which Transaction execution moves between nodes to reach statically-mapped data, is used for Update transactions and Read-Only transactions when the K-Shared caching

is unsuccessful. The Data-Flow model is used exclusively for Read-Only transactions and works to maintain K-Opaque copies of remote partitions and allow Read-Only transactions to bypass remote communications.

BkTM uses a control flow model to guarantee strict correctness for Update transactions in Fig. 5 and for Read-Only transactions that miss the K-Shared cache. Transaction execution is divided into a "Fast Lane", which optimistically assumes there will be no off-device communication on the critical execution path for the transaction. When this assumption fails (i.e. in the event of a partition-miss), execution is halted and moved to the "Slow Lane", either by sending the input variables alone (as a Forward) or sending transaction-local variables to create a remote nested transaction. BkTM aims to keep execution in the Fast Lanes by allowing partition-misses to use K-Opaque copies of remote transactions as if they were the current version. Fig. 5 refers to Read-Only transactions that have avoided remote communication by exploiting K-opacity as being in the "KO Fast lane".

Fig. 6 shows the components used in the KoDTM data flow model. K-Opaque copies of remote partitions are stored on-device in the K-Opaque Cache. System-wide enforcement of the k-consistency is controlled by the Broadcaster thread on the host CPU, which scatters values to remote devices without halting GPU execution in six step process:

1) The broadcaster thread copies the local version shadow entries into host memory and then 2) copies the current local partition values. This ensures that the values copied are newer than or equal to the version numbers copied. The MRSW structure in Fig. 4a allow this copy to occur entirely asynchronously without disrupting GPU transactions. 3) Versions and values are combined into a single message that is MPI Scattered to all nodes in possession of a K-Shared copy. 4) KoDTM listener threads receive the new values, CUDA asynchronous memcpy them into device memory, and then use a stream-synchronize to ensure the values are visible. 5) The original broadcasting thread waits at an MPI Barrier until all KoDTM listener threads have copied the updated values into device memory and arrive at the barrier, indicating all threads in the system with access to the K-Shared copies now see the scattered versions. 6) The versions copied into host memory in step 1 are now copied back into device memory, now indicating the oldest existing K-Shared versions. Compared to Transaction execution, this is long latency process and update transactions may be pessimistically aborted if they are expected to cause KO violations if allowed to proceed. However, this strategy allows remote communications to start as soon as possible rather than waiting until they are initiated by the GPU. Broadcaster threads repeatedly loop through these steps for each shared partition during application execution.

To limit the negative impact on Update transactions, we allow the programmer to limit the maximum number of K-Shared copies using the "K-Share limit" parameter which is explored in our experiments.

3.3 Strict Correctness with Approximate Consistency

GPU KoSTM and KoDTM both exploit relaxed consistency by allowing Read-Only transactions to execute as if they completed at a more "convenient" time. That is to say: K-consistent values are never approximate nor are they invalid, but they may be outdated at the time of commit. Furthermore, approximate consistency does not guarantee that there was a specific "instant" where all read versions existed at the same time, rather that there was an instance where all read versions were within K modifications from the current version. KoSTM and KoDTM do not actually modify the transaction execution order histories, but instead allows certain types of conflicting transactions to proceed uninterrupted. Relaxed consistency always results in an execution order and shared memory state that are strictly correct.

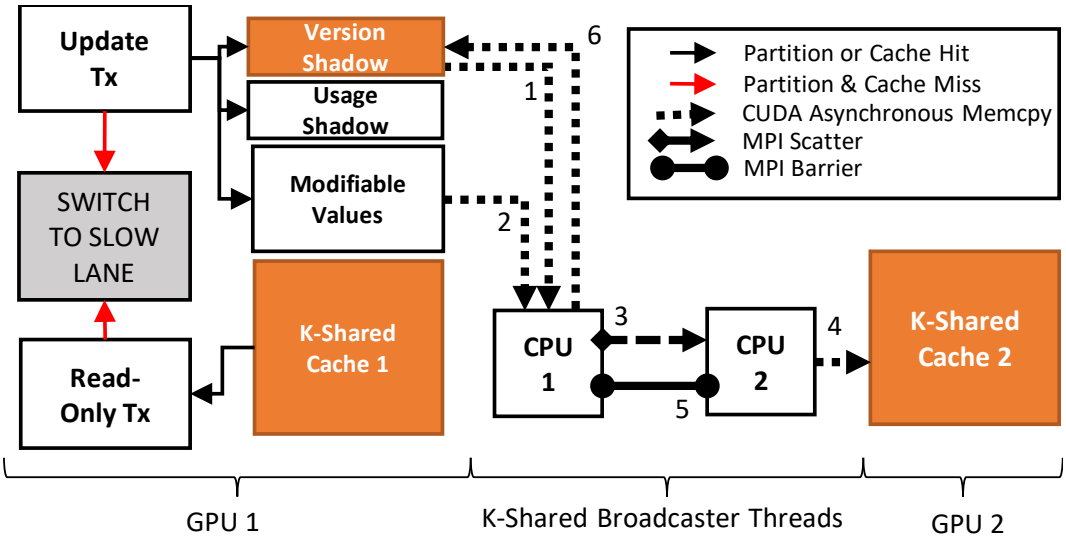


Figure 6 The Data-Flow Model is used to guarantee K-Stale values in the K-Shared cache that is used only by Read-Only transactions. The Host CPU broadcasts the local Version Shadow, and Modifiable Values to remote nodes. These broadcasts are received by remote CPUs and copied into the local K-Shared Cache. No transaction ever initiates data movement; this process is performed automatically by the system.

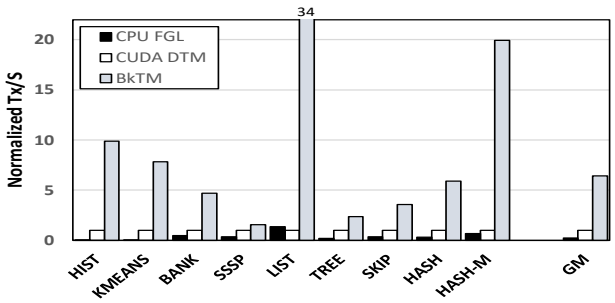
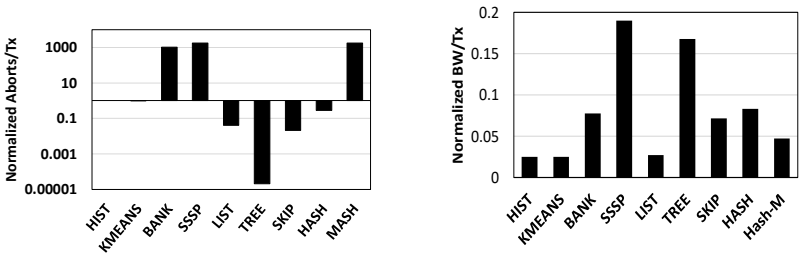


Figure 7 Performance comparisons for workloads with a 75% read-intensity (K=8, 32 Devices, No K-share limit).

Note that no "error" is ever introduced into shared memory through the use of Approximate Consistency. Any transaction that might modify a shared object is required to maintain strict atomicity, consistency, correctness, etc. The programmer must explicitly mark regions where utilization of approximate consistency is acceptable. Whether or not approximate consistency is acceptable for a given application can be subjective and requires a deep understanding of the underlying thread behavior. Programmers can only use approximate consistency in scenarios where stale values cannot be used to introduce error into shared memory.

4 Experimental Analysis

To evaluate the performance of BifurKTM we use a 16 node cluster, where each node contains two CPUs, 2.8GHz E5-2689v2 Xeon processors, and two GPUs, NVIDIA k20x. Nodes are connected using a 56 Gigabit/sec Infiniband network. Benchmarks are written in CUDA



(a) Normalized Abort rate.

(b) Normalized GPU-to-GPU Communication Overhead.

Figure 8 BkTM greatly reduces the GPU-to-GPU communication overheads, but has inconsistent effect on abort rate. Values are normalized against CUDA DTM.

and then compiled using CUDA 9.2 and MPICH 3.1.1 in Red Hat Enterprise Linux 6. We use 9 irregular memory access and dynamic data structure benchmarks used in prior works [6, 13, 10, 11]. BifurKTM is implemented in CUDA, using MPI for communication, and uses pre-processor directives to make multiple transformations of each transaction at compile-time.

The benchmarks used in this work, their workload configurations, and shared memory sizes are shown in Table 1. We consider benchmarks with behaviors favorable to the GPU (compared to the CPU) despite the presence of irregular memory accesses requiring atomic operations. Inputs are randomly generated in place to simulate the behavior of a server receiving random requests from clients. Update transactions are unchanged from their implementations in prior works. We run each benchmark in each configuration three times and report the average.

Read-Only transactions are typically short and involve retrieving a single value from an array or a dynamic data structure: HIST randomly generates a number and reads the number of times it has been generated previously; KMEANS reads the location of the cluster that is closest to a given point; BANK reads the current balance for a given bank account; SSSP, LIST, TREE, and SKIP search a dynamic data structure for a given key and return the associated value if it is found; and finally, HASH+S and HASH+M read one and 20 values from a shared hash table. Hash+M has a larger read/write set in an effort to configure the benchmark for antagonistic behavior. Note that the Read-Only transactions require atomic operations due to the presence of Update transactions that could change any value at any time.

In our first experiment, we compare the performance of BkTM to CPU FGL, where work is only done by the CPUs in the cluster and correctness is assured using Fine-Grained locking, and to CUDA DTM using Pessimistic Software Transactional Memory using the cluster configurations shown in Table 2. Performance values are normalized by the performance of the CUDA DTM. Fig. 7 shows that BkTM+K8 achieves a geometric mean speed up of 6.5x over CUDA DTM, and a speedup of 18x over CPU FGL, when using an K=8 and having no limit on the number of K-Shared read-only copies. BkTM's speedups are primarily attributed to significant reductions in GPU-to-GPU communication overheads despite some increases in the abort rate as shown in Fig. 8. Note that aborting local Update Transactions to ensure cluster-wide K-Opacity requires no communication and thus these aborts are much cheaper than Remote Nested Transaction aborts. BifurKTM eliminates some expensive Remote-Nested aborts in favor of many cheap local aborts, making aborts cheaper on average compared to CUDA DTM. Compared to CPU FGL, BkTM additionally benefits from the GPU's 100x higher computational throughput and 5x higher memory bandwidth.

Table 1 Benchmark Configurations.

	Read-Only	Update		Shared Data	
Benchmark	Tx Reads	Tx Reads	Tx Writes	Object Size (B)	Total Size
HIST	1	1	1	4	70MB
KMEANS	1	1	1	4K	70MB
BANK	1	2	2	4	500MB
SSSP	1	2	1	8	150MB
LIST	1	1	2	8	150MB
TREE	1	1	2	12	100MB
HASH-S	1	1	2	8	150MB
SKIP	1	8	9	36	150MB
HASH-M	20	20	20	4	500MB

HIST and KMEANS perform very long computations before performing very short transactions with low conflict rates. We observe that BkTM has no noticeable impact on the abort rate in Fig. 8a, and reduces communication overheads by 98%, yielding a 7x speedup over CUDA DTM.

BANK and SSSP have very short critical sections with very little time spent outside of the transaction. These behaviors are very disadvantages for GPU clusters unless the workloads exhibit high partition-locality, which limits the usefulness of the K-Shared cache. Having short transactions and high transactional through-puts, the BANK and SSSP both incur massive increases in the abort rate, as update transactions must be pessimistically aborted to ensure K-consistency between K-shared broadcasts. Both benchmarks achieve a modest speedup owing to the 80% reduction in GPU-to-GPU communication overheads.

Dynamic data structure benchmarks show strong speedups proportional to the average search time of the structure. LIST, TREE, SKIP, and HASH all see large decreases in the abort rate as well as the communication overheads owing to the compounding advantages of reduced local contention from 1) GPU KoSTM preventing conflicts between Read-Only and Update transactions, 2) KoDTM allowing at least 80% of remote communication to be avoided, 3) fewer changes to the shared data structure prevents non atomic insertion/retrieval location searches from being repeated. TREE achieves the lowest speedup of 2.4x owing to the very short O(logN) search times and corresponding high update frequency; while list, with a significantly longer O(N) search time, achieves the largest speedup of 34x.

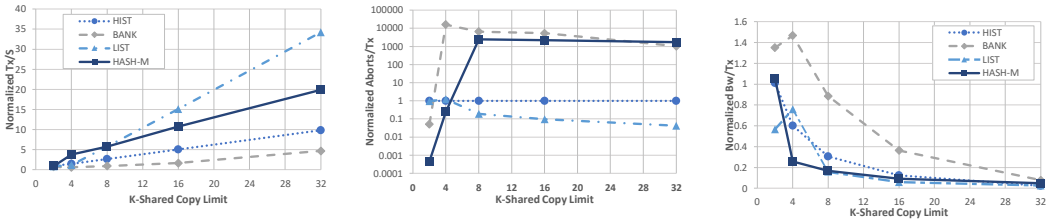
Finally, we observe a 20x speedup for the HASH-M benchmark, which in our experiments performance 20 simultaneous insertions into a shared hash table backed by linked lists. HASH-M's abort rate is greatly increased, as 20 modifications per Update transaction forces abortions to prevent KO violations. The large speedup is attributed to the 95% reduction in GPU-to-GPU communication overheads as shown in Fig. 8b, as all Read-Only transactions hit the K-Share cache. Note that the communication overheads discussed in this work do not include CPU-initiated communications which are used for broadcasting K-Shared values. These broadcasts do compete for communication bandwidth, but they are not along the critical path for any particular transaction and thus do not directly impact performance like a GPU-initiated communication does.

In our experiments, we observe large performance improvements despite huge increases in the abort rate due to the cost asymmetry between a local abort-and-retry, which requires no off-device communication, and the creation of a remote-nested transaction when reading data outside of the local partition which can have a six order-of-magnitudes longer latency caused by remote communication on the critical path. At larger cluster sizes or shared data sizes, the bandwidth between devices will become a critical bottleneck preventing the usage of a read-only copy on every remote node.

Table 2 Cluster Configurations.

Name	Description	Devices	Max Threads per device
CPU FGL	CPU cluster using Locking	64	20
CUDA DTM	GPU cluster using DTM	64 + 64	4096x1024 + 20 on Host
BkTM+KX	GPU cluster usingX-Opaque BkTM	64 + 64	4096x1024 + 20 on Host

In this event, the programmer can either 1) restructure data accesses to decrease partition-miss-rate, or 2) relax K until bandwidth no longer limits update frequency (as shown in Fig. 10b).

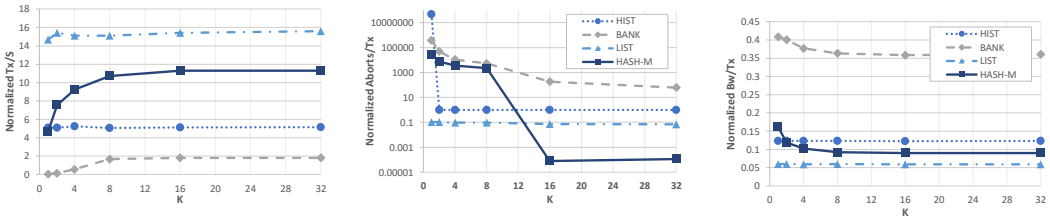


(a) Speedup over BkTM with strict consistency.

(b) Aborts per Committed Transaction.

(c) Communication overhead.

Figure 9 Impact of limiting the number of K-Shared copies on BkTM performance (K=8).



(a) Speedup over BkTM with strict consistency.

(b) Aborts per Committed Transaction.

(c) Communication overheads.

Figure 10 Impact of relaxed consistency on BkTM performance (K-Share limit of 16).

4.1 Balancing Staleness and Throughput

In this section, we study the effect of relaxing consistency and limiting the number of K-Opaque read-only copies of each partition for workloads with a read-intensity of 75%. These two levers allow the programmer to balance two key trade-offs of the BkTM design. We use the shorthand "BkTM+KX" to refer to BkTM configurations with different degrees of approximate consistency, where X=0 refers to "strict consistency".

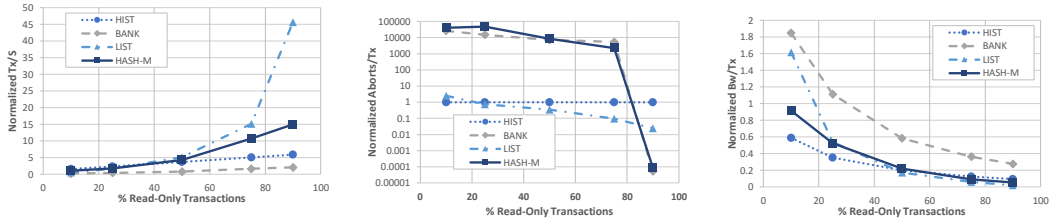
In Fig. 9 we observe that BkTM+K8 achieves a geometric mean speedup of 13.3x over BkTM+K0 due to a 96% reduction in communication overheads. To enforce cluster-wide K-Opacity while using K-Shared data copies, BkTM+K8 incurs a massive increase in the abort-rate. The abort-rate can be mitigated allowing RO transactions to read staler versions by increasing K. We observe that the best performing version of all benchmarks has an unlimited number of K-Shared partition copies, despite the large increases in abort rate, while still enforcing a modest staleness limit of K=8.

As the system allows more K-Shared copies of each partition, the time required to update those copies increases and the limit on the rate at which shared objects can change is lowered. In Fig. 9b we observe a sharp increase in the abort rate for the HASH-M benchmark at a K-Share limit of 2: the time required to update 2 remote partition copies becomes a bottleneck for local update transaction performance. Similarly, maintaining even one K-Shared remote copy hurts BANK update transaction performance. For both benchmarks, the increased throughput of read-only transactions offsets the negative impact on update transactions.

On the other hand, in Fig. 9c we observe spikes in the communication overheads for the BANK and LIST benchmarks as the costs of KO-violation abort & retry are not offset by the RO advantages. We observe that maintaining the K-Shared cache hurts BkTM performance by 24% when allowing only one K-Shared copy of each partition. BkTM+K8 does not show a speedup for the BANK benchmark until 8 devices are allowed to have K-Shared copies of remote partitions.

In Fig. 10, we observe that increasing the degree of Opacity, K, allows update transactions to make more modifications between partition broadcasts to remote readers which can either 1) reduce the number of local aborts caused to ensure global K-Opacity; OR 2) increasing the number of K-Readers, which requires larger, slower broadcasts. Increasing K has a limited impact on RO transactions on K-Sharing nodes as the local copy is guaranteed to be K-Opaque on first access. We observe that increasing K slightly reduces communication overheads in Fig. 10c due to the significantly reduced abort rate.

Increasing K will have a binary effect on update transaction throughput: either K is greater than the number of updates that occur between K-Reader broadcasts or it is not. In Fig. 10b we observe that HIST's update frequency is low and thus increasing K has no impact on the abort rate. In contrast, BANK has a very high frequency and even K=32 is not enough to prevent aborts between K-Reader updates. We see HIST and HASH-M cross break-even points at K=2 and K=16 respectively, where we see 99.9% reductions in their abort-rates. Whether or not a possible staleness of 15 is worth the increased update-throughput is at the discretion of the programmer.



(a) Speedup over BkTM without relaxed consistency.

(b) Aborts Rate.

(c) Communication Overheads.

Figure 11 Impact of workload composition on BkTM performance (K=8, 32 Read-Only Copies).

4.2 Workload Sensitivity

In this study we study the performance of BkTM, normalized by BkTM without relaxed consistency, as the read-intensity of the workload changes. We here define read-intensity as the percentage of all committed transactions that anticipated that no changes would be made to shared memory before the transaction began. The STM system is informed of anticipated read-only execution using information provided by the programmer. Here we use K=8 and allow all 32 nodes to store K-Opaque read-only copies of the full PGAS.

In Fig. 11, we observe large speedups driven primarily by huge reductions in communication overheads for very read-intensive workloads, despite large increases in the abort rate in some cases. With a read-intensity of 10% we observe that BkTM+K8 achieves only 70% of the performance of BkTM+K0 as the majority-update workload does not benefit from the read-optimizations. Poor performance can be attributed to the maintenance of remote read-only partition copies, which can only be K versions behind the local version. These overheads can be hidden with a read-intensity of 25% where BkTM+K8 gains a 1.3x speedup.

At the maximum read-intensity test, 90%, BkTM+K8 shows a geometric mean speedup of 9.6x over BkTM+K0. The LIST benchmark shows particularly large performance improvements owing to the compounding effects of low update frequency, more reliable pre-transaction insertion-point searches, greatly reduced communication overheads, and a greatly reduced abort rate from prevent conflicts between read-only transactions.

In Fig. 11b all benchmarks except HIST show exponentially decreasing abort rates as the read-intensity increases; the abort rate and update frequency of HIST are both so low that the abort rate is unaffected by varying the read-intensity. HASH-M and BANK show greatly increased caused by the high shared-object update frequency causing update transactions to abort and restart while waiting for remote partition copies to be updated. With a read-intensity of 90%, the update frequency is finally low enough that the broadcaster no longer causes aborts to ensure K-Opacity. LIST crosses a similar breaking point at read-intensity of 25%. The exact value for this update frequency is different for each benchmark and is determined by the bandwidth between devices, shared object sizes, and how easily messages can be batched and sent to the same destination node.

5 Conclusion

In this work, we explore the performance of BifurKTM (BkTM), which allows programmers to improve GPU cluster performance by allowing Read-Only transactions to execute at a more convenient time. By rearranging the apparent execution order, BkTM can greatly reduce the conflict rate between transactions and nearly eliminate the communication overheads. We demonstrate that a GPU cluster using BkTM, even workloads with a low read-intensity, can greatly outperform a CPU cluster despite irregular memory accesses and the overheads of accessing distributed shared memory.

Though GPU clusters using BKTM remain sensitive to workload composition, we increase GPU cluster flexibility and achieve a greater high performance range using relaxed consistency. This work relies on the programmer to recognize such opportunities by understanding the underlying application behavior. Finally, the BkTM incurs at least a 200% memory overhead due to the requirement for storing multiple copies of shared objects. We expect future technologies to further relax these constraints by increasing GPU on-device memory, off-chip bandwidth, and increasing the overall diversity of GPU applications.

References

1 Basem Assiri and Costas Busch. Approximate consistency in transactional memory. *International Journal of Networking and Computing*, 8(1):93–123, 2018.

2 Daniel Castro, Paolo Romano, Aleksandar Ilic, and Amin M Khan. Hetm: Transactional memory for heterogeneous systems. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 232–244. IEEE, 2019.

3 Daniel Cederman, Philippas Tsigas, and Muhammad Tayyab Chaudhry. Towards a software transactional memory for graphics processors. In *EGPGV*, pages 121–129, 2010.

4 Sui Chen and Lu Peng. Efficient gpu hardware transactional memory through early conflict resolution. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 274–284. IEEE, 2016.

5 Sui Chen, Lu Peng, and Samuel Irving. Accelerating gpu hardware transactional memory with snapshot isolation. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 282–294. IEEE, 2017.

6 Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

7 Wilson WL Fung, Inderpreet Singh, Andrew Brownsword, and Tor M Aamodt. Hardware transactional memory for gpu architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 296–307. ACM, 2011.

8 Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 289–300, 1993.

9 Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20:195–208, 2007.

10 Anup Holey and Antonia Zhai. Lightweight software transactions on gpus. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 461–470. IEEE, 2014.

11 Samuel Irving, Sui Chen, Lu Peng, Costas Busch, Maurice Herlihy, and Christopher Michael. Cuda-dtm: Distributed transactional memory for gpu clusters. In *Proceedings of the 7th International Conference on Networked Systems*, 2019.

12 Jiri Kraus. An introduction to cuda-aware mpi. *Weblog entry]. PARALLEL FORALL*, 2013.

13 Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46. IEEE, 2008.

14 Sudhanshu Mishra, Alexandru Turcu, Roberto Palmieri, and Binoy Ravindran. Hyflowcpp: A distributed transactional memory framework for c++. In *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*, pages 219–226. IEEE, 2013.

15 John Nickolls, Ian Buck, and Michael Garland. Scalable parallel programming. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 40–53. IEEE, 2008.

16 Mohamed M Saad and Binoy Ravindran. Snake: control flow distributed software transactional memory. In *Symposium on Self-Stabilizing Systems*, pages 238–252. Springer, 2011.

17 Alejandro Villegas, Angeles Navarro, Rafael Asenjo, and Oscar Plata. Toward a software transactional memory for heterogeneous cpu–gpu processors. *The Journal of Supercomputing*, pages 1–16, 2017.

The Impact of Precision Tuning on Embedded Systems Performance: A Case Study on Field-Oriented Control

Gabriele Magnani
DEIB, Politecnico di Milano, Italy

Daniele Cattaneo
DEIB, Politecnico di Milano, Italy

Michele Chiari
DEIB, Politecnico di Milano, Italy

Giovanni Agosta
DEIB, Politecnico di Milano, Italy

Abstract

Field Oriented Control (FOC) is an industry-standard strategy for controlling induction motors and other kinds of AC-based motors. This control scheme has a very high arithmetic intensity when implemented digitally – in particular it requires the use of trigonometric functions. This requirement contrasts with the necessity of increasing the control step frequency when required, and the minimization of power consumption in applications where conserving battery life is paramount such as drones. However, it also makes FOC well suited for optimization using precision tuning techniques. Therefore, we exploit the state-of-the-art FixM methodology to optimize a miniapp simulating a typical FOC application by applying precision tuning of trigonometric functions. The FixM approach itself was extended in order to implement additional algorithm choices to enable a trade-off between execution time and code size. With the application of FixM on the miniapp, we achieved a speedup up to 278%, at a cost of an error in the output less than 0.1%.

2012 ACM Subject Classification Hardware → Power estimation and optimization; Software and its engineering → Compilers; Applied computing → Consumer health

Keywords and phrases Approximate Computing, Field-oriented control, Precision Tuning

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2021.3

Funding Work supported by the FET-HPC project *RECIPE*, G.A. n. 801137.

1 Introduction

Approximate Computing is an increasingly popular approach to achieve large performance and energy improvements in error-tolerant applications [1, 27, 13]. This class of techniques aims at trading off computation accuracy for performance and energy. In particular, precision tuning is an approximate computing technique that trades off the accuracy of mathematical operations for performance and energy by employing less precise data types, e.g. fixed point instead of floating point, or bfloat16 [20] instead of standard IEEE-754 32-bit floating point numbers.

This non-trivial task is usually performed manually by embedded systems programmers, and in general by software developers that need to achieve high performance with limited resources. However, this operation is error-prone and tedious, especially when large code bases are involved. Thus, a significant research effort has been spent over the recent years to build compiler-based tools to support or entirely replace the programmer effort [10]. In particular, recent advances optimize mathematical functions whose computation is usually off-loaded to a library [9].

However, these advances are generally proven on benchmark suites, which are not necessarily representative of specific embedded systems domains. In particular, the PolyBench suite [35] is primarily composed of kernels from the High Performance Computing domain, whereas the CPU variant of AxBench only provides one kernel for each domain addressed, for a total of seven kernels. Therefore, there is a distinct lack of analyses of the performance impact of automated precision tuning on embedded-specific application domains. Given the wide variety of such domains, the trade-off between generality and accuracy of the benchmark can be addressed by exploiting the *miniapp* concept [28, 19, 18], which provides a middle ground between the full application (which provides accuracy at the expense of generality) and kernels (which provide some generality at the expense of accuracy).

Among microcontroller-based embedded application domains, the control of electrical motor drives – a classical topic in power electronics – remains very relevant also thanks to the wide diffusion of drones and the maker culture. *Field-oriented control* (FOC) is one of the two main techniques employed to control such motors. Although it is more computationally intensive than its competitor, Direct Torque Control, the availability of cheap but powerful microcontrollers makes it relevant [32, 4]. We select a FOC controller as a miniapp on which we exercise the full capabilities of a modern compilation toolchain supporting precision tuning for operation on low-power microcontrollers, which are usually not endowed with a floating point unit to save area and power. FOC controllers are mainly composed of the Clarke and Park transforms, in both the direct and inverse form, coupled to proportional/integral (PI) controllers. The two transforms extensively leverage trigonometric functions and square root computation, posing a challenge for the optimization that is best addressed through tunable generation of mathematical functions.

Contribution

In this work, we provide two main contributions. First, we explore the practical applicability of modern precision tuning tools to embedded systems applications, focusing on the specific domain of motor control systems through a dedicated case study.

Second, we improve the FixM library beyond the proof-of-concept provided in [9] by adding the capability of choosing between multiple algorithms. In particular, FixM can now choose between the industry-standard CORDIC [31] algorithm and a simple look-up-table (LUT) implementation.

Organization of the paper

The rest of this paper is organized as follows. In section 2 we briefly survey the existing tools for precision tuning, to select the most appropriate for the optimization of the FOC miniapp. In section 3 we provide a brief background on FOC and a characterization of the FOC miniapp employed as a case study in our work. In section 4 we report on our experimental evaluation, while in section 5 we draw some conclusions and highlight future directions.

2 Related Work

With the growing interest in recent years towards precision tuning as a technique for performance and energy optimization, and with the growing spread of error-tolerant applications in a diverse range of application domains, the literature has seen a number of approaches to automating this task. A full discussion of the topic is beyond the scope of this work, therefore the interested reader is referred to recent surveys such as [2, 10, 27].

Precision tuning approaches can be divided into two broad categories: static and dynamic. Dynamic approaches such as [3, 11, 34, 25, 16], while providing more fine tuning of the data type width, lead to excessive overheads for low-power embedded applications. [11, 25, 16] recompile computational kernels just-in-time to fine-tune them to the current input. The time and energy overhead of recompilation makes this approach sub-optimal for low-power, realtime settings. [34] performs an offline noise-sensitivity analysis to identify error-tolerant areas of the program, and then dynamically reduces the accuracy of floating-point computations at runtime. The main drawbacks of this work is that the offline analysis requires a dataset representative of real world inputs, which might not be available, and hardware support for floating-point precision scaling. The requirement of a representative input dataset is also a drawback of *Autoscaler for C* [22] and *PetaBricks* [3].

Another class of approaches more directly tailored for embedded systems is that of hardware/software codesign approaches, where the hardware computing units are generated according to the minimum necessary precision [21, 26]. These approaches, while very effective, are not applicable to many real world cases. Indeed, the industrial scenario of embedded systems is predominantly composed of small and medium sized system integration companies that work with a range of platforms from large semiconductor manufacturers such as Texas Instruments and ST Microelectronics, with limited opportunities for full-scale hardware/software codesign. Furthermore, this scenario requires to avoid the introduction of custom programming languages and runtimes, both because of the limitations of the underlying hardware and because embedded systems developers often have focused competences on domain-specific tools that generate C code, or directly write embedded C code. This makes dedicated languages such as PetaBricks unsuitable in this scenario.

Thus, we constrain the discussion to tools that can be used for *static precision tuning*, i.e. to provide a single mixed precision version of the original code that satisfies the user-defined precision requirements while optimizing a given performance metric. Such tools gather the information required to apply their optimizations to the code without requiring extensive testing, but rather through static analyses. Among them, the most representative of the state of the art are *Precimonious* [23], *Daisy* [15], and *TAFFO* [12], which are all candidates for use in embedded systems scenarios. Of these, Daisy operates as a source-to-source compiler, which can be considered a drawback, since it may prevent information from the source from reaching the compiler optimization phases directly, possibly introducing overheads. Precimonius and TAFFO operate as LLVM plugins, thus providing a greater degree of integration. Finally, only TAFFO provides dedicated support for mathematical library optimizations [9]. Thus, we select TAFFO with the FIXM extension as the tool for our investigation.

3 Application scenario

The topic of motor control systems is as old as the invention of the first brushed direct current (DC) motor. For over a century, since the mid-1800s, this kind of motors were the favored technology for applications where some degree of control of motor speed and torque was required, because it could be easily achieved through simple techniques such as split windings in the motor and rheostats. Two-phase alternating current (AC) motors or induction motors were much harder to control electrically with the technology of the time. With the advent of solid-state power electronics in the mid-1970's, and the development of control theory, it became possible to electronically control AC motors. The most efficient of such controllers are *active*, in other words they employ feedback from sensors in the motor itself to achieve greater precision in the behavior of the motor.

To this day, one of the most popular state-of-the-art control schemes for AC induction motors is Field-Oriented Control (FOC). FOC was first proposed by Blaschke [5], and it belongs to a wider class of motor control approaches named *variable-frequency drive* (VFD), as it involves variation of the frequency of the electrical power fed into the motor. The main alternative to FOC for motor control is Dynamic Torque Control (DTC), which was developed by Takahashi et al. [30]. This control scheme is simpler to implement. However, it is less effective at low speeds, and produces higher ripple in the torque and the current [8]. A digital implementation of FOC was described by Gabriel et al. in 1980 [17].

In this section, we briefly discuss the relevance of FOC in the current industrial landscape, and then we enter into details with respect to its implementation. Finally, we describe the structure of the miniapp we use, and the improvements to FixM that we implemented.

3.1 Relevant applications of FOC

The applications where FOC is most relevant are all those cases where it is desired to efficiently operate a motor to achieve a set torque or rotation speed. Nowadays, FOC is seeing increasing adoption because the higher computational power required by digital implementations was compensated by the development of high-performance microcontrollers which are able to execute the control loop at ever higher frequencies. In fact, far from standing still, the field of its applications has seen a considerable expansion.

In industrial applications, the last two decades saw the widespread adoption of FOC for all AC motors, where in the past passive control schemes were employed. This development was spurred by an increased preference for permanent-magnet (PM) brushless DC and AC motors, alongside with induction motors (IMs) and other such high-efficiency motors that require the use of FOC to achieve their highest rated torque [6].

Another application field where FOC is in massive use is in high-power electric propulsion systems employing induction motors, such as electric trains and automobiles. While electric trains are a consolidated presence in the public transport scenario, the increased preference of electric engines in automobiles over internal combustion engines is a recent phenomenon because of the reduced environmental impact and the development of battery technology that significantly lifted previous range limitations. FOC applied to automobile-grade induction motors has been successfully employed in the industry, for example in vehicles such as the General Motor EV1 and the Tesla Model S. Therefore, FOC is the key to the impressive acceleration and range efficiency performance of this class of automobiles [29].

An additional use case of FOC that has arisen in the last decade are drones, utility devices that are increasingly replacing heavyweight solutions such as helicopters in applications such as surveillance, video production and more [24].

In the eolic industry, FOC is also used as a control approach for the machine-side-converter component of permanent magnet synchronous generators for wind turbines. Control of the torque of the turbine is key to achieving the highest possible efficiency in all conditions [33].

3.2 Principle of Operation

FOC targets induction motors or permanent magnet synchronous motors. In such motors the drive coils are mounted on the stator, and the rotor is free to rotate around the coils. Motion is achieved by attraction or repulsion of a permanent magnet affixed to the rotor through the magnetic field produced by the current passing through the coils in the stator. Indeed, the output of the FOC control equations is the voltage to be applied to these coils as a function of time. To produce a continuous rotation, the coils – or electromagnets – must alternate their magnetic polarity at a precisely controlled rate.

The central idea behind FOC is to analyze the magnetic flux generated by the current passing through the coils of the motor through vectors in the complex space. In motor space, the frame of reference considered by FOC is composed by three two-dimensional current vectors i_a, i_b and i_c which are laid out with a 120° angle between them. These three vectors model the three electromagnets used by a typical induction motor. This frame of reference is commonly denoted as the *three-phase system axis*, or *abc*-space. The complex current $\bar{i_s}$ induced through the stator is therefore expressed as:

$$\bar{i_s} = i_a + e^{\frac{2j\pi}{3}} i_b + e^{\frac{4j\pi}{3}} i_c$$

where $j = \sqrt{-1}$ is imaginary unit.

The three-phase system describes the current passing through the coils in a geometric and therefore time-variant means. Since in this three-dimensional frame of reference one of the base vectors can be constructed as the linear combination of the other two, it is possible to reduce this space to a simpler still time-variant two-dimensional frame of reference. This operation is performed through the *Clarke transformation*. Given the three-phase currents i_a, i_b and i_c in a balanced system where $i_a + i_b + i_c = 0$, such transform calculates equivalent currents i_α and i_β in the two-phase orthogonal stator space as:

$$\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{\sqrt{3}} & \frac{2}{\sqrt{3}} & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix}$$

To convert this time-dependent frame of reference to a time-independent rotating space, a second transform is employed, named the *Park transformation*. Such transformation operates as follows, given i_α and i_β and a rotor flux angle ϑ:

$$\begin{bmatrix} i_d \\ i_q \end{bmatrix} = \begin{bmatrix} \cos\vartheta & \sin\vartheta \\ -\sin\vartheta & \cos\vartheta \end{bmatrix} \begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix}$$

In induction motors, the rotor flux angle ϑ is measured using a pair of Hall sensors around the perimeter of the motor. The i_d and i_q variables represent respectively the flux component and the torque component of the rotation of the motor. To achieve control of the motor, the FOC approach uses a PI regulator or another kind of regulator on the i_d and i_q variables to compute the *pq*-space voltage components v_d and v_q. We denote the target i_d and i_q as i_{dRef} and i_{qRef}. These voltages are transformed to time-variant $\alpha\beta$-space by applying the reverse Park transform, and then the inverse Clarke transform or another modulation scheme such as PWM to obtain the voltages to the coils in the starting *abc*-space.

3.3 Structure of the Miniapp

From the description of the operation of the FOC control approach, we easily determine that it has a very high arithmetic intensity. In particular, the computation of the Park and the inverse Park transform requires four computations of sin and four computations of cos of a variable angle measured from sensors. Finally, we can deduce that FOC is error tolerant from the application of control engineering principles, and from fact that the inputs to the control system are intrinsically uncertain sensor measurements. Therefore, this application is particularly suited for the application of the FixM methodology, as it matches both effectiveness requirements of such methodology – error tolerance and high arithmetic intensity.

The miniapp we use for analyzing the ability of FixM to optimize the FOC control scheme is composed by:

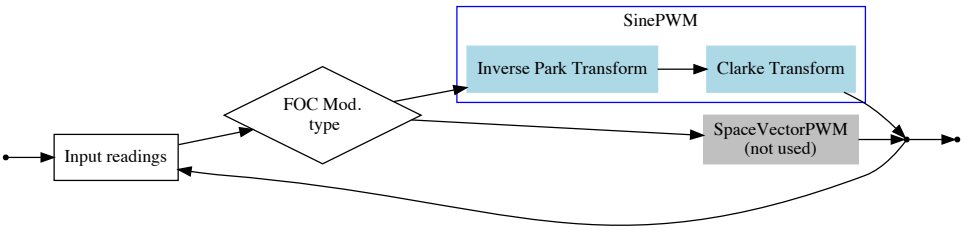


Figure 1 Structure of the FOC miniapp. Light blue elements are the computationally intensive kernel. Shaded elements are not used, but are present in the miniapp structure to maintain compatibility with the Arduino *SimpleFOCLibrary*, from which the original code has been extracted.

- An input generator, which is tasked with producing simulated values of i_a, i_b, ϑ, i_{qRef} and i_{dRef} as required by the FOC system.
- The computational kernel, which performs a single discrete pass of the FOC control algorithm using the inputs generated by the previous component of the miniapp.

The input generator is designed to be lightweight, in order to approximate the computational load which would be required in a realistic application for reading the sensors and the target i_{dRef} and i_{qRef} from an external system or another software component running on the same microcontroller. Additionally, the input generator is deterministic, in order to allow comparisons of the quality of the control action across different precision settings.

The miniapp performs a fixed number of iterations of the FOC kernel combined with the input generator, reporting the last values of i_a, i_b and i_c generated by the FOC controller at regular intervals. At the end of the miniapp's execution cycle, the measured number of clock cycles required for the computation are printed. This allows the comparison of the execution time between differently optimized versions of the miniapp. Figure 1 show a block diagram of the miniapp control flow graph.

3.4 Enhancements to FixM

As presented in [9], FixM is only capable of optimizing sin and cos trigonometric functions by replacing floating point implementations with customized fixed point code depending on the required precision. The fixed point versions of these functions always employed the CORDIC [31] algorithm, as it is fast – it executes in constant time given a fixed amount of bits in the output – and has a very small code size. However, we observe that when the number of different bit partitionings used in the optimized program are small, it is worthwhile to penalize code size in exchange for improved execution time. In particular, we can replace the implementation based on CORDIC with a look-up-table (LUT) based implementation. A LUT is implemented by computing and storing a customizable number of sin values from the input range $[0\text{-}\frac{\pi}{2}]$ at compile time. Since storing the function's value for each possible input may take up too much memory, only values for evenly-spaced inputs are stored. Their granularity determines the precision of the implementation, at the expense of memory consumption. Additionally, the size of the LUT is reduced by exploiting the periodicity and symmetry properties of trigonometric functions. At runtime, the angle to lookup is adjusted using trigonometric transformations to fit the right range and compute the correct function. Thus, LUTs essentially nullify the constant factors intrinsic in a complex

algorithm such as CORDIC, because they only involve a single memory lookup with some minor prior computation to perform a calculation. This approach comes at the cost of a much larger data segment, and the higher the precision, the larger the code size.

In order to allow the automatic trade-off of code size and execution time, we added a parameter to FixM called Z. The Z parameter expresses the proportion of space available for additional code, excluding any occupation attributable by the optimizations performed by FixM. Depending on the value of Z, FixMAGE decides at compile time whether to generate a look-up-table or a CORDIC implementation of a given trigonometric function. By default, Z is set to 0, and in that setting it forces FixMAGE to always generate CORDIC implementations. Conversely, $Z = 1$ will always generate LUTs instead of using CORDIC. Intermediate values let FixMAGE decide which implementation to choose depending on the frequency of use of that implementation and its cost in terms of bytes.

More in detail, FixMAGE models such decision process as a knapsack problem, where each item i is a function instance, its value v_i is equal to the number of times the function is used, and its weight w_i is equal to the estimated size occupied by the LUT computed as:

$$w_i = M_i \cdot d_d \cdot \frac{1 - Z}{d_c \cdot N + d_f \cdot \dot{N}}$$

where M_i is the number of items in the LUT, N is the number of LLVM-IR instructions in the program, $\dot{N}$ is the number of functions in the program, and d_c, d_f and d_d are weights for each instruction, function and LUT entry respectively. The additional parameters d_c, d_f and d_d depend on the architecture. In particular, d_c is the average code density, d_f is the function call prologue and epilogue overhead, and d_d is the size of a single LUT table item. Therefore, the term $A = d_c \cdot N + d_f \cdot \dot{N}$ estimates the code size of the program being compiled. The term $B = M_i \cdot d_d$ estimates the size of the LUTs. For a conventional 32-bit ARM architecture we use $d_c = 4$, $d_f = 64$ and $d_d = 4$. In order to minimize the compilation time, the knapsack problem is solved using the greedy algorithm proposed by Dantzig [14].

4 Experimental Evaluation

In this section, we evaluate the effectiveness of the enhancements presented in Section 3.4 in the optimization of a FOC controller. We evaluate its benefits in terms of energy consumption, execution time and code size of the optimized program. We also measure the impact on computation accuracy, to make sure the optimization does not induce errors in the output that are large enough to compromise the controller's functionality.

Energy consumption and code size are especially important for this application, since the typical hardware platforms on which FOC controllers operate are (possibly) battery-powered embedded systems with limited memory. In this respect, we evaluate FixMAGE by varying the proportion of trigonometric functions implemented with CORDIC vs. LUT, to assess the ability of the approach to modulate the output code size arbitrarily. Execution time is also of primary importance, not only because it directly influences energy consumption, but because – for the controller to be effective – it must provide updated control variables to the motor at an appropriate frequency.

The miniapp benchmark application used in the evaluation has been assembled as described in Section 3.3. In particular, the computational kernel has been obtained by extracting the FOC controller code from Arduino *SimpleFOCLibrary*,[1] version 2.0. This

[1] `https://github.com/simplefoc/Arduino-FOC`

library provides a robust FOC implementation, compatible with a wide set of the most common Microcontroller Unit (MCU) architectures. Arduino *SimpleFOCLibrary* is written in the C++ programming language, and computes all the transforms required by FOC in single-precision floating-point arithmetic. Therefore, it is well suited to the application of FixM for converting all floating-point computations into fixed-point types, whereas other implementations may contain a hand-written fixed-point-based implementation.

No modifications have been made to the extracted code, besides the insertion of a few variable annotations required by TAFFO.

4.1 Hardware Setup

We run our benchmark on two different hardware platforms, representative of the hardware class of low-medium-range MCUs, with limited central memory and CPU frequency. This is the hardware class on which a FOC application could be typically run.

The first platform, codenamed *F2* in the following, is an STM3220G-EVAL evaluation board featuring a 120 MHz Cortex-M3 ARM CPU, with 128 KB of internal RAM, 2 MB of SRAM, and 1 MB of flash memory. This board's CPU has no native support of floating-point arithmetic, which must be completely implemented in software.

The second platform, codenamed *F4* in the following, is a STM32F4-Discovery board with an 168 MHz Cortex-M4 ARM CPU and 192 KB of RAM. Although the CPU is slower than that of the first board, it implements floating-point arithmetic in-hardware.

4.2 Software Configuration

For each platform, we compiled the benchmark application with TAFFO and our enhanced version of FixM. The version of the LLVM toolchain which we employed was version 10.0.1. First of all, we must observe that the implementation of FOC found in the *SimpleFOCLibrary* library is optimized such that only two trigonometric function evaluations are required in each kernel loop iteration. The miniapp was compiled with several different FixM settings to evaluate the tradeoff of using the implementation based on look-up-tables as opposed to the implementation based on CORDIC:

C2 This configuration always uses CORDIC for both trigonometric calls

L1C1 This configuration uses a LUT for the first trigonometric call, but a CORDIC implementation for the second call

C1L1 This configuration uses a LUT for the second trigonometric call, but a CORDIC implementation for the first

L2 This configuration always uses LUTs for both trigonometric calls

We evaluated our approach by comparing such versions of the benchmark with two different baselines. The first one has been obtained by compiling the application with the standard GCC-based compiler toolchain of the platform, as distributed by ST Microelectronics, using only floating-point arithmetic, and with the implementation of the standard mathematical library provided by *newlib* version 2.5.0. The second one consists of the application compiled against the LLVM toolchain with TAFFO, which enables the conversion of floating-point arithmetic into fixed-point computations, but still employs the standard floating-point mathematical library for the trigonometric functions.

4.3 Evaluation Methodology

The execution time of the benchmark was measured by instrumenting it appropriately with code that queries the internal clock of each device. To obtain more consistent measurements, the reported execution times are the average 100 runs of the benchmark. To compare the version of the benchmark optimized with FixM to the baselines, we compute its *speedup*. Let t_1 and t_2 be two time measurements: the speedup of t_1 on t_2 is:

$$S = 100 \left(\frac{t_2}{t_1} - 1 \right)$$

Additionally, we use the speedup to evaluate the potential for energy savings. Indeed, since the speedup grows inversely proportional with the amount of clock cycles spent in the execution of the computational kernel, and the amount of clock cycles grows proportionally with the energy consumption [7], it transitively follows that a high speedup is a strong indication of lower energy consumptions.

The benchmark application was arranged so that it outputs the desired values of the motor control variables as a vector of numbers. We evaluated the accuracy of such results by computing the average Relative Error (RE) between the three versions of the benchmark. Let $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$ be the result vectors of two different versions of the benchmark. We first compute the average absolute error as:

$$AE = \operatorname*{avg}_{1 \leq i \leq n} |x_i - y_i|$$

and then the relative error as:

$$RE = \frac{AE}{\operatorname{avg}_{1 \leq i \leq n} |e_i|}$$

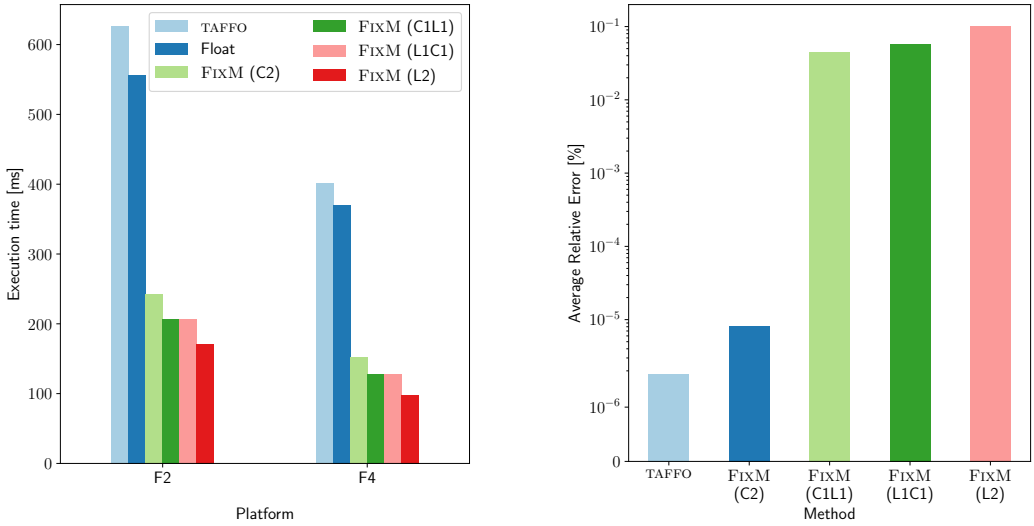
Finally, the percentage relative error can be computed simply by multiplying RE by 100.

4.4 Discussion

Figure 2 shows the performance and accuracy results of our experimental campaign, which confirms that FixM is needed to achieve a speedup on the FOC miniapp. Indeed, TAFFO alone suffers from the impact of the mathematical functions, which require to convert back and forth between floating and fixed point. The introduction of LUTs provides a reasonable benefit in performance, at a limited accuracy impact with respect to CORDIC. The results are confirmed on both platforms, showing the robustness of the FixM framework.

Table 1 shows the impact in memory footprint. The "All Code" column shows the code size (without data) including external support libraries (for example the C standard libraries, and the ST Microelectronics CMSIS-compliant HAL). The "Appl. Code" column shows the size of the code of the FOC miniapp, without counting any external library. Finally, the "Constants" column shows the size of the constant data section of the executable. While TAFFO and FixM do not exhibit a significant overhead – actually FixM with CORDIC even saves some space – the use of LUTs does impose an overhead due to the lookup tables themselves. In fact, the size required by the LUTs alone is higher than the size of the code of the application.

In order to evaluate the efficacy of the trade-off between code and execution time, we compare the code size estimated by FixMAGE to the actual size of the application code after compilation. With parameters $d_c = 4$, $d_f = 64$ and $d_d = 4$, the estimated code size A is



(a) Comparison of execution times by platform and approximation method.

(b) Comparison of average percentage relative errors by approximation method.

Figure 2 Experimental evaluation: performance and accuracy.

Table 1 Size of the generated code in bytes, split across code and constants. The code figures are further split into the code that is part of the FOC miniapp (*Appl. Code*) and the full code size when also taking into consideration the platform-specific support libraries (*All Code*).

	F2			F4		
Method	All Code	Appl. Code	Constants	Code	Appl. Code	Constants
Float	7424	1892	672	6840	1892	696
TAFFO	8912	832	688	8216	900	704
FIXM (C2)	6000	2538	476	5132	2548	496
FIXM (L1C1)	6404	2204	8668	5608	2252	8688
FIXM (C1L1)	6452	2158	8668	5620	2240	8688
FIXM (L2)	6048	1804	8412	5200	1832	8432

equal to 2020 bytes ($N = 281$, $\dot{N} = 14$) which matches the experimentally determined code size for the Float baseline within a margin of error of $\approx 6\%$. The LUT size B is exact (8192 bytes) as the LUTs are generated by FIXMAGE itself.

Figure 3 shows the design space of precision tuning according to the three metrics considered in this work – performance (*Exec Time* in ms), accuracy (*Error* in percentage), and memory footprint (*Size* in bytes). While FIXM using CORDIC provides good performance and small code size at minimum accuracy loss, and is therefore a preferable solution to the use of floating point and TAFFO alone, the use of LUTs provides further performance at the expense of both precision and memory footprint. Mixed solutions (using CORDIC for one operation and LUT for the other) provide an intermediate point that is not Pareto-dominated by either FIXM with CORDIC or FIXM with both LUTs. Thus, the expansion of FIXM to generate also LUTs proves a valuable addition that expands the design space, providing the designer with much needed choices, which can be exerted to cope with specific application constraints. E.g., in case space is tight due to the need to pack more application kernels

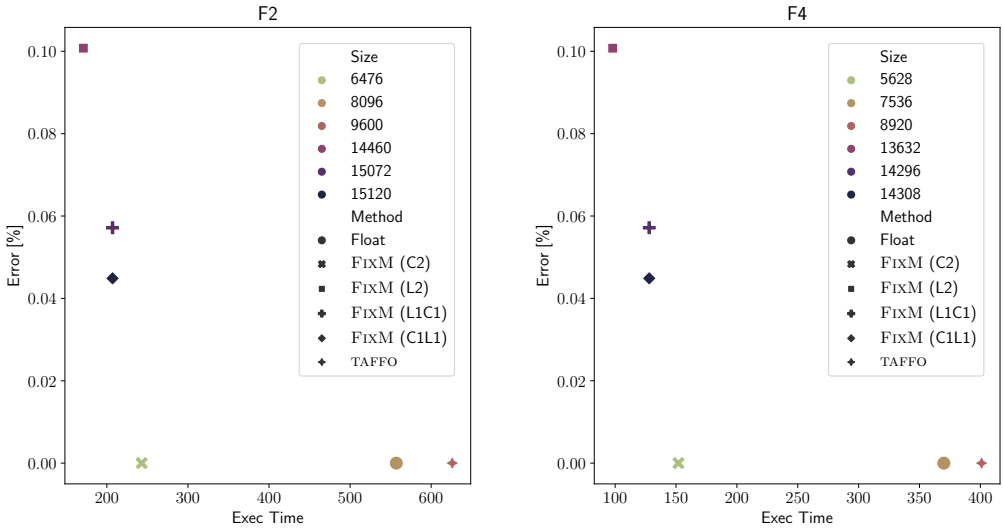


Figure 3 Design space for the two platforms in terms of execution time, error, and memory footprint. Execution time is measured in ms, Size in bytes, Error in percentage.

on a small platform, CORDIC can be prioritized, whereas if the response time is critical more space can be poured into the design to improve performance with the use of one or two LUTs.

5 Conclusions

In this paper, we have explored the impact of precision tuning of arithmetic operations in the application domain of induction motor drive control, through a dedicated miniapp based on a popular Open Source implementation of Field-oriented control. We extended the FixM methodology to manage the trade-off between execution time and code size, achieving a speedup up to 278%, at the cost of a minimal reduction in output error – lesser than 0.1%. Future directions involve the identification of other application domains for which miniapps could be needed, and therefore the construction of a library of domain-specific miniapps. Furthermore, FixM can be further extended to cover more mathematical functions, depending on the needs of the applications.

References

1 A. Agrawal et al. Approximate computing: Challenges and opportunities. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8, 2016.

2 Massimo Alioto, Vivek De, and Andrea Marongiu. Energy-quality scalable integrated circuits and systems: Continuing energy scaling in the twilight of moore's law. *IEEE J. Emerg. Sel. Topics Circuits Syst.*, 8(4):653–678, December 2018. doi:10.1109/JETCAS.2018.2881461.

3 Jason Ansel, Yee L. Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Int. Symp. on Code Generation and Optimization (CGO 2011)*, pages 85–96, April 2011. doi:10.1109/CGO.2011.5764677.

4 Vladislav M. Bida, Dmitry V. Samokhvalov, and Fuad Sh Al-Mahturi. PMSM vector control techniques—a survey. In *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 577–581. IEEE, 2018.

5 Felix Blaschke. Das verfahren der feldorientierung zur regelung der asynchronmaschine. *Siemens Forschungs und Entwicklungsberichte*, 1972.

6 I. Boldea. Electric generators and motors: An overview. *CES Transactions on Electrical Machines and Systems*, 1(1):3–14, 2017. doi:10.23919/TEMS.2017.7911104.

7 Carlo Brandolese, Simone Corbetta, and William Fornaciari. Software energy estimation based on statistical characterization of intermediate compilation code. In *Proceedings of the 2011 International Symposium on Low Power Electronics and Design, 2011, Fukuoka, Japan, August 1-3, 2011*, pages 333–338, 2011.

8 D. Casadei, F. Profumo, G. Serra, and A. Tani. Foc and dtc: two viable schemes for induction motors torque control. *IEEE Transactions on Power Electronics*, 17(5):779–787, 2002. doi:10.1109/TPEL.2002.802183.

9 Daniele Cattaneo, Michele Chiari, Gabriele Magnani, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. FixM: Code generation of fixed point mathematical functions. *Sustainable Computing: Informatics and Systems*, 29:100478, 2021. doi:10.1016/j.suscom.2020.100478.

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

11 Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Giovanni Agosta. Dynamic precision autotuning with TAFFO. *ACM Trans. Archit. Code Optim.*, 17(2), May 2020. doi:10.1145/3388785.

12 Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. TAFFO: Tuning assistant for floating to fixed point optimization. *IEEE Embedded Syst. Lett.*, 12(1):5–8, 2019. doi:10.1109/LES.2019.2913774.

13 Stefano Cherubin et al. Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error. In *Parallel Computing is Everywhere*, volume 32: Advances in Parallel Computing, pages 297–306, March 2018. International Conference on Parallel Computing (ParCo), Sep 2017. doi:10.3233/978-1-61499-843-3-297.

14 George B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):266–288, 1957. doi:10.1287/opre.5.2.266.

15 Eva Darulova, Einar Horn, and Saksham Sharma. Sound mixed-precision optimization with rewriting. In *Proc. 9th ACM/IEEE Int. Conf. on Cyber-Physical Systems*, ICCPS '18, pages 208–219, 2018. doi:10.1109/ICCPS.2018.00028.

16 Marco Festa, Nicole Gervasoni, Stefano Cherubin, and Giovanni Agosta. Continuous program optimization via advanced dynamic compilation techniques. In *Proceedings of the 10th and 8th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, pages 1–6, 2019.

17 R. Gabriel, W. Leonhard, and C. J. Nordby. Field-oriented control of a standard AC motor using microprocessors. *IEEE Transactions on Industry Applications*, IA-16(2):186–192, 1980. doi:10.1109/TIA.1980.4503770.

18 Davide Gadioli et al. Tunable approximations to control time-to-solution in an hpc molecular docking mini-app. *The Journal of Supercomputing*, pages 1–29, 2020.

19 Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.

20 IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, August 2008. doi:10.1109/IEEESTD.2008.4610935.

21 H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE: A fixed-point design and simulation environment. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '98, pages 429–435, 1998.

22 Ki-Il Kum, Jiyang Kang, and Wonyong Sung. AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Trans. Circuits Syst. II. Analog Digit. Signal Process.*, 47(9):840–848, September 2000. doi:10.1109/82.868453.

23 Cindy Rubio-González et al. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 27:1–27:12, November 2013. doi:10.1145/2503210.2503296.

24 Jack Shandle. Field-Oriented Control of Small DC Motors put Drones on a Rising Flight Path, 2015. Accessed November 28, 2020. URL: https://www.digikey.com/en/articles/field-oriented-control-of-small-dc-motors-put-drones-on-a-rising-flight-path.

25 C. Silvano et al. The antarex tool flow for monitoring and autotuning energy efficient hpc systems. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 308–316, 2017. doi:10.1109/SAMOS.2017.8344645.

26 N. Simon, D. Menard, and O. Sentieys. ID.Fix-infrastructure for the design of fixed-point systems. In *University Booth of the Conference on Design, Automation and Test in Europe (DATE)*, volume 38, 2011. URL: http://idfix.gforge.inria.fr.

27 Phillip Stanley-Marbell, Armin Alaghi, Michael Carbin, Eva Darulova, Lara Dolecek, Andreas Gerstlauer, Ghayoor Gillani, Djordje Jevdjic, Thierry Moreau, Mattia Cacciotti, Alexandros Daglis, Natalie Enright Jerger, Babak Falsafi, Sasa Misailovic, Adrian Sampson, and Damien Zufferey. Exploiting errors for efficiency: A survey from circuits to applications. *ACM Computing Surveys*, 53(3), June 2020. doi:10.1145/3394898.

28 Andrew Stone, John Dennis, and Michelle Strout. Establishing a miniapp as a programmability proxy. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 333–334, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2145816.2145881.

29 Xiaoli Sun, Zhengguo Li, Xiaolin Wang, and Chengjiang Li. Technology development of electric vehicles: A review. *Energies*, 13(1):90, December 2019. doi:10.3390/en13010090.

30 I. Takahashi and T. Noguchi. A new quick-response and high-efficiency control strategy of an induction motor. *IEEE Transactions on Industry Applications*, IA-22(5):820–827, 1986. doi:10.1109/TIA.1986.4504799.

31 Jack Volder. The CORDIC computing technique. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 257–261, 1959.

32 Fengxiang Wang, Zhenbin Zhang, Xuezhu Mei, José Rodríguez, and Ralph Kennel. Advanced control strategies of induction machine: Field oriented control, direct torque control and model predictive control. *Energies*, 11(1):120, 2018.

33 V. Yaramasu, A. Dekka, M. J. Durán, S. Kouro, and B. Wu. PMSG-based wind energy conversion systems: survey on power converters and controls. *IET Electric Power Applications*, 11(6):956–968, 2017. doi:10.1049/iet-epa.2016.0799.

34 Serif Yesil, Ismail Akturk, and Ulya R. Karpuzcu. Toward dynamic precision scaling. *IEEE Micro*, 38(4):30–39, July 2018. doi:10.1109/MM.2018.043191123.

35 Tomofumi Yuki. Understanding PolyBench/C 3.2 kernels. In *International workshop on Polyhedral Compilation Techniques (IMPACT)*, 2014.

Resource Aware GPU Scheduling in Kubernetes Infrastructure

Aggelos Ferikoglou
Microprocessors and Digital Systems Laboratory, ECE,
National Technical University of Athens, Greece

Dimosthenis Masouros
Microprocessors and Digital Systems Laboratory, ECE,
National Technical University of Athens, Greece

Achilleas Tzenetopoulos
Microprocessors and Digital Systems Laboratory, ECE
, National Technical University of Athens, Greece

Sotirios Xydis
Department of Informatics and Telematics, DIT, Harokopio University of Athens, Greece

Dimitrios Soudris
Microprocessors and Digital Systems Laboratory, ECE,
National Technical University of Athens, Greece

Abstract

Nowadays, there is an ever-increasing number of artificial intelligence inference workloads pushed and executed on the cloud. To effectively serve and manage the computational demands, data center operators have provisioned their infrastructures with accelerators. Specifically for GPUs, support for efficient management lacks, as state-of-the-art schedulers and orchestrators, threat GPUs only as typical compute resources ignoring their unique characteristics and application properties. This phenomenon combined with the GPU over-provisioning problem leads to severe resource under-utilization. Even though prior work has addressed this problem by colocating applications into a single accelerator device, its resource agnostic nature does not manage to face the resource under-utilization and quality of service violations especially for latency critical applications.

In this paper, we design a resource aware GPU scheduling framework, able to efficiently colocate applications on the same GPU accelerator card. We integrate our solution with Kubernetes, one of the most widely used cloud orchestration frameworks. We show that our scheduler can achieve **58.8%** lower end-to-end job execution time 99%-ile, while delivering **52.5%** higher GPU memory usage, **105.9%** higher GPU utilization percentage on average and **44.4%** lower energy consumption on average, compared to the state-of-the-art schedulers, for a variety of ML representative workloads.

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

Keywords and phrases cloud computing, GPU scheduling, kubernetes, heterogeneity

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2021.4

Funding This work has been partially funded by EU Horizon 2020 program under grant agreement No 825061 EVOLVE (`https://www.evolve-h2020.eu`).

1 Introduction

In recent years, the adoption of artificial intelligence (AI) and machine learning (ML) applications is increasing rapidly. Several major Internet service companies including Google, Microsoft, Apple and Baidu have observed this trend and released their own intelligent personal assistant (IPA) services, e.g. Siri, Cortana etc., providing a wide range of features.

Compared to traditional cloud applications such as web-search, IPA applications are significantly more computationally demanding [13]. Accelerators, such as GPUs, FPGAs, TPUs and ASICs, have been shown to be particularly suitable for these applications from both performance and total cost of ownership (TCO) perspectives [13]. With the increase in ML training and inference workloads [18, 13], cloud providers begin to leverage accelerators in their infrastructures, to catch up with the workload performance demands. This trend is also evident as Amazon AWS and Microsoft Azure have started offering GPU and FPGA based infrastructure solutions.

In particular, for the case of ML inference oriented tasks, public clouds have provisioned GPU resources at the scale of thousands of nodes in data-centers [25]. Since GPUs are relatively new to the cloud stack, support for efficient management lacks. State-of-the-art cluster resource orchestrators, like Kubernetes [9], treat GPUs only as a typical compute resource, thus ignoring their unique characteristics and application properties. In addition, it is observed that users tend to request more GPU resources than needed [3]. This tendency is also evident in state-of-the-art frameworks like Tensorflow which by default binds the whole card memory to an application. This problem, also known as *over-provisioning*, combined with the resource agnostic scheduling frameworks lead to under-utilization of the GPU-acceleration infrastructure and, thus, quality of service (QoS) violations for latency critical applications such as ML inference engines. To overcome the aforementioned issues, real-time monitoring, dynamic resource provisioning and prediction of the future status of the system is required, to enable the efficient utilization of the underlying hardware infrastructure by guiding the GPU scheduling mechanisms.

In this paper, we propose a novel GPU resource orchestration framework that utilizes real-time GPU metrics monitoring to assess the real GPU resource needs of applications at runtime and based on the current state of a specified card decide whether two or more application can be colocated. We analyze the inherent inefficiencies of state-of-the-art Kubernetes GPU schedulers concerning the QoS and resource utilization. The proposed framework estimates the real memory usage of a specified card and predicts the future memory usage, enabling better inference engine colocation decisions. We show that our scheduler can achieve **58.8%** lower end-to-end job execution time 99%-ile for the majority of used inference engine workloads, while also providing **52.5%** higher GPU memory usage, **105.9%** GPU utilization percentage average and **44.4%** lower energy consumption compared with the Alibaba GPU sharing scheduler extension.

2 Related Work

The continuous increase in the amount of containerized workloads uploaded and executed on the cloud, has revealed challenges concerning the container orchestration. Workload co-location and multi-tenancy exposed the interference agnostic nature of the state-of-the-art schedulers [26] while the integration of accelerator resources for ML applications revealed their resource unawareness [25]. To enable better scheduling decisions, real-time [8] or even predictive [19] monitoring is required to drive the orchestration mechanisms. Extending to the case of GPU accelerators, real-time GPU monitoring can allow the colocation of containers on the accelerator in a conservative manner to avoid out-of-memory issues [25].

Container orchestration on GPU resources has been in the center of attention of both academia and industry. Throughout the years, various GPU scheduling approaches have been proposed. Ukidave et al. [27] and Chen et al. [10] have proposed GPU runtime mechanisms to enable better scheduling of GPU tasks either by predicting task behavior or reordering

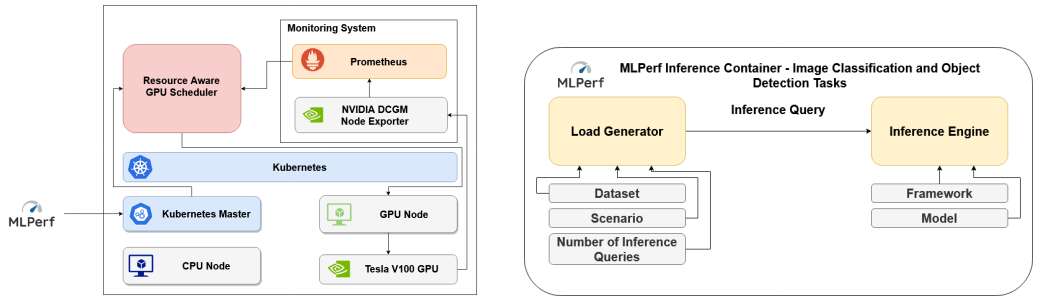
queued tasks. More recent works [17, 11] have introduced docker-level container sharing solutions by allowing multiple containers to fit in the same GPU, as long as the active working set size of all the containers is within the GPU physical memory capacity. As distributed deep neural network (DNN) training based applications have started taking advantage of multiple GPUs in a cluster, the research community proposed application specific schedulers [20] that focus on prioritizing the GPU tasks that are critical for the DNN model accuracy. Hardware support for GPU virtualization and preemption were also introduced. Gupta et al. [12] implemented a task queue in the hypervisor to allow virtualization and preemption of GPU tasks while Tanasic et al. [24] proposed a technique that improves the performance of high priority processes by enabling GPU preemptive scheduling. The integration of GPU sharing schemes on GPU provisioned cloud infrastructures managed by Kubernetes is a trend that is also observed. Yeh et al. proposed KubeShare [29], a framework that extends Kubernetes to enable GPU sharing with fine-grained allocation, while Wang et al. [28] introduced a scheduling scheme that leverages training job progress information to determine the most efficient allocation and reallocation of GPUs for incoming and running jobs at any time.

Regarding container orchestration within GPU environments, Kubernetes itself includes experimental support for managing AMD and Nvidia GPUs across several nodes. Kubernetes GPU scheduler extension [4] exposes a card as a whole meaning that a container can request one or more GPUs. Even though this implementation does not provide fractional GPU usage, it allows better isolation and ensures that applications using a GPU are not affected by others. To overcome this problem, the authors in [1] proposed a GPU sharing scheduling solution which relies on the existing working mechanism of Kubernetes. Alibaba GPU sharing extension aims to improve the utilization of GPU resources by exposing the memory of a card as a custom Kubernetes resource, thus, allowing containers to specify their required amount of memory. Even though this approach allows the concurrent execution of multiple containers, its resource agnostic nature makes it dependable on the credibility of the memory requests. Kube-Knots [25] overcomes this limitation by providing a GPU-aware resource orchestration layer that addresses the GPU orchestration problem. Kube-Knots dynamically harvests spare compute cycles by enabling the co-location of latency-critical and batch workloads, thus, improving the overall resource utilization. This way, it manages to reduce QoS violations of latency critical workloads, while also improving the energy consumption of the cluster. However, its predictive nature fails to face the problem of container failures due to incorrect memory usage predictions and thus GPU memory starvation.

3 Experimental Setup & Specifications

We target high-end server systems equipped with GPU acceleration capabilities found under today's data-center environments. Specifically, our work targets an ML-inference cluster, where a GPU-equipped node is responsible for serving the computational demands of inference queries effectively. In the proposed framework, whenever an inference engine arrives on the cluster, the Kubernetes master redirects it to our custom resource aware GPU scheduler. By leveraging real-time GPU monitoring and prediction, our scheduler decides whether to schedule it on the GPU, or enqueue the task on a priority queue and delay the execution until there are enough GPU resources available. Figure 1 shows an overview of our experimental setup.

Hardware Infrastructure Characterization. All of our experiments have been performed on a dual-socketed Intel® Xeon® Gold 6138 server equipped with an NVIDIA V100 GPU accelerator card, the specifications of which are shown in Table 1. On top of the physical



(a) Proposed Scheduling Framework. **(b)** MLPerf Inference Engine Architecture.

Figure 1 Proposed scheduling framework and MLPerf inference engine architecture.

machine we have deployed three virtual machines, which serve as the nodes of our cluster, using KVM as our hypervisor. The V100 accelerator is exposed on the inference-server VM (24 vCPUs, 32GB RAM) using the IOMMU kernel configuration, while the rest of the VMs (8 vCPUs, 8GB RAM each) are utilized to deploy critical components of our system, such as the master of our Kubernetes cluster and our monitoring infrastructure.

Software & Monitoring Infrastructure Characterization. On top of the VMs, we deploy Kubernetes container orchestrator (v1.18) combined with Docker (v19.03) which is nowadays the most common way of deploying cloud clusters at scale [15]. Our monitoring system consists of two major components, NVIDIA's Data-Center GPU Manager exporter (DCGM) [5] along with Prometheus [6] monitoring toolkit. DCGM exports GPU metrics related to the frame buffer (FB) memory usage (in MiB), the GPU utilization (%) and the power draw (in Watts). In particular, a DCGM exporter container is deployed on top of each node of the cluster through Kubernetes. This container is responsible for capturing and storing the aforementioned metrics into our Prometheus time-series database every specified interval. We set the monitoring interval equal to 1 second to be able to capture the state of our underlying system at run-time. Finally, metrics stored in the Prometheus time-series are accessed from our custom Kubernetes scheduler by performing Prometheus-specific PromQL queries, as described in section 5.

Inference Engine Workloads. For the rest of the paper, we utilize MLPerf Inference [21] benchmark suite for all of our experiments, which is a set of deep learning workloads performing object detection and image classification tasks. As shown in Figure 1b, each MLPerf Inference container instance consists of two main components, *i)* the Inference Engine and *ii)* the Load Generator. The Inference Engine component is responsible for performing

Table 1 CPU & GPU Specifications.

Intel® Xeon® Gold 6138	
Cores/Threads	20/40
Sockets	2
Base Frequency	2.0 GHz
Memory (MHz)	132 GB (2666)
Hard Drive	1 TB SSD
OS (kernel)	Ubuntu 18 (4.15)

NVIDIA V100	
Architecture	Volta
Comp. Cap.	7.0
CUDA Cores	5120
Memory Size	32 GB HBM2
Interface	PCIe 3.0 x16
Sched. Policy	Preemptive

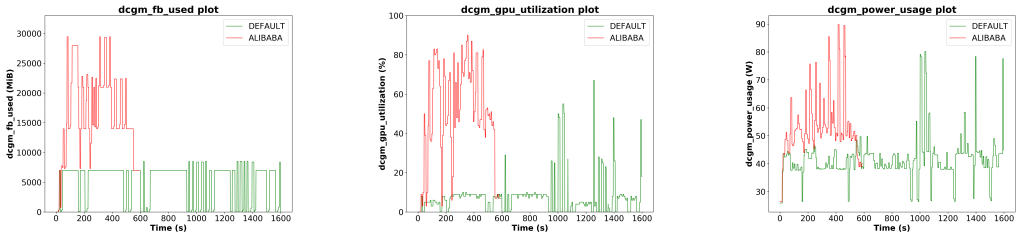
the detection and classification tasks. It receives as input the pre-trained DNN model used during inference (e.g. ResNet, Mobilenet etc.) as well as the corresponding backend framework (e.g. PyTorch, Tensorflow etc.). The Load Generator module is responsible for producing traffic on the Inference Engine and measure its performance. It receives as input the validation dataset (e.g. Imagenet, Coco) as well as the examined scenario and the number of inference queries to be performed. The scenario can be either Single stream (Load Generator sends the next query as soon as the previous is completed), Multiple stream (Load Generator sends a new query after a specified amount of time if the prior query has been completed, otherwise the new query is dropped and is counted as an overtime query), Server (Load Generator sends new queries according to a Poisson distribution) and Offline (Load Generator sends all the queries at start). Considering the above inputs, the Load Generator performs streaming queries to the Inference Engine and waits for the results. For the rest of the paper, we utilize the Single Stream scenario and evaluate our inference engine through the 99%-ile of the measured latency.

4 Motivational Observations and Analysis

Latest advancements in the micro-architecture of NVIDIA's GPUs allow the transparent, cooperative execution of CUDA applications on the underlying accelerator, either through CUDA's streams [2] or through CUDA's Multi-Process Service (MPS) [22] capabilities. These functionalities increase the utilization of GPU accelerators, thus, offering increased computing capacity, yet, state-of-the-art frameworks, such as Kubernetes do not provide mechanisms that expose them to end-users. In fact, Kubernetes default GPU scheduler [4] mechanism provides exclusive access to applications requesting GPU accelerators. Even though, this approach allows isolation and ensures that applications using a GPU do not interfere with each other, it can cause high resource under-utilization or QoS violations, especially in deep-learning inference scenarios on high-end GPUs, which have low requirements in terms of CUDA cores and memory. In order to allow more prediction services to share the same GPU and, thus, improve their QoS and the utilization of the card, partitioning of the GPU memory resource is required. Towards this direction, Alibaba offers a GPU sharing extension [1], which allows the partitioning of the GPU memory. This scheduler allows end-users to define the requirements of their workloads in terms of GPU memory and combines this information with the total available memory of the GPU to decide whether two or more inference engines can be colocated or not.

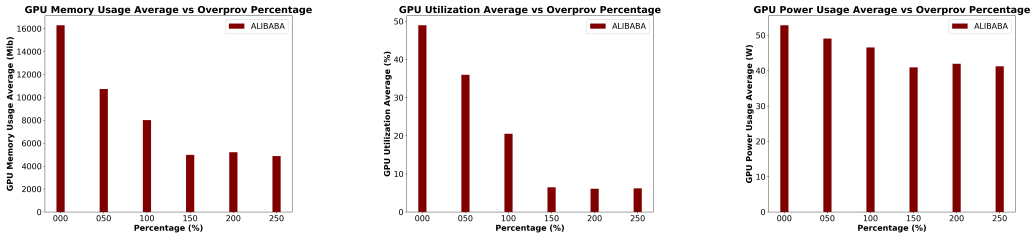
To demonstrate the inefficiency of the Kubernetes GPU scheduler extension compared with Alibaba GPU sharing extension, we perform a straight comparison between them for the scheduling of a workload that consists of 6 inference engines from the MLPerf suite.

Figure 2 shows the GPU memory utilization (MB), the CUDA cores utilization (%) and the power usage signals of the inference engine workload for the above-mentioned schedulers. As shown, the Kubernetes GPU scheduler extension has an average memory utilization of 5GB, which can be considered relatively low compared to the available 32GB memory of the underlying GPU card. The same observation can be made for the GPU utilization signal (7.22% on average) and the power consumption (41.5 Watts on average), as the GPU binding per inference engine leads to resource under-utilization. On the other hand, the average GPU memory usage for the Alibaba GPU sharing extension is 16GB, which is x3.24 higher. Similarly, we see an average of 49% utilization improvement (x6.8 increment) and an average of 52.9 Watts higher power consumption (x1.28 increase). It also leads to a 52.8% decrease of the average energy consumption as Kubernetes GPU scheduler extension consumption is



(a) GPU Memory Usage Signal **(b)** GPU Utilization Percentage Signal **(c)** GPU Power Usage Signal

Figure 2 GPU memory usage, utilization percentage and power usage signals for Kubernetes GPU scheduler extension and Alibaba GPU sharing extension.



(a) Memory Usage Average **(b)** GPU Utilization Average **(c)** Power Usage Average

Figure 3 Memory usage, GPU utilization and power consumption averages vs over-provisioning percentage for Alibaba GPU sharing scheduler extension.

66.4 kJ and Alibaba GPU sharing extension consumption is 31.3 kJ. Finally, we observe that the overall inference engine workload duration using the Alibaba GPU sharing extension is x2.67 faster than the Kubernetes GPU scheduler extension, meaning that the card sharing improves the overall workload duration.

Even though Alibaba's scheduler outperforms the default one, it highly depends on the provisioning degree of the inference engine memory request. For example, if an inference engine requests more memory than it actually needs, this may affect future GPU requests of other inference engines, which will not be colocated, even though their memory request can be satisfied. To better understand the impact of the resource over-provisioning problem within Alibaba's scheduler, we perform 6 different experiments, where we schedule the same inference-engine task, each time with a different memory over-provisioning percentage, ranging from 0% to 250%. Figure 3 depicts the memory usage, the utilization percentage and the power usage averages. For low over-provisioning percentages, Alibaba GPU sharing extension leads to high resource utilization due to the inference engine colocation. However, as shown, it is not able to efficiently sense and handle user-guided over-provisioning scenarios.

5 Resource-aware GPU Sharing Kubernetes scheduler

Figure 4 shows an overview of our proposed resource-aware GPU-sharing scheduler. Whenever an inference engine is scheduled from the custom scheduler, the corresponding workload enters a priority queue which defines their scheduling order. The inference engine assigned priority is proportional to the corresponding GPU memory request. As a result the scheduler always tries to schedule the inference engines with the bigger memory requests. If a workload is chosen to be scheduled, the following three co-location mechanisms are successively executed:

Resource Agnostic GPU Sharing. Our custom scheduler holds a variable that is used as an indicator of the available GPU memory. This variable is initialized to the maximum available memory of the used card in the GPU node. If the inference engine memory request is smaller than the value of this variable, the request can be satisfied and the workload can be scheduled. Whenever an inference engine is scheduled, the value of the indicator variable is decreased by the amount of the memory request. Resource agnostic GPU sharing does not face the memory over-provisioning problem as it is not possible to know a priory that the amount of requested memory is actually the amount that the workload needs to run properly. In our proposed scheduler, we overcome this problem by using real-time memory usage data by our GPU monitoring sub-system. The monitoring system data are collected by performing range queries to Prometheus time series database.

Correlation Based Prediction. Correlation Based Prediction (CBP) [25] provides an estimation for the real memory consumption on a GPU node. The estimation is defined from the 80%-ile of the GPU memory usage rather than the maximum usage. The basic idea of this algorithm is that GPU applications, on an average, have stable resource usage for most of their execution, except for the times when the resource demand surges. In addition, the whole allocated capacity is used for a small portion of the execution time while the applications are provisioned for the peak utilization. CBP scheduler virtually resizes the running workloads for a common case, letting more pending inference engines to be colocated.

In order to have an accurate estimation, low signal variability is required. The signal variability is calculated using the coefficient of variation (CV) metric [7]. If CV is lower than a defined threshold, the memory usage is defined by calculating the 80%-ile of the signal. The free GPU memory estimation is equal to the difference of the maximum available GPU memory and the memory usage estimation. Finally, if the memory request can be satisfied the workload is scheduled. Otherwise the Peak Prediction algorithm is used.

Peak Prediction. Peak Prediction (PP) [25] relies on the temporal nature of peak resource consumption within an application. For example, a workload that requires GPU resources will not allocate all the memory it needs at once. So, although the GPU memory request cannot be satisfied at the scheduling time, it may be satisfied in the near future. The memory usage prediction is based on an auto regressive model (AR) [23]. For an accurate prediction the auto correlation value of order k is calculated. If the auto correlation [16] value is larger than a defined threshold, auto regression of order 1 is performed using linear regression (LR) [14]. If the predicted GPU memory request can be satisfied from PP, then the workload is scheduled. Otherwise, the workload is put into the priority queue and our algorithm attempts to schedule the next available workload from the queue. As we see, PP scheduling decisions depend on the accuracy of the used auto-regressive model and thus linear regression. Even though linear regression is a simplistic approach for predicting the unoccupied memory of the GPU, it can accurately follow the memory utilization pattern (as we further analyze in section 6). In addition, its low computing and memory requirements, allows the PP mechanism to provide fast predictions at runtime with minimal resource interference.

6 Experimental Evaluation

We evaluate our custom scheduler through a rich set of various comparative experiments. We consider inference engine workloads for differing intervals between consecutive inference engine arrivals. In each comparative analysis the exact same workload is fed to the Kubernetes GPU scheduler extension [4], the Alibaba GPU sharing extension [1] and the custom scheduler multiple times. Each time a different memory over-provisioning percentage is used.

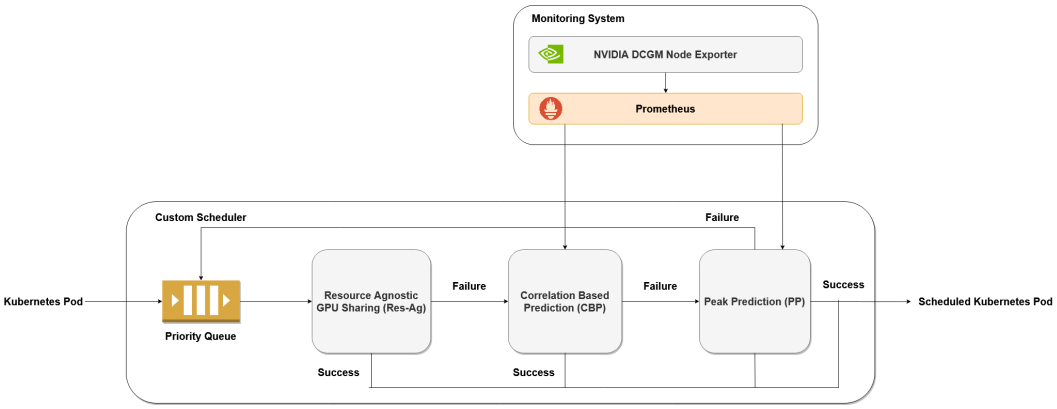
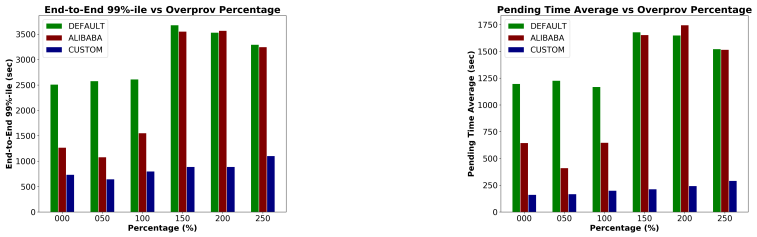


Figure 4 Resource Aware GPU Colocation Algorithm.



(a) End-to-End Job Execution 99%-ile.

(b) Pending Time Average.

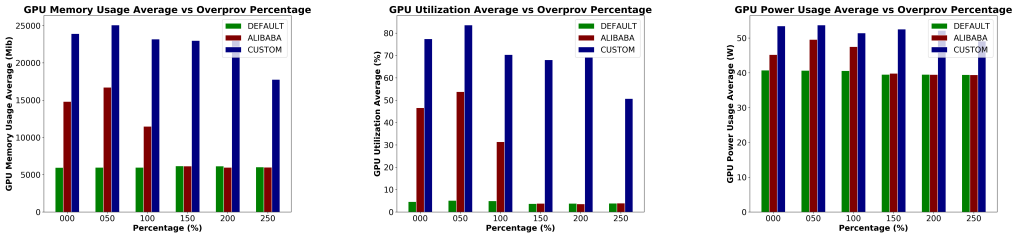
Figure 5 End-to-end job execution 99%-ile and pending time average vs over-provisioning percentage homogeneous workload with MIN=5 and MAX=10.

We provide analysis for homogeneous, i.e., scaling out a single inference service, and heterogeneous workload scenarios. Each workload creates a different inference engine by using the MLPerf inference container we described in section 3. An inference engine is fully defined from the used backend (e.g. Tensorflow, PyTorch etc.), the pre-trained model, the dataset, the scenario, the GPU memory request and the number of inference queries that are going to be executed. The interval between two consecutive inference engine arrivals is defined by the values MIN and MAX (random number in [MIN, MAX] interval in seconds).

6.1 Homogeneous Workload Evaluation

For homogeneous workload, we consider the Tensorflow ssd-mobilenet engine which uses the Coco (resized 300x300) dataset while each inferences engine executes 1024 queries. Each inference engine requires approximately 7 GB of GPU memory meaning that in a card with 32 GB memory, only 4 can be safely colocated.

Figure 5 shows the end-to-end 99%-ile and the pending time average for all the available schedulers, for different over-provisioning percentages. Custom scheduler offers x6.6 (on average) lower pending time average and x3.6 (on average) lower end-to-end 99%-ile from Kubernetes default GPU scheduler extension. It also offers x5.2 (on average) lower pending time average and x2.8 (on average) lower end-to-end 99%-ile from Alibaba GPU sharing scheduler extension. However, due to the colocation of multiple inference engines, custom scheduler's decisions lead to higher inference engine 99%-ile average.



(a) Memory Usage Average. **(b)** GPU Utilization Average. **(c)** Power Usage Average.

Figure 6 Memory usage average, GPU utilization average and power consumption averages vs over-provisioning percentage for homogeneous workload with MIN=5 and MAX=10.

To understand the above mentioned results, the way each mechanism schedules workloads to the GPU should be analyzed. Kubernetes default GPU scheduler extension allocates the whole GPU resource for each inference engine, leading to severe increase of the pending time average (the average time an inference engine waits in the priority queue). The Alibaba GPU sharing scheduler extension uses a resource agnostic colocation mechanism to schedule workloads in the same card. In particular, for over-provisioning percentage equal to 0 % (7 GB memory request) 4 inference engines can be collocated, for 50 % (10 GB memory request) 3 inference engines can be colocated, for 100 % (14 GB memory request) 2 inference engines can be colocated and for 150 %, 200 % and 250 % each inference engine allocates the whole GPU resource. As a result, Alibaba GPU share scheduler extension has similar results with our custom scheduler for over-provisioning percentages equal to 0 % and 50 %. Custom scheduler handles the memory over-provisioning problem in a better way because of its resource aware nature. Figure 5 shows that the proposed scheduler has similar behavior concerning the quality of service metrics for all the different over-provisioning percentages.

Figure 6 shows the memory usage, the utilization percentage and the power consumption averages for all the available schedulers for different over-provisioning percentages. Custom scheduler leads to x3.7 higher memory usage, x16 higher GPU utilization and x1.3 higher power consumption from Kubernetes default GPU extension. It also leads to x2.2 higher memory usage, x2.9 higher GPU utilization and x1.2 higher power consumption from Alibaba GPU sharing scheduler extension. Although we observe an increase in the power usage average, it should be clear that due to the lower overall workload duration the average energy consumption is x2.6 lower from the Kubernetes GPU scheduler extension and x2.2 lower from the Alibaba GPU sharing extension.

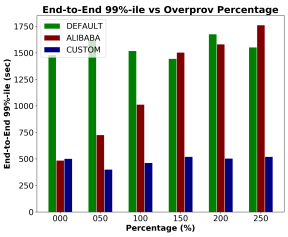
In particular, Kubernetes default GPU extension has the lower resource utilization because each inference engine allocates the whole GPU resource. Alibaba GPU share scheduler extension has similar results with our custom scheduler only for 0 % and 50 % over-provisioning percentages. This is expected, since for these over-provisioning percentages the scheduler can effectively colocate workloads. The higher the over-provisioning percentage is, the closer the resource utilization is to Kubernetes default GPU extension. Finally, we observe that our custom scheduler has similar behavior concerning the resource utilization for all the different over-provisioning percentages.

6.2 Heterogeneous Workload Evaluation

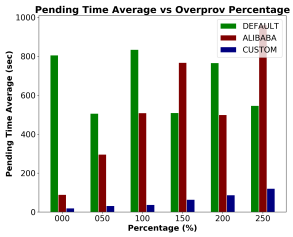
For heterogeneous workload, we consider different inference engines, where each one of them performs a different number of inference queries, as shown in Table 2. Figure 7 shows the quality of service metrics for the heterogeneous inference engine workload. Our proposed

Table 2 Inference engines used for heterogeneous workload evaluation.

Model	Dataset	Queries/Engine (#Engines)
mobilenet	Imagenet	1024 (2), 2048 (2)
mobilenet quantized	Imagenet	256 (2), 512 (2)
resnet50	Imagenet	4096 (2), 8192 (2)
sd-mobilenet	Coco (resized 300x300)	128 (3), 1024 (2)
ssd-mobilenet quantized finetuned	Coco (resized 300x300)	64 (2), 1024 (2)
ssd-mobilenet symmetrically quantized finetuned	Coco (resized 300x300)	512 (2), 4096 (2)



(a) End-to-End Job Execution 99%-ile.



(b) Pending Time Average.

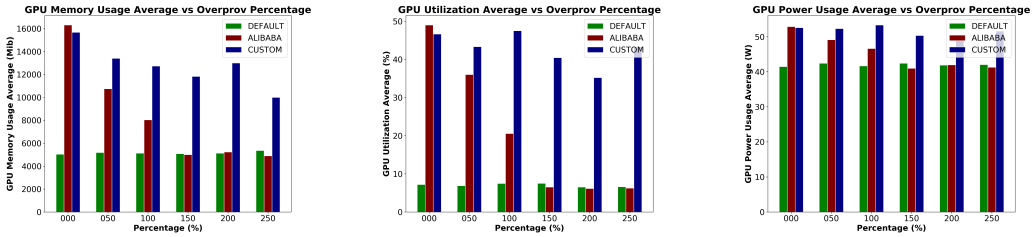
Figure 7 End-to-end job execution 99%-ile and pending time average vs over-provisioning percentage heterogeneous workload with MIN=5 and MAX=10.

scheduler offers x11 lower pending time average and x3.2 lower end-to-end 99%-ile and x8.6 lower pending time average and x2.4 lower end-to-end 99%-ile on average compared to the Kubernetes default and Alibaba's GPU schedulers respectively. Moreover, Figure 8 shows the respective GPU metrics. We see that, our scheduler leads to x2.5 higher memory usage, x6.1 higher GPU utilization and x1.2 higher power consumption compared to Kubernetes default GPU extension and x1.5 higher memory usage, x2.1 higher GPU utilization and x1.1 higher power consumption compared to Alibaba's GPU sharing scheduler extension.

Container Restarts Analysis. As it was mentioned in section 5, CBP involves the risk of incorrect scheduling decisions and thus inference engine failures. CBP's prediction accuracy depends on how representative is the free memory signal it receives as input. Since accelerated applications do not always request GPU resources at the beginning of their execution, it is possible that the used signal does not depict the real load of the node. Although several container restarts occured in the previous experiment, we observe that our proposed scheduler still offers better QoS and GPU resource utilization from the baseline state-of-the-art GPU schedulers.

7 Conclusion

In this paper, we design a resource aware GPU colocation framework for Kubernetes inference clusters. We evaluate the inference engine colocation algorithm using workloads that consist of inference engines using different scenarios. We identify and explain the disadvantages of the correlation based prediction (CBP) and peak prediction (PP) scheduling schemes. Finally, we show that our custom scheduling framework improves the defined quality of service metrics while also increases the GPU resource utilization.



(a) Memory Usage Average. **(b)** GPU Utilization Average. **(c)** Power Usage Average.

Figure 8 Memory usage average, GPU utilization average and power consumption averages vs over-provisioning percentage for heterogeneous workload with MIN=5 and MAX=10.

References

1 Alibaba GPU Sharing Scheduler Extension. URL: https://www.alibabacloud.com/blog/594926.

2 CUDA Streams. URL: https://leimao.github.io/blog/CUDA-Stream/.

3 GPU Memory Over-provisioning. URL: https://www.logicalclocks.com/blog/optimizing-gpu-utilization-in-hops.

4 Kubernetes GPU Scheduler Extension. URL: https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/.

5 NVIDIA Data Center GPU Manager. URL: https://developer.nvidia.com/dcgm.

6 Prometheus. URL: https://prometheus.io/docs/introduction/overview/.

7 B. S. Everitt A.Skrondal. *The Cambridge Dictionary of Statistics*. Cambridge University Press, 2554. URL: http://library1.nida.ac.th/termpaper6/sd/2554/19755.pdf.

8 Rolando Brondolin, Marco D Santambrogio, and Politecnico Milano. A Black-box Monitoring Approach to Measure Microservices Runtime Performance. *ACM Transactions on Architecture and Code Optimization*, 17(4), 2020.

9 Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Commun. ACM*, 59(5):50–57, April 2016. doi:10.1145/2890784.

10 Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 02-06-Apri:681–696, 2016. doi:10.1145/2872362.2872368.

11 James Gleeson and Eyal de Lara. Heterogeneous GPU reallocation. *9th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2017, co-located with USENIX ATC 2017*, 2017.

12 Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *USENIXATC'11: Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, 2011.

13 Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. *SIGARCH Comput. Archit. News*, 43(1):223–238, March 2015. doi:10.1145/2786763.2694347.

14 Howard J. Seltman. Experimental Design and Analysis. *Revista*, 20(2), 2016. doi:10.35699/2316-770x.2013.2692.

15 VMware Inc. Containers on virtual machines or bare metal ? *Deploying and Securely Managing Containerized Applications at Scale, White Paper*, December 2018.

16 John A. Gubner. *Probability and Random Processes for Electrical and Computer Engineers*. Cambridge University Press, 2554. URL: http://library1.nida.ac.th/termpaper6/sd/2554/19755.pdf.

17 D Kang, T J Jun, D Kim, J Kim, and D Kim. ConVGPU: GPU Management Middleware in Container Based Virtualized Environment. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 301–309, 2017. doi:10.1109/CLUSTER.2017.17.

18 Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. GrandSLAm. *EuroSys '19: Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019. doi:10.1145/3302424.3303958.

19 D Masouros, S Xydis, and D Soudris. Rusty: Runtime Interference-Aware Predictive Monitoring for Modern Multi-Tenant Systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):184–198, January 2021. doi:10.1109/TPDS.2020.3013948.

20 Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. *Proceedings of the 13th EuroSys Conference, EuroSys 2018*, 2018-Janua, 2018. doi:10.1145/3190508.3190517.

21 Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. MLPerf Inference Benchmark, 2019. arXiv:1911.02549.

22 Multi-process Service. Multi-process service, 2020.

23 Robert Shumway and David Stoffer. *Time Series Analysis and Its Applications: With R Examples*. Springer, 2017. doi:10.1007/978-3-319-52452-8.

24 I Tanasic, I Gelado, J Cabezas, A Ramirez, N Navarro, and M Valero. Enabling preemptive multiprogramming on GPUs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 193–204, 2014.

25 Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2019-Septe:1–13, 2019. doi:10.1109/CLUSTER.2019.8891040.

26 Achilleas Tzenetopoulos, Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris. Interference-Aware Orchestration in Kubernetes. In *International Conference on High Performance Computing*, pages 321–330. Springer, 2020.

27 Y Ukidave, X Li, and D Kaeli. Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 353–362, 2016. doi:10.1109/IPDPS.2016.73.

28 Shaoqi Wang, Oscar J Gonzalez, Xiaobo Zhou, and Thomas Williams. An Efficient and Non-Intrusive GPU Scheduling Framework for Deep Learning Training Systems. *Sc*, 2020.

29 Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, pages 173–184, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3369583.3392679.

HPC Application Cloudification: The StreamFlow Toolkit

Iacopo Colonnelli
Computer Science Department, University of Torino, Italy

Barbara Cantalupo
Computer Science Department, University of Torino, Italy

Roberto Esposito
Computer Science Department, University of Torino, Italy

Matteo Pennisi
Electrical Engineering Department, University of Catania, Italy

Concetto Spampinato
Electrical Engineering Department, University of Catania, Italy

Marco Aldinucci
Department of Computer Science, University of Pisa, Italy

Abstract

Finding an effective way to improve accessibility to High-Performance Computing facilities, still anchored to SSH-based remote shells and queue-based job submission mechanisms, is an open problem in computer science.

This work advocates a cloudification of HPC applications through a cluster-as-accelerator pattern, where computationally demanding portions of the main execution flow hosted on a Cloud Finding an effective way to improve accessibility to High-Performance Computing facilities, still anchored to SSH-based remote shells and queue-based job submission mechanisms, is an open problem in computer science.

This work advocates a cloudification of HPC applications through a cluster-as-accelerator pattern, where computationally demanding portions of the main execution flow hosted on a Cloud infrastructure can be offloaded to HPC environments to speed them up. We introduce StreamFlow, a novel Workflow Management System that supports such a design pattern and makes it possible to run the steps of a standard workflow model on independent processing elements with no shared storage.

We validated the proposed approach's effectiveness on the CLAIRE COVID-19 universal pipeline, i.e. a reproducible workflow capable of automating the comparison of (possibly all) state-of-the-art pipelines for the diagnosis of COVID-19 interstitial pneumonia from CT scans images based on Deep Neural Networks (DNNs).

2012 ACM Subject Classification Computer systems organization → Cloud computing; Computing methodologies → Distributed computing methodologies

Keywords and phrases cloud computing, distributed computing, high-performance computing, streamflow, workflow management systems

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2021.5

Category Invited Paper

Supplementary Material *Software (Stable)*: `https://github.com/alpha-unito/streamflow`
archived at `swh:1:dir:34c00b970f3326937c7b1cb6467a4a2c4f5dbdec`
Other (StreamFlow website): `https://streamflow.di.unito.it`

Funding This work has been undertaken in the context of the DeepHealth project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 825111. This work has been partially supported by the HPC4AI project funded by Regione Piemonte (POR FESR 2014-20 - INFRA-P).

12th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 10th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2021).
Editors: João Bispo, Stefano Cherubin, and José Flich; Article No. 5; pp. 5:1–5:13

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

Acknowledgements We want to thank Emanuela Girardi and Gianluca Bontempi who are coordinating the CLAIRE task force on COVID-19 for their support, and the group of volunteer researchers who contributed to the development of CLAIRE COVID-19 universal pipeline, they are: Marco Calandri and Piero Fariselli (Radiomics & medical science, University of Torino, Italy); Marco Grangetto, Enzo Tartaglione (Digital image processing Lab, University of Torino, Italy); Simone Palazzo, Isaak Kavasidis (PeRCeiVe Lab, University of Catania, Italy); Bogdan Ionescu, Gabriel Constantin (Multimedia Lab @ CAMPUS Research Institute, University Politechnica of Bucharest, Romania); Miquel Perello Nieto (Computer Science, University of Bristol, UK); Inês Domingues (School of Sciences University of Porto, Portugal).

1 Introduction

If the technical barriers to Cloud-based infrastructures lowered substantially with the advent of the *-as-a-Service model, most High-Performance Computing (HPC) facilities worldwide are still anchored to SSH-based remote shells and queue-based job submission mechanisms.

Finding an effective way to improve accessibility to this class of computing resources is still an open problem in computer science. Indeed, the multiple layers of virtualisation that characterise modern cloud architectures introduce significant processing overheads and make it impossible to apply adaptive fine-tuning techniques based upon the underlying hardware technologies, making them incompatible with performance-critical HPC applications. On the other hand, HPC centres are not designed for general purpose applications. Only scalable and computationally demanding programs can effectively benefit from the massive amount of processing elements and the low-latency network interconnections that characterise HPC facilities, justifying the high development cost of HPC-enabled applications. Moreover, some seemingly trivial applications are not supported in HPC environments, e.g. exposing a public web interface for data visualisation in an air-gapped worker node.

In this work, we promote a *cluster-as-accelerator* design pattern, in which cluster nodes act as general interpreters of user-defined tasks sent by a general-purpose host executor residing on a Cloud infrastructure, as a way to offload computation to HPC facilities in an intuitive way. Moreover, we propose *hybrid workflows*, i.e. workflows whose steps can be scheduled on independent and potentially not intercommunicating execution environments, as a programming paradigm to express such design pattern.

The StreamFlow toolkit [8], a Workflow Management System (WMS) supporting workflow executions on hybrid Cloud-HPC infrastructures, is then used to evaluate the effectiveness of the proposed approach on a real application, a Deep Neural Network (DNN) training pipeline for COVID-19 diagnosis from Computed Tomography (CT) scans images.

In more detail, Section 2 gives a general background on workflow models and discusses the state of the art, while Section 3 introduces the proposed approaches. Section 4 describes the StreamFlow toolkit, which has been used to implement the COVID-related use case described in Section 5. Finally, Section 6 summarises conclusions and future works.

2 Background and related work

A workflow is commonly represented as an acyclic digraph $G = (N, E)$, where nodes refer to different portions of a complex program and edges encode *dependency relations* between nodes, i.e. a direct edge connecting a node m to a node n means that n must wait for m to complete before starting its computation.

When expressing a program in terms of a workflow model, it is necessary to distinguish between two different classes of semantics [16]:

- The *host semantics* defining the subprograms in workflow nodes, usually expressed in a general-purpose programming language such as C++, Java or Python;
- The *coordination semantics* defining the interactions between such nodes, i.e. expressing the workflow model itself.

Tools in charge of exposing coordination semantics to the users and orchestrating workflow executions are known as Workflow Management Systems (WMSs).

Given their extreme generality, workflow models represent a powerful abstraction for designing scientific applications and executing them on a diverse set of environments, ranging from the practitioners' desktop machines to entire HPC centres. In this vision, WMSs act as an interface between the domain specialists and the computing infrastructure.

The WMS landscape is very variegated, as it embraces a broad range of very diverse scientific domains. Nevertheless, when considering the coordination semantics exposed to the workflow designers, two main classes of tools can be identified: high-level products implementing a strict separation of concerns between workflow description and application code, and low-level distributed libraries directly intermingling workflow definition with business logic.

In the first category, many tools provide a simplified Domain Specific Language (DSL) for coordination. Some WMSs adopt a Unix-style approach to define workflows in a technology-neutral way, using a syntax similar to Make [2, 14], while others provide a dataflow programming model to express parallelism easily [11, 22]. Coordination DSLs are quite flexible, but they introduce a different formalism that users must learn. For this reason, some solutions prefer to hide or replace coordination languages with a higher level Graphical User Interface (GUI), trading off complexity against flexibility [1, 17, 7].

Since product-specific DSLs and GUIs tightly couple workflow models to a single WMS, limiting portability and reusability, there are also efforts in defining vendor-agnostic coordination languages or standards. The Common Workflow Language (CWL)[1] [6] is an open standard for describing analysis workflows, following a declarative JSON or YAML syntax. Many products offer support for CWL, either alongside a proprietary coordination DSL [15] or as a higher-level semantics on top of low-level APIs [21].

A strict separation between host code and coordination logics offers a considerable abstraction level, promoting ease of understanding, portability and reproducibility across diverse execution infrastructures, and code reusability for similar purposes. Nevertheless, the significant overhead introduced by this separation of concerns makes these approaches suitable only for coarse-grained workflows, where each step performs a considerable amount of computation.

An alternative to complex and feature-rich WMSs, which puts performances first, is represented by low-level distributed libraries [23, 20, 19], which directly expose coordination programming models in the host code. Such libraries commonly allow for the execution of many interdependent tasks on distributed architectures, supporting scenarios with low-latency or high-throughput requirements. Despite being very efficient in terms of performances, these libraries are hard to use for domain experts without programming experience.

When considering targeted execution environments, both categories of solutions come with some limitations that prevent full support for hybrid architectures, i.e. mixed Cloud-HPC infrastructures. High-level products usually come with a set of pluggable connectors

[1] `https://www.commonwl.org`

targeting a broad range of infrastructures, e.g. public cloud services, batch schedulers (e.g. HTCondor, PBS, SLURM) and Kubernetes clusters. Nevertheless, in most competing approaches different steps of the same workflow cannot be managed by different connectors, forcing users to stick with the same infrastructure for the whole execution flow.

On the other side, low-level distributed libraries commonly require the presence of a shared file-system accessible by all worker nodes in the cluster (e.g. LUSTRE or HDFS), which is hardly the case in hybrid Cloud-HPC settings. Moreover, inter-node communication protocols require a bidirectional connection between the controller and the worker agents, which is not compliant with air-gapped computing nodes that usually characterise HPC facilities.

Even if some tools [10, 13] offer support for automatic data transfers among worker nodes, therefore being compatible with hybrid architectures, they rely on specific transfer protocols (e.g. GridFTP, SRM or Amazon S3) or delegate workflow management to an external batch scheduler such as HTCondor, actually constraining the set of supported configurations.

3 Hybrid workflows

The workflow abstraction has already been explored for offloading computation to HPC facilities in a transparent way. Indeed, as discussed in Section 2, many of the existing WMSs come with a diverse set of *connectors*, some of them addressing cloud environments and some others more HPC-oriented. Nevertheless, a far smaller percentage can deal with hybrid cloud/HPC scenarios for executing a single workflow.

Unfortunately, HPC facilities are not well suited for every kind of application. When executing complex workflows, it is common to have computation-intensive and highly parallelisable steps alternate with sequential or non-compute-bound operations. When scheduling an application of this kind to an HPC centre, only a subset of workflow steps will effectively take advantage of the available computing power, resulting in a low cost-benefit ratio. Moreover, some operations are not supported in HPC facilities, e.g. exposing a web interface for data visualisation in an air-gapped data centre.

In this vision, a WMS capable of dealing with *hybrid workflows*, i.e. able to schedule and coordinate different steps of a workflow on different execution environments [9], represents a crucial step towards a standard task-based interface to distributed computing. Indeed, the possibility to assign each portion of a complex application to the computing infrastructure that best suits its requirements strongly reduces the necessary tradeoffs in relying on such high-level abstraction, both in terms of performances and costs.

A fundamental step to realising a hybrid WMS is to waive the requirement for any shared data access space among all the executors, which is instead a constraint imposed by a broad range of WMSs on the market. Indeed, it is hardly the case that an HPC centre and a Cloud infrastructure can share a common file-system. This requirement can be removed by letting the runtime system automatically handle data movements among different executors whenever needed, keeping the only constraint for the WMS management infrastructure to be able to reach the whole execution environment.

This strategy enables data movements between every pair of resources, and guarantees that each of them requires at most two transfer operations: one from the source to the management infrastructure and one from the management infrastructure to the destination. Nevertheless, a two-step copy can represent an unsustainable overhead when dealing with massive amounts of data, a common scenario in modern scientific pipelines.

Making the WMS aware of a workflow's hybrid nature and letting it autonomously manage data movements are crucial aspects for performance optimisation in such scenarios. First of all, scheduling policies privileging data locality can minimise the number of required

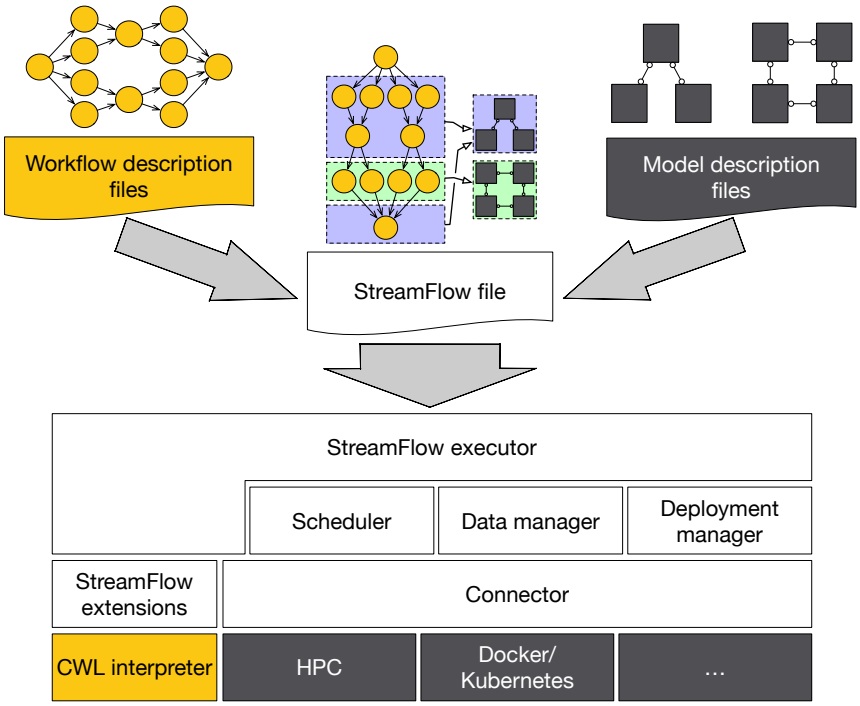


Figure 1 StreamFlow toolkit's logical stack. The yellow area is related to the definition of the workflow's dependency graph, the dark grey area refers to the execution environments, and the white portions are directly part of StreamFlow codebase.

data transfer operations, moving computation near data whenever possible. Moreover, if a WMS is aware of the underlying infrastructure topology, it can use the two-step copy strategy only as a last resort, privileging direct communication channels whenever available.

The hybrid workflows paradigm allows software architects to adopt a *cluster-as-accelerator* design pattern [12], in which cluster nodes act as general interpreters of user-defined tasks sent by a host executor. A similar pattern has been proven very effective to offload computationally heavy operations to dedicated hardware. For example, cryptographic accelerators have long been used for offloading security-related computations from the CPU, while GPGPUs programming models adopt an accelerator pattern to cooperate with the host execution flow.

In this sense, we envision hybrid workflows as a way to enable *cloudified* HPC applications, where domain experts can interact with the host execution flow through the user-friendly *-as-a-Service paradigm, but computationally demanding steps can be easily offloaded to data centers by means of high-level coordination primitives.

4 The StreamFlow toolkit

The StreamFlow[2] tookit [8], whose logical stack is depicted in Figure 1, has been specifically developed to orchestrate hybrid workflows on top of heterogeneous and geographically distributed architectures.

Written in Python 3, it can seamlessly integrate with the CWL coordination standard [6] for expressing workflow models. Alongside, one or more execution environments can be described in well-known formats, e.g. Helm charts for Kubernetes deployments or Slurm

[2] `https://streamflow.di.unito.it/`

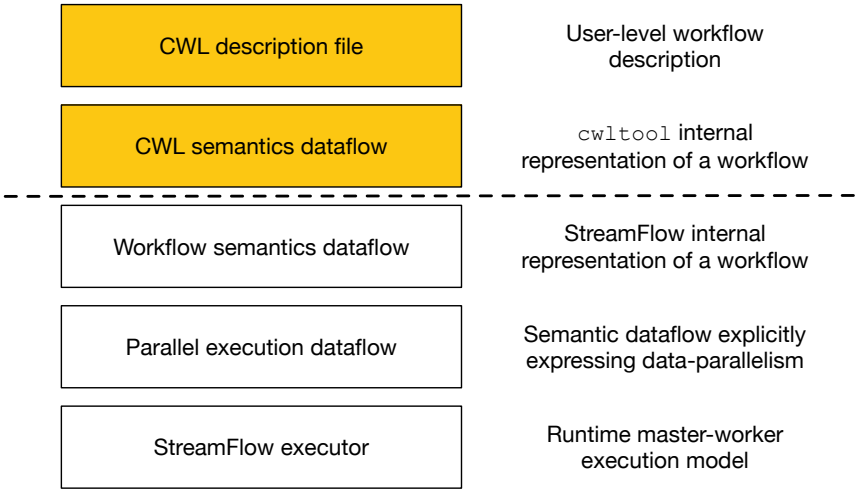


Figure 2 StreamFlow toolkit's layered dataflow model. Yellow blocks refer to the CWL runtime library's workflow representations, called cwltool, while the white ones are internal representations adopted by the different layers of the StreamFlow toolkit.

files for queue jobs. A `streamflow.yml` file, the entry point of a StreamFlow run, is then in charge of relating each workflow step with the best suitable execution environment, actually plugging the hybrid layer in the workflow design process.

CWL semantics can be used to describe a workflow through a declarative JSON or YAML syntax, written in one or more files with `.cwl` extension. Plus, an additional configuration file contains a list of input parameters to initialise a workflow execution. The CWL reference implementation, called `cwltool`, is in charge of translating these declarative semantics into an executable workflow model, such that the runtime layer can efficiently execute independent steps concurrently.

A commonly adopted executable representation of a workflow is the *macro dataflow graph* [4]. Each node of this graph can be represented as a tuple $\langle c_k, In(c_k), Out(c_k)\rangle$, where:

- c_k is a command encoding a coarse-grained computation;
- $In(c_k) = \{i_{kj} : j \in [1, m]\}$ is the set of *input ports*, i.e. the input dependencies of c_k;
- $Out(c_k) = \{o_{kj} : j \in [1, n]\}$ is the set of *output ports*, i.e. the values returned by c_k.

Each edge $\langle o_{kj}, i_{lh}\rangle$ of the graph encodes a *dependency relation* going from node k to node l, meaning that node l will receive a token on its input port i_{lh} from the output port o_{kj} of the node k. When a node receives a token on each of its input ports, it enters the *fireable* state and can be scheduled for execution. At any given time, all fireable nodes can be concurrently executed by the runtime layer, provided that enough compute units are available.

The `cwltool` library natively translates the CWL semantics in a low-level macro dataflow graph, and it also implements a multi-threaded runtime support. Nevertheless, even if CWL is the primary coordination language in StreamFlow, the integration with additional workflow design tools and formats is in plans, making it worthwhile to avoid too tight coupling between CWL logics and the StreamFlow runtime.

For this reason, the StreamFlow toolkit adopts a *layered dataflow model* [18], as depicted in Figure 2. As a first step, StreamFlow translates the CWL dataflow semantics into an internal workflow representation, explicitly modelling a macro dataflow graph. It is worth noting that this representation supports much broader semantics, including loops, stream-based input ports and from-any activation policies.

When dealing with explicit parallel semantics, whether they are data-parallel constructs like scatter/gather or stream-parallel patterns like pipeline executions, the same node of a dataflow graph can be executed multiple times. Therefore, the runtime support needs a lower, parallelism-aware layer, capable of representing each workflow step as the set of its execution units. In StreamFlow, such execution units are called *jobs* and are the only entities directly visible to the underlying runtime components for scheduling, execution and fault tolerance purposes.

Another distinctive feature of the StreamFlow WMS is the possibility to manage complex, multi-agent execution environments, ensuring the *co-allocation* of multiple heterogeneous processing elements to execute a single workflow step. The main advantage is introducing a unique interface to a diverse ecosystem of distributed applications, ranging from MPI clusters running on HPC facilities to microservices architectures deployed on Kubernetes.

To provide enough flexibility, StreamFlow adopts a three-layered hierarchical representation of execution environments:

- A `model` is an entire multi-agent infrastructure and constitutes the *unit of deployment*, i.e. all its components are always co-allocated when executing a step;
- A `service` is a single agent in a model and constitutes the *unit of binding*, i.e. each step of a workflow can be offloaded to a single service for execution;
- A `resource` is a single instance of a potentially replicated service and constitutes the *unit of scheduling*, i.e. each step of a workflow is offloaded to a configurable number of service resources to be processed.

Each model is deployed and managed independently of the others by a dedicated `Connector` implementation, which acts as a proxy for an external orchestration library. All `Connector` classes inherit from a unique base interface, so that support for different execution environments can be added to the codebase by merely developing a new implementation and plugging it in the StreamFlow runtime. In particular, the `DeploymentManager` class has the role of invoking the appropriate `Connector` implementation to create and destroy models whenever needed.

When a step becomes fireable, and the corresponding model has been successfully deployed, a `Scheduler` class is in charge of selecting the best resource to execute it. Indeed, even if only a single target service can be specified for each task, multiple replicas of the same service could exist at the same time and, if the underlying orchestrator provides auto-scaling features, their number could also change in time. Therefore, the `Scheduler` class relies on the appropriate `Connector` to extract the list of compatible resources for a given step.

Then, a scheduling policy is required to choose the best one. Given the very complex nature of hybrid workflows, a universally best scheduling strategy hardly exists. Indeed, many different factors (e.g. computing power, data locality, load balancing) can affect the overall execution time. A `Policy` interface has been introduced to allow for pluggable scheduling strategies, with a default implementation trying to minimise the data movement overhead by privileging data locality aspects.

Finally, the `DataManager` class must ensure that each service can access all its input dependencies and perform data transfers whenever necessary. As discussed in Section 3, it is always possible to move data between resources with a two-step copy operation involving the StreamFlow management infrastructure. In particular, StreamFlow always relies on this strategy for inter-model data transfers. Conversely, intra-model copies are performed by the `copy` method of the corresponding `Connector`, which is aware of the infrastructure topology and can open direct connections between resources whenever possible.

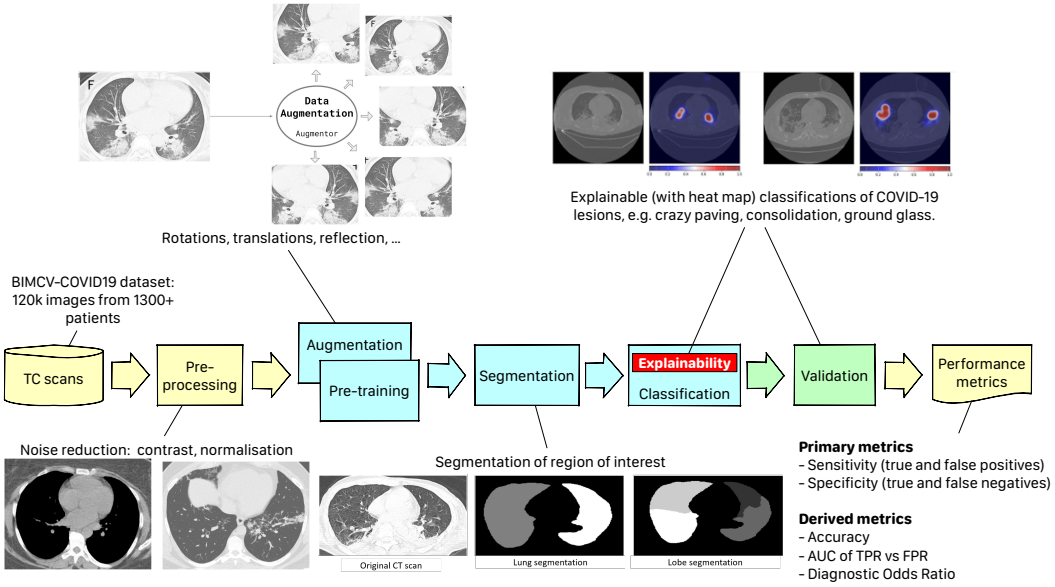


Figure 3 The CLAIRE COVID-19 universal pipeline.

It is also worth noting that all communications and data transfer operations are started and managed by the StreamFlow controller, removing the need for a bidirectional channel between the management infrastructure and the target resources. Therefore, tasks can also be offloaded to HPC infrastructures with air-gapped worker nodes, since StreamFlow directly interacts only with the frontend layer.

Moreover, StreamFlow does not need any specific package or library to be installed on the target resources, other than the software dependencies required by the host application. Therefore, virtually any target infrastructure reachable by a practitioner can serve as a target model, as long as a compatible `Connector` implementation is available.

5 Use case: the CLAIRE COVID-19 universal pipeline

To demonstrate how StreamFlow can help bridge HCP and AI workloads, we present the *CLAIRE COVID-19 universal pipeline* developed by the task force on AI & COVID-19 during the first COVID-19 outbreak to study AI-assisted diagnosis of interstitial pneumonia.

COVID-19 infection caused by the SARS-CoV-2 pathogen is a catastrophic pandemic outbreak worldwide with an exponential increase in confirmed cases and, unfortunately, deaths. When the pandemic broke out, among the initiatives aimed at improving the knowledge of the virus, containing its diffusion, and limiting its effects, the Confederation of Laboratories for Artificial Intelligence Research in Europe (CLAIRE)[3] task force on AI & COVID-19 supported the set up of a novel European group to study the diagnosis of COVID-19 pneumonia assisted by Artificial Intelligence (AI). The group was composed of fifteen researchers in complementary disciplines (Radiomics, AI, and HPC) led by Prof. Marco Aldinucci [3].

At the start of the pandemic, several studies outlined the effectiveness of radiology imaging for COVID-19 diagnosis through chest X-Ray and mainly Computed Tomography (CT), given the pulmonary involvement in subjects affected by the infection. Considering

[3] `https://https://claire-ai.org/`

the extension of the infection and the number of cases that daily emerged worldwide, the need for fast, robust, and medically sustainable diagnosis appeared fundamental. Applying artificial intelligence to radiology images to make the whole diagnosis process automatic, while reducing the efforts required by radiologists for visual inspection was relatively straight.

Even if X-Ray represents a cheaper and most effective solution for large scale screening, its low resolution led AI models to show lower accuracy than those obtained with CT data. Therefore, CT scan has become the gold standard for investigation on lung diseases. Several research groups worldwide began to develop deep-learning models for the diagnosis of COVID-19, mainly in the form of deep Convolutional Neural Networks (CNN), applying lung disease analysis from CT scans images.

As soon as we started analysing all the solution proposed, it was evident that it is impossible to select the most promising ones, due to the use of different architectures, pipelines and datasets. So, we started working on defining a reproducible workflow capable of automating the comparison of state-of-the-art deep learning models to diagnose COVID-19.

The workflow subsequently evolved towards the CLAIRE COVID-19 universal pipeline (Figure 3). This pipeline can reproduce different state-of-the-art AI models existing in the literature for the analysis of medical images. They include the pipeline for the diagnosis of COVID-19 interstitial pneumonia and other diseases. The pipeline is designed to compare the different training algorithms and therefore to define a baseline for these techniques allowing the community to quantitatively measure the progress of AI in the diagnosis of COVID-19 and similar diseases.

The universal pipeline comprises two initial steps: *Image Preprocessing* and *Augmentation*, where standard techniques for cleaning and generating variants of training images are applied. The final stages are also typical pipeline components implementing *Validation* of results and *Performance Metrics* collection.

The core steps are DNN-based. They are *Pre-training*, *Segmentation* and *Classification*. Pre-training is an unsupervised learning step and aims to generate a first set of weights for the next two steps, typically based on supervised learning. The segmentation step isolates the region of interest (e.g. lung from other tissues), and the classification step labels each image with a class identified with a kind of lesion that is typical of the disease. Each of the steps can be implemented using different DNNs, generating different variants of the pipeline. For each of these stages we selected the best DNNs that have been experimented in literature, together with a systematic exploration of networks hyperparameters, allowing a deeper search for the best model. As it can be deduced from Figure 4, the resulting number of the CLAIRE COVID-19 pipelines variants is 990.

Theoretically, the universal pipeline can reproduce the training of all the best existing and forthcoming models to diagnose pneumonia and compare their performance. Moving from theory to practice requires two non-trivial ingredients: a supercomputer of adequate computational power equipped with many latest generation GPUs and a mechanism capable of unifying and automating the execution of all variants of the workflow on a supercomputer.

To set up experiments on the pipeline, we chose the biggest dataset publicly available related to COVID-19's pathology course, i.e. BIMCV-COVID19[4], with more than 120k images from 1300 patients. Supposing to train each pre-trained model for 20 epochs on such dataset, a single variant of the pipeline takes over 15 hours on a single NVidia V100 GPU, one of the most powerful accelerators in the market with more than 5000 CUDA cores.

[4] `https://bimcv.cipf.es/bimcv-projects/bimcv-covid19/`

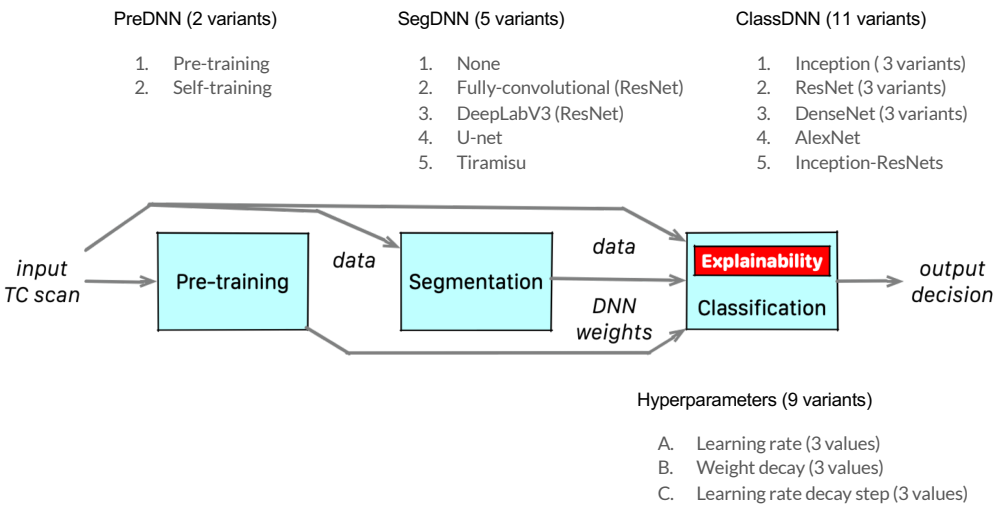


Figure 4 Core components of the CLAIRE COVID-19 universal pipeline and their variants.

Therefore, exploring all the 990 pipeline variants would take over two years on the most powerful GPU currently available and it should also be considered that tuning models for each variant would need more than one execution, too.

Fortunately, since the universal pipeline has an embarrassingly parallel structure (Figure 5), using a supercomputer could reduce the execution time down to just one day. In the best case, running all the variants concurrently on 990 different V100 GPUs only takes 15 hours of wall-clock time. Nevertheless, post-training steps like performance metrics extraction and comparison are better suited for a Cloud infrastructure, as they do not require much computing power and can significantly benefit from web-based visualisation tools. Therefore, the optimal execution of the pipeline advocates a cluster-as-accelerator design pattern.

The bridging of AI and HPC execution models has been solved by managing the universal pipeline with StreamFlow. The pipeline is naturally modelled as a hybrid workflow, offloading the training portions to an HPC facility and collecting the resulting networks on the host execution flow for visualisation purposes. As an interface towards Cloud-HPC infrastructures, StreamFlow automatically manages data movements and remote step execution, providing fault tolerance mechanisms such as checkpointing of intermediate results and replay-based recovery.

At the time of writing, the experiments have begun, and we see the first encouraging results. We performed the analysis of about 1% of the variants (10 of 990) on the High-Performance Computing for Artificial Intelligence (HPC4AI) at the University of Torino, a multi-tenant hybrid Cloud-HPC system with 80 cores and 4 GPUs per node (T4 or V100-SMX2) [5]. Results show that the CLAIRE COVID-19 universal pipeline can generate models with excellent accuracy in classifying typical interstitial pneumonia lesions due to COVID-19, with sensitivity and specificity metrics over 90% in the best cases.

6 Conclusion and future work

HPC is an enabling platform for scientific computing and Artificial Intelligence and a fundamental tool for high impact research, such as AI-assisted analysis of medical images, personalised medicine, seismic resiliency, and the green new deal. HPC and Cloud computing

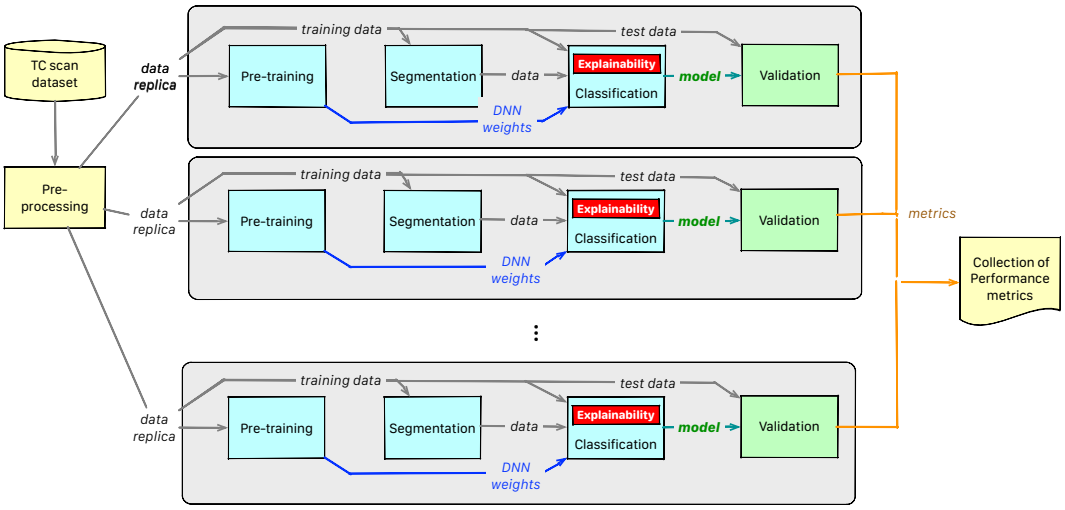


Figure 5 CLAIRE-COVID19 universal pipeline unfolded.

are at the frontier of EU digital sovereignty, which, together with the green new deal, are two cornerstones of the EU agenda.

With EU 8B€ funding, HPC ranks first for the funding volume in the EU Digital Europe 2021-27. Propelled by Artificial Intelligence, the HPC market analysis reports an overall CAGR19-24 of 32.9%[5], where health is among the driver domains, and where the high-end HPC (supercomputing) market grows much faster than other segments. However, Europe struggles to mature an HPC value chain, from advanced research to innovation.

The HPC ecosystem is partitioned into applications, system software, and infrastructures. We believe that the mainstream industrial adoption of HPC requires a system software part, enabling technology to transform applications into easily usable services hence into innovation. While in scientific computing the modernisation of HPC applications is a scientific desideratum required to boost industrial adoption, in AI the shift toward the Cloud model of services is a must. AI applications are already modern, and they will not step back.

In the HPC landscape, AI requires a significant shift in current software technology. Computing resources should be available on-demand as a service, and data should be kept secure in public and shared infrastructures. StreamFlow, leveraging modern virtualisation technologies, advocates a new methodology to assemble existing legacy codes in a portable and malleable way.

In this work, we propose the cluster-as-accelerator design pattern as a way to enable HPC applications cloudification, allowing practitioners to minimise the price-performance ratio. Moreover, we advocate hybrid workflow models as an intuitive programming paradigm to express such design pattern, reducing technical barriers to HPC facilities for domain experts without a strong computer science background.

The CLAIRE-COVID19 universal pipeline has been designed according to these principles, offloading training-related steps to an HPC centre and collecting back the resulting networks on the host execution flow, located on a Cloud infrastructure, for visualisation purposes.

[5] Compound Annual Growth Rate.
Source: `https://orau.gov/ai_townhall/presentations/1115am-Hyperion_Research_AI_Research.pdf`

The StreamFlow toolkit, a Workflow Management System supporting hybrid workflow executions, has been successfully used to perform a first portion of the planned experiments, proving the proposed approach's effectiveness in the AI domain.

The experiments will continue to execute all the variants on HPC4AI, applying the pipeline on the complete dataset. However, the main goal will be setting a comprehensive framework, where StreamFlow could manage the pipeline execution on different hybrid HPC infrastructures.

References

1 Enis Afgan, Dannon Baker, Bérénice Batut, Marius van den Beek, Dave Bouvier, Martin Cech, John Chilton, Dave Clements, Nate Coraor, Björn A. Grüning, Aysam Guerler, Jennifer Hillman-Jackson, Saskia D. Hiltemann, Vahid Jalili, Helena Rasche, Nicola Soranzo, Jeremy Goecks, James Taylor, Anton Nekrutenko, and Daniel J. Blankenberg. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. *Nucleic Acids Res.*, 46(Webserver-Issue):W537–W544, 2018. doi:10.1093/nar/gky379.

2 Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, SWEET@SIGMOD 2012, Scottsdale, AZ, USA, May 20, 2012*, page 1, 2012. doi:10.1145/2443416.2443417.

3 Marco Aldinucci. High-performance computing and AI team up for COVID-19 diagnostic imaging. https://aihub.org/2021/01/12/high-performance-computing-and-ai-team-up-for-covid-19-diagnostic-imaging/, January 2021. Accessed: 2021-01-25.

4 Marco Aldinucci, Marco Danelutto, Lorenzo Anardu, Massimo Torquati, and Peter Kilpatrick. Parallel patterns + macro data flow for multi-core programming. In *Proc. of Intl. Euromicro PDP 2012: Parallel Distributed and network-based Processing*, pages 27–36, Garching, Germany, February 2012. IEEE. doi:10.1109/PDP.2012.44.

5 Marco Aldinucci, Sergio Rabellino, Marco Pironti, Filippo Spiga, Paolo Viviani, Maurizio Drocco, Marco Guerzoni, Guido Boella, Marco Mellia, Paolo Margara, Idillio Drago, Roberto Marturano, Guido Marchetto, Elio Piccolo, Stefano Bagnasco, Stefano Lusso, Sara Vallero, Giuseppe Attardi, Alex Barchiesi, Alberto Colla, and Fulvio Galeazzi. HPC4AI, an AI-on-demand federated platform endeavour. In *ACM Computing Frontiers*, Ischia, Italy, May 2018. doi:10.1145/3203217.3205340.

6 Peter Amstutz, Michael R. Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, John Kern, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, Matt Scales, Stian Soiland-Reyes, and Luka Stojanovic. Common workflow language, v1.0, 2016. doi:10.6084/m9.figshare.3115156.v2.

7 Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel. KNIME: the Konstanz Information Miner. In *Data Analysis, Machine Learning and Applications - Proceedings of the 31st Annual Conference of the Gesellschaft für Klassifikation e.V., Albert-Ludwigs-Universität Freiburg, March 7-9, 2007*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 319–326. Springer, 2007. doi:10.1007/978-3-540-78246-9_38.

8 Iacopo Colonnelli, Barbara Cantalupo, Ivan Merelli, and Marco Aldinucci. StreamFlow: cross-breeding cloud with HPC. *IEEE Transactions on Emerging Topics in Computing*, August 2020. doi:10.1109/TETC.2020.3019202.

9 Rafael Ferreira da Silva, Rosa Filgueira, Ilia Pietri, Ming Jiang, Rizos Sakellariou, and Ewa Deelman. A characterization of workflow management systems for extreme-scale applications. *Future Gener. Comput. Syst.*, 75:228–238, 2017. doi:10.1016/j.future.2017.02.026.

10 Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and R. Kent Wenger. Pegasus, a workflow management system for science automation. *Future Generation Comp. Syst.*, 46:17–35, 2015. doi:10.1016/j.future.2014.10.008.

11 Paolo Di Tommaso, Maria Chatzou, Evan W. Floden, Pablo P. Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35(4):316–319, April 2017. doi:10.1038/nbt.3820.

12 Maurizio Drocco, Claudia Misale, and Marco Aldinucci. A cluster-as-accelerator approach for SPMD-free data parallelism. In *Proc. of 24th Euromicro Intl. Conference on Parallel Distributed and network-based Processing (PDP)*, pages 350–353, Crete, Greece, 2016. IEEE. doi:10.1109/PDP.2016.97.

13 Thomas Fahringer, Radu Prodan, Rubing Duan, Jürgen Hofer, Farrukh Nadeem, Francesco Nerieri, Stefan Podlipnig, Jun Qin, Mumtaz Siddiqui, Hong Linh Truong, Alex Villazón, and Marek Wieczorek. ASKALON: A development and grid computing environment for scientific workflows. In *Workflows for e-Science, Scientific Workflows for Grids*, pages 450–471. Springer, 2007. doi:10.1007/978-1-84628-757-2_27.

14 Johannes Köster and Sven Rahmann. Snakemake - a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012. doi:10.1093/bioinformatics/bts480.

15 Michael Kotliar, Andrey V Kartashov, and Artem Barski. CWL-Airflow: a lightweight pipeline manager supporting Common Workflow Language. *GigaScience*, 8(7), July 2019. doi:10.1093/gigascience/giz084.

16 E.A. Lee and T.M. Parks. Dataflow process networks. *Proc. of the IEEE*, 83(5):773–801, May 1995. doi:10.1109/5.381846.

17 Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew B. Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. doi:10.1002/cpe.994.

18 Claudia Misale, Maurizio Drocco, Marco Aldinucci, and Guy Tremblay. A comparison of big data frameworks on a layered dataflow model. *Parallel Processing Letters*, 27(01):1–20, 2017. doi:10.1142/S0129626417400035.

19 Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 561–577, 2018.

20 Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M. Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. PyCOMPSs: Parallel computational workflows in Python. *Int. J. High Perform. Comput. Appl.*, 31(1):66–82, 2017. doi:10.1177/1094342015594678.

21 John Vivian, Arjun A. Rao, Frank A. Nothaft, et al. Toil enables reproducible, open source, big biomedical data analyses. *Nature Biotechnology*, 35(4):314–316, April 2017. doi:10.1038/nbt.3772.

22 Justin M. Wozniak, Michael Wilde, and Ian T. Foster. Language features for scalable distributed-memory dataflow computing. In *Proceedings of the 2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, DFM '14, page 50–53, USA, 2014. IEEE Computer Society. doi:10.1109/DFM.2014.17.

23 Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016. doi:10.1145/2934664.