16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures

14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms

PARMA-DITAM 2025, January 22, 2025, Barcelona, Spain

Edited by
Daniele Cattaneo
Maria Fazio
Leonidas Kosmidis
Gabriele Morabito

Editors

Daniele Cattaneo
Politecnico di Milano, Italy
daniele.cattaneo@polimi.it

Maria Fazio
University of Messina, Italy
mfazio@unime.it

Leonidas Kosmidis
Barcelona Supercomputing Center (BSC), Spain
leonidas.kosmidis@bsc.es

Gabriele Morabito
University of Messina, Italy
gamorabito@unime.it

ACM Classification 2012
Computer systems organization → Parallel architectures; Software and its engineering → Real-time schedulability; Software and its engineering → Parallel programming languages; Hardware → Methodologies for EDA; Hardware → High-level and register-transfer level synthesis; Hardware → Very large scale integration design; Hardware → Reconfigurable logic and FPGAs; Computer systems organization → Embedded systems; Computer systems organization → Embedded hardware; Computer systems organization → Reliability; Software and its engineering → Compilers; Computing methodologies → Graphics processors; General and reference → Cross-computing tools and techniques; Computer systems organization → Embedded and cyber-physical systems; Applied computing → Aerospace; Software and its engineering → Software safety

ISBN 978-3-95977-363-8

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/978-3-95977-363-8.

Publication date
February, 2025

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.2025.0
ISBN 978-3-95977-363-8 **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

Contents

Papers

16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2025).
Editors: Daniele Cattaneo, Maria Fazio, Leonidas Kosmidis, and Gabriele Morabito
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume collects the papers presented at the 16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures, and the 14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2025). The workshop is co-located with the 2025 edition of the HiPEAC conference and was held on the 22nd of January, 2025, that took place in Barcelona, Spain.

The current trend towards many-core and the emerging accelerator-based architecture requires a global rethinking of software and hardware design, which turn out to be more than ever before strongly entangled.

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 and boost the performance of such (possibly heterogeneous, (re-)programmable and/or (re-)configurable) many-core processor architectures from embedded to high performance computing platforms and cyber physical systems.

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

- T1: Parallel programming models, languages, and applications for many-core platforms
- T2: Compiler and virtualization techniques for novel computing architectures
- T3: Run-time modeling, monitoring, adaptivity, power and memory management
- T4: Design of heterogeneous and reconfigurable many-core architectures
- T5: Methodologies, design tools, and high-level synthesis for heterogeneous architectures
- T6: Hardware/software co-design and design space exploration
- T7: Case studies, success stories and applications applying T1–T6

16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2025).
Editors: Daniele Cattaneo, Maria Fazio, Leonidas Kosmidis, and Gabriele Morabito
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Alejandro J. Calderón (1)
Ikerlan Technology Research Center, Mondragón, Spain

Pau López Castillón (4)
Universitat Politècnica de Barcelona (UPC), Spain;
Barcelona Supercomputing Center (BSC), Spain

Serena Curzel (2)
Politecnico di Milano, Italy

Daniele Frigioni (3)
University of L'Aquila, Italy

Marco Gribaudo (2)
Politecnico di Milano, Italy

Xavier Caricchio Hernández (4)
Universitat Politècnica de Barcelona (UPC), Spain;
Barcelona Supercomputing Center (BSC), Spain

Leonidas Kosmidis (1, 4, 5)
Barcelona Supercomputing Center (BSC), Spain;
Universitat Politècnica de Barcelona (UPC), Spain

Adrian Munera (6)
Barcelona Supercomputing Center, Spain

Vittoriano Muttillo (3)
University of Teramo, Italy

Alberto Perugini (5)
Barcelona Supercomputing Center (BSC), Spain;
Universitat Politècnica de Catalunya (UPC), Barcelona, Spain

Josep Pinot (6)
Barcelona Supercomputing Center, Spain

Luigi Pomante (3)
University of L'Aquila, Italy

Marcos Rodriguez (1)
Ikerlan Technology Research Center, Mondragón, Spain;
Universitat Politècnica de Catalunya, Barcelona, Spain

Sara Royuela (6)
Barcelona Supercomputing Center, Spain

Marco Santic (3)
University of L'Aquila, Italy

Vincenzo Stoico (3)
Vrije Universiteit Amsterdam, The Netherlands

Giacomo Valente (3)
University of L'Aquila, Italy

Irune Yarza (1)
Ikerlan Technology Research Center, Mondragón, Spain

Chenle Yu (6)
Barcelona Supercomputing Center, Spain

16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2025).
Editors: Daniele Cattaneo, Maria Fazio, Leonidas Kosmidis, and Gabriele Morabito
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Analysis of GPU Memory Allocation Characteristics

Marcos Rodriguez
Ikerlan Technology Research Center, Mondragón, Spain
Universitat Politècnica de Catalunya, Barcelona, Spain

Irune Yarza
Ikerlan Technology Research Center, Mondragón, Spain

Leonidas Kosmidis
Barcelona Super Computing Centre (BSC), Spain

Alejandro J. Calderón
Ikerlan Technology Research Center, Mondragón, Spain

Abstract

The number of applications subject to safety-critical regulations is on the rise, and consequently, the computing requirements for such applications are increasing as well. This trend has led to the integration of General-Purpose Graphics Processing Units (GPGPUs) into these systems. However, the inherent characteristics of GPGPUs, including their black-box nature, dynamic allocation mechanisms, and frequent use of pointers, present challenges in certifying these applications for safety-critical systems.

This paper aims to shed light on the unique characteristics of GPU programs and how they impact the certification process. To achieve this goal, several allocation methods are rigorously evaluated to determine which one is best suited to an application, regarding the program characteristics within the safety-critical domain.

By conducting this evaluation, we seek to provide insights into the complexities of GPU memory accesses and its compatibility with safety-critical requirements. The ultimate objective is to offer recommendations on the most appropriate allocation method based on the unique needs of each application, thus contributing to the safe and reliable integration of GPGPUs into safety-critical systems.

2012 ACM Subject Classification Computer systems organization → Parallel architectures; Software and its engineering → Real-time schedulability; Software and its engineering → Parallel programming languages

Keywords and phrases CUDA, Memory allocation, Rodinia, Embedded

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2025.1

Supplementary Material
Dataset (Experiment results): `https://github.com/marcosrc92/MARS-data` [20]
archived at `swh:1:dir:0b2fbba6fbdfa60cb3d84175a2dd102bfb293ff2`

Funding The research presented throughout this paper has received funding from the European Commission's Horizon Europe programme under the METASAT project (grant agreement 101082622), the Basque Government through the ELKARTEK programme within the framework of the AUTOTRUST project (grant number KK-2023/00019) and by the CERVERA programme within the framework of the MEDUSA project (grant number CER-20231011).

1 Introduction

Safety-critical systems have long been integral to various sectors, including aviation, nuclear power generation, healthcare, and more. In today's world, these systems are even more prevalent in our daily lives with the development of autonomous systems. Therefore, it is crucial to ensure that the development and deployment prioritise safety and adhere to rigorous standards to protect occupants, pedestrians, and workers on industrial environments.

High Performance Computing (HPC) platforms are those which are designed to accelerate data processing to solve complex and computationally intensive problems, some architectures include the use of Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs) or Tensor Processing Units (TPUs). These accelerators are increasingly establishing their presence on these sectors by delivering accelerated computations within real-time constraints. However, these heterogeneous systems have inherent limitations, with GPU schedulers exhibiting a black-box behaviour. Additionally, a well-known issue is the bottleneck that arises in the data transfer process between the host system (CPU side) and the device (accelerator side), leading to unpredictable data access times. Over the years, independent authors and companies have dedicated their efforts to develop memory allocation algorithms and architectures specifically aimed at achieving faster data accessing and deterministic behaviour of GPUs.

To ensure the reliability and uniformity of outcomes, this paper conducts an evaluation of various allocation methods using the Rodinia benchmark suite [10, 11, 18, 1]. The assessment encompasses both static and dynamic attributes extracted from the benchmarks, employing tools offered by NVIDIA and other contributors. Static metrics, such as data size, the number of allocations, copies, kernel launches, cache hit rate, and memory coalescence, are taken into account. Moreover, the dynamic aspects inherent in any code, such as allocation, data copying, kernel launching API usage, and overall execution time, are meticulously documented to discern potential connections with the static ones.

A significant enhancement to the benchmark analysis includes a desirable feature for safety-critical systems, the ability to control the timing of memory allocations. Leveraging the *XeroZerox* a tool introduced by [8], that allows that the entire memory allocation is executed only once and exclusively at the beginning of the execution for the NVIDIA allocation methods, we were able to achieve that behaviour. That tool has support to create this memory pool using Unified Memory (UM) and zero-copy (ZC). In this work, we add support to the traditional allocation method to grant an equal comparison between all results gathered from each memory configuration. Our analysis aims to provide conclusions regarding the most suitable allocation method based on the identified program characteristics, regarding mean measurements for those metrics to reveal the swiftest method, reinforced with assessments of standard deviation and histogram representations to gauge predictability, thereby providing valuable insights for informed decision-making.

The organisation of this paper is as follows: Section 2 introduces the most relevant previous works that inspired this study. Section 3 presents the memory managing methods that have been selected to conduct the time analysis on memory accesses that is presented in this work. Section 4 describes *XeroZerox* highlighting our modifications. Section 5 describes the static characteristics and dynamic metrics used for evaluating how they are related and affect the timing response. Section 6 exposes the data treatment, the value extracted from executing every benchmark with different memory configuration and a comparative analysis extracted from the results. Finally, Section 7 summarises the most important ideas and outline future work to be undertaken.

2 Related Work

In this section we present the prior work in the field, aiming to contextualise our research, identify gaps, and build on established frameworks. We identify two categories of memory management, a) studies which aim into new allocation strategies and b) studies which extract conclusions from reverse engineering the GPU. Our work is influenced by two streams: studies focused on new allocation strategies and those aimed at understanding memory through reverse engineering. We employ these techniques to get metrics, using tools designed to alter memory behaviour, to highlight which characteristics are relevant during program execution.

2.1 Allocation strategies

A. Calderón introduces a tool named *XeroZerox*, which is specifically designed for embedded platforms [8]. This open source tool carries out a two-phase analysis and modification process on the source code. In the first phase, it intercepts explicit GPU memory allocation calls and copies creating a mapping between CPU and GPU variables. In the second phase, leveraging this information, it strategically replaces the allocation method sections, opting for UM or ZC allocation instead of the default GPU memory allocation method. *XeroZerox* also creates a pool of memory to allocate all the data at once at the beginning of the process and free it at the end, being a desirable behaviour for safety-critical applications, first for avoiding the timing overhead and non-time deterministic nature of the memory allocations which can impact the worst case execution time of the program, as well as to ensure that the size of the memory allocations is fixed, and therefore can always be satisfied at program deployment.

Another strategy is proposed by Sven Widmer et al. [23]. They developed an allocator focused on enhancing SIMD scalability for small, frequent memory allocations, minimising branch divergence. The system-wide default allocator performs well with few simultaneous requests, and this approach optimises data accesses by utilising one superblock shared among warp threads. A voting mechanism selects a worker thread, reducing simultaneous memory requests and invocations. This design eliminates the need for superblock header data, streamlining memory allocation and minimising synchronisation and memory overhead. Aggregating memory requests within a warp ensures efficient cache utilisation, aligning with the goal of minimising the use of the default allocator for improved performance.

Another approach is described by Andrew Adinetz [3]. HAlloc is a statically sized memory pool, which is subdivided into chunks during initialization, while the handling of large allocations relies on the CUDA dynamic memory allocator. For each allocation size, only one active bin from which to allocate is kept by HAlloc. When a configurable threshold for usage within a bin is reached, it is replaced with a new active bin, maximizing the chances of subsequent allocations finding an available block in the active bin. Lists of bins that are almost-exhausted and almost-empty are also kept by HAlloc for each size. Bins are moved between these two lists during free operations, and bins in the almost-empty list are used to select new active bins when needed. Per-size bins are also maintained by their fine-grained allocator, but a linked list is used to track all active bins, avoiding costly active bin replacement operations.

Zaid Qureshi et al. [19] develop the BaM System (Big accelerator Memory). This tool allows programmers to access big data sets which exceed the GPU memory available in the system, by accessing data stored in storage devices in an on-demand and fine-grained manner, while improving the access time. The authors call this “accelerator-centric” architecture. Threads on GPU can bring data wherever it is stored, either on CPU or any other storage system. This reduces the use of page-faulting mechanism from CPU and it is demonstrated using NVMe SSDs.

In terms of predictability, Björn Forsberg et al. [14] show how to enhance cache hit rate through the adept management of prefetching and evicting using Predictable Execution Models (PREM). To enhance predictability, they introduce a division into a memory phase and a compute phase, a strategy akin to that of XeroZerox. They leverage a "replacement policy" that "selects which data to evict when new data is requested".

2.2 Reverse engineering

On reverse engineering, Jake Choi et al. [13] performed a comparative analysis between UM and ZC in relation to the traditional *cudaMalloc* over a NVIDIA Jetson TX2 SoC, an embedded platform based on Pascal architecture. By evaluating those allocators against three benchmarks of the Rodinia suite, they extract and compare metrics such as memory usage and execution time. Their study concludes that the traditional method should not be the default choice for programmers. In fact, the optimal allocator depends on the specific application being considered.

Calderón et al. [6] created a tool for the reverse engineering of the default CUDA memory allocator in terms of functionality and timing behaviour. They showed that similar to CPU allocators, the CUDA allocator works with power of two bin sizes and that GPU memory allocations which fall into a newly allocated or reallocated bin affects the execution time of the subsequent GPU kernel call. Their open source tool is able to extract the bin sizes and memory pools of NVIDIA GPUs and has been demonstrated with both desktop and embedded GPUs. Moreover, later the tool was extended to OpenCL and non-NVIDIA GPUs [7].

3 Allocation Methods

Over time, there has been a concerted effort from both industry players and academic researchers to devise quicker algorithms, all with the common goal of enhancing data access. In the following we provide a succinct overview of the algorithms under evaluation. It's worth noting that these particular algorithms were selected for their seamless integration and straightforward implementation on our embedded GPU platform i.e. open source availability of their code and compatibility with embedded NVIDIA GPUs. Interestingly, the first three algorithms hail from the research and development efforts of NVIDIA, while the fourth stems from an independent researchers' work.

CUDA traditional allocation is the native allocator of CUDA C programming language [12]. It is used through *cudaMalloc*, which allocates device global memory of a specified size. The operating system looks for space in the memory pool using one of the following policies: first-fit, best-fit, worst-fit or next-fit. Still, data movements must be explicitly done using *cudaMemcpy*.

Zero-Copy. In order to transfer data from host (CPU) to device (GPU) or in the reverse direction, the memory has to be copied to a page-locked buffer. This process can be avoided by allocating paged-locked (aslo known as pinned memory) with *cudaMallocHost* or *cudaHostAlloc* calls, so data can be directly accessed by the device using DMA (Direct Memory Access). Enabled by Unified Virtual Addressing (UVA) released on CUDA 4, it makes data accessible through PCI-e avoiding *cudaMemcpy* calls.

Unified Memory. Supported since CUDA 6, this allocation method [15] joins both Central Processing Unit (CPU) and GPU memory address spaces in a single one. Using *cudaMallocManaged* function, data is allocated into that space, returning a pointer accessible both from host and device. Similar to ZC there is no need of explicit copy declaration because the system migrates data automatically.

ScatterAlloc. This algorithm [22] organises a memory pool into a structure called *Super Block*. This structure has a fixed size of memory that is also split into equal size pages. Its author also proposes a method to keep track of free memory in two levels of hierarchy. At higher level a *Page usage table* is used that stores which pages are in use and freed. At lower level, inside each page, there is another table which keeps track of chunks within a page. In order to make allocations faster, the algorithm changes the storing region when 90% of the pages are filled. In addition, the author introduces an approach to reduce simultaneous access from different threads to the same memory region.

4 Memory analysis and reconfiguration

To introduce a safety framework and enable consistent comparisons between allocation methods, this work uses the *XeroZerox* tool [8] developed by A. J. Calderón mentioned in Section 2, extended with support to traditional memory allocation to fit the needs of our analysis. This tool offers numerous benefits: it allows us to modify memory models without altering the original benchmark code. Additionally, it supports the use of a single, preallocated memory pool at the program start-up. This is a highly desirable feature for safety-critical systems, as it enables control over memory reservation and the ability to calculate the worst case execution time of this process. In the utilisation of the *XeroZerox* tool for assessing allocation methods, a critical observation emerges: the original tool initially overlooked the case of the traditional allocation model. This discrepancy poses a substantial challenge to achieving a comprehensive comparison among allocation strategies. In this work we address this issue in order to perform a fair comparison. Next, we delineate our methodology for utilising the *XeroZerox* tool and address the gap it initially presents concerning the traditional allocation model. To enhance a robust comparison between allocation methods, we have devised an approach to incorporate this gap within our analysis framework.

XeroZerox analyses GPU applications and determines the size of a centralised memory pool that can serve all its needs. This pool is allocated at the beginning of the application using zero-copy or unified memory allocation and it is released upon its completion. Serving as a sub-allocator, *XeroZerox* intercepts traditional memory allocations and substitutes them with allocations served consecutively – i.e. similar to a bump allocator – from the centralised memory pool, with minimal and constant runtime cost. This approach accommodates legacy GPU applications within critical setups' memory management constraints, without any code modifications. Moreover, *XeroZerox* priorities minimising memory consumption, effectively reducing both the memory footprint and the runtime overhead associated with memory management for these applications.

To intercept the target memory functions, the analysis library employs a technique referred to in the literature as *interposition* [9]. This method involves substituting the target functions with user-defined wrapper functions. These wrappers serve to augment the original functions with additional functionality, such as extracting information from their arguments, which is particularly pertinent to our objectives. The analysis library is executed only the first time of executing the application and generates a comprehensive report including details like the maximum memory utilisation, the count of memory pool instances generated, the

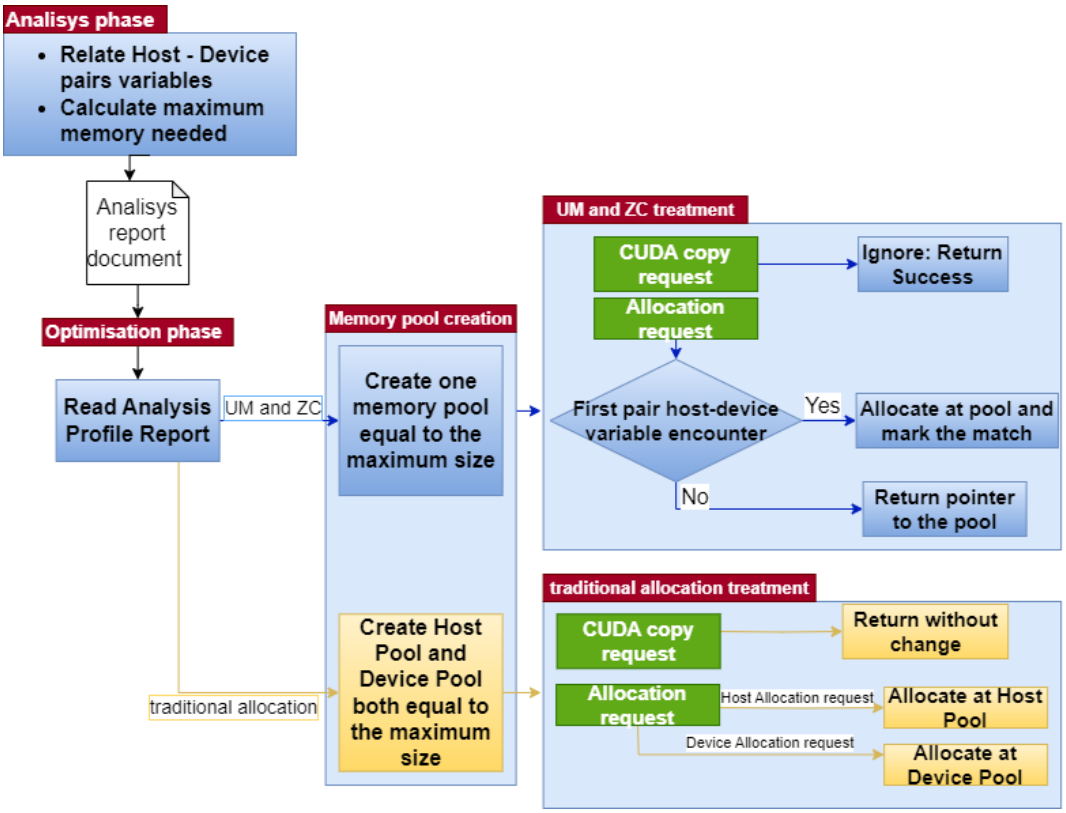


Figure 1 XeroZerox tool behaviour.

frequency of memory transfers and the detection of any memory leaks. On a second phase, which is the only one used for subsequent executions, eg. at deployment, once the centralised memory pool is established in the initialisatiton of the GPU application, *XeroZerox* assumes the role of a sub-allocator, handling allocation requests from the application in accordance with the matches loaded from the optimisation profile. Notably, *XeroZerox* adopts a strategy where it fulfils memory allocation requests solely for the initial request it receives. Subsequent allocation requests prompt *XeroZerox* to return a pointer to the memory region already allocated for the preceding request, effectively optimising memory utilisation by reusing allocated space. This approach minimises redundant allocations and contributes to more efficient memory management within the application.

In this paper, we extended *XeroZerox* with support for the traditional allocation method. We took advantage from the analysis phase, to identify the CUDA calls. At this point, instead of creating just one memory pool to be allocated using ZC (Zero Copy) or UM (Unified Memory) method at the beginning of the optimisation phase, host and device pools are created separately as it is performed when the default CUDA allocation method is used. During an allocation call, the tool discerns whether it is a GPU or CPU allocation, storing the data in the corresponding pool through a linked list. Additionally, the incorporation of support for copy calls has been essential due to the existence of two distinct allocated pools. This ensures that whenever such a call is activated, the tool can reference the variables to the pre-allocated pools, facilitating the migration of data.

As illustrated in Figure 1, the dark blue boxes denote components of the original *XeroZerox* tool, representing the baseline framework. Whereas, the yellow boxes highlight additional elements we added to ensure a fair comparison among different allocation methods.

5 Benchmark's characterisation

The Rodinia benchmark suite is a collection of parallel applications designed to evaluate the performance and scalability of computer systems, particularly those with multicore processors or GPU accelerators. It was developed by researchers at the University of Virginia as a resource for evaluating and comparing different parallel computing architectures.

This benchmarking suite is one of the most widely used GPU bencmarking suites used in the literature. Since our purpose is to analyse the general memory allocation behaviour of GPU software and find out the optimal memory allocation method for use in safety critical systems, we intentionally avoid using a GPU benchmarking suite targeting explicitly safety critical systems like GPU4S Bench and OBPMark, wich have a predictable, easy to analyse memory allocation behaviour, similar to the one achieved by *XeroZerox*.

Rodinia contains 23 benchmarks, characterised by their computation patterns, named *Dwarfs* by Paul Springer [21] and introduced by Asanovic et al. [4]. They refer to recurring structures or strategies used to solve problems and perform tasks in computational systems. These patterns provide standardized ways to approach common computational problems, making it easier to design, implement, and understand. Despite their utility in classification, our analysis indicates that these patterns do not have any relevance on our results. Furthermore, during the execution of 13 out of 23 Rodinia benchmarks, runtime errors or system crashes were encountered, causing the hardware to reboot. To maintain the integrity of the results, we opted not to modify the benchmark code to force compatibility with the ARM-based embedded system used. Those evaluated are presented in Table 1 along with the name of the benchmark and the data loaded is provided by Rodinia's developers.

Table 1 Evaluated Rodinia benchmarks.

Applications	Dwarfs	Datasets
Back Propagation	Unstructured Grid (UG)	65536 elements
Gaussian Elimination	Dense Linear Algebra (DLA)	3x3 matrix and 1024x1024 matrix
LU Decomposition	Dense Linear Algebra (DLA)	64x64 matrix and 2048x2048 matrix
Kmeans	Dense Linear Algebra (DLA)	100 and 819200 elements x 34 columns
Breadth-First Search	Graph Traversal (GT)	1 million nodes and 4096 nodes
SRAD_v1	Structured Grid (SG)	512x512 image
Hotspot	Structured Grid (SG)	Two square matrices of 64x64, 512x512 and 1024X1024 each
Heart Wall	Structured Grid (SG)	104 frames: 609x590 pixels
Leukocyte	Structured Grid (SG)	600 frames: 640x480 pixels
Myocyte	Structured Grid (SG)	16 parameters 1 instance 100 ms

Taking into consideration prior research, we have curated a set of characteristics that define a program for our experimental setup, categorized into static and dynamic metrics. Static metrics provide intrinsic insights into program structure, encompassing cache hit rates, shared memory usage, coalescence, memory access frequency, and data transfer sizes between CPU and GPU. On the other hand, dynamic metrics focus on timing operations, including overall program execution time, kernel execution time, CUDA APIs execution time, and memory operations execution time. This comprehensive approach enables a thorough evaluation of program performance across various dimensions, facilitating informed comparisons between different allocation methods and program configurations. Crossing results from both categories is the key to obtain conclusions of how program characteristics influence its timing results. The following subsections give a deeper understanding of these categories.

5.1 Static metrics

These metrics provide intrinsic insights into program structure, highlighting how a program is coded and its inherent characteristics. These metrics are crucial for understanding the efficiency and resource usage of a program. The following points explain each static metric in detail:

1. **Cache Hit Rates:** This metric assesses the efficiency of data retrieval from cache memory. A high cache hit rate indicates that most data requests are satisfied by the cache, leading to faster data access and improved performance. Analysing cache hit rates helps in optimizing memory hierarchy and reducing latency.
2. **Shared Memory Usage:** This metric evaluates the utilization of shared memory resources within the system. Efficient use of shared memory can reduce global memory accesses and increase the speed of data processing. Understanding shared memory usage is key to optimizing memory allocation and parallel processing capabilities.
3. **Coalescence:** This metric examines the degree to which memory accesses are coalesced, which optimizes data transfer efficiency by grouping multiple memory requests into a single transaction. High coalescence reduces the number of memory transactions, improving bandwidth utilization and reducing latency.
4. **Memory Access Frequency:** This metric quantifies the frequency of memory accesses, indicating the demand for data retrieval during program execution. High memory access frequency can highlight potential bottlenecks and guide optimizations to minimize redundant memory operations and enhance overall performance.
5. **Data Transfer Sizes Between CPU and GPU:** This metric measures the volume of data exchanged between the central processing unit (CPU) and the graphics processing unit (GPU). Large data transfers can introduce significant overhead and latency. Understanding and optimizing data transfer sizes are crucial for improving inter-device communication and overall program efficiency.

5.2 Dynamic metrics

On the other hand, dynamic metrics focus on timing operations, providing insights into various aspects of program execution. The following points explain each dynamic metric in detail:

1. **Overall Program Execution Time**: This metric captures the total duration from the start to the end of the program's execution.
2. **Kernel Execution Time:** This metric measures the time taken to execute computational kernels, which represent the core processing tasks of the program. Analysing kernel execution time helps in understanding the efficiency of the computational workload and identifying areas for optimisation within the kernels.
3. **CUDA APIs Execution Time:** This metric examines the time spent executing CUDA Application Programming Interface (API) calls. These calls manage GPU resources and operations, so their execution time reflects the overhead associated with GPU management. Specifically, this includes the time taken for kernel launches, memory allocations, data transfers, and, in some cases, synchronization operations.
4. **Memory Operations Execution Time:** This metric measures the duration required for memory read and write operations. It is indicative of data transfer efficiency and memory access latency. Optimizing memory operations execution time can significantly enhance overall program performance, particularly in data-intensive applications.

6 Experimental Results

In this section, we present the results of testing various allocators on the Rodinia benchmark suite using their standard input set. Subsequently, in the following sections of this section, graphical representations of the data are provided to facilitate a time comparison between the selected allocators for each metric across all benchmarks. This analysis examines the performance characteristics, strengths, and weaknesses of each allocator in relation to the benchmarks' nature.

Due to the extensive volume of collected data, we have opted to store it in a dedicated GitHub repository [2]. Detailed information from each of the 500 iterations, as well as summarised characteristics in Excel files, can be accessed through this repository. This approach allows for transparency and facilitates access to the comprehensive dataset for those interested in further analysis or replication of the study.

6.1 Experimental setup

For each allocation method outlined in section 3, we performed multiple iterations of each benchmark on an NVIDIA Jetson Orin AGX, an embedded GPU platform certified for use in the automotive sector. Due to the inherent variability in GPU execution times, as discussed in section 5, each benchmark was executed 500 times with the same data inputs. This number of runs allows for a reliable calculation of the mean and standard deviation for each selected parameter, ensuring statistically meaningful results. The variability in execution times was verified through time histograms, which support the assumption of a Gaussian distribution. These histograms are available for review on our GitHub repository [2]. In contrast, obtaining static metrics was more straightforward, as these remain consistent regardless of when the data collection tool was launched or the memory configuration selected.

Only memory access methods interacting with *XeroZerox* have been studied, using the unmodified benchmarks as baseline. This approach was chosen due to a key feature of *XeroZerox* that aligns it with safety-critical systems: memory allocation is managed by ensuring that all allocations occur at the start of the execution.

Using the *NVIDIA NSight Systems* and *NVIDIA NSight Compute* tools alongside each benchmark, we generated a timing report containing dynamic metrics and a characterization report featuring static metrics. The benchmarks from Rodinia were compiled using CUDA version 11.4 and automated with Python scripts version 3.8.10. Metric extraction was performed using *NVIDIA NSight Systems* version 2022.4.2.1 and *NVIDIA NSight Compute* version 2023.2.0.0 build 32895467.

6.2 Profiling

The initial set of static characteristics was extracted from the profiling tool *NVIDIA NSight Systems*, and the results are presented in Table 2. This table displays basic static characteristics, such as the number of allocations and deallocations performed using *cudaMalloc* and *cudaFree*, the number of copies made by *cudaMemcpy*, the quantity of kernels launched, and the size of the data moved. The data related to other native allocation instructions, like *cudaBindTexture* or *cudaMemcpyToSymbol*, has been retained in its original form. This decision is grounded in the belief that these instructions offer essential functionality required by programmers. To aid interpretation, certain cells in the table have been colour-coded and star-marked: blue cells (*) are the value believed to be the edge on the allocator choice, while a green background (**) represent values that exceeded the threshold established for influencing allocator selection, this is founded on the conclusions presented in the following subsection.

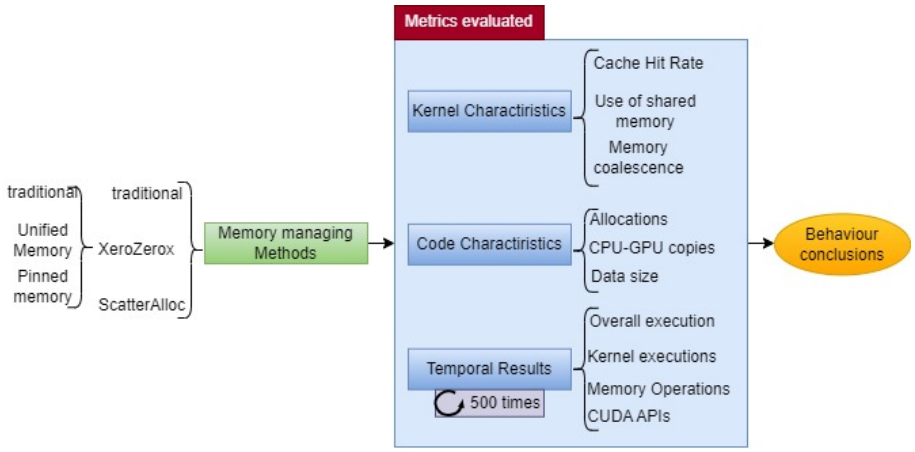


Figure 2 Metrics extracted for every evaluated benchmark.

During our work, the necessity of acquiring a deeper understanding of the behaviour of each benchmark was recognised. For this reason, we have made use of *NVIDIA NSight Compute* to acquire a second set of static metrics related to kernels. To make sure that the allocation method does not interfere with these results by comparing the kernels' characteristics, binaries run from a clean build and from binaries interfered with *XeroZerox*, which are shown in Table 3. In three benchmarks (*gaussian* with 1024 size matrix, *srad_v1* and *myocyte*) the NVIDIA's tool could not complete the analysis. For these two cases the report was empty despite how many times we run the test. We suspect that this might be due to the high number of kernels launched, due to this limitation we could not provide information about these tests.

We observed an anomalous L2 cache hit rate for kernel 2 in the *leukocyte* benchmark, consistently reported by the *NVIDIA NSight Compute* tool, despite multiple reruns of the benchmark. Since we lack access to the internal workings of this tool, we are unable to provide a definitive explanation for this irregularity.

At the core of our analysis lies a meticulous process of data collection of the dynamic metrics during each iteration of the benchmarking procedure. This data, intricately tied to the associated process and variable, serves as the foundation for subsequent calculations of mean and standard deviation. An example of results from those executions is illustrated in Figure 3 for API's executions and in Figure 4 for memory and kernel operations.

Figure 3 represents a benchmark with a specific input set size. Vertical axis (y) represents the normalised time consumed on running every API taking as baseline benchmarks without being modified with a value of 1.00, called *traditional alloc* at the picture. The x axis represents each allocation method evaluated, from left to right: traditional alloc, traditional *cudaMalloc* with XeroZerox optimisation, *ScatterAlloc*, *ZC* with XeroZerox optimisation and *UM* with XeroZerox optimisation. The z axis represents every API call considered relevant, from front to rear: Kernel launch time, allocation API (every method has its own), copy API, synchronisation API (not every benchmark uses this instruction) and overall execution time.

On the other hand, Figure 4 illustrates kernel and memory operations execution times. Similar to the representation used in the APIs figures, these figure differ in that one of the horizontal axes has been adjusted to show which operations are being evaluated. Each kernel call and the normalised time it takes to transfer data from the CPU to the GPU and vice versa are depicted, using the traditional allocation method as a baseline with a value of 1.00. For UM and ZC, the transfer time is not included due to the unique behaviour of these methods, as explained in Section 3.

Table 2 Benchmark Characteristics.

Dwarf	Benchmark	Data	Allocations	Copies	Kernels launched	Data sizes
DLA	Gaussian	Matrix 3x3	3	6	4	$\leq 64kB$
		Matrix 1024x1024	3	6	2046 **	4 MB, $\leq 64kB$ *
	LUD	Matrix 64x64	1	2	10	$\leq 64kB$
		Matrix 2048x2048	1	2	382 **	16,777 MB **
	Kmeans	List 100x34 elements	4	7	3	$\leq 64kB$
		List 819200x34 elements	4	7	3	11.4 MB, 3,277 MB **
SG	srad_v1	512x512 image	12	206 **	502 **	0.920 MB , $\leq 64kB$
	hotspot	Two squared matrix 64x64	3	3	1	$\leq 64kB$
		Two squared matrix 512x512	3	3	1	1.049 MB
		Two squared matrix 1024x1024	3	3	1	4.149 MB *
	heartwall	104 frames: 609x590 pixels	623	50	20	1.952 MB , $\leq 64kB$
	leukocyte	600 frames: 640x480 pixels	34	39	7	0.561 MB , 0.472 MB , $\leq 64kB$
	myocyte	16 parameters 1 instance 100 ms	4	16500 **	3900 **	$\leq 64kB$
UG	backpropagation	65536 elements	6	8	2	4.457 MB , 0.262 MB , $\leq 64kB$ *
GT	BFS	4096 nodes	7	23	16	98 kB , $\leq 64kB$
		1 million nodes	7	31	24	24 MB , 8 MB , 4 MB , 1 MB , $\leq 64kB$ **

Complete information is contained in a Github repository [2], where Excel documents contain detailed information of the mean and standard deviation arranged in folders ordered by DWARFS alongside with 3D chart representations to condense data.

6.3 Comparative Analysis

In this subsection we discuss the results extracted from the correlation between static and temporal analysis, taking two distinct scenarios. The first revolves around an analysis just concerning the overall execution time. In contrast, the second is oriented to those systems characterised by continuous operation, like safety-critical systems responsible for monitoring the environment from startup to the moment the machine is shutdown. These systems typically entail a single allocation at initialisation, followed by recurrent data movement and kernel launches throughout operation, culminating in memory deallocation upon shutdown.

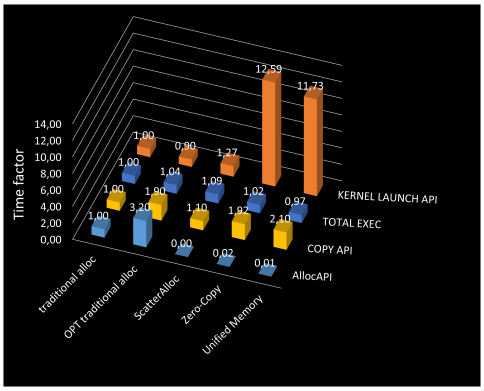


Figure 3 APIs-hotspot 64x64 matrix.

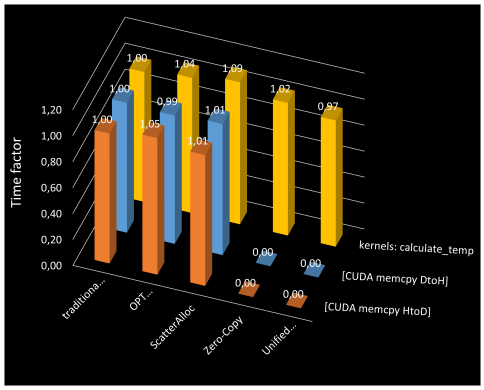


Figure 4 Mem/Kernel Ops-hotspot 64x64.

Table 3 Kernels Characteristics.

Dwarf	Benchmark	Data	L1 cache hit rate	L2 cache hit rate	Coalescent memory	Excesive sectors accessed
DLA	Gaussian	Matrix 3x3	kernel 1: 33,33% kernel 2: 60%	kernel 1: 56,77% kernel 2: 66,47%	Uncoalesced Global Accesses	kernel 1: no kernel 2: 2 (20%)
		Matrix 1024x1024	no data	no data	no data	no data
	LUD	Matrix 64x64	kernel 1: 48,39% kernel 2: 39,24% kernel 3: 25%	kernel 1: 40,51% kernel 2: 42,87% kernel 3: 61,14%	Uncoalesced Shared Accesses	kernel 1: 562 (41%) kernel 2: 6216 (70%) kernel 3: no
		Matrix 2048x2048	kernel 1: 48,39% kernel 2: 58,22% kernel 3: 53,29%	kernel 1: 40,51% kernel 2: 49,38% kernel 3: 64,42%	Uncoalesced Shared Accesses	kernel 1: 562 (41%) kernel 2: 263144 (70%) kernel 3: no
	Kmeans	List 100x34 elements	kernel 1: 82,08% kernel 2: 86,49%	kernel 1: 54,79% kernel 2: 27,73%	Uncoalesced Global Accesses	kernel 1: 3009 (70%) kernel 2: no
		List 819200x34 elements	kernel 1: 52,78% kernel 2: 41,07%	kernel 1: 71,98% kernel 2: 67,62%	Uncoalesced Global Accesses	kernel 1: 24371200 (78%) kernel 2: no
SG	srad_v1	512x512 image	no data	no data	no data	no data
	hotspot	Two squared matrix 64x64	16,13%	62,54%	Uncoalesced Global Accesses	864 (27%)
		Two squared matrix 512x512	4,65%	68,69%	Uncoalesced Global Accesses	67192 (31%)
		Two squared matrix 1024x1024	3,54%	68,67%	Uncoalesced Global Accesses	273184 (31%)
	heartwall	104 frames: 609x590 pixels	95,48%	97,14%	Uncoalesced Global Accesses	9958 (19%)
	leukocyte	600 frames: 640x480 pixels	kernel 1: 99,23% kernel 2: 98,78% kernel 3: ≈ 98%	kernel 1: 22,51% kernel 2: 172,91% kernel 3: ≈ 80%	Uncoalesced Global Accesses	kernel 1: 3051 (19%) kernel 2: 122584 (88%) kernel 3: ≈ 580000 (18%)
	myocyte	16 parameters 1 instance 100 ms	no data	no data	no data	no data
UG	backprop	65536 elements	kernel 1: 58,30% kernel 2: 71,18%	kernel 1: 52,52% kernel 2: 54,17%	Uncoalesced Global Accesses in both kernels	kernel 1: 73728 (18%) kernel 2: 163847 (15%)
GT	BFS	4096 nodes	kernel 1: 19,40% kernel 2: 7,19%	kernel 1: 50,19% kernel 2: 49,48%	No coalescence warning	
		1 million nodes	kernel 1: 0,04% kernel 2: 0,01%	kernel 1: 0,91% kernel 2: 0,93%	No coalescence warning	

6.3.1 Analysis regarding overall execution time

By correlating static results (explained in section 5) and dynamic results from the time analysis, we derived programming guidelines for GPU usage, presenting distinct conclusions based on overall execution time and considerations more pertinent to safety-critical systems. Generally speaking, the traditional method with the *XeroZerox* optimisation out stands over the other allocation methods regarding the total execution time of the benchmark. Additionally, there are other characteristics that had been seen relevant to choose a method of memory management. It appears that the most relevant one is the cache hit rate. This can be observed in the cases of both the *heartwall* and *leukocyte* benchmarks, where the hit rates for both L1 and L2 caches exceed 90%. In such cases, the traditional allocation method without optimisation yields the lowest overall execution times.

In general, when a substantial number of kernels is launched, and some data involved in copies that exceed 4 MB, the zero-copy method with *XeroZerox* proves to be the optimal allocation method, resulting in the lowest overall execution times. Conversely, when the number of copies and kernel launches is small, and the data copied does not exceed 4 MB, both zero-copy and unified memory yield similar results, with unified memory demonstrating the lowest execution times for sizes below that threshold.

There are, however, exceptions. For instance, the *BFS* benchmark should have a clear allocator preference. When using 4096 nodes, unified memory is preferred, while when loading 1 million nodes, zero-copy performs better. Remarkably, the execution times are

very similar for both allocators, with the size having only a minor impact. This benchmark is unique in that it exhibits coalescent memory, thereby minimising excessive sector accesses, which could explain the observed behaviour.

Given these observations, benchmarks such as *gaussian* (loading a 3x3 matrix), *LUD* (loading a 64x64 matrix), *kmeans* (loading 100x34 elements), and *hotspot* (loading a 64x64 matrix) were expected to behave similarly. However, temporal results reveal that *hotspot* and *LUD* perform better with unified memory, while *kmeans* and *gaussian* yield better overall execution times with zero-copy. It was found that the first couple make use of shared memory and experience a higher number of warps stalled compared to the latter two benchmarks.

Below, observations are presented concerning overall execution time, categorized by their respective levels of importance, with the highest level of importance listed first. Each observation is paired with a recommended allocator:

1. L1 and L2 cache hit exceed 90% -> traditional allocation method
2. Coalescent memory -> zero-copy + XeroZerox or Unified Memory + XeroZerox
3. When a substantial number of kernels (measured at 2 in blue or green) are launched & data size greater or equal than 4 MB -> zero-copy + XeroZerox
4. When number of kernels launched is small & data size less or equal than 4 MB -> Unified Memory + XeroZerox
5. Use of shared memory -> Unified Memory + XeroZerox
6. High number of warps stalled -> zero-copy + XeroZerox

6.3.2 Analysis regarding kernels and copies execution times

Another interpretation of the results can be made if the programmer is looking to adapt the code into safety-critical systems. One important recommendation is to perform memory allocations only once at the beginning, a behaviour achieved automatically with the *XeroZerox* optimisation. Moreover, memory freeing is not a relevant characteristic in this case, because the expected execution is that the system runs continuously, reading and writing data, moving it between CPU and GPU and executing functions and kernels. So considered metrics here are the time that the copy and kernel launch APIs are active, and the kernel and copy execution times.

Upon examining the active time of the copy API and the kernel launch API, it's observed that **conventional method without XeroZerox intervention and ScatterAlloc** typically emerge as the faster options. However, there are instances where other methods yield similar times. The observed behaviour aligns closely with the previous findings, with one notable exception found in the *backpropagation* benchmark. Here, both ZC and UM methods show comparable performance to the traditional method.

When looking at copying operations, the first thing noticed is that there is no data when using ZC or UM, presumably because there is no explicit copy between CPU and GPU, so the delay depends on the related API. On the other cases, the time grows proportionally with the size copied being in any case in the same magnitude order, no special correlation is found here.

To verify the accuracy of the mean and standard deviation as appropriate measures, histograms were generated for each process, also available on the mentioned GitHub repository [2]. Notably, the traditional allocation method exhibited considerable variation in results, suggesting a lack of consistency. As a result, **it is advisable to avoid the traditional allocation method** in systems targeting safety-critical or real-time scheduling, regardless of the mean results. Also when employing the traditional allocation method in any form, the

histograms for copying and allocating APIs exhibited a multi-modal distribution, while ZC and UM usually exhibit one gaussian bell shape, so in terms of predictability it is advised to use these last two methods in case that *ScatterAlloc* is not the preferable method.

7 Conclusion

In this work, we analysed the dynamic memory behaviour of GPU programs for safety critical systems, targeting the widely used suite GPU benchmark suite, Rodinia. Studying all this data has revealed some reasons behind the allocator's behaviour have been identified through a static analysis of the benchmarks.

The specific characteristics and requirements of each benchmark and kernel influence the choice of GPU memory allocation method. Factors such as cache hit rates, data sizes, and the number of kernel launches may play a crucial role in determining which allocation method is the most suitable for a given scenario.

For future research and validation of the conclusions presented in this work, the following avenues can be explored:

1. Microbenchmarks for Specific Scenarios: Conducting microbenchmarks designed to test each specific scenario and characteristic identified in this research can provide a more detailed and comprehensive validation of the conclusions. This can help in fine-tuning allocation methods for precise use cases.
2. GPU Direct RDMA [16] and GPUDirect Storage [17]: Investigating the use of NVIDIA's GPUDirect RDMA and GPUDirect Storage methods represents a promising direction. These technologies facilitate direct GPU access to data stored in storage units such as SSDs, circumventing CPU involvement. This approach holds the potential to deliver substantial performance benefits and minimize latency in data-intensive applications. The work of J. Bakita et al.[5] is directly relevant to this topic and can be utilised to propel advancements in this area.
3. Evaluation on other platforms: Consider how these allocation methods perform on different GPU architectures and platforms, as compatibility and performance can vary.

References

1 Rodinia: Accelerating compute-intensive applications with accelerators, 2018. URL: `https://rodinia.cs.virginia.edu/doku.php`.

2 Mars-data, 2024. URL: `https://anonymous.4open.science/r/MARS-data-568D/README.md`.

3 Andrew V Adinetz. Halloc: a high-throughput dynamic memory allocator for gpgpu architectures, 2014.

4 Krste Asanovic. The landscape of parallel computing research: A view from berkeley. Report, Electrical Engineering and Computer Sciences University of California at Berkeley, 2006.

5 Joshua Bakita. Enabling GPU memory oversubscription via transparent paging to an NVMe SSD*. *Real-Time Systems Symposium*, 2022.

6 Alejandro J. Calderón, Leonidas Kosmidis, Carlos F. Nicolás, Francisco J. Cazorla, and Peio Onaindia. Understanding and exploiting the internals of GPU resource allocation for critical systems. In David Z. Pan, editor, *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, pages 1–8. ACM, 2019. `doi:10.1109/ICCAD45719.2019.8942170`.

7 Alejandro J. Calderón, Leonidas Kosmidis, Carlos F. Nicolás, Francisco J. Cazorla, and Peio Onaindia. GMAI: understanding and exploiting the internals of GPU resource allocation in critical systems. *ACM Trans. Embed. Comput. Syst.*, 19(5):34:1–34:23, 2020. `doi:10.1145/3391896`.

8 Alejandro J. Calderón. *Real-Time High-Performance Computing for Embedded Control Systems*. Thesis, Universitat Politècnica de Catalunya, 2022.

9 A. Chatterjee. Function interposition in c with an example of user defined malloc, 2017. URL: `https://www.geeksforgeeks.org/function-interposition-in-c-with-an-example-of-user-defined-malloc`.

10 Shuai Che. Rodinia: A benchmark suite for heterogeneous computing, 2009. `doi:10.1109/IISWC.2009.5306797`.

11 Shuai Che. A characterization of the rodinia benchmark suite with comparison to contemporary CMP workloads, 2010. `doi:10.1109/IISWC.2010.5650274`.

12 John Cheng. *Professional CUDA C Programming*. John Wiley & Sons, Inc., 2014.

13 Jake Choi. Comparing unified, pinned, and host/device memory allocations for memory-intensive workloads on Tegra SoC. *Concurrency and Computation: Practice and Experience*, 2020. `doi:10.1002/cpe.6018`.

14 Björn Forsberg, Luca Benini, and Andrea Marongiu. *Taming Data Caches for Predictable Execution on GPU-based SoCs*. IEEE, 2019. `doi:10.23919/DATE.2019.8715255`.

15 NVIDIA. Unified memory in cuda for beginners, 2017. URL: `https://developer.nvidia.com/blog/unified-memory-cuda-beginners/`.

16 NVIDIA. Gpudirect rdma, 2020. URL: `https://docs.nvidia.com/cuda/gpudirect-rdma/index.html`.

17 NVIDIA. Gpudirect storage, 2020. URL: `https://developer.nvidia.com/gpudirect-storage`.

18 System Optimization and Riverside Computer Architecture Laboratory at the University of California. Complete rodinia benchmark suite v3.1, 2017. URL: `https://github.com/socal-ucr/Rodinia/tree/3.1`.

19 Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Brian Park, Jinjun Xiong, Chris J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William J. Dally, and Wen-mei W. Hwu. Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 325–339. ACM, 2023. `doi:10.1145/3575693.3575748`.

20 Marcos Rodriguez. marcosrc92/MARS-data. Dataset, swhId: `swh:1:dir:0b2fbba6fbdfa60cb3d84175a2dd102bfb293ff2` (visited on 2025-01-13). URL: `https://github.com/marcosrc92/MARS-data`, `doi:10.4230/artifacts.22756`.

21 Paul L. Springer. Berkeley's dwarfs on cuda, 2012. URL: `https://api.semanticscholar.org/CorpusID:44643311`.

22 Markus Steinberger. Scatteralloc: Massively parallel dynamic memory allocation for the gpu, 2012.

23 Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele. Fast dynamic memory allocator for massively parallel architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, GPGPU-6, Houston, Texas, USA, March 16, 2013*, GPGPU-6, pages 120–126, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2458523.2458535`.

Custom Floating-Point Computations for the Optimization of ODE Solvers on FPGA

Serena Curzel
Politecnico di Milano, Italy

Marco Gribaudo
Politecnico di Milano, Italy

Abstract

Mean Field Analysis and Markovian Agents are powerful techniques for modeling complex systems of distributed interacting objects, for which efficient analytical and numerical solution algorithms can be implemented through linear systems of ordinary differential equations (ODEs). Solving such ODE systems on Field Programmable Gate Arrays (FPGAs) is a promising alternative to traditional CPU- and GPU-based approaches, especially in terms of energy consumption; however, the floating-point computations required are generally thought to be slow and inefficient when implemented on FPGA. In this paper, we demonstrate the use of High-Level Synthesis with automated customization of low-precision floating-point calculations, obtaining hardware accelerators for ODE solvers with improved quality of results and minimal output error. The proposed methodology does not require any manual rewriting of the solver code, but it remains prohibitively slow to evaluate any possible floating-point configuration through logic synthesis; in the future, we will thus implement automated design space exploration methods able to suggest promising configurations under user-defined accuracy and performance constraints.

2012 ACM Subject Classification Hardware → Methodologies for EDA; Hardware → High-level and register-transfer level synthesis; Computer systems organization → Architectures; Hardware → Very large scale integration design; Hardware → Reconfigurable logic and FPGAs

Keywords and phrases Differential Equations, High-Level Synthesis, FPGA, floating-point

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2025.2

Supplementary Material *Software*: `https://github.com/ferrandi/PandA-bambu`

Funding Funded by the European Union – NextGenerationEU – PNRR – M4 – C2 – I1.3 – SERICS PE00000014 – Cascade Funding SPOKE 8 – (MAM-CYD) – CUP J33C22002810001.

1 Introduction

Analytical modeling techniques such as Mean Field Analysis (MFA) and Markovian Agents (MA) can be applied to predict and optimize the performance of systems composed of many interacting objects, including e.g., cyber-physical systems. MFA [29] describes the transient evolution and the stationary behavior of such systems dividing their constituent objects into classes, each one describing a specific behavior [6, 12]. MA extends MFA by allowing objects, also called agents, to be distributed in a space that can be either continuous or discrete [22]; each agent has its own local behavior, which is influenced by mutual interactions with other agents. MA provides a powerful and scalable technique for modeling complex systems of distributed objects, and as such it has been applied e.g., to study sensor networks [9], Covid-19 diffusion [23], and forest fire monitoring [10].

Both MFA and MA models are analyzed using linear systems of ordinary differential equations (ODEs). One (large) vector is used to count the number of objects in each state for each class in MFA models, and MAs extend this representation by repeating these components for each considered spatial location. A kernel function defines how each element in the

state vector will evolve in time according to the definitions of the individual components, and the transient evolution of the system is computed by integrating this kernel. This representation can easily be parallelized, which motivated us to explore hardware acceleration on Field Programmable Gate Array (FPGA) or Application-Specific Integrated Circuits (ASICs) as a faster and more energy-efficient solution than software execution on general-purpose processors. ASICs are the best solution in terms of performance, but they incur higher development costs; FPGAs are more accessible and can be quickly reconfigured, allowing to update accelerators according to the requirements of new applications or to try multiple configurations in a prototyping phase before committing to long and expensive ASIC manufacturing.

The design flow we envisioned for the implementation of hardware accelerators that will solve the MFA/MA ODE systems is based on High-Level Synthesis (HLS), which allows a faster and more reliable process than manual hardware design. HLS tools, in fact, automatically generate hardware accelerators starting from a software description (e.g., written in C/C++), greatly increasing developers' productivity and allowing application experts to obtain efficient designs without being experts in low-level circuit design [11].

A typical optimization opportunity available in HLS tools is the usage of custom data types instead of the standard IEEE floating-point types used in software. In fact, FPGA implementations of floating-point functional units are usually slower than the specialized floating-point units present in modern CPUs, and they require a considerable amount of resources. If the computational precision of the application allows it, fixed-point calculations are to be preferred as they can be implemented through simpler logic. While a few previous attempts at implementing fixed-point ODE solvers exist, in this paper we focus on the exploration of custom floating-point types, i.e., types with a non-standard number of bits for mantissa and exponent, through the TrueFloat framework [19] integrated into the Bambu HLS tool [17]. The main strength of TrueFloat compared to existing libraries of floating-point components is its integration within the HLS flow, allowing deeper optimization of the functional units during the process of generating the accelerator datapath. TrueFloat types allow us to improve the quality of results (QoR) of the generated accelerators while maintaining the desired accuracy, and we demonstrate it by testing the application of different TrueFloat configurations on a proxy ODE solver representative of the type of computation performed in MFA/MA models.

TrueFloat provides the required support to implement approximate applications in HLS, but it fully relies on the user to specify the desired floating-point configuration. A key step in the design process is thus to determine the minimal precision required to maintain an acceptable output error. We argue, however, that the configuration space is too wide to be explored exhaustively and that previous research in design space exploration (DSE) tools for approximate computing are too specific to software, so further research will be required to equip Bambu and TrueFloat with a useful DSE tool able to suggest good floating-point configurations without long logic synthesis and implementation runs.

In summary, this paper makes the following contributions:

- We present a practical application of TrueFloat custom floating-point computations in the synthesis of a non-trivial program;
- We highlight the need for automated methods for the search of good floating-point configurations in a wide design space;
- We demonstrate the QoR improvement provided by custom low-precision types for an ODE solver accelerated on FPGA.

The rest of the paper is structured as follows: Section 2 summarizes related work on the acceleration of ODE solvers on FPGA and on custom-precision floating-point formats, Section 3 describes our HLS-based methodology and the exploration of different formats targeting improved QoR and minimal effect on accuracy, Section 4 presents the results we obtained after FPGA synthesis, and Section 5 concludes the paper with final remarks and future research directions.

2 Related work

2.1 Solving ODEs on FPGA

ODE systems are a fundamental component of many scientific applications in high-performance computing, and a significant amount of research focuses on improving the performance of the numerical methods used to solve them. FPGA-based solutions have been explored because of their potential to provide high throughput and low energy consumption, making them an attractive alternative to GPUs and multi-core CPUs despite the steep learning curve of low-level hardware design.

The Differential Equation Processing Element (DEPE) co-processor [27, 28, 26] has been proposed as an alternative to customizing an FPGA accelerator for the solution of a specific ODE system. The co-processor was designed as a no-instruction-set computer together with its compiler and it solves ODEs with a Runge-Kutta fourth-order method implemented through fixed-point computations; the evaluation focuses on its low area consumption compared to HLS-generated designs. Another co-processor based on the RISC-V architecture was proposed targeting a specific class of ODEs [25], implementing Euler and Runge-Kutta methods through single-precision floating-point computations and obtaining faster execution time than a single-core CPU. Other research works present custom solver units where both the solver method and the ODEs are hard-coded into the accelerator [4, 5].

HLS tools raise the level of abstraction required to design FPGA accelerators [11], and they have been applied to implement accelerators for various ODE solvers [35]. Another possibility to simplify the design flow of ODE accelerators is to rely on domain-specific languages and map their primitives to optimized register-transfer level (RTL) primitives [3] or to exploit commercial tools provided within Matlab/Simulink to generate RTL components from model-based representations [1].

New computational models have also been proposed in place of conventional numerical solvers, as they might provide an advantage when implemented in custom hardware. For example, since analog components can solve ODEs faster than numerical methods, a possibility is to simulate analog components in FPGA logic through digital differential analyzers [15, 21]. The Euler solver can also be approximated through the use of stochastic integrators, which can be efficiently implemented as RTL components [30].

2.2 Approximate computing

Research in the field of approximate computing led to a variety of techniques that aim at improving the energy efficiency of an application through reduced precision, and tools that analyze the application to understand the impact of approximation on the quality of its results [31, 13].

The problem of finding the best approximation scheme has been addressed through both analytical models and learning-based approaches, mostly focusing on the few precision levels available in existing general-purpose CPUs (single- and double-precision floating-point

as defined by IEEE standards). Precimonious [33], for example, implements a dynamic program analysis pass that, given a set of representative inputs and a target accuracy, aims at improving the runtime of the program by reducing its precision. The tool automatically performs a search of the configuration space and suggests a floating-point type for each variable in the program which can be implemented with lower precision (converting from `long double` to `double` or `float`). The work presented by Ho et al. [24] focuses on reducing the number of mantissa bits to be assigned to floating-point variables through the GNU Multiple Precision Floating-Point Routines (MPFR) [20]; the search for the best solution is implemented in Python by running multiple versions of the application with different precision levels, and it can be extended to find a fixed-point configuration suitable for FPGAs. SmartFPTuner [8], instead, introduces a machine learning component that predicts the output error generated by a reduced precision configuration, and uses mathematical programming to search for the smallest possible format for each variable among those supported by a custom RISC-V platform; such a combined approach results in much faster time-to-solution.

For what concerns FPGA implementations of approximate computing, multiple RTL and HLS libraries exist that provide optimized implementations of fixed- and floating-point operators with arbitrary precision. VFLOAT [37] and FloPoCo [14] provide VHDL components for custom-precision floating-point arithmetic that users have to manually integrate into their designs. The AdaptivFloat representation [36] was motivated by the precision requirements of deep learning applications on FPGA, but the implementation of corresponding arithmetic units still requires significant manual effort. Proprietary HLS tools map fixed- and floating-point C++ types with arbitrary precision onto a back-end RTL library during the synthesis process [2, 34]. TrueFloat [19] stands out as it not only provides a library of customizable floating-point operators with arbitrary precision, but also is fully integrated within the HLS tool Bambu [17]; its results are competitive with hand-designed library operators when applied to isolated functional units, and it outperforms commercial HLS tools when synthesizing whole applications because it allows Bambu to optimize floating-point operators rather than treating them as black boxes.

3 Methodology

3.1 High-Level Synthesis

We based our design flow on High-Level Synthesis (HLS) because it allows us to quickly translate existing MFA- and MA-based C applications into FPGA accelerators, and to evaluate different configurations without having to manually modify the RTL design. For example, the output of HLS tools is a description in low-level Verilog/VHDL, tailored to a specific FPGA board to extract the best performance in terms of latency (number of cycles, clock frequency), resource consumption (number of FPGA resources used among the different categories of logic and memory elements available), or power consumption. It is possible to specify directives that prioritize one of these metrics over the other, to change the hardware target, or to request specific backend optimizations starting from the same input program with no manual rewriting of the code.

FPGA vendors typically offer HLS as part of their design toolkits (e.g., Vitis HLS from AMD/Xilinx or the Intel HLS compiler), but such tools only support FPGA boards from a single vendor, and they cannot be modified, as their source code is proprietary. Instead, we exploit the open-source HLS framework Bambu [17], which facilitates research into new design automation methods. Bambu has been an invaluable tool in previous projects where knowledge of the internal HLS process and the possibility of modifying it were crucial to generate better accelerators in a specific domain (e.g., big data [32], aerospace [18]).

Bambu supports most C/C++ constructs, including function calls, access to arrays and structs, parameters passed by reference or copy, pointer arithmetic, dynamic resolution of memory accesses, and module sharing. Moreover, it can also take as input intermediate representations from the GCC and Clang compilers, leading to the possibility of direct integration between Bambu and compiler-based toolchains [7]. The HLS flow in Bambu is similar to a software compilation process, beginning with a high-level specification and generating low-level code through a series of analysis and optimization steps divided into three phases (front-end, middle-end, and back-end). In the front-end, Bambu parses the input code and translates it into an intermediate representation (IR), while numerous target-independent analyses and optimizations are performed in the middle-end. The back-end performs the actual synthesis of Verilog/VHDL code ready for simulation, logic synthesis, and implementation on FPGA or ASIC through external tools.

When a software description is translated into a hardware accelerator through HLS, there are ample opportunities to include optimizations that drastically impact the QoR in terms of performance, area, and energy consumption. One common example is loop unrolling: replicating the instructions of independent loop iterations allows implementing them in parallel in the accelerator, increasing performance at the expense of resource consumption. Alongside techniques that aim at exploiting different degrees of parallelism present in the input applications, another possibility to generate efficient accelerators in terms of performance per area is to explore the usage of custom data types, avoiding the generation of floating-point functional units which are usually inefficient when compared to the specialized floating-point arithmetic units of modern CPUs and GPUs.

In this paper, we focus on the customization of floating-point computations in isolation, disregarding all other optimization opportunities available in Bambu. While this will likely result in sub-optimal results in absolute terms, which would not justify the choice of offloading computations to FPGA, our aim is to conduct an ablation study to understand the impact of this specific type of optimization on the QoR of generated accelerators, and on the application accuracy. We will then exploit information gathered from this study during future experiments with MFA- and MA-based applications to be accelerated.

3.2 TrueFloat

Experiments with custom data types are usually limited by the back-end libraries supported by HLS tools, mostly focused on fixed-point types; however, Bambu also integrates the dedicated TrueFloat framework for the generation of custom floating-point types. TrueFloat allows users to specify custom formats for floating-point computations and automatically synthesizes corresponding optimized arithmetic units; different TrueFloat encodings can be specified for different parts of the input application, resulting in the generation of multi-precision accelerators. The main strength of TrueFloat is the integration within the HLS process, providing effortless translation between different floating-point encodings through simple command-line options and integration with other optimization techniques present in the HLS flow. TrueFloat also opens the possibility of generating an equivalent representation of the synthesized accelerator at a higher level of abstraction, which could be used for fast and accurate software simulation.

Figure 1 describes the TrueFloat synthesis flow within Bambu. The input code is in one of the standard formats supported by Bambu, and it contains standard floating-point operations and types. The user expresses one or more required custom representations (one for each function in the input code) through command-line options, and the HLS flow autonomously handles type replacement, conversions, and custom arithmetic units generation, avoiding manual and error-prone code rewriting.

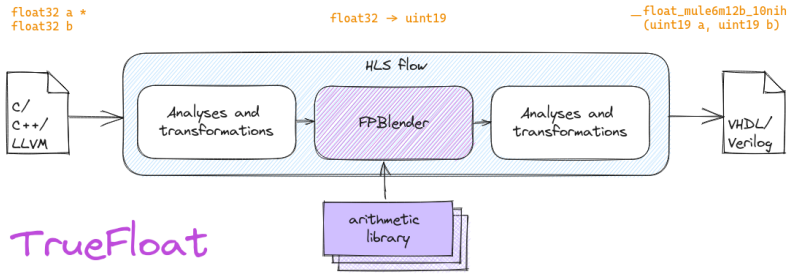


Figure 1 TrueFloat design methodology as presented in [19].

A compiler step called FPBlender handles all floating-point operations within the HLS flow in Bambu, exploiting information generated during previous analysis steps on the intermediate representation of the input program and allowing subsequent steps to apply more accurate optimizations after the custom floating-point format has been implemented. FPBlender generates ad-hoc functional units exploiting the TrueFloat library of templatized components, which contains optimized implementations for basic arithmetic operators (addition, subtraction, multiplication, division, and comparison) and bidirectional type conversion operators (floating-point to integer, integer to floating-point, and floating-point to floating-point). The TrueFloat library components are soft-float implementations in C built by combining basic integer operations; input and output parameters are defined as unsigned integers as well. All functions have arguments representing standard operands followed by a set of eight specialization arguments to indicate the number of exponent bits, fractional bits, the exponent bias, the rounding mode, the exception mode, whether hidden one is enabled, whether subnormals are enabled, and the sign mode.

Users of TrueFloat should explicitly define a floating-point format for each function they want to customize in the input code through the `-fp-format` command-line option, which Bambu will use to replace the standard single- or double-precision data type present in the input file. In particular, `-fp-format` requires the name of the function that will be customized and a string that encodes the requested format. Functions called by the selected function will be implemented with standard types unless –fp-format-propagate is set, instructing Bambu to propagate the custom data type to all called functions. Besides the choice of the number of bits for mantissa and exponent, TrueFloat also allows tuning settings such as whether subnormals are supported or not, as they result in simpler logic and lower resource consumption. The format string following the function name is composed as follows:

```
e<exp_bits>m<frac_bits>b<exp_bias><rnd_mode><exc_mode><spec><sign>
```

The number of bits requested for the exponent and for the mantissa are set through `exp_bits` and `frac_bits`, and `exp_bias` indicates the bias added to the unsigned value represented by the exponent bits. The rounding mode `rnd_mode` can be either nearest even, which is the IEEE standard rounding mode, or truncate, where no rounding is applied. The exception mode `exc_mode` can be set to require IEEE standard exceptions, to saturation, where infinite is replaced with the highest possible value and not-a-number results in undefined behavior, or to overflow, where both infinite and not-a-number result in undefined behavior. Finally, with `spec` it is possible to select whether to enable the IEEE standard representation with hidden one and subnormal numbers, while `sign` specifies whether all values should be considered as negative numbers, positive numbers, or if IEEE dynamic sign should be used.

Accelerators that use floating-point types with lower precision have higher performance and to use fewer FPGA resources with respect to designs based on standard `float` or `double` calculations. Moreover, TrueFloat arithmetic operators have been shown to have similar or better performance than other implementations from state-of-the-art libraries [19]. However, to the best of our knowledge, this is the first time that TrueFloat gets applied to a realistic input application beyond the single arithmetic operation or small kernel.

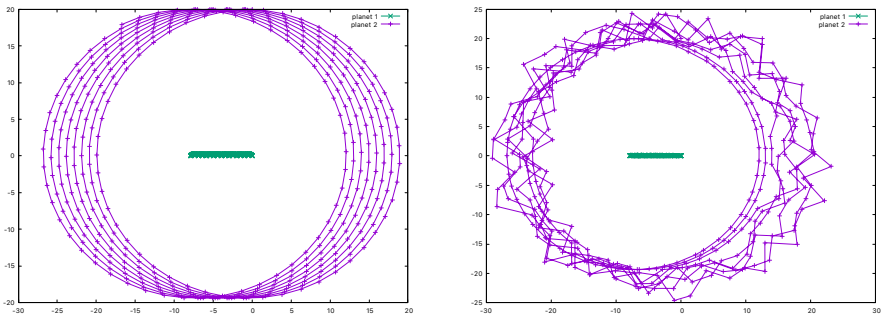
TrueFloat provides the required support to implement approximate applications in HLS, but it fully relies on the user to specify the desired floating-point configuration. Unfortunately, none of the approaches described in Section 2.2 can be directly applied to find an optimal TrueFloat configuration within an accuracy constraint. In fact, tools and techniques designed for software applications tend to consider only a few possible formats, which are the ones supported by the target CPU, while the design space of possible TrueFloat formats is much larger: it is possible to specify the exponent bitwidth and the mantissa bitwidth independently, and tune several other configuration options. Moreover, TrueFloat controls floating-point precision on a per-function level instead of variable by variable. Finally, the performance target for a hardware accelerator possibly includes resources consumption besides latency, and both metrics can only be assessed reliably after long logic synthesis runs. A learning-based approach would also not be feasible due to the absence of a reference dataset for our target application. Under these circumstances, manually sampling a limited number of points in the design space remains the simplest but most effective method to explore the trade-off between accelerator performance and accuracy of the results. Fortunately, this does not require any manual code rewriting, as TrueFloat automatically replaces the types of all variables and operations according to the specified Bambu command-line options.

4 Experimental results

4.1 Target application and configuration exploration

To keep synthesis times within reasonable limits (e.g., less than one hour per configuration), we evaluated a proxy application in place of the performance modeling use case. The proxy application solves the n-body problem through a Runge-Kutta-Fehlberg method with adaptive step size (RKF45) [16], and it has been implemented in C. The gravity interactions between bodies simulated by the application resemble the patterns of interaction among multiple agents in our target MFA/MA model, and the same RKF45 method will be used to solve the ODEs modeling the evolution of the cyber-physical system under investigation. Therefore, results from the synthesis of the n-body application can be used to decide which floating-point configurations shall be evaluated in the synthesis of the MFA/MA accelerator, keeping in mind that the requested precision may change when the number of interacting agents increases.

Our target application contains the kernel function to be accelerated (`solve`) together with a main program that runs the n-body simulation and plots its outputs. By looking at the output plots, it is easy to assess if the solution is converging correctly. Figure 2 shows the output plots obtained running simulations of the motion of two planets, i.e., the evolution of their orbits starting from an initial condition where the planet with the smallest mass has non-zero speed (purple trace) and the planet with the biggest one is still (green trace). The correct plot in Figure 2a is obtained with double-precision floating-point calculations, while using an extremely low precision in part of the simulation may lead to an incorrect plot such as the one in Figure 2b. Thanks to its advanced co-simulation features, Bambu is able to use the main function of the target application as a testbench to verify the correct behavior of



(a) Correct output. **(b)** Inaccurate output.

Figure 2 Plots generated by the target application with different floating-point accuracies.

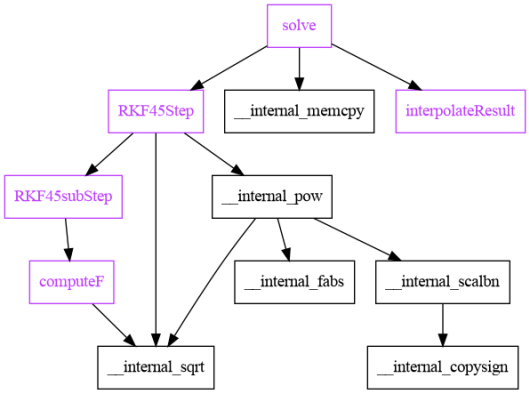


Figure 3 Call graph of the target application.

the accelerated kernel function; it is therefore possible to plot similar output graphs with results from the RTL simulation of the generated accelerator. TrueFloat configurations with unacceptably low precision will then be discarded either because the simulation does not produce a result within a given time limit or by comparing their outputs with Figure 2a.

As TrueFloat allows controlling floating-point configurations at the function level, it is important to analyze the call graph of the target application (Figure 3). The `solve` kernel function repeatedly calls subfunctions `interpolateResult` and `RKF45Step`; the latter contains a loop calling subfunction `RKF45SubStep`, which in turn calls multiple times the `computeF` function. The application also includes calls to floating-point mathematical functions from the C standard library. The amount of floating-point operations executed by the application partially depends on the floating-point format used, as there are loops that terminate based on the achieved precision; in a double-precision run, there are approximately 200.000 floating-point additions and just as many multiplications, floating-point divisions and other mathematical functions are less than 1000.

We kept the double-precision version of the target application as a baseline (configuration 0 in Table 1) and selected six TrueFloat configurations to compare against it, all of which produced correct simulation results. We first reduced the number of bits for mantissa and exponent for the whole accelerator, specifying a format for `solve` in the Bambu command-line options and requesting the propagation of the format to all called functions. The specialization string was selected to obtain IEEE-compliant behavior. The minimum number of bits was found to be 22 for the mantissa and 7 for the exponent, scaling the bias accordingly

Table 1 List of floating-point configurations to be evaluated.

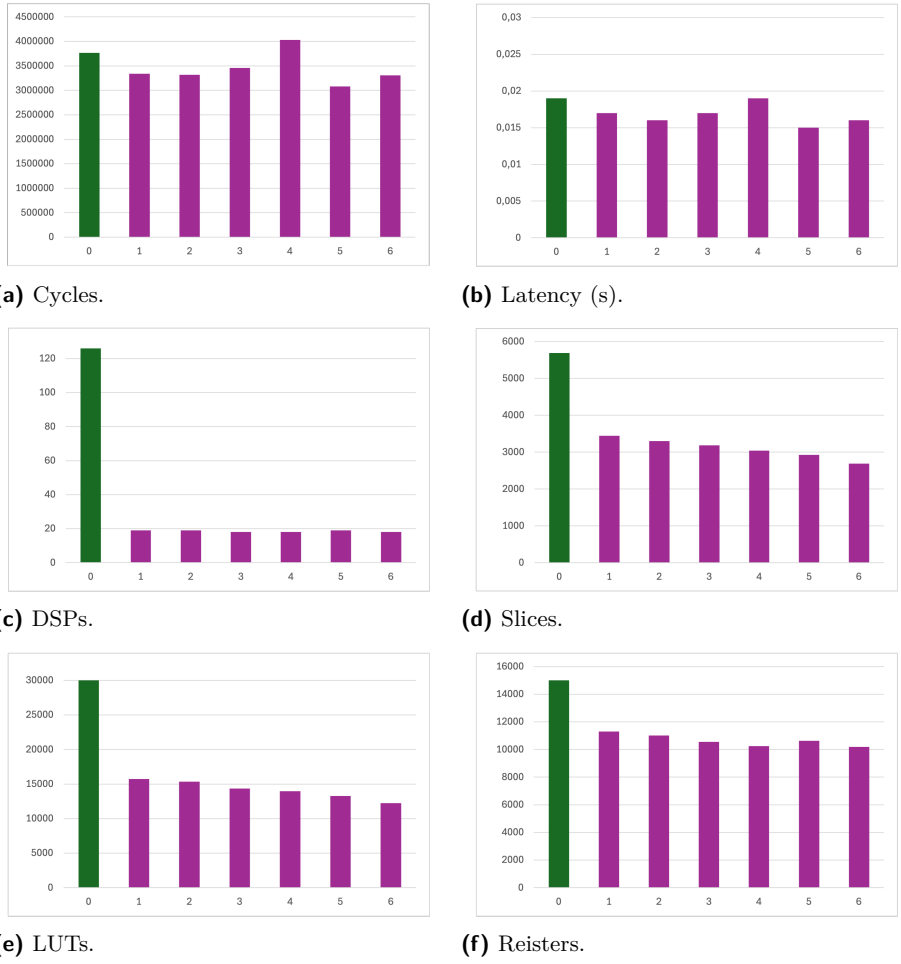
Configuration	Function affected	Exponent bits (bias)	Mantissa bits	Specialization string
0	`solve`	11 (-1023)	52	nihs
1	`solve`	8 (-127)	23	nihs
2	`solve`	7 (-63)	22	nihs
3	`solve`	7 (-63)	22	nihs
	`interpolateResult`	6 (-31)	10	nihs
4	`solve`	7 (-63)	22	nihs
	`interpolateResult`	6 (-31)	10	nihs
	`RKF45subStep`	6 (-31)	21	nihs
5	`solve`	7 (-63)	22	tih
6	`solve`	7 (-63)	22	tih
	`interpolateResult`	6 (-31)	10	tih

(configuration 2); trimming the bitwidths further resulted in no convergence or unacceptable errors in the output plots. Interestingly, this means that IEEE single-precision floating-point would also be an acceptable configuration, and so we included that in the evaluation (configuration 1). Next, we tested whether we could further reduce the number of bits with respect to configuration 2 by selecting a different format for one or more subfunctions (configurations 3 and 4); in fact, lowering the precision of functions that contain fewer operations has a smaller impact on the accelerator QoR, but also a smaller impact on the application accuracy. Finally, we acted on the specialization string of configurations 2 and 3 removing support for subnormal numbers and using truncation instead of rounding to nearest even (configurations 5 and 6), as this was previously shown to be beneficial, especially in terms of area consumption [19].

4.2 Synthesis results

We synthesized six versions of the n-body application with Bambu according to the configurations of Table 1 and evaluated the QoR of the generated accelerators after place-and-route (p&r). We chose an Alveo U55C FPGA from AMD/Xilinx as target and requested a clock period of 5ns, which was achieved by all configurations; all other Bambu options were left as default in order to focus on the effects of customizing floating-point formats on performance. We evaluated and reported in Figure 4 the number of clock cycles from simulation, the latency considering the achieved frequency post p&r, and area consumption in terms of number of digital signal processing blocks (DSPs, the scarcest resource on FPGA and required to implement floating-point multiplications), slices, lookup tables (LUTs), and registers.

Green bars represent synthesis results for configuration 0, i.e., the 64-bit double-precision baseline; all other configurations use floating-point formats with fewer bits, which translates directly into fewer resources consumed (Figures 4c-4f). The comparison is especially striking in Figure 4c, as the other configurations use $\sim 85\%$ fewer DSPs. Looking at performance (Figures 4a and 4b), however, configuration 0 is not the slowest one: configurations 3, 4, and 6 perform worse than configurations 1, 2, and 5, with configuration 4 even running slower than the double-precision baseline. Despite using custom formats with fewer bits, in fact, these configurations have to pay the price of converting data between different types every time that one of the affected subfunctions is called. (It is likely possible to restructure the code to mitigate this issue, e.g., by unrolling loops that call the same function multiple times.) The suggestions of moving from rounding to truncation and of dropping support for subnormals were proven to be beneficial, as configurations 5 and 6 result in better QoR than configurations 2 and 3 along all considered metrics.



(a) Cycles. **(b)** Latency (s).

(c) DSPs. **(d)** Slices.

(e) LUTs. **(f)** Reisters.

Figure 4 Synthesis results for six different floating-point configurations.

From these results, we can conclude that using custom floating-point formats generated by TrueFloat is highly beneficial to reduce the area consumption of ODE solvers accelerated on FPGA. The latency of the generated accelerators can decrease when reducing the number of bits used for floating-point computations, but other factors, such as the amount of required conversion operators, may counter the improvement. In absolute terms, the accelerators we generated are quite slow ($\sim$ 7 times higher latency than software execution); however, we did not exploit any of the available parallelism in the application, and thus we are confident that it would not take too much effort to obtain higher-performance versions of the accelerator. Even in the worst case that we evaluated, the resources required occupy less than $\sim$ 4% of the target FPGA, suggesting that, for example, aggressive unrolling of loops should be possible (considering also that the application is highly compute-intensive and thus memory bandwidth should not be a bottleneck if intermediate results are kept on-chip).

5 Conclusion

We applied the TrueFloat framework to an ODE solver performing double-precision floating-point calculations to generate FPGA accelerators with custom precision, aiming at studying the effect of lower mantissa and exponent bitwidth on performance and area consumption.

The results we obtained highlight the possible savings in terms of resources and latency but also the care required to choose the best floating-point configuration in a wide design space where the loss of accuracy in the application output is also a concern; future research on automated design space exploration will undoubtedly improve the usability of TrueFloat.

References

1 Hassan Al-Yassin, Mohammed A. Fadhel, Omran Al-Shamma, and Laith Alzubaidi. Solving Lorenz ODE System Based Hardware Booster. In *Intelligent Systems Design and Applications (ISDA)*, pages 245–254, 2019. doi:10.1007/978-3-030-49342-4_24.

2 AMD/Xilinx. Arbitrary Precision Data Types Library, 2024. URL: https://docs.amd.com/r/en-US/ug1399-vitis-hls/Arbitrary-Precision-AP-Data-Types.

3 Silas Bartel and Matthias Korch. Generation of logic designs for efficiently solving ordinary differential equations on field programmable gate arrays. *Software: Practice and Experience*, 53(1):27–52, 2023. doi:10.1002/spe.3043.

4 Soham Bhattacharya and Dwaipayan Chakraborty. Design-Space Exploration of the Runge-Kutta Hardware Accelerator for Solving Ordinary Differential Equation. In *2023 IEEE International Conference on Electrical, Automation and Computer Engineering (ICEACE)*, pages 260–264, 2023. doi:10.1109/ICEACE60673.2023.10442673.

5 Soham Bhattacharya and Dwaipayan Chakraborty. Implementation of a Hardware Accelerator with FPU-Based Euler and Modified Euler Solver For an Ordinary Differential Equation. In *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1106–1112, 2023. doi:10.1109/CSCI62032.2023.00182.

6 Andrea Bobbio, Marco Gribaudo, and Miklós Telek. Analysis of Large Scale Interacting Systems by Mean Field Method. In *2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 215–224, 2008. doi:10.1109/QEST.2008.47.

7 Nicolas Bohm Agostini, Serena Curzel, Jeff Jun Zhang, Ankur Limaye, Cheng Tan, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Brooks, Gu-Yeon Wei, and Antonino Tumeo. Bridging Python to Silicon: The SODA Toolchain. *IEEE Micro*, 42(5):78–88, 2022. doi:10.1109/MM.2022.3178580.

8 Andrea Borghesi, Giuseppe Tagliavini, Michele Lombardi, Luca Benini, and Michela Milano. Combining learning and optimization for transprecision computing. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pages 10–18, 2020. doi:10.1145/3387902.3392615.

9 Dario Bruneo, Marco Scarpa, Andrea Bobbio, Davide Cerotti, and Marco Gribaudo. Analytical modeling of swarm intelligence in wireless sensor networks through Markovian agents. In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, 2009. doi:10.4108/ICST.VALUETOOLS2009.7672.

10 Lelio Campanile, Mauro Iacono, Fiammetta Marulli, Marco Gribaudo, Michele Mastrioianni, et al. A DSL-Based Modeling Approach For Energy Harvesting IoT/WSN. In *36th International ECMS Conference on Modelling and Simulation*, pages 317–323, 2022. doi:10.7148/2022-0317.

11 Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology and Systems*, 15(4):1–42, 2022. doi:10.1145/3530775.

12 Francesca Cordero, Daniele Manini, and Marco Gribaudo. Modeling Biological Pathways: An Object-Oriented like Methodology Based on Mean Field Analysis. In *2009 Third International Conference on Advanced Engineering Computing and Applications in Sciences*, pages 117–122, 2009. doi:10.1109/ADVCOMP.2009.25.

13 Ayad M. Dalloo, Amjad Jaleel Humaidi, Ammar K. Al Mhdawi, and Hamed Al-Raweshidy. Approximate Computing: Concepts, Architectures, Challenges, Applications, and Future Directions. *IEEE Access*, 12:146022–146088, 2024. doi:10.1109/ACCESS.2024.3467375.

14 F. de Dinechin and B. Pasca. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, 2011. doi:10.1109/MDT.2011.44.

15 Alireza Fasih, Tuan Do Trong, Jean Chamberlain Chedjou, and Kyandoghere Kyamakya. New computational modeling for solving higher order ODE based on FPGA. In *2009 2nd International Workshop on Nonlinear Dynamics and Synchronization*, pages 49–53, 2009. doi:10.1109/INDS.2009.5227969.

16 Erwin Fehlberg. *Low-order classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problems*, volume 315. National aeronautics and space administration, 1969.

17 Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, et al. Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1327–1330, 2021. doi:10.1109/DAC18074.2021.9586110.

18 Fabrizio Ferrandi, Michele Fiorito, Claudio Barone, Giovanni Gozzi, and Serena Curzel. High-Level Synthesis Developments in the Context of European Space Technology Research. In *15th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 13th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2024)*, volume 116, pages 1:1–1:12, 2024. doi:10.4230/OASIcs.PARMA-DITAM.2024.1.

19 Michele Fiorito, Serena Curzel, and Fabrizio Ferrandi. TrueFloat: A Templatized Arithmetic Library for HLS Floating-Point Operators. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 23rd International Conference, SAMOS 2023, Samos, Greece, July 2–6, 2023, Proceedings*, pages 486–493, 2023. doi:10.1007/978-3-031-46077-7_35.

20 Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Softw.*, 33(2), 2007. doi:10.1145/1236463.1236468.

21 Jonathan Garcia-Mallen, Shuohao Ping, Alex Miralles-Cordal, Ian Martin, Mukund Ramakrishnan, and Yipeng Huang. Towards an Accelerator for Differential and Algebraic Equations Useful to Scientists. *IEEE Computer Architecture Letters*, 22(2):185–188, 2023. doi:10.1109/LCA.2023.3332318.

22 Marco Gribaudo, Davide Cerotti, and Andrea Bobbio. Analysis of On-off policies in Sensor Networks Using Interacting Markovian Agents. In *2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 300–305, 2008. doi:10.1109/PERCOM.2008.100.

23 Marco Gribaudo, Mauro Iacono, and Daniele Manini. COVID-19 Spatial Diffusion: A Markovian Agent-Based Model. *Mathematics*, 9(5), 2021. doi:10.3390/math9050485.

24 Nhut-Minh Ho, Elavarasi Manogaran, Weng-Fai Wong, and Asha Anoosheh. Efficient floating point precision tuning for approximate computing. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 63–68, 2017. doi:10.1109/ASPDAC.2017.7858297.

25 Andrew Hollabough and Dwaipayan Chakraborty. An Open-Source Co-processor for Solving Lotka-Volterra Equations. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1690–1694, 2022. doi:10.1109/ISCAS48785.2022.9937835.

26 Chen Huang, Bailey Miller, Frank Vahid, and Tony Givargis. Synthesis of networks of custom processing elements for real-time physical system emulation. *ACM Trans. Des. Autom. Electron. Syst.*, 18(2), 2013. doi:10.1145/2442087.2442092.

27 Chen Huang, Frank Vahid, and Tony Givargis. A Custom FPGA Processor for Physical Model Ordinary Differential Equation Solving. *IEEE Embedded Systems Letters*, 3(4):113–116, 2011. doi:10.1109/LES.2011.2170152.

28 Chen Huang, Frank Vahid, and Tony Givargis. Automatic synthesis of physical system differential equation models to a custom network of general processing elements on FPGAs. *ACM Trans. Embed. Comput. Syst.*, 13(2), 2013. doi:10.1145/2514641.2514650.

29 Thomas G. Kurtz. Solutions of Ordinary Differential Equations as Limits of Pure Jump Markov Processes. *Journal of Applied Probability*, 7(1):49–58, 1970. URL: `http://www.jstor.org/stable/3212147`.

30 Siting Liu and Jie Han. Hardware ODE Solvers using Stochastic Circuits. In *Proceedings of the 54th Annual Design Automation Conference (DAC)*, 2017. `doi:10.1145/3061639.3062258`.

31 Sparsh Mittal. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.*, 48(4), 2016. `doi:10.1145/2893356`.

32 Christian Pilato, Subhadeep Banik, Jakub Beránek, Fabien Brocheton, Jeronimo Castrillon, Riccardo Cevasco, et al. A System Development Kit for Big Data Applications on FPGA-based Clusters: The EVEREST Approach. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2024. `doi:10.23919/DATE58400.2024.10546518`.

33 Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, et al. Precimonious: Tuning assistant for floating-point precision. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013. `doi:10.1145/2503210.2503296`.

34 Siemens Digital Industries Software. HLS Libs, 2024. URL: `https://hlslibs.org/`.

35 Ioannis Stamoulias, Matthias Möller, Rene Miedema, Christos Strydis, Christoforos Kachris, and Dimitrios Soudris. High-Performance Hardware Accelerators for Solving Ordinary Differential Equations. In *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2017. `doi:10.1145/3120895.3120919`.

36 T. Tambe, E. Y. Yang, Z. Wan, Y. Deng, V. Janapa Reddi, et al. Algorithm-Hardware Co-Design of Adaptive Floating-Point Encodings for Resilient Deep Learning Inference. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. `doi:10.1109/DAC18072.2020.9218516`.

37 Xiaojun Wang and Miriam Leeser. VFloat: A Variable Precision Fixed- and Floating-Point Library for Reconfigurable Hardware. *ACM Trans. Reconfigurable Technol. Syst.*, 3(3), 2010. `doi:10.1145/1839480.1839486`.

System-Level Timing Performance Estimation Based on a Unifying HW/SW Performance Metric

Vittoriano Muttillo
University of Teramo, Italy

Vincenzo Stoico
Vrije Universiteit Amsterdam, The Netherlands

Giacomo Valente
University of L'Aquila, Italy

Marco Santic
University of L'Aquila, Italy

Luigi Pomante
University of L'Aquila, Italy

Daniele Frigioni
University of L'Aquila, Italy

Abstract

The rapidly increasing complexity of embedded systems and the critical impact of non-functional requirements demand the adoption of an appropriate system-level HW/SW co-design methodology. This methodology tries to satisfy all design requirements by simultaneously considering several alternative HW/SW implementations. In this context, early performance estimation approaches are crucial in reducing the design space, thereby minimizing design time and cost. To address the challenge of system-level performance estimation, this work presents and formalizes a novel approach based on a unifying HW/SW performance metric for early execution time estimation. The proposed approach estimates the execution time of a C function when executed by different HW/SW processor technologies. The approach is validated through an extensive experimental study, demonstrating its effectiveness and efficiency in terms of estimation error (i.e., lower than 10%) and estimation time (close to zero) when compared to existing methods in the literature.

2012 ACM Subject Classification Computer systems organization → Embedded systems; Computer systems organization → Embedded hardware

Keywords and phrases embedded systems, hw/sw co-design, performance estimation, lasso, machine learning

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2025.3

Supplementary Material *Software (Source Code)*: `https://github.com/hepsycode/SLIDE-x` [22] archived at `swh:1:rev:0907cf39f7d023d0dc2e8307e1ffef6115d0377a`

Funding This research work has been funded by the Electronic Components and Systems for European Leadership Joint Undertaking (ECSEL JU) through the project AIDOaRt, grant agreement No. 101007350, and the Key Digital Technologies Joint Undertaking (KDT JU) through the project MATISSE, grant agreement No. 101140216.

1 Introduction

In the last thirty years, there has been an exponential increase in the exploitation of embedded systems in everyday life. This increase has led to a rise in the complexity of such embedded systems due to: (i) the continuous demand for additional improvements in both functional and non-functional requirements and (ii) the growing design automation with the application of embedded systems in various domains (e.g., Automotive, Aerospace) [11, 25].

16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2025).
Editors: Daniele Cattaneo, Maria Fazio, Leonidas Kosmidis, and Gabriele Morabito; Article No. 3; pp. 3:1–3:14

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

Therefore, designing these systems is even more a critical task and so early-stage HW/SW performance estimation for rapid design space exploration at higher abstraction levels becomes crucial [10, 29].

In this context, numerous studies have explored the use of Machine Learning (ML) techniques for performance estimation [2, 14, 15, 17, 28, 34]. The adoption of these techniques has been driven by the challenges associated with creating an accurate analytical model of the HW/SW micro-architecture, which is often error-prone or sometimes impossible due to the lack of detailed documentation and necessary human expertise for model design [2]. Despite this scenario, the current state of the art lacks, to the best of our knowledge, of a unified HW/SW model capable of facilitating rapid performance estimation across several platforms at the system level.

For the above reasons, this study investigates how to overcome the limitations of the existing methods, particularly those restricted to specific application domains or technologies, through an approach that allows performance estimation of different HW/SW designs at the system level of abstraction. The provided approach uses the LASSO model to estimate the CC4CS performance metric presented and validated in [26]. CC4CS is a statement-level metric that can be used to quantify and, therefore, compare the performance of different processor technologies (i.e., Commercial Off-the-Shelf – COTS 8/32-bit embedded processors and HW components synthesized on FPGAs). CC4CS is defined as the ratio between the clock cycles and statements executed by a C function. In addition, this study presents a preliminary evaluation of the accuracy of the LASSO model in estimating CC4CS values. The evaluation of the accuracy of LASSO is carried out by using the **SLIDE-x**[1] framework, which executes a benchmark of well-known C functions across a set of 3 processors, namely Intel 8051, Atmega328p, Leon3, and an FPGA, i.e., the Artix7. Consequently, SLIDE-x outputs the CC4CS values that are used to train the LASSO model. Finally, the accuracy of LASSO is evaluated by comparing its predictions against the measurement profiled using SLIDE-x. The results are promising as they show a Mean Absolute Percentage Error (MAPE) of less than 10% for the Intel 8051, Leon3, and Artix7, with a maximum speed-up of up to 32x compared to the traditional HLS flow. In summary, our paper offers the following contributions: (1) formal HW/SW processor characterization through statistical analysis; (2) a detailed regression-based approach for evaluating HW/SW design performance; (3) a preliminary assessment of the accuracy of our performance predictions. This work is useful for system designers, helping them evaluate multiple HW/SW solutions and reduce design space exploration overhead.

2 Related works

In this section, we review the current state of research focused on two key areas within embedded systems design: predicting the timing performance of processors built to execute a given *Instruction Set Architecture (ISA)* (i.e., General Purpose Processors – GPPs, called SW processors), and of processors designed to directly execute application functions (i.e., Single Purpose Processors - SPPs, called HW processors) at the system level of abstraction.

To describe a SW processor and its behavior, several levels of abstraction can be considered. Accordingly, several timing estimations can be performed [27]. In such a context, the authors in [14] use a linear regression technique based on an application analysis performed at the Register Transfer Level (RTL) internal representation of the GNU GCC compiler (i.e.,

[1] SLIDE-x repository: `https://github.com/hepsycode/SLIDE-x`

needs for micro-architectural knowledge of the system). Zhang et Al. [34] use a linear regression model to estimate the performance of a given embedded software executed by the RISC-V processor, using metrics related to (assembly) instruction level. The final speed-up in comparison to the cycle-accurate simulation is up to 5x for RV32I and 4.2x for RV32IM. Finally, Amalou et Al. [2] present several approaches: (1) Ithemal: a tool that uses a Recurrent Neural Network (RNN) architecture with a hierarchical Long Short-Term Memory (LSTM) approach to predict the throughput of a set of instructions considering the opcodes and operands of instructions in a basic block (BB); (2) CATREEN: an RNN predictive algorithm able to predict the steady-state execution time of BBs in a program; (3) ORXESTRA: a tool that predicts the execution time of BBs within compiled binaries using a ML technique named Transformers XL, a recurrent variant of Transformers.

In the HW domain, the use of *High-Level Synthesis* (HLS) tools has become of vital importance [3, 16]. HLS tools provide automatic transformation of C/C++/SystemC specifications into *Hardware Description Languages* (HDL) like Verilog or VHDL, significantly boosting productivity in custom hardware development [2][3]. However, for large-scale systems, the time needed to perform HLS can often become a bottleneck [27]. Additionally, fast platform selection remains a significant challenge for developers due to the significant performance variations among platforms for the same workload [8].

To address this issues, Makrani et al. [15] introduced the Cross-Platform Performance Estimation (XPPE) tool based on ML. XPPE uses the resource usage reported by the Xilinx HLS tool and predicts application acceleration on various platforms using a Neural Network (NN) model, considering both application characteristics and FPGA platform parameters. The authors of [28] propose HLSPredict, an ML-based cross-platform estimator. Unlike XPPE, HLSPredict uses workloads as inputs to estimate performance on an FPGA by executing them on a Commercial Off-The-Shelf (COTS) host CPU. Finally, [17] presents Pyramid, a tool that uses ML to estimate optimal performance and resource usage of HLS designs, with the Random Forest (RF) outperforming other ML models.

2.1 State-of-the-Art Limitations

Table 1 compares our work with state-of-the-art ML studies by examining prediction errors. While existing studies focus on reducing errors through SW implementation or HW synthesis, none offer a unified HW/SW model with low prediction times and errors. This gap highlights the need for a unified model to compare HW and SW processors at the system level. Our paper aims to address these limitations.

3 Preliminaries

Our work introduces a method for system-level execution time estimation of C functions across different HW/SW processor technologies, using the CC4CS performance metric [26]. This metric, already used in literature for HW/SW Co-Design methodologies [21, 24], simplifies performance estimation and comparison by abstracting the execution of "generic C statements". Our approach is built on the model in [5], which defines a "generic C statement" as a combination of fundamental units, called "atoms". Atoms are the basic components of statements, and the complexity of a statement depends on the number of atoms it con-

[2] Panda/Bambu Project: `https://panda.dei.polimi.it/`
[3] Vitis HLS: `https://www.xilinx.com`

Table 1 Comparison of literature Timing Estimation works. L/NL:= Linear/Non-Linear, LR:= Linear Regression.

Work	Target	Approach	Error (%)
[14]	ARM926EJ-S (SW) LEON3 (SW)	LR	$8.25\% \leq \cdots \leq 15.45\%$ $8.03\% \leq \cdots \leq 13.60\%$
[34]	RV32I (SW) RV32IM (SW)	LR	7.03% 5.27%
[2]	ARM Cortex M4, M7, A53, A72 (SW)	ITHEMAL CATREEN ORXESTRA	$9.1\% \leq \cdots \leq 18.2\%$ $8.9\% \leq \cdots \leq 13.4\%$ $6.2\% \leq \cdots \leq 8.9\%$
[28]	Artix7 (HW)	L/NL ML	$1.88\% \leq \cdots \leq 9.79\%$
[15]	20 Xilinx FPGA (HW)	NN	$5.1\% \leq \cdots \leq 9\%$
[17]	3 Xilinx FPGA (HW)	L/NL ML	$3.5\% \leq \cdots \leq 4.8\%$
Our Work	8051 (SW - CISC), ATmega (SW - RISC), LEON3 (SW - RISC)) Bambu (HW - Artix7)	System-Level Linear ML (Unified HW/SW Approach)	$4.26\% \leq \cdots \leq 6.53\%$ $6.99\% \leq \cdots \leq 22.04\%$ $0.34\% \leq \cdots \leq 2.58\%$ $6.54\% \leq \cdots \leq 11.84\%$

tains. Although the complexity of a C statement is not strictly predefined [26], factors like programmer experience, coding style, and standards [13] usually keep it at a "low/medium average complexity". A "generic C statement" reflects the common way programmers write statements. When a C function runs with input data set D_k, each atom and statement executes a certain number of times, enabling the collection of profiling data, such as through tools like Gcov.

3.1 Performance Model for SW Processors

This subsection introduces a general mathematical model representing the execution time of a C function executed by a basic GPP (i.e., no advanced microarchitecture features, such as pipeline), refining the model proposed in [5]. To perform timing performance estimation, a model for the approximate (ideal) execution time $\overline{T}_k$ is required. Therefore, in a basic GPP, the ideal execution time of a generic C function is:

$$\overline{T}_k^{SW} = N_k^I \cdot CC_j \cdot \tau_j \tag{1}$$

where CC_j is the average number of clock cycles per statement, τ_j is the GPP processor's clock period, and N_k^I represents the number of executions of all assembly instructions in the generic C function when run with input data set D_k.

3.2 Performance model for HW Processors

HW implementation (i.e., SPP) of a generic C function can be done using HLS tools like Bambu[2], LegUp [6], or Vitis HLS[3]. As noted in [27], common HLS practices use an intermediate representation to capture the control and data flows of the C code. Basic Blocks (BBs) represent the code control flow at the statement level.

The visual representation of control and data flow using BBs is called Control and Data Flow Graph (CDFG). Each operation is assigned to a Functional Unit (FU) capable of executing it [1], and FUs are encapsulated within the BBs of the CDFG model, as shown in Figure 1. Each datapath within the q-th BB contains $L_{j,q}^{FU}$ functional units of type $FU_{j,q,v}$

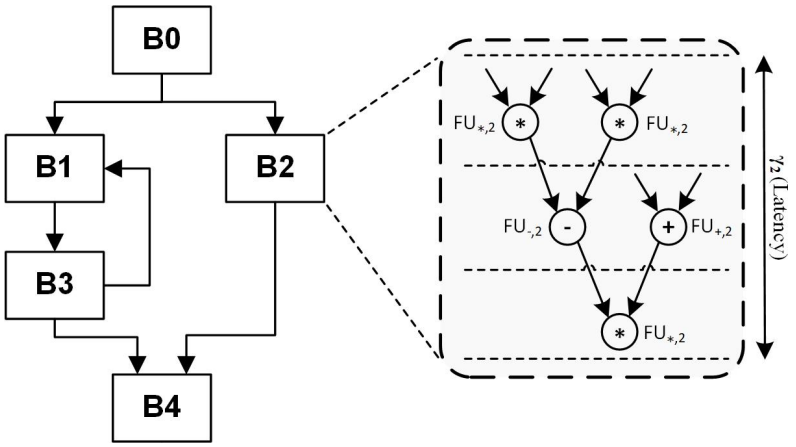


Figure 1 CDFG representation. $FU_{*,2}$ is one of the FU of basic block B2, while γ_2 is the total latency of basic block B2.

(e.g., sum, mul, sub). The total propagation time (latency) depends on Integrated Circuit (IC) technologies and micro-architecture. $L_{j,q}^{FU}$ is obtained from code analysis on non-scheduled DFGs, while delays from registers and multiplexers are ignored. The actual execution time of a generic C function synthesized as an SPP is defined as:

$$T_k^{HW} = \sum_{q=1}^{N_k^{BB}} \sum_{h=1}^{N_{j,q,k}^{st}} \frac{1}{N_{j,q,k}^{st}} \sum_{v=1}^{L_{j,q}^{FU}} \omega_{j,q,v} \cdot \gamma_{j,q,v,k} \cdot \tau_j$$
$$\text{where } \omega_{j,q,v} = \begin{cases} 1 \text{ if } FU_{j,q,v} \in \text{longest data path} \\ 0 \text{ otherwise} \end{cases} \quad (2)$$

where T_k^{HW} in Eq. 2 is the execution time of the q-th BB in a generic C function synthesized as SPP p_j with input data set D_k. N_k^{BB} is the total number of executed BBs, $N_{j,q,k}^{st}$ is the number of executed statements in the q-th BB, and $\gamma_{j,q,v,k}$ is the latency of the v-th FU in the q-th BB. The clock period τ_j depends on IC technology, micro-architecture, and scheduling policy. Assuming no multi-cycling, pipelining, or chaining, τ_j is given by the following equation.

$$\tau_j = \max_{\forall\{v,q\}} t(FU_{j,q,v})$$

An average slack time (i.e., idle time of operations in a control step) can be used in multi-cycling and pipelined implementations [27] to calculate τ_j. HLS tools allow setting a desired clock period τ_j (e.g., Bambu[2]), and aim to minimize the difference between the desired τj and the actual τj. From Eq. 2, the simplified execution time model for a generic C function synthesized as SPP can be approximated as follows:

$$\overline{T}_{k,h}^{HW} = \sum_{h=1}^{L^{st}} \overline{T}_{k,h}^{HW} = 1/N_k^{st} \cdot \sum_{q=1}^{N_k^{BB}} \sum_{v=1}^{L_{j,q}^{FU}} \omega_{j,q,v} \cdot \gamma_{j,q,v} \cdot \tau_j^* \quad (3)$$

where $\overline{T}_{k,h}^{HW}$ is the average execution time of the h-th statement belonging to the q-th BB in a generic C function synthesized as SPP p_j with data D_k under the desired clock period τ_j^*.

3.3 Proposed Unified HW/SW Performance Model

The proposed unified HW/SW performance model integrates the timing behavior of a generic C function executed on a GPP with that of the same function synthesized as an SPP, achieved through HLS tools.

For the SW side, let $N^I_{h,k,j}$ represent the number of assembly instructions needed to execute statement h of the C function on GPP p_j with input data set D_k. This can be determined using an assembly-level execution trace [20]. The average number of executed assembly instructions $\overline{N}^I_k$ is:

$$\overline{N}^I_k = 1/N^{st}_k \cdot \sum_{h=1}^{L^{st}} N^I_{h,k,j} \quad \text{and} \quad N^I_k = \sum_{h=1}^{L^{st}} N^I_{h,k,j} = \overline{N}^I_k \cdot N^{st}_k$$

N^I_k is the total number of executed assembly instructions, and N^{st}_k is the total number of executed statements for a generic C function with D_k. Therefore, Eq. 1 can be redefined:

$$\overline{T}^{SW}_k = \overline{N}^I_k \cdot N^{st}_k \cdot CC_j \cdot \tau_j = N^{st}_k \cdot \frac{\sum_{h=1}^{L^{st}} N^I_{h,k,j} \cdot CC_j}{N^{st}_k} \cdot \tau_j \tag{4}$$

According to Eq. 4, the expressions for the approximate (ideal) $\overline{T}_k$ can be redefined for basic GPP processors as follows:

$$\overline{T}^{SW}_k = N^{st}_k \cdot \bar{t}(ST_k)^{SW} \tag{5}$$

$$\bar{t}(ST_k)^{SW} = 1/N^{st}_k \cdot \sum_{h=1}^{L^{st}_s} N^I_{h,k,j} \cdot CC_j \cdot \tau_j \tag{6}$$

$$CC^{SW}_{j,k} = \sum_{h=1}^{L^{st}} N^I_{h,k,j} \cdot CC_j \tag{7}$$

Eq. 7 shows the total clock cycles $CC^{SW}_{j,k}$ needed to execute a generic C function on GPP p_j with input D_k. This value is normalized in Eq.6 to the total number of C statements executed with input D_k.

For the HW side, based on Eq. 3, the number of clock cycles $CC^{HW}_{j,k}$ needed to execute a generic C function synthesized as an SPP can be evaluated using HLS tools[2]. These tools generate HDL files (Verilog or VHDL) and provide the required clock cycles for executing C functions with data D_k, as follows:

$$CC^{HW}_{j,k} = \sum_{q=1}^{N^{BB}_k} \sum_{v=1}^{L^{FU}_{j,q}} \omega_{j,q,v} \cdot \gamma_{j,q,v} \tag{8}$$

According to Eq. 5 and Eq. 8, a generic C function with input data D_k executed by a GPP or by an SPP requires an execution time of:

$$\overline{T}^{\frac{HW}{SW}}_k = N^{st}_k \cdot \bar{t}(ST_k)^{\frac{HW}{SW}} = N^{st}_k \cdot \left(\frac{CC^{\frac{HW}{SW}}_{j,k}}{N^{st}_k} \right) \cdot \tau_j \tag{9}$$

The fraction within parentheses in Eq. 9 represents the unified metric *Clock Cycles for C statements* (CC4CS) [26]. The CC4CS metric, at the statement level of abstraction, encompasses both atoms (SW) and blocks (HW). According to Eq. 9, the empirical evaluation of the CC4CS metric across a set of HW/SW processor technologies requires a well-defined methodology for automated and repeatable operations, as shown in Figure 2.

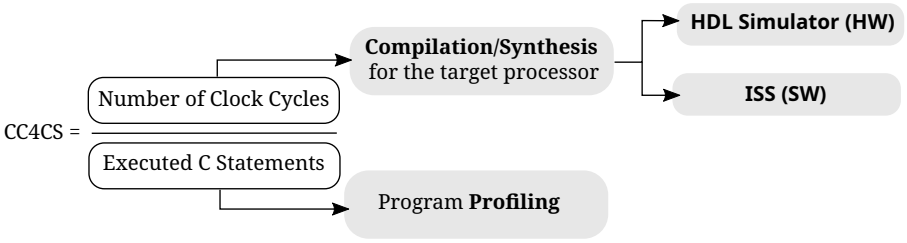


Figure 2 CC4CS Evaluation Methodology.

For a specific processor p_j, each C function is taken from the benchmark. Random input data $D_{s,k}$ is generated and uniformly distributed within a set range. Two parallel processes then determine the clock cycles required by the target HW or SW processor to execute the function ($CC_{j,k}^{\frac{HW}{SW}}$) and the number of C statements executed (N_k^{st}), which depends only on the input data and function, not on the processor. To evaluate CC4CS across HW/SW processors, the process involves: (a) selecting target processors p_j; (b) choosing benchmark C functions; (c) generating input data sets D_k; (d) profiling C functions to find the number of executed statements N_k^{st} (using tools like Gcov); (e) compiling/synthesizing C functions for each processor; (f) performing cycle-accurate simulations to extract the real execution time $\bar{t}(ST_k)$. This is done through ISS or HDL simulations. Each processor p_j will then have a Cumulative Distribution Function (CDF) of $CC4CS_j$. Different compiler optimization flags can be applied, though in this work, the -O0 flag is used as proof of concept, leaving other flags for future exploration.

3.4 Performance Estimation Approach

This work addresses the challenge of determining an estimator, $\hat{T}_{s,k}$, for the actual (real) execution time of a given C function z_s implemented or synthesized through both HW/SW processor technologies. The total actual (real) estimation time of a generic C function is expressed as follows:

$$T_k = \sum_{h=1}^{L^{st}} N_{h,k}^{st} \cdot t(ST_{h,k}) \simeq f(N_k^{st}, \hat{t}(ST_k)) = \hat{T}_k \tag{10}$$

where $\hat{t}(ST_k)$ is the estimated average time to execute a statement ST_k in a generic C function. The error to be minimized over functions and input data sets is:

$$\min_{\forall\{D_k\}} \epsilon_k^2 = \min_{\forall\{D_k\}} (T_k - \hat{T}_k)^2 \tag{11}$$

According to Eq. 10, our proposed solution uses the Least Absolute Shrinkage and Selection Operator (LASSO) [7] to exploit an approach called the CC4CS LASSO Regression Approach (CLRA), as follows:

$$\begin{aligned} &\hat{T}_k = \beta_0 + N_k^{st} \cdot \hat{t}(ST_h) \cdot \hat{\beta} + \lambda \cdot |\hat{\beta}| \\ &\hat{t}(ST_h, z_s) = \bar{t}(ST_k) + \delta \\ &\bar{t}(ST_k) = g(CC4CS_j, \tau_j) \text{ AND } \delta = h(\theta, CC4CS_j, \tau_j) \end{aligned} \tag{12}$$

where θ depends on correction functions like, e.g., the affinity value defined in [4]. LASSO regression performs an L1 regularization that adds a penalty equal to the absolute value of the magnitude of the coefficients. LASSO solutions are quadratic programming problems best solved with dedicated software tools (e.g., Matlab). According to Eq. 11 and Eq. 12, we define the final estimation problem as follows:

$$\min_{\beta_0,\hat{\beta}} \quad 1/2d\sum_{k=1}^{d}[T_k - \beta_0 - N_k^{st} \cdot \hat{t}(ST_h) \cdot \hat{\beta}]^2 + \lambda \cdot |\hat{\beta}| \tag{13}$$

where d is the number of observations (i.e., number of C function executions), $T_{s,k}$ is the execution time with input data set D_k, N_k^{st} is the number of executed C statements with input data set D_k, λ is a non-negative regularization parameter. The parameters β_0 and $\hat{\beta}$ are scalar values.

4 Experimental Activities

This section outlines the experimental activities used to validate the proposed processor characterization and performance estimation approach. Based on Eq. 9 and Figure 2, we developed the **SLIDE-x** (**S**ystem-**L**evel **I**nfrastructure for HW/SW **D**ataset **E-x**traction) framework to evaluate CC4CS across various processors. While implementation details are beyond the scope of this paper, the source code is freely available on GitHub[1]. All experiments were performed on a PC with an Intel® Xeon CPU E3-1225 v5 @ 3.30 GHz, 32 GB memory, and 128KB L1, 1 MB L2, and 8 MB L3 caches.

The benchmark includes 15 control- and data-dominated C functions from well-established HW/SW benchmarks [27]. Each function was tested with various data types (namely: *int8, int16, int32, int64* from stdint library, single precision IEEE 754 floating point data types) and randomly generated input files. A total of $6 * 10^4$ inputs were generated via uniform random distribution, with additional tests using $6 * 10^5$ and $6 * 10^6$ inputs showing no significant difference. The benchmark avoids function calls, recursion, external files, or library routines, and input ranges were set to prevent overflows.

CC4CS was evaluated for specific HW/SW processor technologies. For GPPs, we considered: (1) Intel 8051 CISC microcontroller[4]; (2) Microchip ATmega328/P[5], a low-power CMOS 8-bit microcontroller; and (3) LEON3[6], a 32-bit SPARC V8-compatible soft processor. The 8051 was simulated using Dalton ISS[4], Atmega328/P with SimulAVR ISS[7], and LEON3 with Cobham Gaisler TSIM ISS[6]. For SPPs, FPGA synthesis for the Xilinx Artix7 XC7A35T-1CPG236C was done using Bambu HLS[2].

4.1 Processor Characterization Results

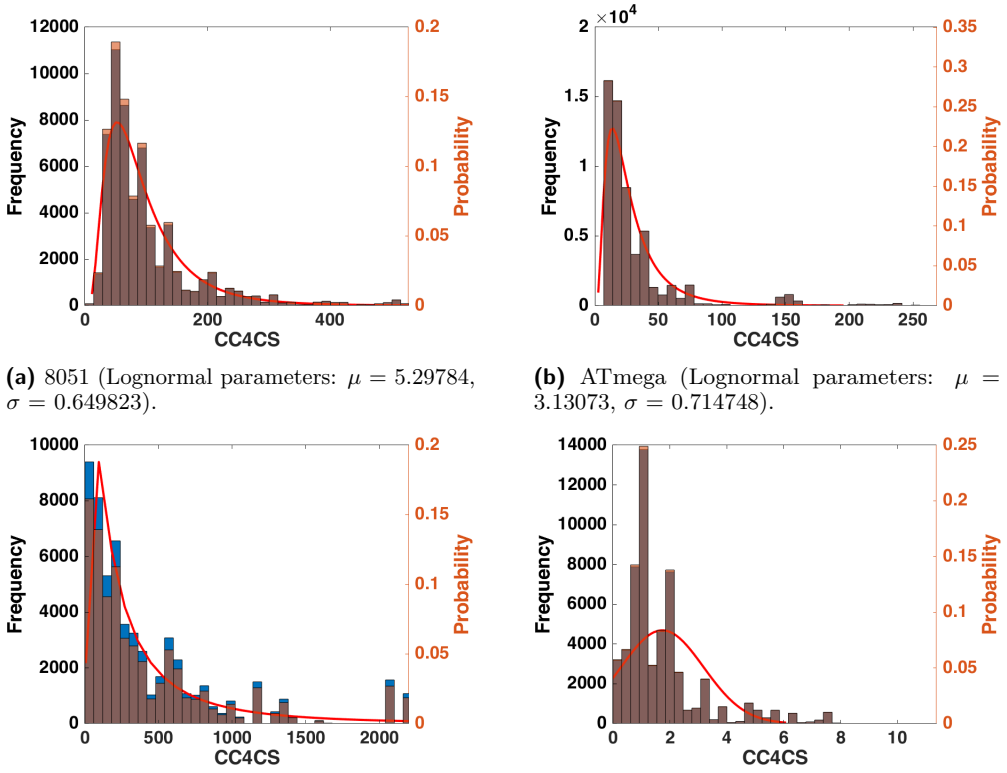
In our work, we aimed to identify which classical probability distribution best fits the empirical cumulative $CC4CS_j$ distributions obtained via the SLIDE-x framework (e.g., *Normal Gaussian*, *Lognormal*, *Beta*, *Weibull*). These distributions were evaluated using *Goodness-Of-Fit* (GOF) metrics, including *NLogN*, *BIC*, *AIC*, and *AICc*. The analysis revealed that the Lognormal distribution is best for GPPs, while the Normal distribution suits SPPs, as shown in Figure 3.

We then outlined an approach to characterize GPPs and SPPs, focusing on estimating distribution parameters (mean μ, standard deviation σ) for specific processors. To derive the values for GPPs, it has been applied the *Moment Matching Approximation* (MMA) method [33], which approximates the statistics of an empirical distribution function, with mean $\hat{\mu}$ and square mean $\hat{\mu}_2$, with a Lognormal random variable $Z = e^x$ such that $X \sim N(\mu_x, \sigma_x^2)$.

[4] U. of California, Dalton Project: `https://newit.gsu.by`
[5] M. Technology, ATMega328/P: `https://www.microchip.com`
[6] Gaisler Website: `https://www.gaisler.com/`
[7] SimulAVR: `http://savannah.nongnu.org`

$$\begin{cases} \hat{\mu} \triangleq E[Z] = E[e^x] \\ \hat{\mu}_2 \triangleq E[Z^2] = E[e^{2x}] \end{cases} \quad \begin{cases} \hat{\mu} = e^{\mu_x + \frac{1}{2}\cdot\sigma_x^2} \\ \hat{\mu}_2 = e^{2\cdot\mu_x + 2\cdot\sigma_x^2} \end{cases} \quad \begin{cases} \mu_x = log\frac{\hat{\mu}^2}{\sqrt{\hat{\mu}_2}} \\ \sigma_x^2 = log\frac{\hat{\mu}_2}{\hat{\mu}^2} \end{cases} \tag{14}$$

The μ and σ parameters for SPPs were set to the arithmetic mean $\hat{\mu}$ and standard deviation $\hat{\sigma}$ of the empirical distribution, with the fitted distribution being the Normal distribution $N(\hat{\mu}, \hat{\sigma}^2)$. These parameters are shown in Figure 3. This approach allows for the performance characterization of any processor technology using $CC4CS_j$. Such characterizations can be included in datasheets or other relevant materials and made available for further analysis.



(a) 8051 (Lognormal parameters: $\mu = 5.29784$, $\sigma = 0.649823$).

(b) ATmega (Lognormal parameters: $\mu = 3.13073$, $\sigma = 0.714748$).

(c) LEON3 (Lognormal parameters: $\mu = 5.42068$, $\sigma = 1.21775$).

(d) Bambu (Normal parameters: $\mu = 1.7326$, $\sigma = 1.4489$).

Figure 3 $CC4CS_j$ sampling distribution and fitted probability density function.

4.2 CLRA Performance Prediction Results

The predictive equations are given in Eq. 12, where τ_j represents the clock period of the HW/SW processor. We use the cumulative distribution function from Section 4.1 to estimate each function's execution time as follows:

$$\bar{t}(ST_{s,k}) = Q_2 \cdot \tau_j \quad \text{AND} \quad \delta(z_s) = 0 \quad \text{for GPPs} \tag{15}$$

$$\bar{t}(ST_{s,k}) = \mu \cdot \tau_j \quad \text{AND} \quad \delta(z_s) = 0 \quad \text{for SPPs} \tag{16}$$

Q_2 represents the median of the lognormal distribution for the 8051, Atmega328/P, and LEON3 processors, while μ is the mean of the normal distribution for the Bambu SPP. To build the CLRA model, the dataset was split into 80% for training ($48 * 10^3$ inputs) and 20% for testing ($12 * 10^3$ inputs). We then used Matlab R2022b's LASSO function with 10-fold cross-validation and the elastic net method, with alpha = 1.0.

Defining the estimation error as $\epsilon_{s,k} = T_{s,k} - \hat{T}_{s,k}$, we finally evaluate the errors for the different processors p_j using Percentage Error (PE) and Mean Absolute Percentage Error (MAPE) defined as follows [7]:

$$PE_j = \left(\frac{1}{n * d} \sum_{i=1}^{n} \sum_{k=1}^{d} \frac{\epsilon_{s,k}}{T_{s,k}} \right) \cdot 100, \quad MAPE_j = \left(\frac{1}{n * d} \sum_{i=1}^{n} \sum_{k=1}^{d} \frac{|\epsilon_{s,k}|}{T_{s,k}} \right) \cdot 100 \tag{17}$$

In this work, we have used the R^2 measure of goodness of fit metric and defined an additional reliability metric as follows:

$$REL_j = \frac{2 \cdot \sum_{s=1}^{n} \sum_{k=1}^{d} \sum_{r=k+1}^{d} \mu_{s,k,r}}{n \cdot d \cdot (d-1)}, \text{ where } \mu_{s,k,r} = \begin{cases} 0 & \text{if } (\hat{T}_k > \hat{T}_r \text{ and } T_k < T_r) \\ & \text{or if } (\hat{T}_k < \hat{T}_r \text{ and } T_k > T_r) \\ 1 & \text{otherwise} \end{cases}$$

Table 2 shows Pearson correlation and slope values between clock cycles and executed C statements. Correlations for 8051 and ATmega328/P are lower (< 0.9), while LEON3 is close to 1. The slope indicates estimation uncertainty increases with input data bits for Atmega and 8051 but remains stable for LEON3 and Artix-7. The table also shows that 8051 performs worst with float data types due to the lack of an FPU, while Bambu has the lowest correlation ($< 50\%$).

Table 2 CC4CS HW/SW Statistical Analysis results (p-value $\ll 0.001$ for every value).

p_j	Data Type Corr.[1]				Data Type Slope[3]			
	int8	**int16**	**int32**	**float**	**int8**	**int16**	**int32**	**float**
LEON3	0.993	0.919	0.9280	0.973	341.086	335.759	343.400	335.705
ATmega	0.849	0.905	0.976	0.934	8.633	10.755	14.582	24.624
8051	0.994	0.987	0.928	0.747	85.829	106.111	129.371	247.771
Artix7	0.424	0.372	0.362	0.408	2.250	2.300	2.289	3.273

a) [1] Corr.: Pearson Correlation; [2] Slope: Regression Slope Parameter;

Table 3 shows the results from the CLRA approach. Generally, the table reports high reliability and R^2 values for most input types, with some exceptions due to underfitting caused by data inconsistency or imbalance (e.g., Atmega int32 or Bambu int8). Despite lower reliability and R^2 compared to other SW processors, LEON3 has the lowest mean error in PE (from -1.37% to 6.08%) and MAPE (from 0.34% to 2.58%) due to caches and pipelines creating a stronger linear link between executed statements and clock cycles. Atmega shows the largest errors and p-values, while 8051 has smaller errors due to its CISC architecture, which has a more linear dependency between statements and clock cycles. Artix 7 shows a smaller PE range (from -2.12% to 0.36%) but higher MAPE (from 6.54% to 11.84%) and stronger R^2 and reliability compared to SW processors. This is because synthesis in HW depends on data size rather than input values. Despite Bambu's low correlation and higher errors for 8-bit types, the approach performs well for HW processors, with errors consistently below 10% for data types larger than 8 bits. Based on the works listed in Table 1, our approach consistently outperforms other works for SW processors like the 8051 and LEON3. For HW processors, other techniques may give better results but at the cost of longer execution times and greater resource demands [17]. Our approach, leveraging the CC4CS metric, provides accurate estimations for both HW and SW technologies at the system level. Errors are always below 10% for 8-bit CISC and 32-bit RISC processors with caches and pipelines (e.g., 67% more accurate than [14] for LEON3 with -O0 flag). Our approach also eliminates the need to compile/run target code for each architecture, using

a linear LASSO model to represent processor behavior. Although computing the CC4CS metric for new processors can take hours, once completed, estimation time for new data is minimal. Solving the CLRA optimization takes around 1 minute, with execution time estimation negligible.

Table 3 CLRA Performance results across various data type sizes and architectural targets. H_0 : MAPE $\geq$ 10%.

Target	Intel MCS51					AVR Atmega328/P				
Metrics	**int8**	**int16**	**int32**	**float**	**AVG**	**int8**	**int16**	**int32**	**float**	**AVG**
PE (%)	-16.03	6.06	8.16	-6.10	-1.98	-2.97	-1.39	-14.04	-2.82	-5.30
MAPE (%)	5.13	6.53	6.88	4.26	5.70	9.10	12.15	22.04	6.99	12.57
p-value	0.0211	0.0397	0.0161	2.7E-04	2.7E-06	0.9041	0.9622	0.9889	0.8598	0.9994
Rel	0.925	0.924	0.931	0.925	0.926	0.929	0.930	0.935	0.937	0.933
R^2	0.969	0.963	0.981	0.982	0.974	0.975	0.980	0.986	0.989	0.982
Target	**Sparc-V8 LEON3**					**Xilinx Artix7 (XC7A35T)**				
Metrics	**int8**	**int16**	**int32**	**float**	**AVG**	**int8**	**int16**	**int32**	**float**	**AVG**
PE (%)	6.08	0.18	3.90	-1.37	2.20	-2.12	-0.15	-0.19	0.36	-0.52
MAPE (%)	1.40	2.34	2.58	0.34	1.66	11.84	6.54	7.09	6.65	8.03
p-value	7.2E-07	1.6E-05	2.1E-05	8.9E-20	1.1E-22	0.727	8.6E-04	0.001	9.3E-04	0.0033
Rel	0.922	0.923	0.919	0.935	0.925	0.939	0.937	0.940	0.946	0.940
R^2	0.892	0.906	0.930	0.960	0.922	0.985	0.999	0.999	0.999	0.996

While extracting the initial dataset is time-consuming, our approach greatly reduces prediction times compared to Bambu, lowering prediction errors after data collection and training. The overall speed-up reaches up to 32x ($\approx$ 97%). In comparison, state of the art report a 17% reduction using NNs [15] and 43.78% with RFs [28], both lower than the LASSO model's speed-up. Thus, our approach offers a significant speed-up over traditional HLS methods. For SW processors, the key advantage is model portability and the ability to evaluate performance across various inputs and code complexities.

5 Threats to validity

The *internal validity* may be influenced by how the CLRA model is trained, as MAPE values in Table 3 are based on a dataset where 80% was used for training and 20% for testing. This could reduce the observed error. In the future, we plan to introduce a control group for training. For *external validity*, the main concern is generalizability. Results may vary with different HW/SW environments or workloads outside the training set. To address this, we validated the approach using well-known benchmarks, though future work may need to include more diverse inputs and benchmarks. *Construct validity* may be affected by the characteristics of the selected processors and FPGA (Intel 8051, Atmega328p, LEON3, Artix7). Factors like cache, virtual memory, and external memory could influence performance, and our small set of simple HW limits the generalization of our claims. Additionally, our assumption that CC4CS values follow lognormal and normal distributions may not hold for more complex platforms. *Conclusion validity* concerns the reliability of our findings. We used appropriate statistical tests to avoid biases and errors, and we have made our repository available to reproduce and validate our work[1].

6 Conclusion and Future Work

This work presents a system-level performance estimation approach using the CC4CS, a unified HW/SW metric for early performance estimations. The paper formalizes this metric and proposes an estimator based on statistical analysis. Experiments validate the approach, showing effectiveness with an estimation error below 10% and an estimation time close to 0. As shown in Table 1, our method enables system-level estimation without compiling and running the code on each architecture. It uses statistical analysis and linear regression to model different HW/SW processors. While calculating the CC4CS metric for a new processor may take hours, the subsequent estimates for new C functions are immediate. Furthermore, the estimator provides reliable predictions of execution times with limited error. Future work will (1) increase the amount of data extracted through also the usage of advanced observability mechanisms [32] and generate models for common compiler configurations; (2) integrate more HLS tools and ISSs (e.g., RISC-V, ARM, Vitis HLS), targeting various FPGA families [23], heterogeneous targets [31], and SW processors with advanced micro-architectural features (e.g., pipelines); (3) enrich the reference benchmark by considering different sources [9] across various application domains [12, 30]; (4) improving the exploitation of non-linear ML models (e.g., SVM, Regression Trees, Random Forest, Neural Networks) [19]; (5) extend the approach with additional statistics (e.g., Kolmogorov-Smirnov tests, ANOVA, t-tests) [18].

References

1 Vikas Agrawal, Anand Pande, and Mahesh M. Mehendale. High level synthesis of multi-precision data flow graphs. In *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, pages 411–416, 2001. doi:10.1109/ICVD.2001.902693.

2 Abderaouf Nassim Amalou, Elisa Fromont, and Isabelle Puaut. Fast and accurate context-aware basic block timing prediction using transformers. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, CC 2024, pages 227–237, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3640537.3641572.

3 Yunsheng Bai, Atefeh Sohrabizadeh, Zongyue Qin, Ziniu Hu, Yizhou Sun, and Jason Cong. Towards a comprehensive benchmark for high-level synthesis targeted to fpgas. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2024. Curran Associates Inc.

4 Carlo Brandolese, William Fornaciari, Luigi Pomante, Fabio Salice, and Donatella Sciuto. Affinity-driven system design exploration for heterogeneous multiprocessor soc. *IEEE Transactions on Computers*, 55(5):508–519, May 2006. doi:10.1109/TC.2006.66.

5 Carlo Brandolese, William Fornaciari, Fabio Salice, and Donatella Sciuto. Source-level execution time estimation of c programs. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, pages 98–103, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/371636.371694.

6 Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1950413.1950423.

7 Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline F.Y. Young, and Zhiru Zhang. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 129–132, 2018. doi:10.1109/FCCM.2018.00029.

8 P. Derler, E. A. Lee, and A. Sangiovanni Vincentelli. Modeling cyber–physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.

9 Tania Di Mascio, Luigi Laura, and Marco Temperini. A framework for personalized competitive programming training. In *2018 17th International Conference on Information Technology Based Higher Education and Training (ITHET)*, pages 1–8, 2018. doi:10.1109/ITHET.2018.8424620.

10 Daniele Di Pompeo, Emilio Incerto, Vittoriano Muttillo, Luigi Pomante, and Giacomo Valenete. An efficient performance-driven approach for hw/sw co-design. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 323–326, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3030207.3030239.

11 Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009. doi:10.1109/MC.2009.118.

12 Paolo Giammatteo, Federico Vincenzo Fiordigigli, Luigi Pomante, Tania Di Mascio, and Federica Caruso. Age & gender classifier for edge computing. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4, 2019. doi:10.1109/MECO.2019.8760160.

13 José Andrés Jiménez, José Amelio Medina Merodio, and Luis Fernández Sanz. Checklists for compliance to do-178c and do-278a standards. *Computer Standards & Interfaces*, 52:41–50, 2017. doi:10.1016/j.csi.2017.01.006.

14 Marco Lattuada and Fabrizio Ferrandi. Performance modeling of embedded applications with zero architectural knowledge. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 277–286, 2010. doi:10.1145/1878961.1879010.

15 Hosein Mohammadi Makrani, Hossein Sayadi, Tinoosh Mohsenin, Setareh rafatirad, Avesta Sasan, and Houman Homayoun. Xppe: Cross-platform performance estimation of hardware accelerators using machine learning. In *ASPDAC '19*, ASPDAC '19, pages 727–732, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3287624.3288756.

16 Dimosthenis Masouros, Aggelos Ferikoglou, Georgios Zervakis, Sotirios Xydis, and Dimitrios Soudris. Late breaking results: Language-level qor modeling for high-level synthesis. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3649329.3663500.

17 Hosein Mohammadi Makrani, Farnoud Farahmand, Hossein Sayadi, Sara Bondi, Sai Manoj Pudukotai Dinakarrao, Houman Homayoun, and Setareh Rafatirad. Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 397–403, 2019. doi:10.1109/FPL.2019.00069.

18 Vittoriano Muttillo, Claudio Di Sipio, Riccardo Rubei, Luca Berardinelli, and MohammadHadi Dehghani. Towards synthetic trace generation of modeling operations using in-context learning approach. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 619–630, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3691620.3695058.

19 Vittoriano Muttillo, Paolo Giammatteo, and Vincenzo Stoico. Statement-level timing estimation for embedded system design using machine learning techniques. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '21, pages 257–264, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3427921.3450258.

20 Vittoriano Muttillo, Paolo Giammatteo, Vincenzo Stoico, and Luigi Pomante. An early-stage statement-level metric for energy characterization of embedded processors. *Microprocessors and Microsystems*, 77:103200, 2020. doi:10.1016/J.MICPRO.2020.103200.

21 Vittoriano Muttillo, Luigi Pomante, Marco Santic, and Giacomo Valente. Systemc-based co-simulation/analysis for system-level hardware/software co-design. *Computers and Electrical Engineering*, 110:108803, 2023. doi:10.1016/j.compeleceng.2023.108803.

22 Vittoriano Muttillo and Vincenzo Stoico. SLIDE-x (System-Level Infrastructure for HW/SW Dataset E-xtraction). Software, version 1.0., swhId: `swh:1:rev:0907cf39f7d023d0dc2e8307e1ffef6115d0377a` (visited on 2025-02-12). URL: `https://github.com/hepsycode/SLIDE-x`, `doi:10.4230/artifacts.22912`.

23 Vittoriano Muttillo, Vincenzo Stoico, Marco Santic, Giacomo Valente, Luigi Pomante, and Daniele Frigioni. Slide-x-ml: System-level infrastructure for dataset e-xtraction and machine learning framework for high-level synthesis estimations. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, pages 616–619, 2024. `doi:10.1109/ICCD63220.2024.00098`.

24 Vittoriano Muttillo, Giacomo Valente, Daniele Ciambrone, Vincenzo Stoico, and Luigi Pomante. Hepsycode-rt: a real-time extension for an esl hw/sw co-design methodology. In *Proceedings of the Rapido'18 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '18, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3180665.3180670`.

25 Vittoriano Muttillo, Giacomo Valente, Fabio Federici, Luigi Pomante, Marco Faccio, Carlo Tieri, and Serenella Ferri. A design methodology for soft-core platforms on fpga with smp linux, openmp support, and distributed hardware profiling system. *Eurasip Journal on Embedded Systems*, 2016(1), 2017. `doi:10.1186/s13639-016-0051-9`.

26 Vittoriano Muttillo, Giacomo Valente, Luigi Pomante, Vincenzo Stoico, Fausto D'Antonio, and Fabio Salice. Cc4cs: An off-the-shelf unifying statement-level performance metric for hw/sw technologies. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 119–122, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3185768.3186291`.

27 Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, October 2016. `doi:10.1109/TCAD.2015.2513673`.

28 Kenneth O'Neal, Mitch Liu, Hans Tang, Amin Kalantar, Kennen DeRenard, and Philip Brisk. Hlspredict: Cross platform performance prediction for fpga high-level synthesis. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018. `doi:10.1145/3240765.3240816`.

29 Jürgen Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 2012. `doi:10.1109/JPROC.2011.2182009`.

30 Walter Tiberti, Federica Caruso, Luigi Pomante, Marco Pugliese, Marco Santic, and Fortunato Santucci. Development of an extended topology-based lightweight cryptographic scheme for ieee 802.15.4 wireless sensor networks. *International Journal of Distributed Sensor Networks*, 16(10):1550147720951673, 2020. `doi:10.1177/1550147720951673`.

31 Giacomo Valente, Gianluca Brilli, Tania Di Mascio, Alessandro Capotondi, Paolo Burgio, Paolo Valente, and Andrea Marongiu. Fine-grained qos control via tightly-coupled bandwidth monitoring and regulation for fpga-based heterogeneous socs. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–15, 2024. `doi:10.1109/TPDS.2024.3513416`.

32 Giacomo Valente, Tiziana Fanni, Carlo Sau, Tania Di Mascio, Luigi Pomante, and Francesca Palumbo. A composable monitoring system for heterogeneous embedded platforms. *ACM Trans. Embed. Comput. Syst.*, 20(5), July 2021. `doi:10.1145/3461647`.

33 R. Valentini, P.D. Marco, R. Alesii, and F. Santucci. Cross-layer analysis of multi-static rfid systems exploiting capture diversity. *IEEE Transactions on Communications*, 69(10):6620–6632, 2021. `doi:10.1109/TCOMM.2021.3096541`.

34 Weiyan Zhang, Mehran Goli, and Rolf Drechsler. Early performance estimation of embedded software on risc-v processor using linear regression. In *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 20–25, 2022. `doi:10.1109/DDECS54261.2022.9770144`.

Towards Studying the Effect of Compiler Optimizations and Software Randomization on GPU Reliability

Pau López Castillón
Universitat Politècnica de Barcelona (UPC), Spain
Barcelona Supercomputing Center (BSC), Spain

Xavier Caricchio Hernández
Universitat Politècnica de Barcelona (UPC), Spain
Barcelona Supercomputing Center (BSC), Spain

Leonidas Kosmidis
Barcelona Supercomputing Center (BSC), Spain
Universitat Politècnica de Barcelona (UPC), Spain

Abstract

The evolution of Graphics Processing Unit (GPU) compilers has facilitated the support for general-purpose programming languages across various architectures. The NVIDIA CUDA Compiler (NVCC) employs multiple compilation levels prior to generating machine code, implementing intricate optimizations to enhance performance. These optimizations influence the manner in which software is mapped to the underlying hardware, which can also impact GPU reliability.

TASA is a source-to-source code randomization tool designed to alter the mapping of software onto the underlying hardware. It achieves this by generating random permutations of variable and function declarations, thereby introducing random padding between declarations of different types and modifying the program memory layout. Since this modifies their location in the memory, it also modifies their cache placement, affecting both their execution time (due to the different conflicts between them, which result in a different amount of cache misses in every execution), as well as their lifetime in the cache.

In this work, which is part of the HiPEAC Student Challenge 2025, we first examine the reproducibility of a subset of data presented in the ACM TACO paper "Assessing the Impact of Compiler Optimizations on GPU Reliability" [10], and second we extend it by combining it with our proposal of software randomization. The paper indicates that the -O3 optimization flag facilitates an increased workload before failures occur within the application. By employing TASA, we investigate the impact of GPU randomization on reliability and performance metrics.

By reproducing the results of the paper on a different GPU platform, we observe the same trend as reported in the original publication. Moreover, our preliminary results with the application of software randomization show in several cases an improved Mean Waiting Before Failure (MWBF) compared to the original source code.

2012 ACM Subject Classification General and reference → Reliability; Software and its engineering → Compilers; Computing methodologies → Graphics processors

Keywords and phrases Graphics processing units, reliability, software randomization, error rate

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2025.4

Funding This work was supported by the ESA funded project "Open Source Software Randomisation Framework for Probabilistic WCET Prediction and Security on (multicore) CPUs, GPUs and Accelerators" as well as European Commission's METASAT Horizon Europe project (grant agreement 101082622). Moreover, it was also partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under the grant IJC2020-045931-I.

16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2025).
Editors: Daniele Cattaneo, Maria Fazio, Leonidas Kosmidis, and Gabriele Morabito; Article No. 4; pp. 4:1–4:10
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

With the latest advancements in GPU architectures and their compilers, there are many optimization opportunities, which allow the mapping of complex general purpose GPU (GPGPU) code on these architectures in a significantly different manner. This results not only in a performance difference, but also in different reliability properties [10].

Fernando Fernandes Dos Santo et al. in their ACM TACO article "Assessing the Impact of Compiler Optimizations on GPUs Reliability" [10] demonstrated how different compiler optimization flags impact the final reliability of GPU applications and used GPU fault injection to estimate the MWBF (Mean Work Between Failures) value.

Despite knowing that applications compiled with -O3 may have an elevated risk of encountering critical errors due to the higher optimization levels that eliminate redundant or non-executed code, the substantial reduction in execution time – resulting in enhanced performance – allows for a greater volume of work to be accomplished prior to such errors occurring.

On the other hand, TASA [5] is a source-to-source code randomization compiler designed to alter the mapping of software to the underlying hardware without altering its functionality. However, unlike compiler optimizations, TASA preserves the same number of instructions. The primary goal of TASA's development was to streamline the Worst Case Execution Time (WCET) computation for Critical Real Time Embedded Systems. This analysis is often resource-intensive, particularly when performed in a static manner through abstract interpretation and cache analysis, in which all possible execution paths are considered in order to determine whether each memory access will hit, miss or may miss in the cache. Such analysis is very sensitive to the memory layout and in fact requires knowing the exact memory layout of the program.

Instead, TASA randomizes the memory layout for each execution, allowing the use of Measurement Based Probabilistic Timing analysis for the probabilistic WCET (pWCET) estimation. In particular, TASA facilitates the computation of the pWCET by collecting a series of execution times, which are then processed with a statistical method known as Extreme Value Theory (EVT), which can estimate the maximum of a probability distribution.

The randomization of the memory layout creates different mappings in the cache with different conflicts among the contents of the cache lines, which subsequently randomizes the program's execution time.

This is the main reason why analyzing the conclusions presented in the original paper [10], we wondered how applying TASA to the original source code would impact the reliability metrics and how they would be related with the ones presented in [10].

This work has been performed in the context of the HiPEAC Student Challenge 2025. The purpose of this competition is to reproduce the experiments of a recent paper published in the ACM Transactions on Architecture and Code Optimization (TACO) journal on a different platform, and optionally improve or optimize the proposed solution of the paper.

In summary, the contributions of this work-in-progress paper is the following: a) we reproduce a subset of the GPU fault injection experiments of [10] in a different GPU platform, namely an NVIDIA 1080Ti GPU based on the Pascal architecture which was not included in the original publication. Our results follow the same trend reported in the original publication, which is that the -O3 optimization level is beneficial for GPU reliability. b) we extend the original work by repeating the same experiments under software randomization. Our preliminary results show an improved average MWBF (Mean Work Between Failures) compared to the original source code.

The mean work between failures, as its name indicates, is the total amount of work done by the executable before an error occurs. It is defined with the following equation:

$$MWBF = total\ time/errors \tag{1}$$

2 Background and Related Works

This section presents the previous works in the literature on which our proposal is based, as well as similar works.

2.1 Radiation benchmark

In Fernando Fernandez et al. [10], the authors evaluate the likelihood of encountering neutron beam radiation produced errors during the execution of a kernel on a GPU. More concretely, the study aimed on analyzing how this radiation beam would affect the final output of the application running when errors are appearing and which is the impact on the final output of the application, generating one of the following 3 outcomes: a) correct output (masked error), b) wrong output (silent data corruption) and c) critical error that makes the machine halting (Detected Unrecoverable Error).

The authors investigate this issue using two distinct methodologies: the first one involves employing a fault injection framework to simulate radiation generated errors during kernel execution, while the other one entails deploying a GPU server that is exposed to a radiation machine during kernel execution in order to obtain real life values for this radiation impact on GPU's reliability.

In order to do this, they studied the error propagation caused by radiation in multiple well-known benchmarks using different optimization flags. To achieve this, two different methodologies were used.

The fault injection experiments involved using Nvidia's NVBitFI tool [12] to inject bit flip errors into the GPU application, in order to observe how these errors affected the application's execution depending randomly on which instruction they were injected. Specifically, this examined how many functional units (FUs) were used by the instruction, how a random error affected the output, and how different optimization flags generated varying levels of error criticality. These variations made it more likely for errors to manifest as Silent Data Corruption (SDC) or Detected Unrecoverable Errors (DUE) rather than being masked, depending on the optimization flag used or the number of FUs utilized by the GPU.

Once the impact of random bit flip on instructions was analyzed, they also evaluated it using a neutron ray beam machine that generates actual radiation-induced bit flips. This allowed to obtain the bit flip error rate in real-world scenario, by merging these results with the ones obtained previously. The paper concluded that less optimized code fail less than optimized code, but if we take into account the time spending calculating the results, optimized code tends to fail less per time unit.

With respect to the original paper, we reproduced the experiments on a Nvidia's GTX 1080 Ti card, which uses the Pascal microarchitecture as opposed to the the Kepler and Volta architecture used in [10]. Moreover, our card does not have ECC recovery errors, so we omitted the analysis of that part.

For obvious budgetary reasons, we were unable to replicate the beam experiments either, limiting our analysis exclusively to NVBitFI fault injection analysis.

2.2 Radiation Evaluation of Automotive GPUs

In addition to high performance desktop and server GPUs, a series of works from Iván Fernandez et al. has studied the radiation performance of embedded GPUs, particularly targeting the automotive domain, such as NVIDIA Xavier and NVIDIA Orin.

Given the safety critical nature of the automotive sector, automotive products require compliance with high quality manufacturing standards such as AEC-Q100 as well as functional safety standards such as ISO 26262.

Both NVIDIA Xavier and NVIDIA Orin have been certified for ISO 26262 safety standards from TÜV SÜD for the highest automotive assurance level (ASIL D).

In order to achieve these certification, these architectures include several reliability features such as ECC protection not only in the DRAM, as it is the case of GPUs targeting the supercomputer market, but also in almost any hardware structure, e.g. in the caches, communication interfaces etc.

In [9] the authors have evaluated the NVIDIA Xavier under a proton beam, using the matrix multiplication benchmark from the GPU4S Bench [6] / OBPMark Kernels [11, 3] Benchmarking suite. Consistently with the safety oriented design of the NVIDIA Xavier, the authors were not able to identify any wrong output during the irradiation experiments. Therefore, the errors either were corrected by the hardware or the resulted in DUE which caused a system restart.

By exploiting the RAS (reliability and serviceability) feature of the System-on-Chip (SoC) they were able to identify for the first time in the literature the source of DUEs in the radiation testing of such complex architectures, which in this case was the tags of the cache, which were not protected with ECC.

The authors made similar observations in [8], in which the same experiments were performed in the NVIDIA Orin. Given the introduction of ECC in the tag array of the Orin, its reliability was higher.

The main difference between [10] and [9, 8] is that the first one focused on the evaluation on high performance, less reliable GPUs, using a software solution. On the other hand, [9, 8] focused on highly reliable automotive products, in which the hardware provides protection.

2.3 TASA

Kosmidis et al. [5] introduced TASA, which stands for "Toolchain-Agnostic Static Software Randomization". TASA is a source-code level static software randomization tool. This means that it works at the application source code level. It adds a random padding among memory objects and reorders them from run to run resulting in a randomized memory layout and subsequently random execution time. The original purpose of TASA was to enable a measurement based method for assessing the Worst Case Execution Time (WCET)[13] of programs for Critical Real Time Embedded Systems called Measurement Based Probabilistic Timing Analysis (MBPTA) [2].

TASA was first prototyped in a custom compiler parser as a proof of concept [5]. Later, it was re-implemented in the CIL framework[7] adding support for CUDA, but only for its C subset. Currently, TASA is being re-implemented from scratch in the Clang compiler framework within the European Space Agency (ESA) funded project "Open source software randomization framework for probabilistic WCET prediction and security on (multicore) CPUs, GPUs and Accelerators" [4].

In order to randomomize the placement of the program memory objects, instead of doing it in a low level randomization within the compiler banckend, TASA performs the randomization at source code level, by randomizing the placement of their declarations.

This is because, as we know, executables are stored in ELF (executable and linkable) format files. These files distribute code and data declarations across different program sections, which are mapped to the process memory when it is executed. This means that, by grouping these high-level declarations in the source code and distributing them based on their section in the ELF file, randomizing their order within the same section would introduce different memory placements depending on the permutations generated for each section. The reason for this is that as observed by [5], compilers place by default the program elements in the order of their declaration.

In this work in progress, we extended TASA to work with CUDA code instead of C which is supported by [5] and we support at least the constructs found in the benchmarks we use. Then we applied software randomization to the original source files in order to randomize their memory layout and be able to assess the impact of randomization on the program reliability.

3 Reliability Metrics And Evaluation Methodology

In this Section, we outline the evaluation methodology we employed, as well as the tools utilized throughout the process. As already mentioned, our primary objective was to replicate a subset of the data collected from the compiler's optimization reliability on radiation study [10] and compare it with the new data obtained from the execution of the randomization framework. The motivation for reproducing prior data stems from the use of a different GPU, which is discussed in the following section.

3.1 Devices

To conduct these experiments, we used an Nvidia GTX 1080Ti GPU, which is a desktop/server GPU. As a result, this device does not have support for ECC. Moreover, this GPU has a Pascal GPU microarchitecture and compute capability 6.1.

Moreover, as explained in Section 2.2, automotive grade GPUs like NVIDIA Xavier and Orin could not be used, since they don't exhibit the same behavior thanks to their hardware protection.

3.2 Benchmarks, compilers and flags

The GPU applications utilized for testing are a reduced subset from the original paper, due to the massive number of executions required. This decision is primarily influenced by time constraints affecting our ability to execute all benchmarks. We have chosen to evaluate the following compiler optimization flags: -O0, -O1, -O3 and -use_fast_math.

The compiler version we used is NVCC 11.3, which is the same with [10].

The selected benchmarks for execution are BFS from the Rodinia Benchmarking suite [1] and GEMM from the CUDA Toolkit. This choice is based on their fundamentally different behaviors, which allows for a more insightful comparison. One kernel is primarily memory access-bound, while the other is compute-bound, making their contrasting characteristics particularly interesting for analysis.

3.3 Fault Injection Framework

For the fault injection framework, we utilize the same one employed in [10]: NVBitFI. This framework allowed us to gather data regarding execution performance, including instances of successful execution, situations where errors occurred but the execution completed without

terminating, and cases where the kernel encountered an exception and was subsequently terminated by the system. Additionally, this framework enables us to ascertain the failure rate associated with different types of instructions.

3.4 Randomization Framework

As previously discussed, we utilized a software randomization tool to randomize the memory layout. Specifically, we employed TASA [5], which has been previously demonstrated mainly for CPU code and with a GPU application [7]. We used the Clang TASA implementation which is under development in an ESA-funded project [4]. This tool supports C and C++ code, and part of our current work involved adding support for CUDA code. It is worth noting that our current TASA Clang support for CUDA is still work in progress. As such, it is not a complete implementation, but rather an update to enable the execution of the aforementioned programs associated with the experiments. Our objective is to determine whether generating these software randomized binaries yields different results compared to those compiled using only nvcc.

3.5 Methodology Challenges

During the experimentation process, we realized that due to the timing constraints related to the deadline of the HIPEAC challenge's student competition, we could not conduct a more exhaustive experiment of TASA in the context of reliability. This limitation was primarily because the use of NVBitFI introduces a significant execution time overhead for obtaining results, which varied between 2-5 seconds per iteration. While this overhead might be considered acceptable for a 1000-iteration analysis of the original benchmarks from [10], it becomes unacceptable for TASA. Given that we applied 100 different randomizations (which is not a high number of randomizations) and 100 iterations per randomization, analyzing the 7 flags of the original study with 7 benchmarks would result in a computation time much longer than the 3 months of the Student Challenge duration. Nonetheless, we have approached this study as a demonstration of the potential for a more comprehensive analysis that could cover the entirety of the original study. Still, the results obtained meet our expectations and allow us to remain optimistic about a larger-scale study.

3.6 Methodology

In the study, two distinct experiments have been carried out. First, we decided to check whether the error rate with the error injection of NVBitFI from the original paper was correctly replicated in our architecture. To do this, we performed an error injection identical to [10], with 1000 errors per iteration and 1000 iterations. The final results show the average of these 1000 iterations.

As for TASA, with the objective described earlier of presenting the results on time, we performed 100 randomizations on the original source code, and an error injection identical to the one in the original paper, but limited to 100 iterations per randomized source code. Note that performing the original 1000 iterations would increase the total time by a factor of 10, taking 2-3 days per benchmark and flag. Once these results were obtained, we observed the average error rate of each of the TASA randomizations compared to the original source code, as well as the average MWBF compared to the original code.

The compiler version has been the same as the original paper NVCC 11.3 and we generated code for the SM 6.1 compute capability of our GPU.

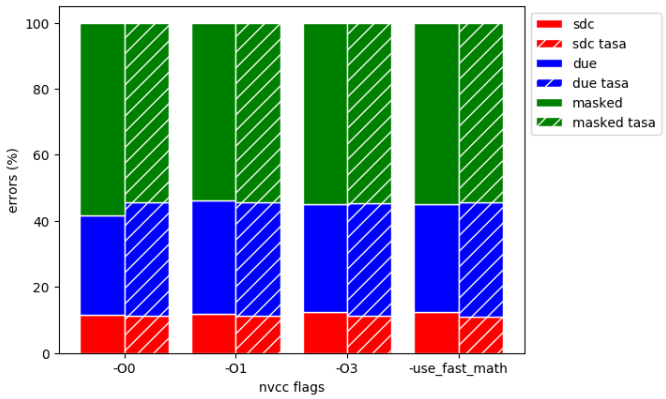


Figure 1 Plot comparing masked, SDC and DUE errors between regularly compiled code and TASA code for BFS using different compiler flags.

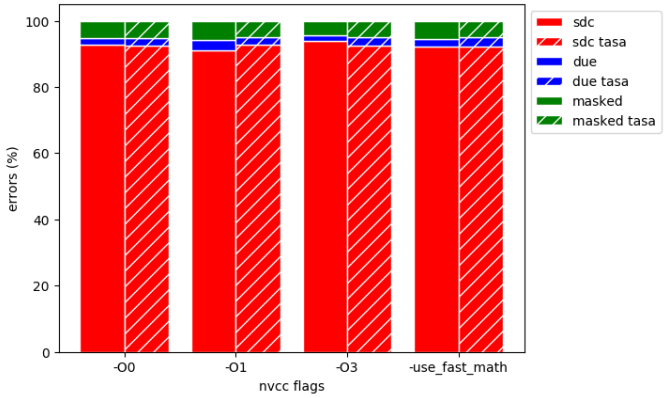


Figure 2 Plot comparing masked, SDC and DUE errors between regularly compiled code and TASA code for GEMM using different compiler flags.

4 Reliability results

Our reproduced results, shown in the left part of Figures **1** and **2** do not differ to much with the presented data from the original paper [10]. The results share a similar pattern and the slight difference could be driven by the use of the different GPU we used.

As illustrated in Figures **1** and **2**, the error patterns for BFS and GEMM differ significantly. GEMM, which is more computationally intensive, tends to experience a higher incidence of Silent Data Corruption (SDC) errors, while the occurrence of Detected Unrecoverable Errors (DUE) is comparatively lower. Conversely, BFS is characterized by a greater number of DUE failures, a trend that may be attributed to its memory-bound nature.

This difference can also be observed in Figures **3** and **4**, where the MWBF metric is presented (higher is better). In the memory bound benchmark (BFS), in which most of the time is spent in memory transfers, the -O0 flag provides better results, because of the redundant memory accesses of the non optimized code. On the other hand, in the computationally intensive benchmark (GEMM), the highest optimization level (-O3), achieves the best performance, therefore increasing the amount of work performed without an error per time unit.

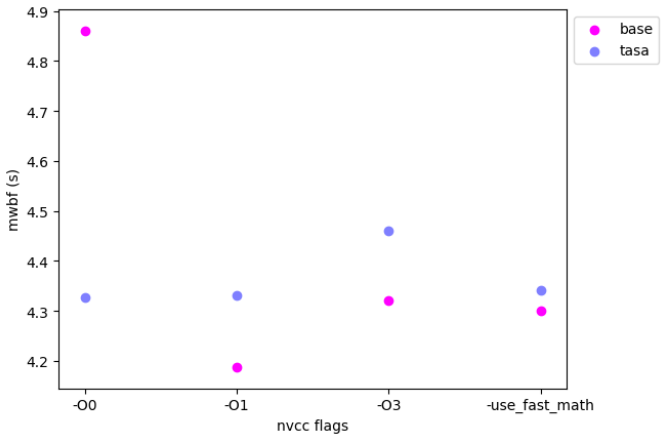


Figure 3 Difference in MWBF between regularly compiled code and TASA code for BFS using different compiler flags.

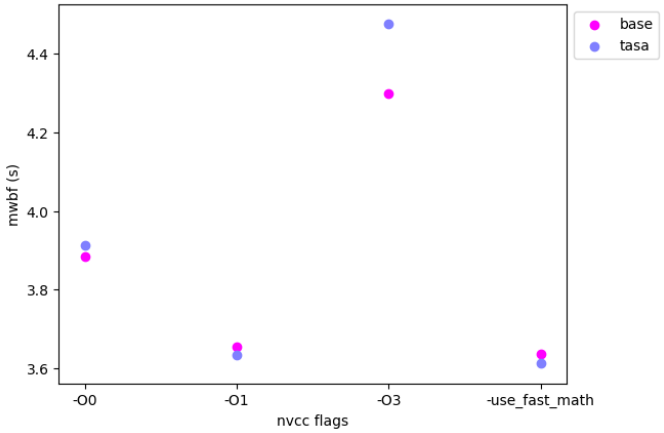


Figure 4 Difference in MWBF between regularly compiled code and TASA code for GEMM using different compiler flags.

5 Evaluation of TASA results

Figures **1** and **2** show the TASA results on the right bar of each compiler flag. Software randomized code experiences slightly less SDCs and a higher percentage of DUEs. A possible reason for this is that software randomization can expose errors that due to the memory layout were masked.

Moreover, as it can be seen in the Figures **3** and **4**, the mean of the TASA results could improve the default result. We observe that the obtained results with different flags are correlated with the ones obtained with TASA. An interesting observation is that TASA executions have better or similar results for MWBF in some configurations. This could be due to the fact that, within the normal distribution of executions produced by TASA, the average execution time is lower than that of the original source code. An increase in the number of randomizations in TASA would be necessary to justify this.

6 Conclusion

In this paper, we partially reproduced the results of [10] in a different GPU architecture, and we observed the same trends, i.e. that the -O3 compiler flag increases the MWBF metric, that is the useful work performed on average before an fault occurs.

Moreover, by introducing software randomization, our preliminary results show that the same trend is observed, and in several cases the MWBF is improved.

7 Future Work

We have to highlight that our conclusions are not decisive, as in this time-limited exercise performed in the HiPEAC Student Challenge 2025 we lacked the resources to conduct an extensive analysis with a larger dataset. As previously mentioned, we have only executed a total of 100 different seeds of the same program. This sample size may not be sufficient to assure the validity of the presented results. Also the original paper uses bigger code samples. The choice of BFS and GEMM is justified for their different behavior. Despite this, the results are pretty optimistic, so as this previous analysis has given us a good tendency, we will increase the total amount of randomizations of the original source code and the iterations per each one, and make an exhaustive analysis including all flags and benchmarks.

References

1 Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. doi:10.1109/IISWC.2009.5306797.

2 Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In Robert Davis, editor, *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 91–101. IEEE Computer Society, 2012. doi:10.1109/ECRTS.2012.31.

3 David Steenari et al. On-Board Processing Benchmarks, 2021. http://obpmark.github.io/.

4 Leonidas Kosmidis, Matina Maria Trompouki, Pau Lopez Castillon, Eric Rufart Blasco, Javier Fernandez Salgado, and Andreas Jung. Open Source Software Randomisation Framework for Probabilistic WCET Prediction on Multicore CPUs, GPUs and Accelerators. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Lecture Notes in Computer Science. Springer, 2024.

5 Leonidas Kosmidis, Roberto Vargas, David Morales, Eduardo Quiñones, Jaume Abella, and Francisco J. Cazorla. TASA: Toolchain-Agnostic Static Software Randomisation for Critical Real-time Systems. In Frank Liu, editor, *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016*, page 59. ACM, 2016. doi:10.1145/2966986.2967078.

6 Ivan Rodriguez, Leonidas Kosmidis, Jerome Lachaize, Olivier Notebaert, and David Steenari. GPU4S Bench: Design and Implementation of an Open GPU Benchmarking Suite for Space On-board Processing. Technical Report UPC-DAC-RR-CAP-2019-1, Universitat Politècnica de Catalunya, 2019. URL: https://www.ac.upc.edu/app/research-reports/public/html/research_center_index-CAP-2019,en.html.

7 Ivan Rodriguez Ferrandez, Alvaro Jover Alvarez, Matina Maria Trompouki, Leonidas Kosmidis, and Francisco J. Cazorla. Worst Case Execution Time and Power Estimation of Multicore and GPU Software: A Pedestrian Detection Use Case. *Ada Lett.*, 43(1):111–117, October 2023. doi:10.1145/3631483.3631502.

8 Ivan Rodriguez-Ferrandez, Leonidas Kosmidis, Maris Tali, David Steenari, Alex Hands, and Camille Bélanger-Champagne. Proton Evaluation of Single Event Effects in the NVIDIA GPU Orin SoM: Understanding Radiation Vulnerabilities Beyond the SoC. In *30th IEEE International Symposium on On-Line Testing and Robust System Design, IOLTS 2024, Rennes, France, July 3-5, 2024*, pages 1–7. IEEE, 2024. doi:10.1109/IOLTS60994.2024.10616076.

9 Ivan Rodriguez-Ferrandez, Maris Tali, Leonidas Kosmidis, Marta Rovituso, and David Steenari. Sources of Single Event Effects in the NVIDIA Xavier SoC Family under Proton Irradiation. In Alessandro Savino, Paolo Rech, Stefano Di Carlo, and Dimitris Gizopoulos, editors, *28th IEEE International Symposium on On-Line Testing and Robust System Design, IOLTS 2022, Torino, Italy, September 12-14, 2022*, pages 1–7. IEEE, 2022. doi:10.1109/IOLTS56730.2022.9897236.

10 Fernando Fernandes Dos Santos, Luigi Carro, Flavio Vella, and Paolo Rech. Assessing the Impact of Compiler Optimizations on GPUs Reliability. *ACM Trans. Archit. Code Optim.*, 21(2), February 2024. doi:10.1145/3638249.

11 David Steenari, Leonidas Kosmidis, Ivan Rodríguez-Ferrández, Álvaro Jover-Álvarez, and Kyra Förster. OBPMark (On-Board Processing Benchmarks) - Open Source Computational Performance Benchmarks for Space Applications. In *2nd European Workshop on On-Board Data Processing (OBDP)*, 2021. doi:10.5281/zenodo.5638577.

12 Timothy Tsai, Siva Kumar Sastry Hari, Michael Sullivan, Oreste Villa, and Stephen W. Keckler. NVBitFI: Dynamic Fault Injection for GPUs. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 284–291, 2021. doi:10.1109/DSN48987.2021.00041.

13 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389.

Evaluation of the Parallel Features of Rust for Space Systems

Alberto Perugini
Barcelona Supercomputing Center (BSC), Spain
Universitat Politècnica de Catalunya (UPC), Barcelona, Spain

Leonidas Kosmidis
Barcelona Supercomputing Center (BSC), Spain
Universitat Politècnica de Catalunya (UPC), Barcelona, Spain

Abstract

The rise in complexity of the algorithms run on space systems, largely attributable to higher resolution instruments which generate a large amount of the data to be processed, as well as to the need for increased autonomy, which relies on Neural Network inference systems in future missions, demand the adoption of more powerful on-board hardware, such as multicores.

At the same time, the correctness and reliability of critical on-board software is of paramount importance for the success of space missions. However, developing such complex software in low-level languages can have a negative impact on these aspects.

For this reason, this paper evaluates the role that the Rust programming language can have in this change, given its memory safety and built in support for parallelism, which allows to better utilise more powerful hardware, in particular multicore cpus, without compromising the programmability and safety of the code.

To this end, the GPU4S benchmarking suite, part of the open source OBPMark benchmarking suite of the European Space Agency (ESA), is ported to Rust, with sequential and parallel implementations. The performance of the ported benchmarks is compared to the existing sequential and parallel implementations in low-level languages to evaluate the trade-offs of the different solutions, and it is evaluated on several multicore platforms which are candidates for future on-board processing systems. A particular focus is put on parallel versions of the benchmarks, where Rust offers solid native support, as well as library support for fast parallelization similar to OpenMP. Finally, in terms of correctness, the Rust implementations are free of recently detected defects in the low-level implementations of the GPU4S benchmarks.

2012 ACM Subject Classification Software and its engineering → Parallel programming languages; Applied computing → Aerospace; Software and its engineering → Software safety

Keywords and phrases Rust, Multicore, Space Systems

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2025.5

Supplementary Material
Software (Source Code): `https://gitlab.bsc.es/aperugin/gpu4s_rust` [15]
archived at `swh:1:dir:57ab27fabeccfe138fadcb55e63a5bea6873de21`

Funding This work was performed within the European Commission's METASAT Horizon Europe project (grant agreement 101082622). Moreover, it was also partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under the grant IJC2020-045931-I.

1 Introduction

The development of more powerful applications for safety critical missions, together with the growth in the number of cores in commercial processors, requires the software to adapt to use more efficiently the newer platforms. Although in the past Ada – a safe language – dominated the space domain, C/C++ have taken their place, but they are older languages which lack more modern features and in particular safety. Memory safety becomes even more relevant in parallel applications, as it is easier to make mistakes in this context. Other

programming languages such as Java and Go, offer better programmability particularly in parallel code, however they are not usable in safety critical systems due to the fact that the garbage collector can cause big latency spikes, and does not allow for *Worst Case Execution Time* calculations.

In this niche domain, the Rust programming language has a real chance of being the best option. It is a memory safe language without garbage collection, which gets rid of latency spikes [9]. Instead of garbage collection, it uses the RAII (Resource Acquisition Is Initialization) paradigm [6] to insure that memory is freed. It offers modern tools and features, and has native parallelism support within the language. Performance is also one of the biggest focuses of Rust, which is important to compete with C and C++.

In this paper we analyse if the potential that Rust has in theory is carried out in practice, by porting the GPU4S benchmarks [16, 14]. These benchmarks, while they will be useful for platform performance evaluation, in the context of this paper are used instead to show a performance and programmability comparison between Rust and C, both in sequential and parallel code. Our code developed is released as open source at [15] and will be included in the next OBPMark release from ESA.

The paper is divided in 6 sections: Section 2 serves as an introduction to the GPU4S benchmarks. Section 3 presents the hosted (i.e. running on an operating system) version of the benchmarks, both in their sequential and parallel versions. Section 4 discusses the platforms on which the code was evaluated, as well as the performance comparisons and finally Section 5 presents the conclusions and future work.

2 The GPU4S benchmarks

In this section we introduce the GPU4S benchmarks suite [14, 16], also known as OBPMark Kernels, part of the open source OBPMark benchmarking suite [18] of the European Space Agency (ESA) hosted at [12], which are the applications we ported from C to Rust in this work. Next we describe their purpose, design and delve into the benchmarks functionality and relevance.

As the amount of data to be processed on board of spacecraft increases and the type of algorithms to be run expand to allow autonomous operation, the need for performance in the embedded system employed is growing significantly. The GPU4S Bench suite was developed to improve the performance testing capabilities of such systems, with a particular focus on Embedded Graphics Processing Units (GPUs) for satellite on-board processing and other safety critical systems, where existing benchmarks were particularly inadequate.

The complete list of the benchmarks available in the suite is: CIFAR 10, Convolution, Correlation, Fast fourier transform, Fast fourier transform window, Finite impulse response filter, LRN, Matrix multiplication, Max pooling, Memory bandwidth, Relu, Softmax, Wavelet transform.

The benchmarks were chosen as representative algorithms used in space-relevant applications from a survey performed among on-board software divisions of Airbus Defence and Space, Toulouse, France [16], with a look also at applications that will become important in the future, such as Computer Vision and Neural Networks. Although initially focusing on GPU benchmarking, GPU4S Bench has been later ported to several programming models including OpenMP for CPUs (used for comparison in this paper), GPUs and FPGAs, CUDA, OpenCL, HIP, Ada and others and it is used by ESA to drive the selection of hardware platforms for future space missions. To signify this platform independence, it is renamed as OBPMark Kernels, and it is used together with OBPMark applications as contractual requirement for reproducible use cases in ESA- funded projects.

Listing 1 Matrix struct in its 1D and 2D version.

```
pub struct Matrix1d<T: Number> {
    data: Vec<T>,
    rows: usize,
    cols: usize,
}

pub struct Matrix2d<T: Number> {
    data: Vec<Vec<T>>,
    rows: usize,
    cols: usize,
}
```

3 Hosted Benchmarks

This section presents the implementation of the hosted version of the benchmarks, which means that they run on top of an OS. First we define the data structures, traits and method implementations of the code to be benchmarked, defined in the `obpmark_library` crate, before delving in the code of the benchmarks executables.

3.1 Data Structures

The benchmarks from the GPU4S Bench project operate on vectors and matrices, in most cases 2D matrices. As a consequence, the main data structure required is a matrix data structure. While the C version of the benchmarks employs a dynamically allocated one-dimensional buffer for this purpose, we decide to develop two distinct versions of the data structures, to compare performance and ease of use of the two approaches: `Matrix1d`, that stores the data in a 1D Vector, and `Matrix2d`, that stores the data in a 2D Vector.

The difference in the memory disposition of the two version depends on the platform and allocator on which the code will run. As the structure will be allocated at the start of the executable, we suspect that the memory will look very similar amongst the different versions. This could make the 2D vector solution preferable, as it will be less bug prone than the 1D version, by allowing an easier use of iterators. We will look further into this when analysing the results of the benchmarks. Listing 1 shows the `struct` code.

Both structures are generic, so that they can contain any type that implements the trait `Number`, which we will discuss in detail in the following section.

When analysing the benchmark executables we will see how the `Matrix` methods will be called from the benchmarks; to allow the benchmark code to not be specialized for the different matrix types, all operations on them are defined in traits. The `BaseMatrix` trait defines basic operations on matrices:

- Initialization of a new matrix from a 2D vector representation.
- Retrieval of a 2D vector representation of the data.
- Initialization of a new matrix populated with zeroes.
- Initialization of a new matrix with random contents based on a seed.
- Input and output operations on files.
- Conversion to a 1D vector data representation, which is needed for verification.
- Reshaping the matrix by adjusting its row and column counts.

Listing 2 The base matrix trait (some code excluded).

```
pub trait BaseMatrix<T: Number> {
    fn new(data: Vec<Vec<T>>, rows: usize, cols: usize)
        -> Self;

    fn get_data(&self) -> Vec<Vec<T>>;

    fn zeroes(rows: usize, cols: usize) -> Self;

    fn from_random_seed(
        seed: u64, rows: usize, cols: usize, min: T, max: T
        ) -> Self;

    fn from_file(path: &Path, rows: usize, cols: usize)
        -> Result<Self, FileError>;

    fn to_file(&self, path: &Path)
        -> Result<(), std::io::Error>;

    fn reshape(&mut self, new_rows: usize, new_cols: usize)
        -> Result<(), Error>;

    fn to_c_format(self) -> Vec<T>;
}
```

3.2 The Number trait

The original C code provides the benchmarks with different data types that are selected at compile time. We decided that the library should not need to be compiled to a specific numeric type, but rather should be generic over the content of the matrix. This allows the user of the library to easily have matrices that contain different types in the same context, and makes the library useful also outside of the benchmarks code.

There are four types that we will need to support for the various benchmarks (not all the types are supported for all benchmarks, we will later show how to deal with this):

- The integer type `i32`.
- The single precision floating point type `f32`.
- The double precision floating point type `f64`.
- The half precision floating point type `f16`; this is not available in the standard library, so we used the crate `half` [2]. The inclusion of the f16 version of the benchmarks is for consistency with the C version, however at the moment the support for intrinsics is very limited making the code very slow. This is an area in which Rust is rapidly moving forward and we should expect the situation to be quite different in a short time after the publication of this paper. During the time of writing of this document the hardware support on x86 has changed already. For this reason we will not discuss further this feature, as it is subject to rapid change.

3.2.1 The `num_traits` crate

The `num_traits`[3] crate is used to help with definitions of common numerical behaviour in Rust programs. While external to the standard library, it enjoys widespread adoption, and many external numeric types implement the crate's traits.

Listing 3 Fundamental trait from `funty` crate.

```
pub trait Fundamental:
    'static
    + Sized
    + Send
    + Sync
    + Unpin
    + Clone
    + Copy
    + Default
    + std::str::FromStr
    + PartialEq<Self>
    + PartialOrd<Self>
    + std::fmt::Debug
    + std::fmt::Display
{
}
```

The num_traits crate offers many useful traits; here we describe the ones we used in defining our own traits:

- `NumRef`: Encompasses basic numeric operations like addition, subtraction, multiplication, and division, both for a type and its reference.
- `NumAssignRef`: Adds assignment-based operations to `NumRef` (e.g. `+=`, `*=`).
- `AsPrimitive<f64>`: Allows casting to f64,serving as a superset of all types to be implemented.
- `PrimInt`: Contains common operations on integer types.
- `Float`: Contains common operations on floating point types.

3.2.2 Limitations of num_traits

While the traits discussed are useful to define common behaviour, not all the operations that someone might want to do on a number are covered by the num_traits crate. The crate funty [1] was inspiration for some of the missing traits, in particular for its `Fundamental` one, that enforces basic behaviour such as `Copy`, `Display` and `Debug`.

Until now we described behaviour that we could anticipate we would need even before writing a single benchmark function, also because they are bundled in existing traits. The additional behaviour we describe from here on was mostly added while developing some specific benchmark. The typical process would be: 1. Do some operation you know you can do on a numeric type, 2. Have the compiler complain that it is not possible for the given bounds, 3. Add a new bound to the existing trait. This can be somewhat frustrating, especially when this is not straight forward, as in some of the cases we will describe later.

The ability to obtain a `Number` from an iterator of `T: Number` and `&T: Number` also needs to be specified as a trait bound. At this point two missing behaviours still remain:

- Obtain a `Number` from a sum operation on an iterator of elements of type `T` where `T: Number` or `&T: Number`. This is done with the following trait bounds:
  `std::iter::Sum<Self>`
  `for<'a> std::iter::Sum<&'a Self>`.

Listing 4 Fundamental trait from `funty` crate.

```
macro_rules! impl_serialize {
    ($type: ty) => {
        impl Serialize for $type {
            type Bytes = [u8; core::mem::size_of::<$type>()];
            ...
            fn to_ne_bytes(self) -> Self::Bytes {
                self.to_ne_bytes()
            }
            fn from_ne_bytes(bytes: Self::Bytes) -> Self {
                <$type>::from_ne_bytes(bytes)
            }
        }
    };
}
```

Listing 5 Integer and Float traits.

```
pub trait Float: Number + num_traits::Float {}
pub trait Integer: Number + num_traits::PrimInt {}
```

- `Serialize`: A way to serialize the numbers to their bytes representation and back; fundamental numeric types all have this behaviour available but it is not behind a trait, as such we need to implement the trait to call the method defined in the standard library. Since the method name is the same for all types, we can achieve this through a macro rule (Listing 4).
- `RngRange`: A way to generate numbers in a given range. In this case a trait does exist (`SampleUniform` from the `rand` crate), with some complications for the `f16` type.

3.2.3 More specific bounds

The `Number` trait is designed to work with all the types supported by the benchmarks. However not all benchmarks support all the different types: as an example the Softmax benchmark only works on floating point types. Another situation that sometimes happens is that the implementation is very different amongst different types. To allow for this, two more specific traits are defined, to differentiate between integer and floating point types. Listing 5 shows how this was easily achieved thanks to the `num_traits` crate.

3.3 Sequential traits

In this section, we will go through the steps required to implement the library code of a benchmark function, and we will analyze a few examples of benchmark traits.

As discussed previously, the matrix methods are defined inside traits, so that the same function can be called on different implementations of the matrix structure from the benchmark code. Each benchmark trait consists of two member functions:

- The main benchmark function, that is called from the executable and is timed to assess the performance of the hardware.
- A function that operates on a subsection of the matrix. This often is an extraction of an internal loop of the algorithm.

Listing 6 Matrix multiplication trait.

```
pub trait MatMul<T> {
    fn multiply_row(&self,
        other: &Self, result_row: &mut [T], row_idx: usize);
    fn multiply(&self, other: &Self, result: &mut Self)
        -> Result<(), Error>;
}
```

Listing 7 `multiply_row` implementation.

```
fn multiply_row(&self,
    other: &Self, result_row: &mut [T], row_idx: usize) {
    let i = row_idx;
    for j in 0..other.cols {
        let mut sum = T::zero();
        for k in 0..self.cols {
            sum += self.data[i * self.cols + k] *
                other.data[k * other.cols + j];
        }
        result_row[j] = sum;
    }
}
```

This code separation does it a little bit less readable, but it allows to reduce code duplication when other implementations, in particular the parallel versions, are coded.

For example, Listing 6 shows the trait for the matrix multiplication operation. In this case the function `multiply_row` calculates one row of the output given the two input matrices and the index of the row to calculate. The `multiply` function, on the other hand, is the one that will be benchmarked, that deals with calling `multiply_row` for each row of the result matrix.

Listing 7 shows the implementation of the `multiply_row` function for the 1D version of the matrix structure. The code is really straight forward, being the inner two loops of a matrix multiplication function, with the value of the outer loop index being passed as the argument `row_idx`.

The `multiply` function then is very easy, since all it has to do is implement the outer loop and call the `multiply_row` function with the correct arguments. We will discuss this further when analysing the parallel implementations, where the design of the functions will become more relevant.

The matrix multiplication example was chosen because representative of most of the benchmarks available, where the inner function operates on a row of the result and the calculation of each row of the output is completely independent of the others. In benchmarks such as the finite impulse response filter, where the input and output are one-dimensional, the inner loop operates on a single element of the output rather than a row, as we can see from the trait definition in Listing 9.

Another case where the design is somewhat different is where, instead of having only operations that can happen in any order, we have the need to enforce some kind of order amongst operations: this is the case for example in the softmax and the wavelet transform benchmarks, and it will become more relevant in the parallel versions of the benchmarks.

Listing 8 `multiply` implementation for `Matrix1d`.

```
fn multiply(&self, other: &Self, result: &mut Self)
    -> Result<(), Error> {
    ...
    result
        .data
        .chunks_exact_mut(self.cols)
        .enumerate()
        .for_each(|(i, result_row)|
            self.multiply_row(other, result_row, i)
        );
    Ok(())
}
```

Listing 9 Finite impulse response filter trait.

```
pub trait FirFilter<T> {
    fn fir_filter_element(&self, kernel: &Self, element_idx: usize)
        -> T;
    fn fir_filter(&self, kernel: &Self, result: &mut Self)
        -> Result<(), Error>;
}
```

3.4 Parallel versions

The parallel version of the benchmarks comes in two flavours: one that uses only the standard library, and one that uses the Rayon crate [7], which allows Rust to operate in parallel in array elements, with minor code modifications, similar to OpenMP. This way we can assess if the added complexity of dealing with data parallelism ourselves is worth it in some metric, such as performance, code maintenance, etc.

3.4.1 Using the standard library

The easiest way to parallelize the code, is to iterate over the result matrix, and spawn a thread for each row to do the computation. This would look something like the code in Listing 10. The advantages of this approach is the readability and simplicity of the code, which becomes even more evident when comparing it with the one in Listing 8; aside from the thread scope and calling the `spawn()` method, the code is identical to the sequential version.

We did not comment the `chunks_exact_mut()` method in the sequential version, because the reason for it being there is the design of the row calculation function, which is suited to parallel code. In the matrix multiplication example in Listing 7, the function modifies the result by receiving mutable slices of its rows, rather than through a mutable reference to the whole result matrix. This is not strictly necessary in the sequential version, as the calls do not overlap with each other. However it is crucial in the parallel versions, because each thread needs a mutable reference to part of the matrix simultaneously: `chunks_exact_mut()` does exactly this while making sure that there is no overlap amongst the different chunks, which makes sure that to avoid any data race between threads.

The disadvantage of the solution presented is that the number of threads spawned is not constant, rather it grows with the size of the matrix, which significantly degrades the performance of this solution if the number of rows is large.

Listing 10 Template for naive parallel implementation.

```
thread::scope(|s| {
        result
                .data
                .chunks_exact_mut(result.cols)
                .for_each(|chunk| {
                        s.spawn(move || {
                                // Do the row calculation here
                    }
                });
        });
});
```

Listing 11 Template for parallel implementation.

```
let rows_per_thread = (self.rows - 1) / n_threads + 1;

thread::scope(|s| {
    result
        .data
        .chunks_mut(result.cols * rows_per_thread)
        .for_each(|chunk| {
            let start_row = chunk_idx * rows_per_thread;
            s.spawn(move || {
                chunk.
                    chunks_exact_mut(result.cols).
                    for_each(|row| {
                        // Do the row calculation here
                    }
                }
            });
        });
});
```

To improve on this, what we need to do is spawn a proper number of threads, which is usually equal to the number of cores, or hardware threads, available on the platform, and assign multiple rows to each thread. This makes the code only slightly more complicated than the previous case, but improves significantly the performance. As shown in Listing 11, each thread has an internal loop (`chunk.chunks_exact_mut(result.cols)`, so that it can call the row function on each row of the chunk. The use of the method `chunks_mut` instead of the exact version in the outside loop, allows us to run the code regardless of the number of rows of the matrix, as it does not need to be a multiple of the number of threads.

3.4.2 The Softmax Benchmark

The softmax benchmark, as mentioned before, has a slightly different design than the one presented with the matrix multiplication case. This is because, before we can calculate the element of the final matrix, we need to now the sum of the exponentials of the whole matrix. For this reason the code has two parallel sections, with a synchronization in the middle so that the sum is available to the second parallel section. As shown in Listing 12, this only requires us to have two threads scopes rather than one, so that the threads are joined and the sum is calculated before the start of the second scope.

Listing 12 Parallel implementation of Softmax benchmark.

```
fn parallel_softmax(&self, result: &mut Self, n_threads: usize)
    -> Result<(), Error> {
    ...
    let rows_per_thread = (self.rows - 1) / n_threads + 1;
    let mut total_sum = T::zero();
    thread::scope(|s| {
        result
            .data
            .chunks_mut(result.cols * rows_per_thread)
            .enumerate()
            .map(|(chunk_idx, chunk)| {
                let start_row = chunk_idx * rows_per_thread;
                s.spawn(move || {
                    let mut sum = T::zero();
                    chunk.chunks_mut(self.cols).enumerate()
                        .for_each(|(i, result_row)| {
                            sum += self.softmax_row(result_row,
                                start_row + i);
                        });
                    sum
                })
            })
            .for_each(|handle| total_sum += handle.join().unwrap());
    });
    thread::scope(|s| {
        result
            .data
            .chunks_exact_mut(result.cols * rows_per_thread)
            .for_each(|chunk| {
                s.spawn(|| {
                    chunk.iter_mut().for_each(|el| {
                        *el /= total_sum;
                    });
                });
            })
    });
    Ok(())
}
```

Listing 13 Parallel matrix multiplication using Rayon.

```
fn rayon_multiply(&self, other: &Self, result: &mut Self)
    -> Result<(), Error> {
    ...
    result
        .data
        .par_chunks_exact_mut(other.cols)
        .enumerate()
        .for_each(|(i, chunk)| {
            self.multiply_row(other, chunk, i);
        });
    Ok(())
}
```

While the code presented is long, especially when compared to the rayon version as we will see, if we consider only the code specific to the benchmark, without the part to spin the threads presented in the previous section, which is the same in all benchmarks, we can see that the code is indeed very straight forward.

3.4.3 Using Rayon

Rayon is a great tool for dealing with the kind of problems we are working on. This becomes evident when comparing the rayon code with the sequential code, for example for the matrix multiplication function (Listing 13), which we included in its sequential version in Listing 8.

The only modification necessary to the code is using Rayon's parallel version of the `chunks_exact_mut()` method from standard library.

The softmax benchmark presented when discussing the standard library versions, is a good example both to show the compactness that Rayon can achieve compared to the standard library code, and to showcase some case where the parallel code is not just adding a `par_` in front of the standard library iterator. The Rayon version of the code is available in Listing 14. The second loop is the same as in the sequential version, while the first needs to use the `reduce()` method for calculating the sum. The syntax of reduce is the typical functional syntax of the method, and allows for the reduction operation itself to be performed in parallel as well, which will not be a major consideration on our target platforms, since they will have relatively few cores (4-8), but could be relevant if we had a very large number of threads. It should be noted that the order in which the reductions will happen is not specified, which means the result could be not exactly the same due to the non-associativity of floating point operations [8], which is why we should use a tolerance when validating the results.

3.4.4 Parallel traits

Just like in the sequential case, the parallel versions of the benchmarks are defined inside traits. The parallel traits are defined inside two modules: `rayon_traits` for the Rayon versions and `parallel_traits` for the standard library versions. In this case the traits only require one member, that being the function to be benchmarked, and the naming scheme used is the following: The traits have the same name as their sequential counter-part, preceded by `Rayon` or `Parallel` depending on the version. The method defined inside the trait has

Listing 14 Parallel softmax using Rayon.

```
fn rayon_softmax(&self, result: &mut Self) ->
        Result<(), Error> {
    ...
    let sum = result
        .data
        .par_chunks_mut(self.cols)
        .enumerate()
        .map(|(i, row)| self.softmax_row(row, i))
        .reduce(|| T::zero(), |partial_sum, next_sum| partial_sum
                            + next_sum);
    result.data.par_iter_mut().for_each(|el| {
        *el /= sum;
    });
    Ok(())
}
```

Listing 15 Parallel traits.

```
pub trait RayonRelu {
    fn rayon_relu(&self, result: &mut Self) -> Result<(), Error>;
}
pub trait ParallelRelu {
    fn parallel_relu(&self, result: &mut Self, n_threads: usize) ->
            Result<(), Error>;
}
```

the same name as the sequential version preceded by `rayon_` or `parallel_` depending on the version. An example of parallel traits for the rectified linear unit benchmark is shown in Listing 15.

3.5 The Benchmark Code

This section deals with the `benchmarks` crate, that contains the binaries' code and utilities used by them. Each benchmark in the project has its own executable that deals with argument parsing, initialization, timing, input/output and validation of the result. Besides the executable, in the benchmarks crate, there is a `lib.rs` file that defines some common functionalities and helper functions.

3.5.1 Type aliasing

The benchmark_utils module defined in the crate deals with some useful code for the benchmarks. Amongst its functionalities, it deals with defining some types based on arguments passed at compile time, in particular a `Number` type that is going to be the content of the matrices, and a type `Matrix`, which is the specific matrix implementation to be used. This is done using `#[cfg(feature = ...)]` directives, that work similarly to `#ifdef` directives in C.

Listing 16 shows the code needed to define the correct type for the `Number` alias. In order to pass the flag using cargo, all it takes is to pass to cargo the flag `-features "feature_name"`, and cargo will then pass the flag to rustc. There is no way yet [5] to have mutually exclusive

Listing 16 Type aliasing based on compile flag.

```
#[cfg(feature = "float")]
pub type Number = f32;
#[cfg(feature = "double")]
pub type Number = f64;
#[cfg(feature = "int")]
pub type Number = i32;
#[cfg(feature = "half")]
pub type Number = half::f16;
#[cfg(not(any(
    feature = "float",
    feature = "double",
    feature = "int",
    feature = "half"
)))]
pub type Number = f32;
```

features, however, if more than one datatype is set we will get a compile error for trying to set a type alias already defined. With the last cfg command, if no type is specified at compilation, we default to `f32`.

4 Results

4.1 Experimental Setup

The evaluation of the code performance has been carried out on both ARM and x86 hosted platforms. All the evaluated platforms are considered good candidates for upcoming high performance aerospace and avionics systems [17]. This section describes the different platforms characteristics. The platforms selection has been made to add to the evaluations of the existing implementations, however since our code does not make use of GPUs, we only use the multicore CPU on each of the selected platforms.

4.1.1 NVIDIA Jetson AGX Xavier

The NVIDIA Jetson AGX Xavier [4] is an embedded platform from NVIDIA which has 8 CPU cores as well as a GPU. The board has different power-modes (Table 1), that affect both the CPU and GPU. In particular we used power-mode 1 and power-mode 2, which maximises the multicore performance of the platform which keeps the board consumption under 15W which has been identified a thermal limit of on-board processing platforms in the GPU4S ESA-funded project [14]. The selected power modes have respectively 2 and 4 CPU cores active and they are same used in similar multicore performance evaluations [17]. The version of the platform we are using has 32 GB of LPDDR4 shared between the CPU and GPU.

The software on the platform is based on Linux Kernel version: 4.9.140 / L4T 32.3.1, Ubuntu version: Ubuntu 18.04 LTS aarch64 and GCC version: 7.5.0 (Ubuntu/Linaro 7.5.0-3ubuntu1 18.04).

Table 1 NVIDIA Jetson AGX Xavier Power Modes.

	Mode		
Property	MAXN	10W	15W
Power budget	n/a	10W	15W
Mode ID	0	1	2
Online CPU	8	2	4
CPU maximal frequency (MHz)	2265.6	1200	1200
GPU TPC	4	2	4
GPU maximal frequency (MHz)	1377	520	670

4.1.2 AMD Ryzen V1605B

The AMD Ryzen V1605B is part of the Ryzen Embedded V1000 family, which offers high performance platforms with a CPU and a Vega GPU in an SoC.

The V1605B has 4 cores, each with 2 hardware threads each, however in our case we did not see improvements by using 8 threads, so we report our results using 4 threads.

The software used on the platform is: Linux kernel: 5.4.0-42-generic, Ubuntu version: Ubuntu 18.04.5 LTS x86_64, GCC version: 7.5.0 (Ubuntu 7.5.0-3ubuntu1 18.04).

4.2 Performance Results

In this subsection we present the performance results of the Rust code compared to the C implementations. We report the result on the AMD Ryzen V1605B and the NVIDIA Jetson AGX Xavier, so that we have two different architectures. All the benchmarks in this subsection, unless differently specified, use a 4096 size and a single precision floating point type, which means that for benchmarks that operate on vectors rather than matrices (FFT, FIR FFT windowed) the number of elements is very small. The decision on which size and datatype to use is for consistency with the standard sizes defined for the GPU4S Bench/OBPMark Kernels Benchmarking Suite [12], which mainly targets existing space processors which have significantly lower performance and memory. Moreover, the same sizes have been used in multicore evaluation of the same platforms using the same benchmarks presented in [17] under the RTEMS SMP space qualified operating system in sequential C and OpenMP, which are used for comparison with our sequential and parallel Rust implementations.

The result on the Xavier in power-mode 2 (so using 4 cores in the parallel benchmarks) are shown in the graphs in Figure 1. If we compare the performance of the sequential versions in C and Rust, we see a few different cases:

- In FFT and Matrix Multiplication the performance of the two implementations is very close, in the case of Matrix Multiplication almost identical. The small differences are probably attributable to slightly different optimization between LLVM, which is used by Rust and GCC, which is used by RTEMS or different memory arrangements between the different allocators.
- In Convolution, LRN, Max Pooling, Relu and Softmax the performance of the Rust sequential versions is from 10% all the way to 130% better than that of the C version. This clearly cannot be just slight differences in the compiled code, but requires the code to use significantly different calculations. A few hints have led us to believe that the difference has to be due to the introduction of more vector operations in the Rust compiled code. In particular, while we were not able to identify the specific functionalities

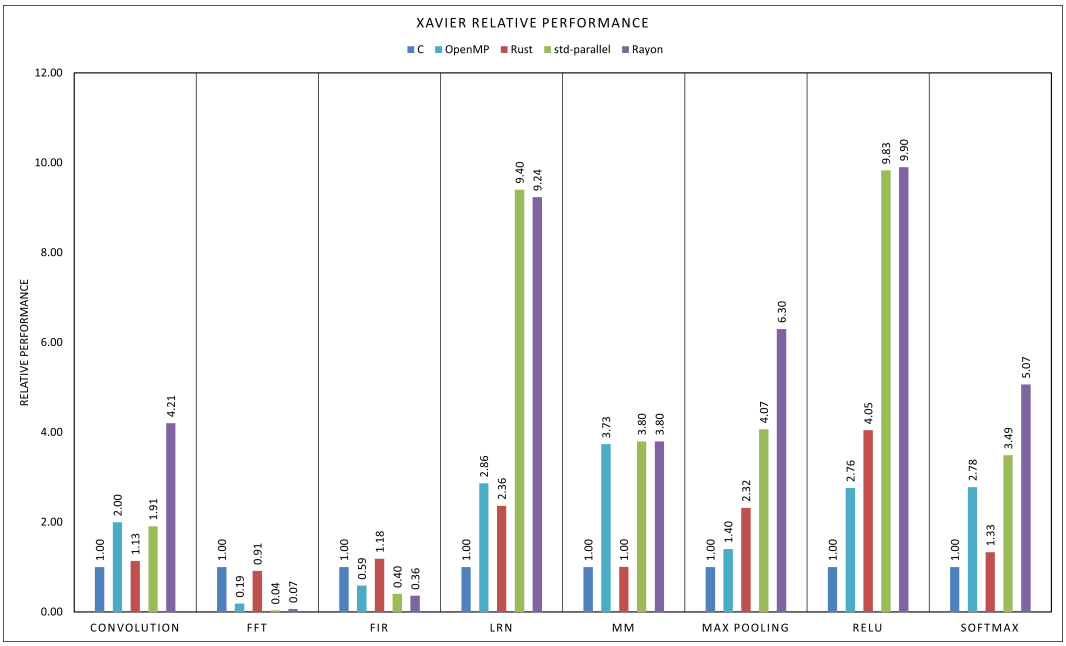


Figure 1 Performance comparison on the Xavier in power-mode 2 of the implementations of the algorithms. The results are normalised to the sequential version, which is shown as 1x.

of the instructions in the disassembled code, we saw an increase in the number of such operations in the code, as well as in some cases we see the verification to have to be with a tolerance to pass, which suggests that the floating point operations are carried out in some different order compared to the C version.
While in the case of Convolution and Softmax the performance difference is not that large, we are convinced there is a real difference in performance due to the difference in the results for the 2 different parallel implementations. We will discuss this in more detail when analysing the parallel versions.

- In FIR (and perhaps FFT) the performance difference measured is hard to quantify with confidence given the very short execution time of the benchmarks, which is in the 0.5 ms range.

If we now look at the parallel implementations, we see some interesting differences in performance. First, let's look at the short execution time benchmarks, FIR and FFT: the parallel versions in this case are slower than the sequential version, and this is true both for the Rust and the OpenMP code. This is likely due to the overhead of spinning up the different threads which is not worth the small improvement in execution time. In FFT in particular, we are not actually able to parallelize the code due to complex task dependencies which are not supported in Rayon. Similarly, the OpenMP version of the code uses homogeneous parallelism (i.e. using parallel for) instead of the OpenMP tasking model, because it provides easier certification for multicore contention in aerospace systems [17].

This means that for the parallel code we use the windowed version of the FFT benchmark as in [17], which is much easier to parallelise with homogeneous parallelism, and offers a way to get to an approximated result. This however shows promise for larger vectors, where we can see an actual speedup. The reason for the choice of the sizes is to use the same values from the testing that has been done on the C versions. It should also be noted that the windowed algorithm cannot compete with the library implementation in FFTW, which is highly optimised, making it perhaps not a good candidate for parallelization.

For Matrix Multiplication, we can see that, as in the sequential case, both Rust parallel implementations have extremely similar performance to the OpenMP version, which makes Matrix Multiplication the most consistent benchmarks between the C and Rust implementations, with a very good 3.8x speedup on 4 cores.

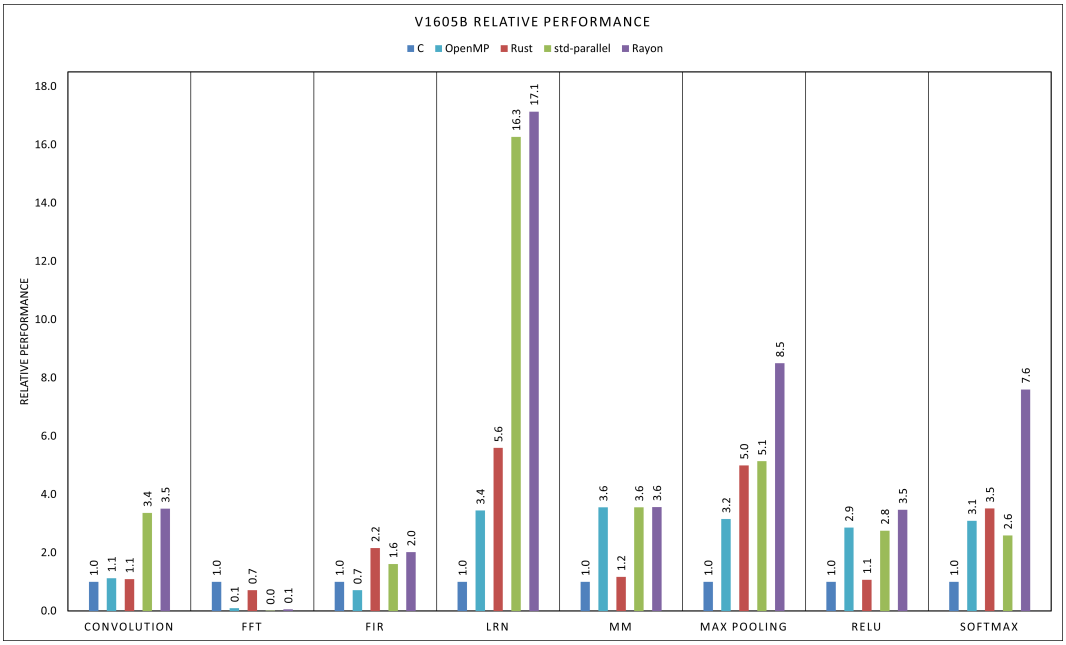


Figure 2 Performance comparison on the AMD V1605B of the implementations of the algorithms. The results are normalised to the sequential version, which is shown as 1x.

Looking at the AMD platform results we can see that in some cases the results are quite similar to the Xavier platform, while in some other we have very different relative performance. Since we are not particularly interested in the performance difference amongst the platforms we do not report the execution times. However it should be noted that the V1605B has much higher performance, with the benchmark taking usually anywhere from half to 1/5 to complete execution compared to the Xavier. The behaviour of Matrix multiplication is still very consistent amongst the different implementations, with speedups that approach the linear case both for OpenMP and the parallel Rust implementations. Similar to the Xavier, LRN, Max Pooling and Softmax show much higher sequential performance compared to the C version, in this case performing even better than the OpenMP version, once again thanks to a higher use of vector instructions. Relu, on the other hand, is quite close to the C versions in this case, both in the sequential and parallel execution, with a slight upper hand of the Rayon code. FFT is significantly slower in the parallel versions on the AMD platform too, for the same reasons discussed above, while FIR manages to have better sequential performance compared to the C code, but the parallel code is still slower than the sequential version due to the small sizes on the input.

4.2.1 Taking a closer look to the parallel implementations

As mentioned in the previous section, the two different parallel implementations of the Rust code can have quite different performance depending on the benchmark.

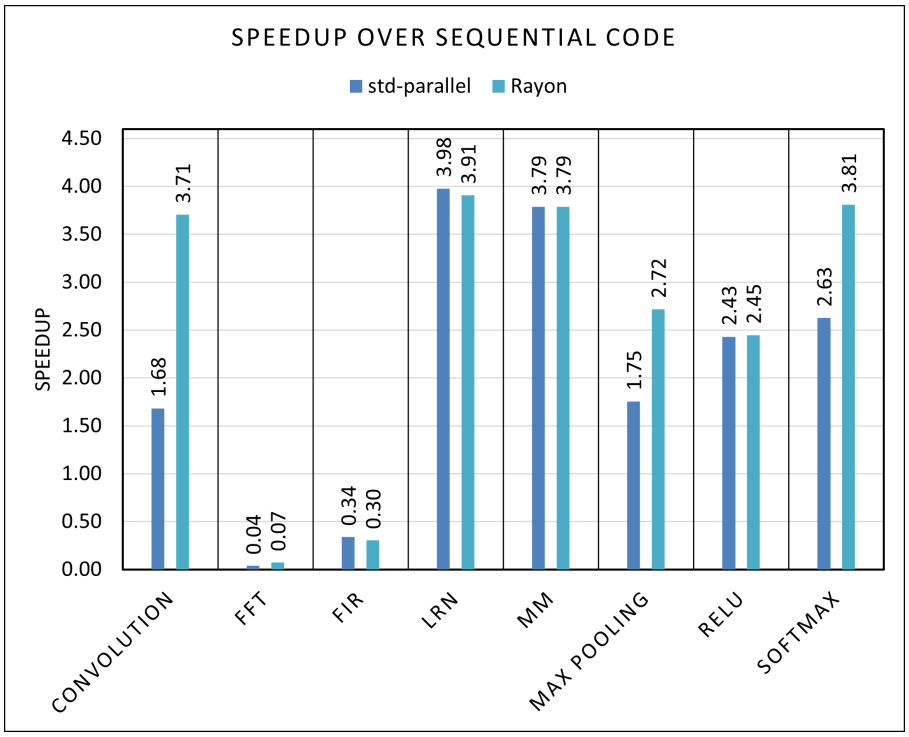


Figure 3 Comparison of the speedup of the parallel Rust implementations on the NVIDIA Xavier.

In Figure 3 we see the speedup over the sequential code of the rayon and std-parallel implementations, which seems to favor significantly Rayon, in particular where the vector instructions made the sequential code faster. Our guess to the cause of this behaviour, is that Rayon does a better job keeping the vector operations, as the library can decide, knowing the hardware features available, if it makes sense to make something parallel and to which thread assign each part of the matrix, while in the std-parallel the programmer takes this decision, which can result in suboptimal performance. This is true in particular in the Convolution and Softmax benchmarks, where the std-parallel code has similar performance to the OpenMP one, even though the Rust sequential version is faster. In LRN and Relu this is not the case, with both the std-parallel and Rayon version showing very impressive speed-ups compared to the sequential C version.

In the AMD results shown in Figure 4, we can see somewhat similar results, but here the std-parallel version of Softmax is slower than the sequential code and in Max Pooling it has a very similar performance, while on the ARM platform we could still see a speedup over the sequential code. As mentioned before, this is likely because the manual subdivision of the input and output matrices to make the parallelization possible interferes with the ability of the compiler to introduce vector instructions, while the more complex Rayon runtime is able to still utilize them. On the other hand, both Rayon and std-parallel manage to perform very well in the LRN benchmark, on both architectures.

Another difference with the performance on the Xavier is that on the V1605B the speedup does not go much higher than three, while in the Xavier case we had some cases, like LRN and Matrix Multiplication, where the performance was close to the theoretical maximum.

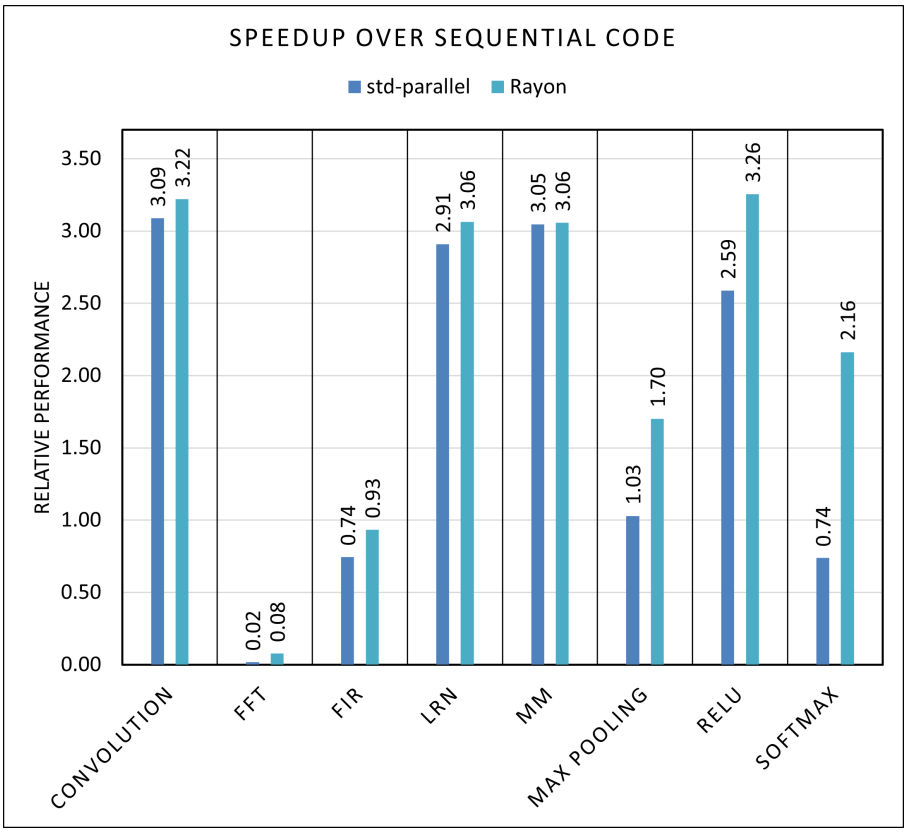


Figure 4 Comparison of the speedup of the parallel Rust implementations on the V1605B.

4.2.2 2D matrices

Until now we mostly considered the 1D version of the Matrix structure to have a better comparison with the C code, however as we mentioned, we developed also a 2D version which improves on programmability and decreases bugs when manually dealing with identifying rows.

Figure 5 shows the speedup of some 2D Matrix algorithms over their 1D counterparts, and as we can see in general there is a performance deficit by having the rows allocated independently from each other. It seems to be the case that this deficit is not the same for all benchmarks, with Softmax actually showing an improvement, though perhaps not statistically significant. In our results the difference is usually pretty small, making the 2D matrix probably preferable in situations where we don't care about very small performance differences. It should be noted though that the difference in the real world *could* be higher, if the structures are allocated during the execution rather than at the start of the program, or with more programs running at the same time, as the OS could place the different rows far away from each other, increasing the cache misses.

5 Conclusions and Future Work

The results presented in the previous section show that the performance of sequential Rust is similar to C in our space-relevant applications. The same holds for the parallel versions, with Rayon showing some of the most promising results both in terms of ease of use and

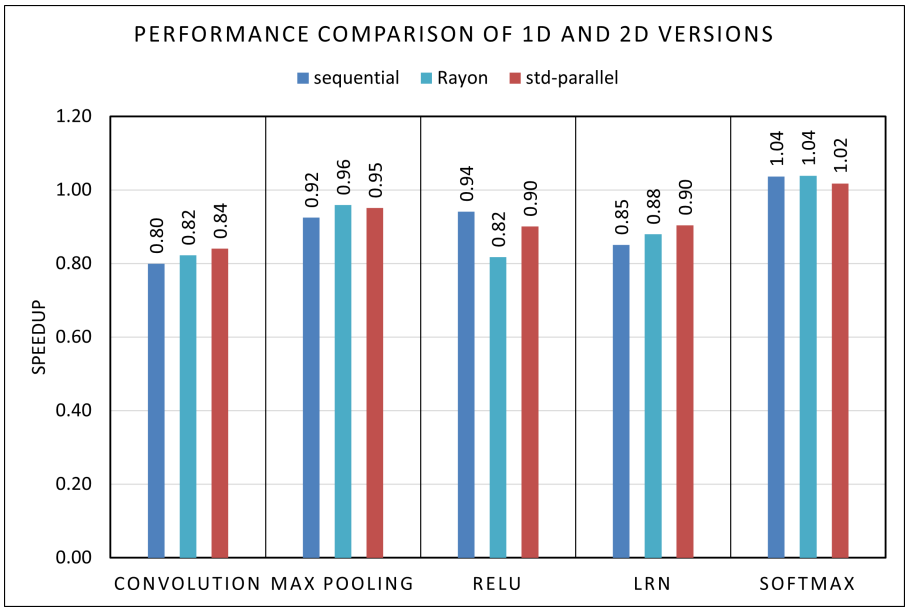


Figure 5 Speedup of the 2D matrix over the 1D version on the AMD V1605B.

performance. These findings together with the memory safety and easier development of Rust make it a promising technology in the space and other safety critical domains, as well as in embedded systems in general.

Another take away comes from the 2D version of the algorithms, that perform only slightly worse but help a lot with programmability. Throughout the development we caught bugs on the 1D version that stemmed from inadequate testing, as the benchmarks only use square matrices, due to the use of the incorrect dimension in iterators; these bugs did not happen in the 2D versions as the there is no need to manually divide the structure in rows.

Similarly, it is worth noting that during the multicore evaluation performed in our group for the [17] publication, a couple of software defects (out of bounds accesses and incomplete initialisation) were found in the C and OpenMP implementations of the FIR GPU4S benchark, which manifested with a crash only on the GR740 space processor under the RTEMS SMP real-time operating system. These latent defects were masked in all other hardware platforms and operating systems combinations. After investigating and correcting these defects in the GPU4S Bench official repository, we checked whether these defects were also present in our Rust port, as well as in the Ada SPARK versions performed in [10]. Interestingly, these defects were not present in the GPU4S Bench ports in these two safe languages [13, 11], since both languages prevent uninitialised memory and out-of-bound accesses.

Rust has a reputation of being a hard language: we would agree that the learning curve can be somewhat steep in the beginning, but in our opinion the very thing that makes Rust hard, i.e. the compile time checks, is what can make the programmer a lot more confident in the resulting code, since once it compiles we know we are not going to get any segmentation faults. This is even more the case in parallel code, where there is much lower risk of forgetting to release a lock, introducing very hard to debug errors and race conditions.

When compared with OpenMP the comparison in ease of parallel code development is less one sided, especially with the wide use of OpenMP in many applications and the larger feature set compared to Rayon. Still, as we saw in Section 3, the parallel code is very easy to obtain from the sequential one and the results are very good, making Rust a viable option.

Another area where we would like to see further development is in libraries for mathematical abstraction: we considered using some crates for mathematical operations, but did not find one that satisfied our requirements, partly due to limited support. We created the `Number` trait to deal with some of these problems, and we think a package that can help with this would be instrumental for the use of Rust in high performance parallel applications.

References

1 funty – Rust Package Registry – crates.io. URL: https://crates.io/crates/funty.

2 half – Rust Package Registry – crates.io. URL: https://crates.io/crates/half.

3 num_traits – Rust Package Registry – crates.io. URL: https://crates.io/crates/num_traits.

4 NVIDIA Jetson Xavier Series – nvidia.com. URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.

5 Pre-RFC: Cargo mutually exclusive features – internals.rust-lang.org. URL: https://internals.rust-lang.org/t/pre-rfc-cargo-mutually-exclusive-features/13182.

6 RAII – cppreference.com. URL: https://en.cppreference.com/w/cpp/language/raii.

7 rayon – Rust Package Registry – crates.io. URL: https://crates.io/crates/rayon.

8 What Every Computer Scientist Should Know About Floating-Point Arithmetic – docs.oracle.com. [Accessed 02-09-2023]. URL: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

9 Why Discord is switching from Go to Rust – discord.com. URL: https://discord.com/blog/why-discord-is-switching-from-go-to-rust.

10 Dimitris Aspetakis. Evaluation of the Ada SPARK Language Effectiveness in Graphics Processing Units for Safety Critical Systems. Bachelor's thesis, Universitat Polytècnica de Catalunya, May 2023. URL: https://upcommons.upc.edu/handle/2117/390672.

11 Dimitris Aspetakis, Leonidas Kosmidis, Matina Maria Trompouki, Jose Ruiz, and Gabor Marosy. Formal Methods for High Integrity GPU Software Development and Verification. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024.

12 D. Steenari et al. On-Board Processing Benchmarks, 2021. https://obpmark.github.io/.

13 Leonidas Kosmidis, Dimitris Aspetakis, and Matina Maria Trompouki. Formal methods for GPU Software Development Using Ada SPARK, presentation at the ESA Software Product Assurance Workshop 2023. URL: https://www.cosmos.esa.int/documents/10939403/13962862/3_updated_Leonidas_Kosmidis_2023_Software+Product+Assurance+Workshop+2023_Formal_Methods_GPUs+(1).pdf.

14 Leonidas Kosmidis, Iván Rodriguez, Alvaro Jover-Alvarez, Sergi Alcaide, Jérôme Lachaize, Olivier Notebaert, Antoine Certain, and David Steenari. GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021.

15 Alberto Perugini and Leonidas Kosmidis. GPU4S Benchmarks Rust Port Source Code Repository. swhId: swh:1:dir:57ab27fabeccfe138fadcb55e63a5bea6873de21 (visited on 2025-02-20). URL: https://gitlab.bsc.es/aperugin/gpu4s_rust.

16 Ivan Rodriguez, Leonidas Kosmidis, Jerome Lachaize, Olivier Notebaert, and David Steenari. GPU4S Bench: Design and Implementation of an Open GPU Benchmarking Suite for Space On-board Processing. Technical Report UPC-DAC-RR-CAP-2019-1, Universitat Politècnica de Catalunya, 2019. URL: https://www.ac.upc.edu/app/research-reports/public/html/research_center_index-CAP-2019,en.html.

17 Marc Solé, Jannis Wolf, Ivan Rodriguez, Alvaro Jover, Matina Maria Trompouki, and Leonidas Kosmidis. Evaluation of the Multicore Performance Capabilities of the Next Generation Flight Computers. In *Digital Avionics Systems Conference (DASC)*, 2023.

18 David Steenari, Leonidas Kosmidis, Ivan Rodríguez-Ferrández, Álvaro Jover-Álvarez, and Kyra Förster. OBPMark (On-Board Processing Benchmarks) – Open Source Computational Performance Benchmarks for Space Applications. In *2nd European Workshop on On-Board Data Processing (OBDP)*, 2021. doi:10.5281/zenodo.5638577.

HiPART: High-Performance Technology for Advanced Real-Time Systems

Sara Royuela
Barcelona Supercomputing Center, Spain

Adrian Munera
Barcelona Supercomputing Center, Spain

Chenle Yu
Barcelona Supercomputing Center, Spain

Josep Pinot
Barcelona Supercomputing Center, Spain

Abstract

Cyber-physical systems (CPS) attempt to meet real-time and safety requirements by using hypervisors that provide isolation via virtualisation and Real-Time Operating Systems that manage the concurrency of system tasks. However, the operating system's (OS) decisions may hinder the efficiency of tasks because it needs more awareness of their specific intricacies. Hence, one critical limitation to efficiently developing CPSs is the lack of tailored parallel programming models that can harness the capabilities of advanced heterogeneous architectures while meeting the requirements integral to CPSs, such as real-time behaviour and safety requirements. While conventional HPC languages, like OpenMP and CUDA, cannot accommodate critical non-functional properties, safety languages, like Rust and Ada, are limited in their capabilities to exploit complex systems efficiently. On top of that, accessibility to the programming task is essential to making the system usable to different domain experts. HiPART tackles these challenges by developing a comprehensive framework holistically addressing efficiency, interoperability, reliability, and sustainability. The HiPART framework, based on OpenMP, provides tailored support for (1) real-time behaviour and safety requirements and (2) the efficient exploitation of advanced parallel and heterogeneous processor architectures. This support is exposed to users through extensions to the OpenMP specification and its implementation in the LLVM framework, including the compiler and the OpenMP runtime library. With this framework, HiPART will contribute to realising more capable and reliable autonomous systems across various domains, from autonomous mobility to space exploration.

2012 ACM Subject Classification General and reference → Cross-computing tools and techniques; Computing methodologies → Parallel programming languages; Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Parallel architectures

Keywords and phrases Cyber-physical systems, OpenMP, Parallel and heterogeneous architectures, Efficiency, Adaptability, Interoperability, Real-time, Resilience, Reliability

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2025.6

Supplementary Material *Software (GuardianOMP Tool for task-based replication with OpenMP)*:
`https://anonymous.4open.science/r/GuardianOMP-4233/README.md` [18]
Software (LLVM for OpenMP event-based synchronization in taskgraphs):
`https://gitlab.bsc.es/ppc-bsc/software/llvm-taskgraph/-/tree/cudaGraph_RTevent` [4]
archived at `swh:1:dir:aa51e2f008a05c550f7f0f9b2359a13165b8094a`
Software (LLVM for adaptive parallelism):
`https://gitlab.bsc.es/ppc-bsc/research/c3po2024-dynamicvariants` [16]
archived at `swh:1:dir:a43d82b3d564a56f1a25347ff9a0beec81def2d5`

Funding This work is part of the HiPART project, with reference PID2023-148117NA-I00, financed by MICIU/AEI /10.13039/501100011033 and FEDER, UE.

16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2025).
Editors: Daniele Cattaneo, Maria Fazio, Leonidas Kosmidis, and Gabriele Morabito; Article No. 6; pp. 6:1–6:15

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

Acknowledgements The authors want to thank the outstanding advisory board of HiPART, which includes Dr. Raúl de la Cruz (Collins Aerospace), Mr. Marc Blanch (Technica Engineering), Mr. Franck Wartel (Airbus Defence and Space), Dr. Michael Klemm (OpenMP ARB), Mr. Roland Weigan (European Space Agency), and Dr. Dirk Ziegenbein (Robert Bosch GmbH).

1 Introduction

The demands of our rapidly evolving society and the ever-expanding scope of industrial applications urge a substantial leap forward in the autonomy and intelligence of complex Cyber-Physical Systems (CPSs), like those used in autonomous mobility and space exploration. The increasing need for High-Performance Computing (HPC) capabilities coupled with the requirements regarding Real-Time (RT) behaviour exacerbates two critical challenges in CPS' development: (1) the coordination of a potentially extensive set of tasks that often require real-time execution and significant computational resources, and (2) the complexities entailed by the parallel and heterogeneous platforms upon which CPS are commonly deployed.

Nowadays, CPSs attempt to meet real-time and safety requirements through hypervisors [13] that provide isolation via virtualisation and Real-Time Operating Systems (RTOS) [11] that manage the concurrency of tasks that increasingly encompass dynamic and resource-intensive computations (e.g., AI flows). However, current software development environments fail to enable a comprehensive analysis of the entire system, impeding the efficient utilisation of highly parallel and heterogeneous architectures.

There is an urgent need in CPS development for parallel programming models tailored to harness the parallel capabilities of advanced processors efficiently while fulfilling the non-functional requirements (NFR) that are integral to CPSs. Unfortunately, conventional programming models, like OpenMP and CUDA, do not support essential NFRs, such as real-time behaviour (e.g., task deadlines and periods, predictability, and event-based execution) and safety requirements (e.g., functional correctness and resilience).

HiPART, depicted in Figure 1, originates in the limitations of current programming systems to provide an unified computing framework equipped with mechanisms for developing, deploying, and executing complex CPSs on parallel and heterogeneous architectures. HiPART considers a holistic approach that integrates primary requirements in CPS [10]: (1) *efficiency*, optimising the amount of resources (e.g., energy and time) required to deliver the expected functionalities; (2) *interoperability*, ensuring seamless compatibility and scalability, and support heterogeneity to compose various components into a cohesive system; (3) *reliability*, operating as expected, even under challenging conditions, providing robustness, availability, and predictability; and (4) *sustainability*, enabling adaptability, resilience, and reconfigurability. The HiPART framework is based on OpenMP and its implementation in the LLVM compilation framework, integrating extensions at the levels of the programming model, the compiler and the runtime system to meet real-time constraints, event-based execution, resilience, efficiency and adaptability.

2 State of the art (SoA)

HiPART builds upon three main pillars: parallel programming models for critical real-time computation, mechanisms to boost the performance and adaptability of the evolving CPS, and mechanisms to meet reliability and resilience expectations.

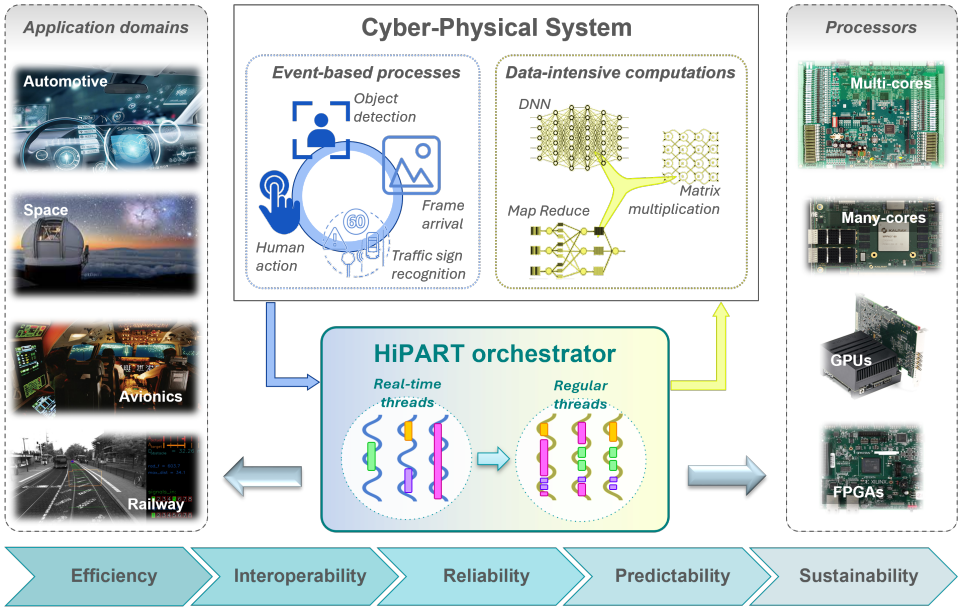


Figure 1 HiPART's overview and intended application.

Parallel programming models and real-time. A programming language is selected based on factors like the system's requirements, performance constraints, and the desired level of control. Languages like Ada prioritise safety, with Ada Ravenscar for determinism and SPARK for formal verification. Despite their merits, these languages, except for the young Rust, fail to achieve popularity. Conversely, widely used languages like C/C++, with fine-grained control for performance-critical applications, and Python, offering a high level of abstraction that simplifies task development, are embraced for their versatility. Despite their popularity and support for parallelism, these languages are either inaccessible to non-expert programmers or cannot provide the required efficiency. Contrarily, models like OpenMP and SYCL are well-adopted for their efficiency but exhibit shortcomings in supporting NFR. Nonetheless, aspects such as real-time behaviour [28, 26, 5, 14] and correctness are already considered [33, 23, 22] for extensions to the OpenMP tasking model (see Section 3).

Efficiency and adaptability in heterogeneous systems. Over the past decade, GPUs have become popular in fields like scientific computing and machine learning, where parallel tasks are frequent. Although parallel programming models like OpenMP offer high-level interfaces with competitive productivity in heterogeneous platforms [7], they may fall short in performance compared to hand-tuned applications using low-level models like CUDA [3]. Challenges arise in achieving performance portability, scalability, and adaptability. New techniques, like highly dynamic task-based parallelism and asynchronous programming, aim to streamline the development process. However, the overhead introduced by the parallel orchestrator [9, 12, 34] and the lack of features to describe adaptability opportunities may impede optimal execution in evolving environments. This is critical in rapidly evolving heterogeneous architectures where specific devices might not be available or even present failures, and the overall conditions of the system constantly vary.

Reliability and resiliency. The development of trustworthy CPSs is challenged by the intricate interaction between the computational and physical realms. Schedulability [27] and fault-tolerance are integral in CPSs, given that missing deadlines and processing errors can potentially lead to catastrophic consequences. Techniques for space redundancy, like N-modular redundancy [30] and task replication [6], and time redundancy, like application-level checkpointing [31] may enhance fault-tolerance but also compromise the schedulability of the system [1] if tasks miss their deadlines due to increased computing requirements. Furthermore, these mechanisms need to be made aware of the structure of the applications and either require error-prone processes for manually determining the checkpointed data [29] or increase overheads and memory footprint due to poorly decided checkpoints.

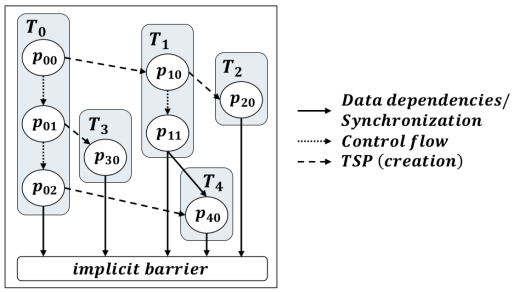
Considering the limitations mentioned above, HiPART leverages proposals and know-how from previous projects that have worked towards converging the HPC and the critical real-time domains using OpenMP, like AMPERE [19], RESPECT [21], HP4S [32], and the yet to finish RisingSTARS [24] and LIONESS [25] projects to revolutionize the landscape of complex CPS. Through extensions to OpenMP and an open-access implementation based on LLVM, HiPART will facilitate the design, deployment, and execution of real-time HPC CPS on advanced parallel and heterogeneous architectures, holistically addressing efficiency, interoperability, reliability, and sustainability.

3 The OpenMP programming model

HiPART builds on OpenMP, the de facto standard for programming shared-memory systems within the HPC community. The model defines an application programming interface (API) with compiler directives to annotate C/C++ and Fortran applications. Figure 2 shows an OpenMP task-based example, with a code snippet in Figure 2a and an extended task dependency graph (TDG) in Figure 2b describing the execution constraints among the different tasks. This example illustrates the most relevant features of OpenMP utilized in the HiPART project in the following paragraphs.

```
1  #pragma omp parallel num_threads(N)
2  #pragma omp single
3  #pragma omp task                          // T0
4  {
5    p00();
6    #pragma omp task depend(out:x)          // T1
7    {
8      p10();
9      #pragma omp task                      // T2
10     { p20(); }
11     p11();
12   }
13   p01();
14   #pragma omp task                        // T3
15   { p30(); }
16   #pragma omp task depend(inout:x)        // T4
17   { p40(); }
18 } // Implicit barrier
```

(a) Sample code.



(b) Task dependency graph (TDG).

Figure 2 OpenMP tasking example.

OpenMP implements fork-join parallelism, i.e., a program starts sequentially until it reaches a `parallel` construct (line 1) and creates a team of threads associated with the parallel region where different mechanisms can distribute work. The *thread model* defines an abstraction of user-level threads that exposes low-level architectural details for exploiting

loop-intensive applications. The *tasking model* provides a high-level abstraction to define independent regions of work, namely *tasks*. The *accelerator model* leverages the tasking model to offload tasks to accelerators and the thread model to exploit parallelism within the accelerator. Given its programmability and productivity [20] and the extensions proposed to adapt it to real-time systems (see Section 2), this work builds on the tasking model.

An OpenMP *task* (lines 3, 6, 9, 14, and 16) is an independent work unit with a block of executable code and its data environment. Tasks can be synchronized through memory fences, like the `taskwait` and `barrier` constructs, including the implicit barriers like those at the end of the single and the parallel regions (line 18), or the data-flow synchronizations defined by the `depend` clause coupled with the `in` (line 6), `out`, and `inout` modifiers (line 16). Tasks can also be nested, where each nesting level entails an isolated domain of synchronization. Task-based program commonly use the `single` construct to allow only one thread in the single region to execute the sequential code. Meanwhile, the rest of the threads wait in the implicit barrier at the end of the region until there is work to do (i.e., tasks are instantiated).

When a thread encounters a `task` construct at run-time, it creates a task instance that becomes ready when all its input dependencies are satisfied. In mainstream implementations, e.g., GCC and LLVM, ready tasks are dynamically scheduled by the runtime system to the available OpenMP threads. Furthermore, threads use work-stealing when they do not have work enqueued but other threads in the team still have work in their queues.

4 The HiPART framework: Extended OpenMP for real-time HPC

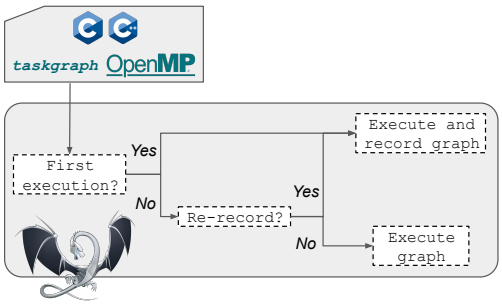
Building on the OpenMP tasking model (introduced in Section 3) and considering the limitations described in Section 2, the HiPART framework leverages and introduces extensions to the OpenMP specification and its implementation in LLVM, including the Clang frontend for processing new directives and clauses, the LLVM compiler for static analysis and code generation, and the OpenMP runtime library for runtime support, to address efficiency, interoperability, reliability, and sustainability. The reminder of this section introduces the extensions leveraged from previous projects and planned to be refined during the project (see Sections 4.1, 4.2, and 4.3) and the extensions already prototyped and preliminary tested since the beginning of the project in September 2024 (see Sections 4.4 and 4.5).

4.1 Graph-based execution

The recently released OpenMP6.0 [2] incorporates the *taskgraph* directive. This functionality implements reusable graphs of tasks to reduce the overhead of task orchestration (e.g., task creation) and minimise contention on shared resources (e.g., task ready queues). The functioning of taskgraphs is illustrated in Figure 3, with a sample code showing a `taskgraph` directive in Figure 3a and the execution flow depicted in Figure 3b. Overall, the `taskgraph` construct encloses a region of code that can be captured as a TDG. Hence, it includes task-generating constructs (e.g., `task` and `taskloop`) that are executed whenever the region is reached and other statements (e.g., control-flow statements) that will only be executed when the region is recorded. When a taskgraph region is encountered at runtime, if a graph of tasks already exists for that region, it is played. Otherwise, or if the user explicitly asks for regenerating the graph through the additional `graph_reset` clause (e.g., when the condition within the clause in line 5 resolves to true), then the system generates the TDG of the region. How the TDG is generated and executed is implementation-defined, so it can either be generated statically, at compile-time, or dynamically, at run-time. In the latter case, it can be created while executing the region or in a preprocessing step to later execute the graph.

```
#pragma omp parallel
#pragma omp single
{
  for (int i = 0; i < N; ++i) {
    #pragma omp taskgraph graph_reset(i%2==0)
    {
      #pragma omp task depend (out:a[i])
      produce_element (a[i]);
      #pragma omp task depend (in:a[i])
      consume_element (a[i]);
    }
  }
}
```

(a) Sample code.



(b) Workflow.

Figure 3 OpenMP *taskgraph* example.

The taskgraph framework has already been tested using several HPC benchmarks including `task` and `taskloop` constructs. The results show speedups of up to 6x compared to the LLVM native implementation of tasking [34]. While upstream LLVM alreay implements a *record and replay* mechanism relying on runtime routines, a prototype implementation of the `taskgraph` construct including static (compile-time) and dynamic (run-time) recording capabilities is publicly available in `https://gitlab.bsc.es/ppc-bsc/software/llvm-taskgraph`.

Graph-based computation has recently become popular as it allows reducing the overheads of task orchestration in CPUs [34] and enhancing the performance of CPU-GPU heterogeneous systems using CUDA graphs (for NVIDIA devices) or HIP graphs (for AMD devices). Leveraging the similarities between taskgraphs and CUDA and HIP graphs, an extension of the OpenMP taskgraph framework has been proposed to deploy OpenMP taskgraphs in GPUs using the GPU native graphs [35], i.e., CUDA or HIP graphs. The results show similar or better performance compared to original *target tasks* exploiting the target threading model due to the reduction of kernel offloads and an optimised orchestration of the tasks. Furthermore, the framework also shows better scalability with the number of processors.

4.2 Adaptability through function variants

Adaptability and performance portability in complex and changing heterogeneous systems cannot only be accomplished through compiler optimisations or runtime mechanisms. OpenMP includes features to define function specialisations. In the standard, different user functions can be linked together through the `declare variant` directive, which establishes different functions to implement a unique functionality and the condition that the compiler must check to statically decide which implementation is used in each function call. Although this presents a step forward, adaptability can only be obtained at compilation time. Therefore, runtime changes (e.g., a permanent failure in a device) cannot be considered.

HiPART relies on an extended interpretation of OpenMP variants that gathers metrics at run-time to dynamically decide among the set of function specialisations provided by the user [15]. This procedure allows for considering the system's dynamic conditions, like workload and energy consumption. To that end, the compiler follows the flow shown in Figure 4, producing two distinct binaries: (1) an instrumented version equipped with runtime calls that collect metrics about resource usage and (2) a version that interprets the metrics gathered by the instrumented version to guide variant selection. The second binary is generated only after the instrumented binary runs and collects the metrics.

The metrics captured during warm-up include average and peak CPU usage (%), average and peak GPU usage (%), thread stack memory consumption (%), heap memory consumption (%), GPU memory consumption (%), and execution time (ms). During the execution of

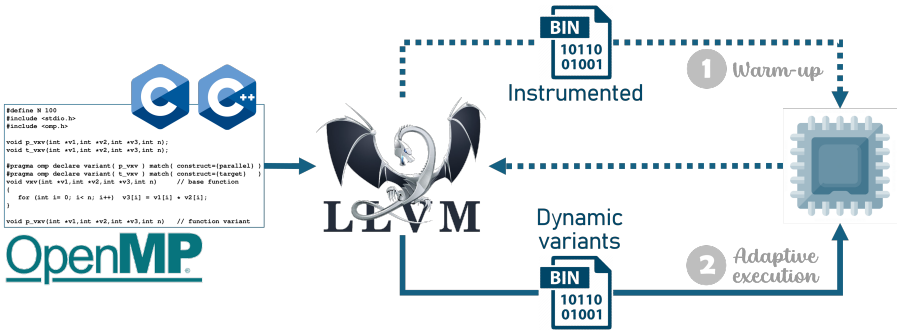


Figure 4 Workflow of the extended LLVM to support dynamically selected function variants.

the final binary, users can provide a list of metrics to guide variant selection, represented as a set of comma separated triplets of the form of `metric:threshold:weight`, where `metric` is *cpu*, *gpu*, *mem*, *stack* or *heap*, `threashold` indicates a percentage that, when reached, forces the runtime to select the variant that stresses less the corresponding metric, and `weight` is an optional parameter that indicates the weight of the metric.

The evaluation of this proposal [15] shows the capability of the system to tune the optimal number of threads for each parallel region, prevent errors or slowdowns in systems with limited memory, and effectively swap between CPU and GPU implementations when one resource is overloaded. However, the method might introduce unbearable overheads in systems with rapid fluctuations, and decisions might soon be outdated. HiPART plans to mitigate these effects with mechanisms like splitting functions. Furthermore, other planned extensions include more refined predictive models that can foresee system state changes and new metrics like power consumption.

4.3 Predictable execution via task-to-thread mapping heuristics

The dynamic nature of the OpenMP task scheduler and the use of work-stealing to balanc the workload of all threads introduce uncertainty in the execution and, although good-enough in general terms, entail certain overheads. Recent works have proposed the use of temporal conditions to derive a more efficient task-to-thread mapping able to reduce system response time and running time and providing better predictability [26]. The scheduling mechanism proposed is depicted in Figure 5. It is split into two phases: (1) *allocation*, assigning each task to a thread, and (2) *dispatching*, selecting a task from the ready task queue. A series of heuristics are proposed for each of these phases, leveraging information about the number of tasks in a thread's ready queue and their execution time, among other aspects.

An evaluation of the proposed heuristics compares to common implementations in OpenMP runtime, including breadth-first scheduler (BFS) and work-first scheduler (WFS), regarding response time and running time. The results show that the response time produced by some heuristics is lower than the default LLVM scheduler in most cases, and the variability in the results given by the heuristics is lower than that of the default scheduler.

HiPART plans to extend this work by providing a schedulability analysis of state-of-the-art mapping strategies and the suggested heuristics in relevant applications. Furthermore, the project considers extending these mechanisms to heterogeneous systems with multiple GPUs.

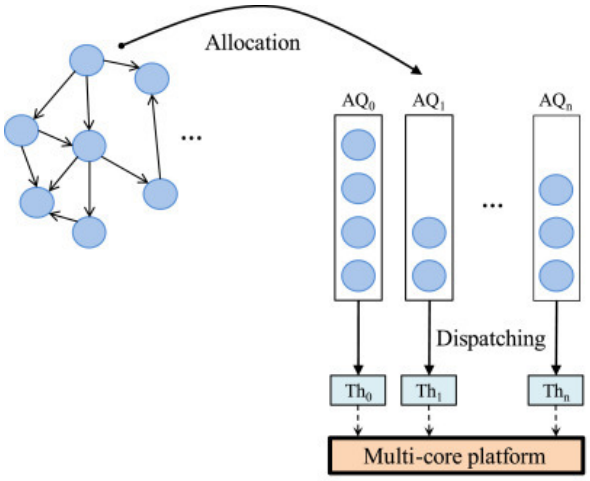


Figure 5 Task-to-thread mapping workflow.

4.4 Fault tolerance

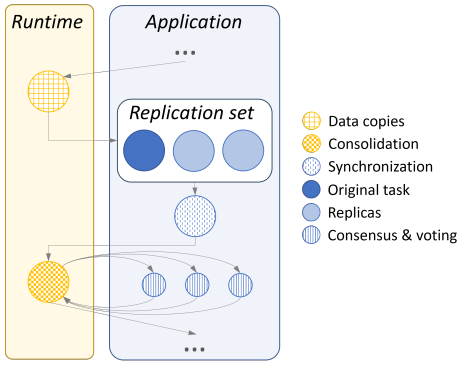
Several CPSs are sensible to transient faults due to the harm these may cause to the safety of people, the environment and the system itself. Unfortunately, models like OpenMP lack the mechanisms to provide fail-operational behaviour. Accordingly, HiPART is developing fault-tolerance techniques to enable software-based task-level replication and user-directed fault detection to mitigate the impact of transient faults.

The proposed fault-tolerance mechanism is based on an extension to OpenMP to define task replication. Figure 6 shows an example of such an extension. The sample code in Figure 6a illustrates the syntax proposed (lines 9 to 11) [17], where:

- the clause `replicated` specifies the number of replicas to create through an integer expression > 1 (3 in the example) that indicates the total number of tasks to be generated (including the original instance and the replicated instances), and the replication strategy through the `spatial`, `temporal`, or `spatial_temporal` optional keywords. *Spatial replication* forces each task to be executed on a different resource, such as a processor core or an architecture, allowing them to run in parallel; *temporal replication* ensures that tasks execute one after the other, enforcing sequential execution; and *spatial_temporal replication* combines both approaches, requiring tasks to run on different resources while also executing sequentially. If no replication type is specified, the default behaviour allows the tasks to run in parallel without restrictions, thus favouring performance.
- the clause `replica_private` defines variables that will be replicated for each task replica, i.e., each task will have its own copy of the variable. By making data private to replicas, this clause prevents data races and ensures each replica operates independently without causing inconsistencies in shared data.
- the clause `replica_firstprivate` specifies a list of variables that must be firstprivate to each replica (i.e., each task will allocate a new space and initialize its value to the value of the original variable at the time the task is instantiated). The compiler performs a shallow copy unless a shaping expression (e.g., $[size]a$, an array a of $size$ elements) is defined, in which case it performs a deep copy considering the corresponding size. This clause ensures that the replicas start with a consistent state, improving fault tolerance while maintaining independent execution for each replica.

```
1  void foo() {
2     int v = 2;
3     int size = sizeof(int)*10;
4     int *a = (int *) malloc (size);
5     #pragma omp parallel
6     #pragma omp single
7     {
8         #pragma omp task \
9             replicated(3, spatial) \
10            replica_private([size]a) \
11            replica_firstprivate(v)
12        {
13            for (int i=0; i<10; i++)
14                a[i] = v;
15        }
16    }
17 }
```

(a) Sample code.



(b) Execution flow.

Figure 6 OpenMP task replication proposal example.

The execution flow in Figure 6b illustrates the behaviour of the replication mechanism. The thread that encounters the task replicated (line 8) creates a *replication set* with three tasks: the original and two replicas. Since the spatial constraint is specified, the OpenMP runtime system prevents threads from executing multiple tasks within the same replication set. Threads can further be bound to cores through the `OMP_PROC_BIND` environment variable for the whole execution or the `proc_bind` clause, attached to a `parallel` construct.

Figure 7 shows preliminary results of the impact of using task replication in the Barcelona OpenMP Task Suite (BOTS) [8], a benchmark suite with eleven benchmarks exposing different memory profiles and CPU consumption when randomly inserting a single bitflip per execution in either memory or registers. The result of a bitflip can be a *benign fault*, when the application completes with the correct result without detecting any error, an *output error* when the application finishes normally but the output is incorrect, a *crash*, when the application terminates unexpectedly due to an internal error, or *timeout*, when the application hangs and does not complete. The results compare a bitflip in the sequential vanilla version, namely *vanilla*, with a bitflip in the replicated version, namely *GuOMP*. Considering that bitflips typically occur in the stack in applications with minimal memory usage, examples like *Floorplan*, *Alignment*, *Fib*, *Nqueens*, and *Knapsack* exhibit high tolerance to faults because the majority of the stack is not actively used. Oppositely, benchmarks like *UTS*, which uses about 1.5MB of stack memory, are more prone to crashes. On the other hand, applications with higher use of dynamic memory see more significant effects from bitflips. Still, this behaviour depends on the specific algorithms and their memory access patterns. For example, a single bitflip in the *Sort* integer array disrupts the output, as sorting algorithms depend on precise memory operations.

HiPART plans to extend the proposal for fault-tolerance in OpenMP with a user-defined consensus and voting mechanism that will provide better accuracy when comparing results from different replicas. Furthermore, replicas will be extended to exploit function variants to boost resilience in heterogeneous systems. Finally, to provide fault-recovery capabilities, HiPART plans to enable communication between the runtime system and the application through runtime routines and OpenMP data structures and offer a checkpointing mechanism to store selected memory objects. These methods combined will enable users to handle errors in the most adequate way for each application.

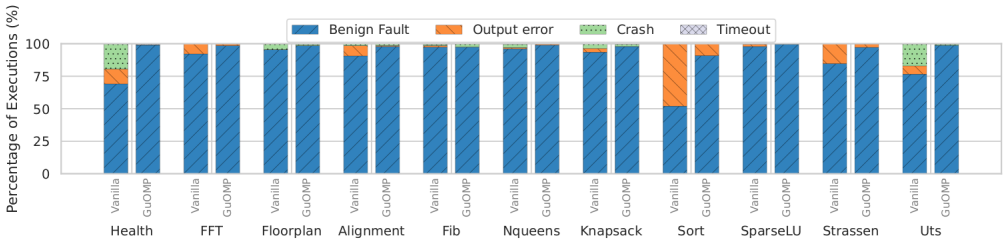
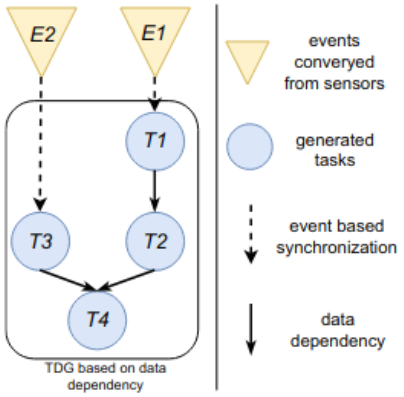


Figure 7 Comparison of the consequences of memory bitflips in BOTS with sequential vanilla and task-replicated versions.

```
1  void tdg_computation() {
2     #pragma omp target nowait \
3           depend(out:dep1) attach(event1)
4     {...} // T1
5     #pragma omp target nowait \
6           depend(in:dep1) depend(out:dep2)
7     {...} // T2
8     #pragma omp target nowait \
9           depend(out:dep3) attach(event2)
10    {...} // T3
11    #pragma omp target nowait \
12          depend(in:dep2,dep3)
13    {...} // T4
14 }
15 void notify() {
16    #pragma fulfill_event gpu_notify(event1)
17    #pragma fulfill_event gpu_notify(event2)
18 }
```

(a) Sample code.



(b) TDG.

Figure 8 OpenMP event-based synchronization proposal example.

4.5 Real-time event-based execution

The throughput-centric design of GPUs poses challenges when integrating them into time-sensitive CPS. Modern systems recently evolved to minimize overheads and interference along the critical path through advanced mechanisms, such as CUDA graphs in NVIDIA devices and HIP graphs in AMD devices. However, GPU vendors provide ecosystems specific to their products, preventing code portability. HiPART has integrated event-based synchronizations into OpenMP and extended the support for OpenMP taskgraph to CUDA/HIP graphs to notably reduce interference and overheads in time-sensitive applications.

Figure 8 shows an example of the proposal for event-based synchronization in OpenMP. Figure 8a depicts an example of generating four interdependent target tasks, two synchronized with events. The corresponding TDG is shown in Figure 8b. Upon the encountering of an `attach` clause, the implementation creates a new *allow-launch* event and connects it to the beginning of the execution of the associated task region. The generated task can only start executing its associated structured block when the *allow-launch* event is fulfilled. This will happen when another thread encounters the `fulfill_event` directive with either the `cpu_notify` or `gpu_notify` clauses taking the same event as the argument.

Preliminary experiments measure the response time and time variability of the Adaptive Optics (AO) real-time controller illustrated in Figure 9. The application combines two functionalities: (1) a pixel processing method that takes raw data from wavefront sensor cameras and processes the pixels with a series of arithmetic kernels, and (2) a series of matrix-vector multiplications that produce a command as a vector sent to the deformable mirror actuators of the physical component, typically a telescope.

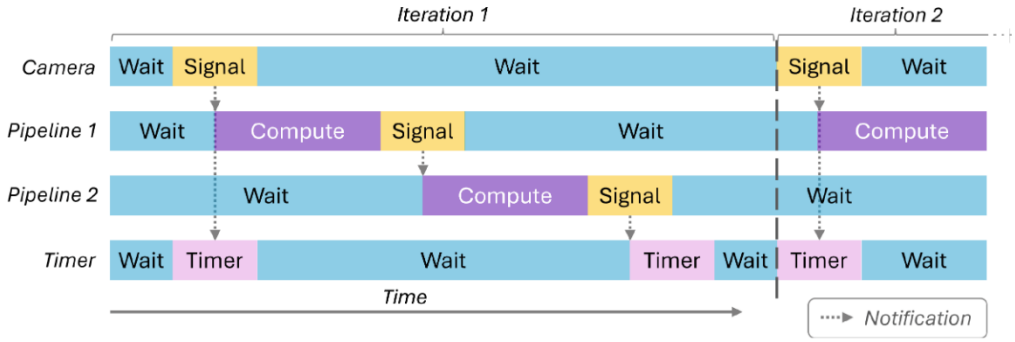


Figure 9 AO pipeline.

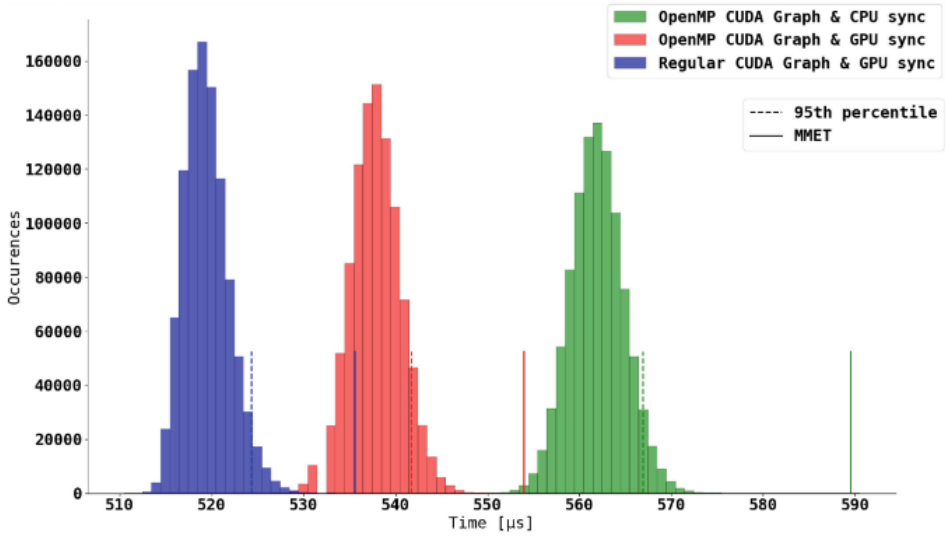
Figure 10 shows the response time of the AO application using CUDA (in Figure 10a) and HIP (in Figure 10b). In NVIDIA devices, our implementation (*OpenMP CUDA Graph & GPU sync*) achieves 16μs max. jitter, with 538μs mean execution time (MET) and 554μs maximum measured execution time (MMET). Although the response time is slightly higher in our OpenMP CUDA version than the native solution (*Regular CUDA Graph & GPU sync*), the max jitter is identical, showing comparable predictability. In AMD devices, however, the OpenMP HIP version (*OpenMP HIP Graph & GPU sync*) achieves a slightly higher max. jitter of 21μs (443μs MET and 464μs MMET). As expected, the third method using the CPU synchronization strategy showed the largest execution times and a max jitter of 28μs (562μs MET and 590μs MMET) with CUDA and 33μs (478μs MET and 511μs MMET) with HIP. All in all, our proposal delivers MET and jitter comparable to the native CUDA/HIP implementations while maintaining a simple, directive-based programming style.

5 Project plan and next steps

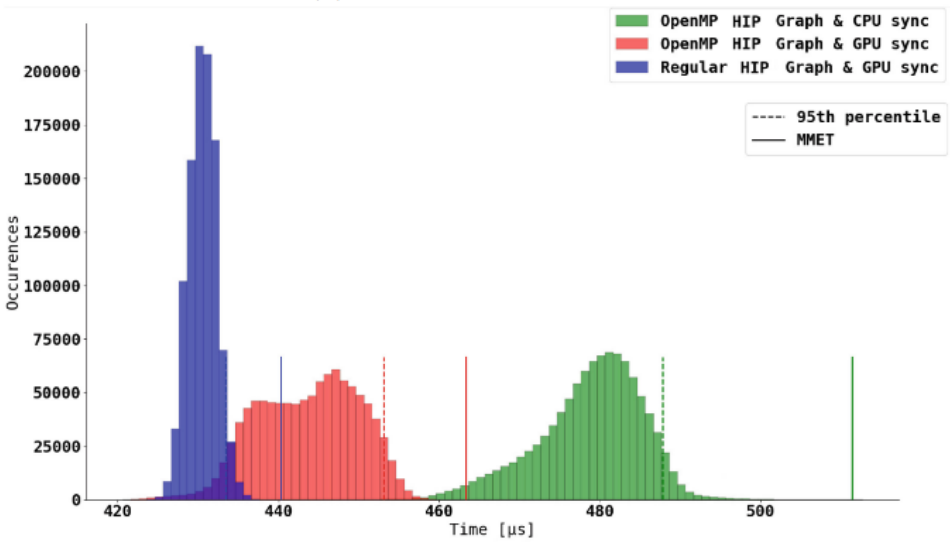
HiPART is organized into three technical Work Packages: (1) *WP1* is dedicated to developing and validating real-time high-performance systems through relevant use cases, (2) *WP2* is dedicated to developing extensions for high-performance, including performance and adaptability, and (3) *WP3* is dedicated to extensions for critical real-time systems, including resilience and predictability. Additionally, there is a dedicated WP for impact maximization and project management, ensuring continuous dissemination of key findings[1] and the smooth development of HiPART. All WPs span the 4 phases of the project, including (1) *phase 1* to define the use cases and the initial design, (2) *phase 2* for the development and isolated testing of software components, ensuring their individual functionality and compatibility with the targeted platforms, (3) *phase 3* for the integration and optimization of the parallel framework, and (4) *phase 4* for the validation and demonstration.

HiPART started in September 2024 and will expand three years of work. It is now in its initial phase, where use cases and system requirements are being defined. This paper presents the project and its intended solution for real-time high-performance systems. The project is developing a unique framework for efficiently deploying advanced CPS in parallel and heterogeneous processor architectures, holistically addressing real-time and HPC requirements. The project leverages and extends OpenMP to accommodate requirements from the critical

[1] Follow HiPART in the LinkedIn profile: `https://www.linkedin.com/company/105114452`.



(a) CUDA on NVIDIA.



(b) HIP on AMD.

Figure 10 Histogram of response time.

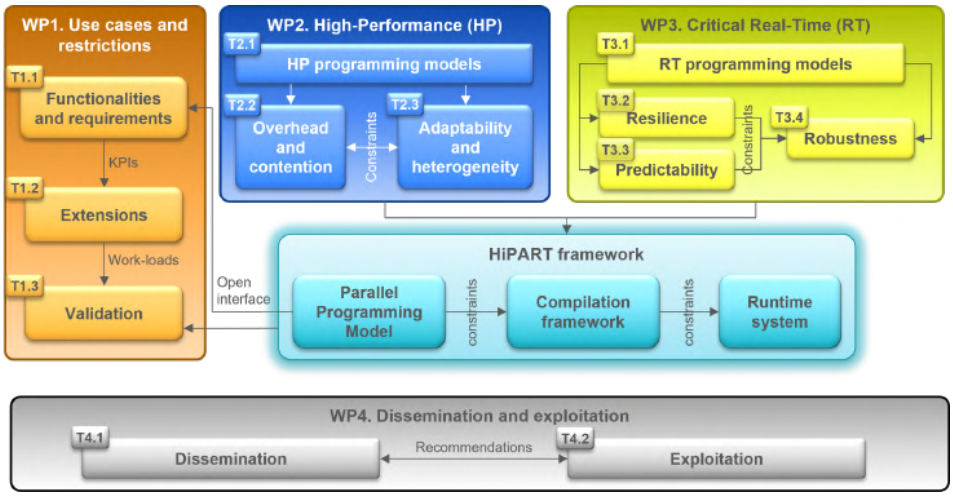


Figure 11 Timeplan of the HiPART project.

real-time domain, including event-based execution, time predictability and resilience, and the HPC domain, including performance portability and adaptability. The project has preliminary results for task-based replication, showing enhancements in detecting and bypassing faults, and event-based execution exploiting heterogeneous OpenMP taskgraphs, showing competitive performance, equivalent jitter and much better programmability than native solutions. Future work includes further extensions for fault-recovery based on consensus-and-voting, N-version programming and checkpointing and scheduling mechanisms to enhance the predictability of heterogeneous systems, among others.

References

1 Jaemin Baek, Jeonghyun Baek, Jeeheon Yoo, and Hyeongboo Baek. An N-modular redundancy framework incorporating response-time analysis on multiprocessor platforms. *Symmetry*, 11(8):960, 2019. doi:10.3390/SYM11080960.

2 OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 6.0, 2024. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf.

3 Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. A comparison of SYCL, OpenCL, CUDA, and OpenMP for massively parallel support vector machine classification on multi-vendor hardware. In *10th International Workshop on OpenCL*, pages 1–12, 2022. doi:10.1145/3529538.3529980.

4 Cyril Cetre, Chenle Yu, and Sara Royuela. LLVM for OpenMP event-based synchronization in taskgraphs. Software, version 1.0. swhId: swh:1:dir:aa51e2f008a05c550f7f0f9b2359a13165b8094a (visited on 2025-02-14). URL: https://gitlab.bsc.es/ppc-bsc/software/llvm-taskgraph/-/tree/cudaGraph_RTevent, doi:10.4230/artifacts.22923.

5 Cyril Cetre, Chenle Yu, Sara Royuela, Rémi Barrere, Eduardo Quiñones, and Damien Gratadour. Event-Based OpenMP Tasks for Time-Sensitive GPU-Accelerated Systems. In *International Workshop on OpenMP*, pages 31–45. Springer, 2024. doi:10.1007/978-3-031-72567-8_3.

6 Jian-Jia Chen, Chuan-Yue Yang, Tei-Wei Kuo, and Shau-Yin Tseng. Real-time task replication for fault tolerance in identical multiprocessor systems. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, pages 249–258. IEEE, 2007. doi:10.1109/RTAS.2007.30.

7 Jose Monsalve Diaz, Swaroop Pophale, Kyle Friedline, Oscar Hernandez, David E Bernholdt, and Sunita Chandrasekaran. Evaluating support for OpenMP offload features. In *47th International Conference on Parallel Processing*, pages 1–10, 2018. doi:10.1145/3229710.3229717.

8 Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *International Conference on Parallel Processing*, pages 124–131. IEEE, 2009. doi:10.1109/ICPP.2009.64.

9 Thierry Gautier, Christian Pérez, and Jérôme Richard. On the impact of OpenMP task granularity. In *Evolving OpenMP for Evolving Architectures: 14th International Workshop on OpenMP (IWOMP)*, pages 205–221. Springer, 2018. doi:10.1007/978-3-319-98521-3_14.

10 Volkan Gunes, Steffen Peter, Tony Givargis, and Frank Vahid. A survey on concepts, applications, and challenges in cyber-physical systems. *KSII Transactions on Internet and Information Systems (TIIS)*, 8(12):4242–4268, 2014. doi:10.3837/TIIS.2014.12.001.

11 Prasanna Hambarde, Rachit Varma, and Shivani Jha. The survey of real time operating system: RTOS. In *International Conference on Electronic Systems, Signal Processing and Computing Technologies*, pages 34–39. IEEE, 2014.

12 Dian-Lun Lin and Tsung-Wei Huang. Efficient GPU computation using task graph parallelism. In *27th International Conference on Parallel and Distributed Computing*. Springer, 2021.

13 Santiago Lozano, Tamara Lugo, and Jesús Carretero. A Comprehensive Survey on the Use of Hypervisors in Safety-Critical Systems. *IEEE Access*, 11:36244–36263, 2023. doi:10.1109/ACCESS.2023.3264825.

14 Brayden McDonald and Frank Mueller. OpenMP-RT: Native Pragma Support for Real-Time Tasks and Synchronization with LLVM under Linux. In *25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 119–130, 2024. doi:10.1145/3652032.3657574.

15 Adrian Munera, Guerau Dasca, Eduardo Quiñones, and Sara Royuela. Adaptive parallelism in OpenMP through Dynamic Variants. In *High Performance Computing: ISC High Performance 2024 International Workshops*, 2024.

16 Adrian Munera, Guerau Dasca, and Sara Royuela. LLVM for adaptive parallelism with dynamic variants. Software, version 1.0. swhId: swh:1:dir:a43d82b3d564a56f1a25347ff9a0beec81def2d5 (visited on 2025-02-14). URL: https://gitlab.bsc.es/ppc-bsc/research/c3po2024-dynamicvariants, doi:10.4230/artifacts.22924.

17 Adrian Munera, Eduardo Quiñones, and Sara Royuela. GuardianOMP: A framework for highly productive fault tolerance via OpenMP task-level replication. In *International Parallel & Distributed Processing Symposium*, 2025.

18 Adrian Munera and Sara Royuela. GuardianOMP. Software, version 1.0. (visited on 2025-02-14). URL: https://anonymous.4open.science/r/GuardianOMP-4233/README.md, doi:10.4230/artifacts.22922.

19 Eduardo Quiñones, Sara Royuela, Claudio Scordino, Paolo Gai, Luís Miguel Pinho, Luís Nogueira, Jan Rollo, Tommaso Cucinotta, Alessandro Biondi, Arne Hamann, et al. The AMPERE Project:: A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimization. In *IEEE 23rd International Symposium on Real-Time Distributed Computing*, pages 201–206. IEEE, 2020.

20 Alejandro Rico, Isaac Sánchez Barrera, Jose A Joao, Joshua Randall, Marc Casas, and Miquel Moretó. On the benefits of tasking with OpenMP. In *15th International Workshop on OpenMP*, pages 217–230. Springer, 2019.

21 Sara Royuela. RESPECT - Reliable Heterogeneous Parallelism for Embedded Critical Systems, 2024. URL: https://sroyuela.github.io/respect-project/.

22 Sara Royuela, Alejandro Duran, Maria A Serrano, Eduardo Quiñones, and Xavier Martorell. A functional safety OpenMP for critical real-time embedded systems. In *13th International Workshop on OpenMP*, pages 231–245. Springer, 2017. doi:10.1007/978-3-319-65578-9_16.

23 Sara Royuela, Roger Ferrer, Diego Caballero, and Xavier Martorell. Compiler analysis for OpenMP tasks correctness. In *ACM International Conference on Computing Frontiers*, 2015.

24 Sara Royuela, Bartomeu Pou, Cyril Cetre, Rémi Barrère, Eduardo Quiñones, and Damien Gratadour. RisingSTARS, RISE International Network for Solutions Technologies and Applications of Real-time Systems. In *ISC High Performance*, 2023.

25 Sara Royuela, Franck Wartel, Sylvain Tiberio, Eric Jenn, Hubert Guérard, and Guy Bois. LIONESS – Improving and leveraging OpenMP for the efficient and safe use of new high-performance hardware platforms. *Ada User Journal*, 45(4), 2024.

26 Mohammad Samadi, Sara Royuela, Luis Miguel Pinho, Tiago Carvalho, and Eduardo Quiñones. Time-predictable task-to-thread mapping in multi-core processors. *Journal of Systems Architecture*, 148:103068, 2024. doi:10.1016/J.SYSARC.2024.103068.

27 Maria A Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, and Eduardo Quiñones. Timing characterization of OpenMP4 tasking model. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE, 2015.

28 Maria A Serrano, Sara Royuela, and Eduardo Quiñones. Towards an OpenMP specification for critical real-time systems. In *14th International Workshop on OpenMP*. Springer, 2018.

29 Faisal Shahzad, Jonas Thies, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard Wellein. CRAFT: A library for easier application-level checkpoint/restart and automatic fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):501–514, 2018. doi:10.1109/TPDS.2018.2866794.

30 Aleksandar Simevski, Rolf Kraemer, and Milos Krstic. Investigating core-level N-modular redundancy in multiprocessors. In *IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs*, pages 175–180. IEEE, 2014. doi:10.1109/MCSOC.2014.33.

31 John Paul Walters and Vipin Chaudhary. Application-level checkpointing techniques for parallel programs. In *International Conference on Distributed Computing and Internet Technology*, pages 221–234. Springer, 2006. doi:10.1007/11951957_21.

32 Franck Wartel and Certain Antoine. HP4S: High Performance Parallel Payload Processing for Space. In *European Workshop on On-board Data Processing*, 2021. URL: https://zenodo.org/records/5639503/files/05.01_OBDP2021_Certain.pdf?download=1.

33 Michael Wong, Michael Klemm, Alejandro Duran, Tim Mattson, Grant Haab, Bronis R de Supinski, and Andrey Churbanov. Towards an error model for OpenMP. In *International Workshop on OpenMP*, pages 70–82. Springer, 2010. doi:10.1007/978-3-642-13217-9_6.

34 Chenle Yu, Sara Royuela, and Eduardo Quiñones. Taskgraph: A low contention OpenMP tasking framework. *Transactions on Parallel and Distributed Systems*, 34(8):2325–2336, 2023. doi:10.1109/TPDS.2023.3284219.

35 Chenle Yu, Sara Royuela, and Eduardo Quiñones. Enhancing heterogeneous computing through OpenMP and GPU graph. In *53rd International Conference on Parallel Processing*, pages 534–543, 2024. doi:10.1145/3673038.3673050.