Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale

WP4 Tool chain for heterogeneous multi-node HPC platform

D4.3 Mixed precision tool suite

http://textarossa.eu

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No 956831

TEXTAROSSA

Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale

Grant Agreement No.: 956831

Deliverable: D4.3 Mixed precision tool suite

Project Start Date: 01/04/2021 **Duration**: 36 months

Coordinator: *AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE - ENEA, Italy.*

Deliverable No	D4.3
WP No:	WP4
WP Leader:	CINI-POLIMI
Due date:	M24 (March 31, 2023)
Delivery date:	07/04/2023

Dissemination Level:

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No 956831

DOCUMENT SUMMARY INFORMATION

Project title:	Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale
Short project name:	TEXTAROSSA
Project No:	956831
Call Identifier:	H2020-JTI-EuroHPC-2019-1
Unit:	EuroHPC
Type of Action:	EuroHPC - Research and Innovation Action (RIA)
Start date of the project:	01/04/2021
Duration of the project:	36 months
Project website:	textarossa.eu

WP7 Dissemination, Communication and Exploitation

Deliverable number:	D4.3					
Deliverable title:	Mixed precision tool suite					
Due date:	M24					
Actual submission date:	07/04/2023					
Editor:	Daniele Cattaneo					
Authors:	D. Cattaneo, G. Agosta					
Work package:	WP4					
Dissemination Level:	Public					
No. pages:	22					
Authorized (date):	07/04/2021					
Responsible person:	Giovanni Agosta					
Status:	Plan	Draft	Working	Final	[✓ Submitted]	Approved

Revision history:

Version	Date	Author	Comment
0.1	2023-03-31	D. Cattaneo	Main text
0.2	2023-04-06	G. Agosta	Updated document metadata
1.0	2023-04-07	D. Cattaneo	Finalization after internal review

Quality Control:

Checking process	Who	Date
Checked by internal reviewer	Barbara Cantalupo	2023-04-07
Checked by Task Leader	Giovanni Agosta	2023-04-07
Checked by WP Leader	Berenger Bramas	2023-04-07
Checked by Project Coordinator	Massimo Celino	2023-04-07

COPYRIGHT

ACKNOWLEDGEMENTS

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement no 956831. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Italy, Germany, France, Spain, Poland.

Please see http://textarossa.eu for more information on the TEXTAROSSA project.

The partners in the project are AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE (ENEA), FRAUNHOFER GESELLSCHAFT ZUR FOERDERUNG DER ANGEWANDTEN FORSCHUNG E.V. (FHG), CONSORZIO INTERUNIVERSITARIO NAZIONALE PER L'INFORMATICA (CINI), INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE (INRIA), BULL SAS (BULL), E4 COMPUTER ENGINEERING SPA (E4), BARCELONA SUPERCOMPUTING CENTER-CENTRO NACIONAL DE SUPERCOMPUTACION (BSC), INSTYTUT CHEMII BIOORGANICZNEJ POLSKIEJ AKADEMII NAUK (PSNC), ISTITUTO NAZIONALE DI FISICA NUCLEARE (INFN), CONSIGLIO NAZIONALE DELLE RICERCHE (CNR), IN QUATTRO SRL (in4). Linked third parties of CINI are POLITECNICO DI MILANO (CINI-POLIMI), Università di Torino (CINI-UNITO) and Università di Pisa (CINI-UNIPI); linked third party of INRIA is Université de Bordeaux; in-kind third party of ENEA is Consorzio CINECA (CINECA); in-kind third party of BSC is Universitat Politècnica de Catalunya (UPC).

The content of this document is the result of extensive discussions within the TEXTAROSSA © Consortium as a whole.

DISCLAIMER

Table of contents

List of Acronyms

Acronym	**Definition**
BP	Business Plan
IP	Intellectual Property
IPR	Intellectual Property Rights
PTC	Program Technical Comittee
DEC	Dissemination and Exploitation Committee
PMB	Project Management Board
ASIC	Application Specific Integrated Circuit
FPGA	Field Programmable Gate Array
CAGR	Compound Annual Growth Rate
HPC	High-Performance-Computing
DM	Dissemination Manager
NoC	Network on Chip
PTM	Project Technical Manager
RTRM	Run-Time Resource Manager
GPGPU	General Purpose computing on Graphics Processing Unit
JIT	Just in Time
TAFFO	Tuning Assistant for Floating and Fixed point Optimization
VRA	Value Range Analysis
DTA	Data Type Allocation
TU	Translation Unit
ILP	Integer Linear Programming
ARE	Average Relative Error

Executive Summary

In order to better exploit the energy efficiency of current computing hardware, as well as to improve the overall quality of the applications in TEXTAROSSA, part of the project involves the development of a mixed-precision tool suite that allows application developers to automatically exploit optimized data-types that maximise the tradeoff between precision and performance.

This document describes the current status of the mixed-precision tool-suite, as delivered. In particular we describe TAFFO, its modifications to support TEXTAROSSA applications based on GPGPUs and parallel applications, and finally we show that the modifications developed indeed result in an improved runtime on a suite of example benchmarks. To prioritize the support of TEXTAROSSA use case applications, the development of CUDA support (in addition to the foreseen OpenCL support) has been prioritized over HLS. Mixed precision with HLS will be delivered in D4.7.

Quantitative goal: Error below 3%, performance greater than baseline. The goal is achieved for over 75% of the PolyBench suite.

Technical goal contribution: Mixed precision contributes to sustained application performance and energy efficiency by providing speedup (and therefore total energy reductions) of up to 100%, depending on the error-tolerance of the application.

1 Introduction

Approximate Computing is an increasingly popular approach to achieve large performance and energy improvements in error-tolerant applications [Sampson2011, Mittal2016]. This class of techniques aims at trading off computation accuracy for performance and energy.

Within Approximate Computing, a key issue is to perform each computation on the most energy and performance-efficient data type that allows to preserve the desired minimum accuracy. This non-trivial task is usually performed manually by embedded systems programmers, and in general by software developers that need to achieve high performance with limited resources.

However, this operation is error-prone and tedious, especially when large code bases are involved. Thus, a significant research effort has been spent over the recent years to build compiler-based tools to support or entirely replace the programmer effort [Cherubin2020].

Most precision tuning tools do not address parallel programming paradigms, both CPU-based and GPGPU based. This is particularly true for tools that perform automatic detection of the region of code to be affected by the precision tuning transformation. Indeed, to guarantee the correctness of the transformed program, precision tuning tools need to reliably detect each parallel region and the sets of variables shared between parallel execution threads. The analysis of the parallel behaviour of the program is particularly difficult for languages such as C, C++ and LLVM-IR, which do not support parallel programming paradigms without an auxiliary support library.

At the same time, the code transformation steps that are required to implement mixed precision in the program must preserve the correctness of parallel-computing-specific constructs when they are affected by the optimization process. Such constructs involve locking, synchronization, and GPU API calls.

TAFFO is the tool employed in the TEXTAROSSA toolchain in order to achieve mixed precision in heterogeneous HPC platform. Therefore, in order to fit within the general architecture of TEXTAROSSA, our primary goal is to improve TAFFO's support for parallel computing paradigms. At compiler level, this translates for support for specific frameworks and/or languages that enable the implementation of programs exploiting such paradigms, such as OpenMP, OpenCL and CUDA. This is done by embedding tool-specific knowledge inside TAFFO in order for the mixed-precision tuning passes to infer the actual data flow across different execution contexts.

This document describes the modifications of TAFFO delivered as part of the TEXTAROSSA project in order to support the GPGPU and parallel computing use cases. We first summarize the architecture of TAFFO, then we describe the new features that allow the tuning of mixed-precision applications employing parallel computing paradigms and GPGPUs. We also show experimental data that demonstrates the effectiveness of the modifications performed.

2 The TAFFO Precision Tuning Framework

TAFFO is an easy to use and state-of-the-art framework that automatically performs precision tuning to exploit the performance/accuracy trade-off, developed at Politecnico di Milano since 2017. TAFFO has a proven optimization efficacy in a variety of hardware configurations and application domains [Fossati2020][Magnani2021] and leverages programmer annotations which encapsulate domain knowledge about the conditions under which the software being optimized will run. TAFFO is modular and based on the solid LLVM technology, which allows extensibility to improved analysis techniques, and comprehensive support for the most common precision-reduced data types and programming languages.

According to the classification given by Cherubin and Agosta in [Cherubin2020], in the state of the art other precision tuning tools focus only on one aspect of precision tuning — such as verification of error constraints, or the generation of a data type assignment — and do not provide a comprehensive applicative solution. This makes TAFFO uniquely suited for integration in pre-existing development environments and wherever it is necessary to perform mixed precision tuning on large codebases.

Additionally, most other precision tuning tools target floating point data types, overlooking other representations. This is appropriate in the high-performance-computing context, but limits applicability in edge computing applications. TAFFO fills this gap, providing an increased set of supported data types — including fixed point. Additionally, static analyses are less common than dynamic analyses. In fact, TAFFO is unique amongst LLVM-based precision tuning tools in exploiting only static analyses.

TAFFO operates during code compilation as a plug-in for the LLVM compiler infrastructure.

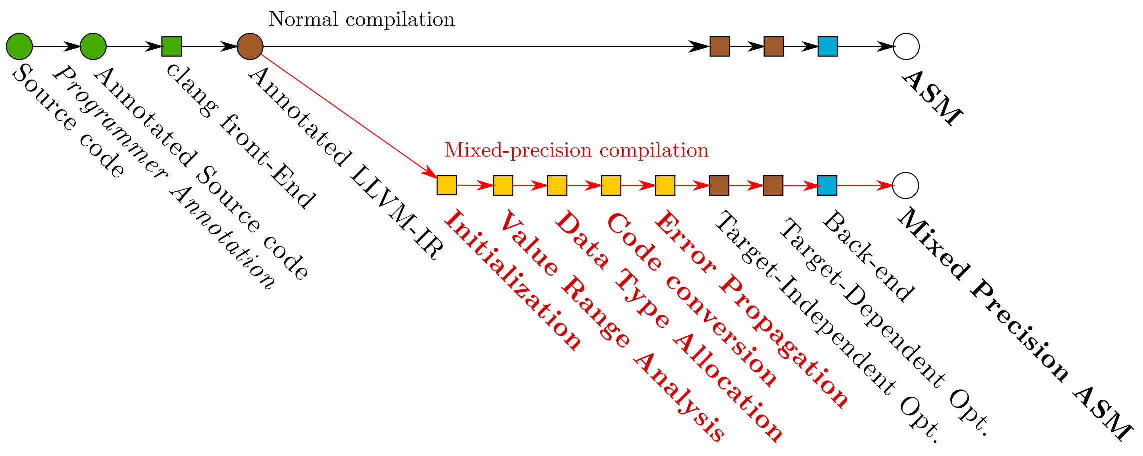
This design decision creates several benefits.

- **Portability**: the software tool is agnostic with respect to source language and target architecture.
- **Fine-Grained**: data types are allocated to each individual operation rather than on each declared variable.
- **Compatibility**: TAFFO may be combined with other optimisation techniques.

TAFFO accepts as input LLVM intermediate code representation, and it produces the same format as output. The LLVM-IR-equivalent version of a C/C++ program can be obtained via *clang*. A plain un-modified version of the *clang* front-end is also able to parse the additional annotations provided by the user in the source code. These annotations will later be processed by the initial stages of TAFFO.

TAFFO is shipped as a set of LLVM passes, i.e. elementary units of the compiler pipeline. The passes can be inserted in any point of the optimization pipeline, but usually they are executed immediately after the frontend. Other normalization and analysis passes are automatically scheduled by LLVM as required. As a user interface, TAFFO provides an eponymous command line program, *TAFFO*, for compiling a C or C++ program or file and automatically perform precision tuning optimizations. A standard invocation of *clang* or *GCC* can be replaced with an invocation to *TAFFO* to enable such optimizations.

The five pipeline stages of the TAFFO architecture are called *Initializer* (INIT), Value Range Analysis (VRA), Data Type Allocation (DTA), Conversion (CONV), and Error Propagation (EP). Their order of execution and position on the overall compilation pipeline is shown in the following figure.



The INIT pass of TAFFO reads annotations and generates the internal metadata structure required by the other passes. VRA conservatively derives from the metadata the numerical intervals of each variable in the program. DTA then determines which reduced-precision data type to use. The DTA pass comes in two operation modes: a peephole-based algorithm in which each variable is assigned a fixed-point data type with the highest valid point position; and an Integer-Linear-Programming-based technique [Cattaneo2021]. CONV modifies the LLVM-IR accordingly with the data type chosen by the previous passes, optionally replacing trigonometric function calls with higher-efficiency custom implementations [Cattaneo2021FixM]. EP statically analyses the error using state-of-the-art estimation methods [Cherubin2020Dynamic].

2.1 Comparison with other tools

To our knowledge, there are no other precision tuning tools that explicitly support OpenMP for parallel applications. However multiple tools do indeed claim to support GPGPUs.

In particular, our approach to GPGPU automated mixed-precision tuning bears the most resemblance to the one presented in GPUMixer [Laguna2019]. While our approach is able to leverage TAFFO to perform data type selection and selection of the mixed-precision configuration, GPUMixer is a ground-up solution and therefore also includes a graph-based methodology for data type selection via a dynamic graph search. This dynamic search intrinsically takes more time and effort than the static analyses utilized by TAFFO. Additionally, GPUMixer supports only double-precision or single-precision data types, while TAFFO also supports fixed point types and half-precision floats. Finally, the code conversion approach in GPUMixer does not allow for the minimization of casts in the generated code, and it is explicitly discussed how the number of casts influence the register pressure and therefore produce non-optimal performance. On the contrary, TAFFO always minimizes the number of casts in the transformed program, producing code that is equivalent in all respects to changing the data types in the source code.

The focus on static analyses found in our work also differentiates us from efforts from Rojek [Rojek2019] and Nobre [Nobre2018], as they only focus on optimizations performed at run-time (dynamically), while our approach is performed at compile-time, although TAFFO can be used as a framework for dynamic optimization as well [Cherubin2020Dynamic]. Finally, our work also supports a set of data types similar to the one supported in Angerd et al. [Angerd2017], however their work is purely theoretical, and they do not demonstrate results on a real GPGPU architecture, opting for CPU-based simulations and assessments instead.

3 Delivered Software Features

This deliverable adds two features to TAFFO: support for CPU-based parallel computing with OpenMP, and support for GPGPU applications employing CUDA and OpenCL.

3.1 CPU-based parallel computing

While TAFFO operates at the intermediate representation level, OpenMP is mainly implemented in the compiler frontend. In fact, OpenMP is used by adding specific *pragma* annotations in a C or C++ program. Depending on the pragma, OpenMP will automatically transform what would normally be a non-parallel C language construct into a parallel one.

The most common pragmas are the *parallel* pragma and the *for* pragma, and as a result we focus on supporting such pragmas in our implementation strategy [Magnani2022]. The *parallel* pragma executes a given code block multiple times in parallel in multiple threads. The number of threads depends on the estimated maximum number of independent threads that can be run on the machine. On the other hand, the *for* pragma must appear before a *for* loop, and it executes each iteration of the loop in parallel with respect to the other.

The implementation of a typical OpenMP library uses a fixed thread pool, a well-known implementation strategy for supporting parallel computations while minimizing the operating-system-level overhead of creating and destroying threads every time a new task must be instantiated.

The *parallel* pragma starts a new task on each available thread in the pool, all tasks running the same piece of code. The *for* pragma is similar, except that each task executes its body multiple times depending on how many threads are in the pool. Since the trip-count of the loop must be known up-front, the induction variable of the loop shall not be modified in the body of the loop itself.

In the context of LLVM, this functionality is supported by the OpenMP runtime library provided with the *clang* compiler. The creation of the thread pools and the enqueueing of the tasks is performed by code in such library, but the code that calls the library functions is generated by the *clang* frontend at compile-time of the program. The block of code that is associated to each parallel task to be executed is outlined by the *clang* compiler to a separate function. A pointer to this function is then passed to a specific runtime function ("__kmpc_fork_call") alongside with the local variables that are used within each thread context.

In order to add support for OpenMP-aware optimizations in LLVM-IR, we modify two passes of TAFFO: Initializer and Conversion. In Initializer, the program is searched for instances of call sites of the OpenMP fork function. At each call site in the translation unit, the OpenMP fork function is temporarily deleted and replaced by a *trampoline* function, whose body simply calls the OpenMP outlined function. This allows TAFFO's existing code to handle OpenMP programs without additional modifications.

Indeed, analyses and transformations in TAFFO are intra-procedural, and can handle mixed-precision across functions and in function arguments. The VRA and DTA passes are able to inspect each call-site independently and derive mixed-precision data types and value ranges for each call argument. This means that call sites of the same function can have different type annotations depending on the surrounding context.

Therefore, the Conversion pass must duplicate each function affected by the mixed-precision transformation a number of times that depends on the number of call sites with unique type assignments.

In the final program, as a result, call sites that in the original program invoked the same function now may invoke different functions, depending on whether the call sites now use different types than before or not. This applies to the OpenMP trampoline functions and outlined functions as well.

To support the OpenMP runtime, an additional process has been implemented in the Conversion pass, which converts the calls to the trampoline function back to the original call to the OpenMP library function. This procedure effectively also replaces OpenMP outlined functions with their mixed-precision cloned equivalent if needed.

For what concerns the handling of trip count of loops, we exploit the fact that the OpenMP library initialization function of *for* loops takes as arguments the lower bound, upper bound and stride of the loop. Therefore, it is easy to analyse the outlined function, detecting the initialization calls and computing the total trip count across all loops with the formula:

$$n = \left\lceil \frac{u - l + 1}{s} \right\rceil$$

Where *n* is the trip count, *s* is the stride, *u* is the upper bound and *l* is the lower bound.

3.2 Support for GPGPU computation

The application programming interface for OpenCL is composed of a C library which allows interfacing with the GPU hardware from the host, and a GPU-oriented dialect of C++ (the OpenCL language) employed for programming the kernels. The host code uses the OpenCL library in order to initialize the hardware, communicate memory-layout information, loading, compiling and executing the kernels. Ordinarily, the OpenCL host library also includes a compiler for the OpenCL language, used for loading new OpenCL kernels at runtime. It is also possible to load previously compiled OpenCL kernels.

In our approach we support both OpenCL and CUDA with a unified solution that exploits the commonalities between the two APIs. Both APIs provide calls in order to instantiate kernels and to copy the arguments from CPU memory to GPU memory and vice-versa, and these APIs are used in similar ways.

In order to support the tuning of both host code and the GPGPU kernels, we assume the two portions of code consist of two separate translation units (TU), which are compiled externally by a third-party compiler such as *clang*. Both the OpenCL and CUDA APIs also offer a Just In Time compiler that, if exploited, allows the programmer to JIT-compile the source code of the kernels from a text string specified in the host code. The JIT approach allows to bundle host code and kernel code in a single TU. The two approaches to kernel code definition (multi-TU and JIT) are functionally equivalent. However, JIT-compilation does not allow for the vendor-provided compiler to be extended with plugins, such as TAFFO. Therefore, in this work we only focus on the multi-TU approach.

TAFFO, just like *clang* or other C compilers, operates on a single TU at a time. However, to achieve the precision tuning of both host code and kernel code, the precision tuning tool must have some visibility of both TUs at the same time, in order for analyses on one piece of code to be able to influence the analyses on the other. Therefore, in order to share information between different runs of TAFFO, we introduce the **Delayed Analysis methodology** or DA in brief. By exploiting DA, TAFFO is able to match scalar variables or buffers present in one translation unit with other variables or buffers present in another translation unit. If the relationships between translation units is known, the match can be performed automatically. This is the case for extern symbols in host programs consisting of multiple source code files. When it is not possible to assess in a conservative way if a translation unit is related to another one, the programmer can register a given variable or buffer for DA manually by exploiting a new kind of annotation, called buffer ID annotation. The buffer ID is a string value associated with a given variable or array. The data type allocation and range information of variables or arrays with the same buffer ID is kept synchronized by TAFFO, even if the variables are part of different translation units. An example of a buffer ID annotation is shown in the following code snippet.

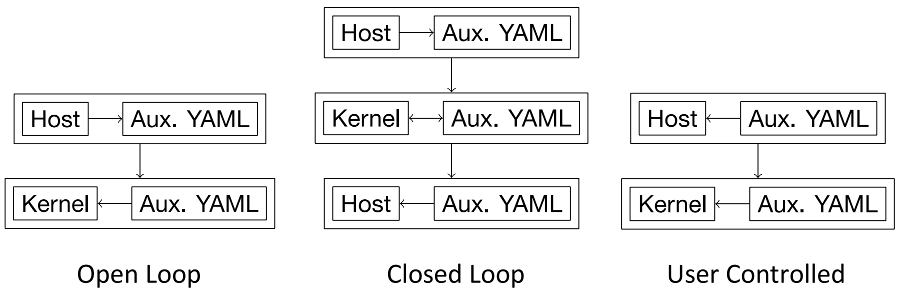
```
__attribute__((annotate("scalar(range(-1,1)) bufferid('a')"))) float *a;
```

The same format of annotation is used for host code and for kernel code.

When using DA, at every full compilation TAFFO collects the currently known value range and data type information for all variables having a buffer ID, and stores it into an auxiliary YAML [YAML] file. This file is then read and updated by subsequent passes of TAFFO in order to take into account the information that can be inferred from other TUs.

Depending on the way in which TAFFO is invoked on each TU, the DA methodology can operate either in three ways: open loop mode, closed loop mode, or user-controlled mode. In open loop mode, the final ranges for every buffer subject to DA are already known, therefore the only analysis being suspended is the data type allocation. Since the data type allocation depends primarily on the ranges [Cattaneo2021], the first execution of TAFFO decides the data types for all variables, while the subsequent executions read the correct types from the auxiliary files. In closed loop mode, the value ranges of the buffer ID variables are not known a-priori. Therefore, TAFFO computes the final ranges of all variables only after two executions: the first one on the host code, the second one on the kernel code. In the second run, the final ranges are known, and the data type allocation is computed and applied to the kernels. One last execution of TAFFO closes the loop and applies the data type allocation to the host code as well. Finally, in user-controlled mode the user establishes a-priori the data type to use for all shared variables between host code and kernel code by writing the DA auxiliary YAML file by hand. Therefore, the role of TAFFO— with respect to the shared variables — is simply to apply the data type choice selected by the user.

The Delayed Analysis modes are shows in the following image. The vertical arrows illustrate the order in which host or kernel code is tuned, while horizontal arrows illustrate the dependency relationship between the code being tuned and the auxiliary YAML file used by DA.



Other modifications were required in order to allow TAFFO to detect which buffers are used to send or retrieve data for the GPGPU, and to automatically adjust the sizes of the buffers sent or received from the GPGPU in case the sizes of the reduced precision data types differ from the originals.

In particular we handle the following OpenCL and CUDA functions:

- clCreateBuffer
- clEnqueueReadBuffer
- clEnqueueWriteBuffer
- clSetKernelArg
- cuMemcpyHtoD
- cuMemcpyHtoD_v2
- cuMemcpyDtoH
- cuMemcpyDtoH_v2

The parameters to these functions are automatically adjusted for consistency with reduced-precision data types. Additionally, specific support needs to be provided for how translation units are formatted for specific GPU compiler architectures. At the moment we only support NVIDIA GPUs, therefore specific support has been added for PTX assembly files.

4 Testing and Evaluation

In this section we briefly show experimental data gathered by exploiting the TAFFO version delivered as part of the TEXTAROSSA project on relevant industry-standard benchmarks in order to evaluate the effectiveness of the modifications oriented to supporting the OpenMP, OpenCL and CUDA frameworks. As quantitative goal, we target an error below 3%, and performance (in terms of time-to-solution) greater than baseline.

4.1 OpenMP

To evaluate our work, we used the PolyBench/ACC benchmark suite [GrauerGray2012]. PolyBench is a collection of several small kernels written in C, covering several computational tasks, such as data mining tasks, linear algebra kernels, BLAS routines and more. PolyBench allows to tune the amount of memory to employ for every test in order to be able to adapt to multiple targets, even memory-constrained ones such as microcontrollers.

We run all the benchmarks on a non-uniform memory access (NUMA) server with a 24 Six-Core AMD Opteron(tm) Processor 8435 (2,6 GHz) with 128GB RAM. The operating system is Ubuntu 20.04 LTS. Not all benchmarks gained a speedup from parallelization — as a result, only a subset of benchmarks were selected, specifically those that can be parallelized without algorithmic changes with respect to the original unmodified PolyBench/C suite, and where parallelization does indeed produce a speedup. These benchmarks are *2mm*, *3mm*, *doitgen*, *gemm*, *syr2k*, and *syrk*.

We compiled the benchmarks in three different configurations, or versions. In all cases, the compiler used was *clang* version 12.0, based on LLVM version 12.0. In the first configuration (denoted with the number zero) both the TAFFO mixed precision optimizations and OpenMP support were disabled, producing a non-parallel benchmark. In the second configuration (denoted with the number 1), OpenMP was enabled, but TAFFO was not used. In the third and final configuration (denoted with the number 2), both OpenMP and TAFFO were employed.

The data from the experiments conducted as described herein are shown in the following table.

Benchmark	**t_0 [s]**	**t_1 [s]**	**t_2 [s]**	**Speedup 1**	**Speedup 2**	**ARE**
2mm	105.375	6.199	1.819	1599.9%	240.7%	8.85×10^{-9} %
3mm	23.760	1.381	0.803	1620.8%	71.9%	7.45×10^{-5} %
doitgen	1.028	0.111	0.080	830.1%	38.1%	1.47×10^{-3} %
gemm	101.646	7.515	0.850	1252.6%	783.7%	8.85×10^{-9} %
syr2k	6.692	2.595	1.027	157.9%	152.8%	8.85×10^{-9} %
syrk	2.285	0.974	0.239	134.6%	308.3%	8.85×10^{-9} %

In the table, t_0 refers to the execution time of the non-parallel kernels, t_1 the execution time of the parallel kernels, and t_2 the execution time of the mixed-precision parallel kernels. Speedup 1 is the speedup of the parallel kernel with respect to the non-parallel kernel, and it measures the execution time improvement due to OpenMP support alone. Speedup 2 is the speedup of the mixed-precision parallel kernel (configuration 2) with respect to the parallel kernel (configuration 1), and it quantifies the improvements due to TAFFO's mixed precision optimization. Finally, we show the Average Relative Error (ARE) introduced by the mixed precision optimization performed by TAFFO.

The results show that, on top of the already considerable speedup derived from the usage of a parallel algorithm, TAFFO is able to improve the speedup considerably, up to an additional 783% for the *gemm* benchmark. The ARE error metric is under 0.01% for all benchmarks, which is in line with previous results obtained with TAFFO without exploiting OpenMP. This meets and exceeds our quantitative goals, therefore we can state that the OpenMP extension to TAFFO is effective in general at automatically achieving mixed-precision computation in most error-tolerant parallel applications.

4.2 OpenCL and CUDA

In order to demonstrate the effectiveness of our approach for automated mixed precision computation in a GPGPU environment, we evaluate our solution again on the PolyBench/ACC benchmark suite [GrauerGray2012]. Apart from the OpenMP implementation, this benchmark suite provides an implementation of the same kernels for both OpenCL and for CUDA, which have been coded following the best programming practices for each of these versions. This feature allows us to neglect differences in performance caused purely by the implementation of the kernels.

Our TAFFO-based solution is tested on two separate machines, one featuring CUDA, one using OpenCL. The machine used for evaluating OpenCL is an HP Z2 G8 tower workstation, with 64 GiB of RAM, a Intel 11th Gen Intel Core i7-11700K running at 3.60 GHz and a NVidia GeForce RTX 3070 GPGPU with Compute Capability 8.6. This machine runs Ubuntu 22.04.2 LTS with LLVM version 15.0.0. The machine used for evaluating CUDA is a tower workstation with 16 GiB of RAM, an AMD Ryzen 5600X Processor running at 3.7 GHz and a NVidia GeForce RTX 3070 Ti GPGPU with Compute Capability 8.6. This machine runs Ubuntu 20.04.6 LTS with LLVM version 15.0.0. These two hardware configurations will be called OpenCL machine and CUDA machine in the following discussion.

In both machines TAFFO was exploited using the closed loop DA methodology variant, in order to compile the entire set of benchmarks in the PolyBench/ACC suite. Multiple compilations were performed in order to characterize various ways of using reduced precision. In particular, we examine both the case in which the kernel code also performs the conversion of the data to the original non-reduced- precision type, and the case in which the kernels return the data in reduced- precision formats. The benchmark implementations provided by PolyBench/ACC already provide the necessary code for measuring time-to-solution, which includes in the measurement the time required for data transfer to and from the GPGPU. We compute the error by comparing the contents of the buffers produced by the evaluated configuration compiled using TAFFO and the unmodified benchmark.

We show the results of the experiments in the following tables:

OpenCL:

Benchmark	Host	Kernel	Time [s]	Speedup	Abs. Error	ARE
2mm	half	half	6.4770e-03	53.5 %	4.0423e+02	0.8 %
3mm	half	half	1.2330e-03	26.2 %	3.6008e+02	1.1 %
adi	float	float	1.4263e-02	0.0 %	0.0000e+00	0.0 %
atax	half	half	1.7390e-03	5.5 %	4.4391e+03	3.3 %
bicg	half	half	2.0920e-03	0.5 %	5.4780e+01	5.4 %
convolution-2d	half	half	2.9300e-04	45.1 %	5.4648e-04	0.4 %
convolution-3d	fix 16	fix 32	7.1100e-04	21.0 %	2.8901e-03	0.0 %
correlation	float	fix 32	1.3572e+00	4.7 %	4.0076e-04	3.4 %
covariance	half	half	1.3634e+00	3.9 %	2.9132e+00	1.7 %
doitgen	half	half	4.1100e-03	1.5 %	4.8665e-01	0.0 %
fdtd-2d	half	half	1.1065e-01	89.0 %	9.5823e-02	0.0 %
gemm	half	float	3.8500e-04	42.1 %	1.3905e+01	0.1 %
gemver	fix 16	fix 16	2.3150e-03	5.4 %	3.4510e-03	10.8 %
gesummv	half	half	1.5640e-03	3.5 %	8.3010e+05	2.2 %
gramschmidt	half	half	2.3286e+00	1.5 %	2.3192e-03	9.2 %
jacobi-1d-imper	fix 16	fix 32	9.9600e-04	75.7 %	4.9805e-04	0.1 %
jacobi-2d-imper	half	half	5.2000e-04	104.8 %	8.4940e-01	0.3 %
lu	half	half	5.1224e-02	46.5 %	6.3399e-02	13.5 %
mvt	fix 16	fix 32	9.0450e-03	43.6 %	7.4449e+02	41.5 %
syr2k	fix 32	float	2.6312e-02	13.1 %	9.2237e+00	0.0 %
syrk	half	float	1.5230e-02	0.3 %	3.4083e+02	0.0 %

CUDA:

Benchmark	Host	Kernel	Time [s]	Speedup	Abs. Error	ARE
2mm	half	half	5.6430e-03	88.2 %	4.0423e+02	0.8 %
3mm	half	half	1.2320e-03	21.3 %	3.6008e+02	1.1 %
adi	half	half	1.4456e-02	1.1 %	3.1309e-04	0.1 %
atax	fix 16	fix 32	3.2370e-03	0.7 %	7.9631e+03	18.7 %
bicg	fix 16	fix 16	1.9230e-03	0.8 %	6.1717e+01	6.0 %
convolution-2d	half	half	2.0300e-04	36.5 %	5.4648e-04	0.4 %
convolution-3d	fix 16	fix 32	1.5840e-03	2.3 %	2.8901e-03	0.0 %
correlation	half	half	1.3106e+00	4.0 %	3.9263e-04	2.3 %
covariance	half	half	1.3057e+00	6.1 %	2.9132e+00	1.7 %
doitgen	half	float	4.0890e-03	2.9 %	4.8669e-01	0.0 %
fdtd-2d	half	half	9.1099e-02	76.0 %	9.5823e-02	0.0 %
gemm	half	half	4.3900e-04	35.5 %	1.0121e+02	0.3 %
gemver	half	half	2.0810e-03	6.0 %	1.0460e+01	0.2 %
gesummv	fix 16	fix 32	1.5160e-03	2.2 %	3.0681e+06	16.2 %
gramschmidt	half	half	2.1154e+00	13.5 %	2.3192e-03	9.2 %
jacobi-1d-imper	fix 32	fix 32	1.0111e-01	12.5 %	0.0000e+00	0.0 %
jacobi-2d-imper	half	half	5.2100e-04	82.0 %	8.4940e-01	0.3 %
lu	fix 16	fix 16	4.7488e-02	36.0 %	9.0684e-02	45.3 %
mvt	half	half	3.2190e-03	1.4 %	1.9312e+01	2.9 %
syr2k	fix 16	fix 16	2.7327e-02	0.4 %	4.6278e+02	0.0 %
syrk	fix 16	fix 16	1.4096e-02	0.6 %	2.3141e+02	0.0 %

In particular, the tables show, for the OpenCL and CUDA machines respectively, the configurations that obtain the greatest speedup irrespective from the error. The host and kernel columns in the tables show the data types exposed to the host and used for the computation in the kernel respectively. The supported data types are denoted with the following names:

- half for IEEE-754 binary16 [14,15],
- float for IEEE-754 binary32,
- fix 16 for all 16-bit-sized fixed point types,
- fix 32 for all 32-bit-sized fixed point types.

The point position of all fixed point types is allocated automatically by the Data Type Allocation pass of TAFFO. The speedup is always calculated with respect to the unmodified PolyBench/ACC code, which is equivalent to a configuration with both host and kernel data types as float.

The data points with the best speedups are reliably the ones employing 16-bit sized data types, although they often have measurably higher errors than other types, especially for benchmarks such as *atax*, *gesummv*, *lu* and *mvt*. The specific configuration choices that obtain the best speedup differ between the two machines, with the CUDA machine having lesser speedups overall. The highest speedup reached is on the OpenCL machine, on the *jacobi-2d-imper* benchmark. The same benchmark achieves the second highest speedup on the CUDA machine, behind the *2mm* benchmark. The Absolute Relative Error (ARE) is generally below 10% except for some benchmarks such as *lu*, whose ARE is 45.3%. Examination of the results of the intermediate computations performed by the benchmark highlighted that the error is due to the usage of fixed point 16-bit types. With these data types, when the benchmark employs long chains of multiplications, the quantization error is easily amplified due to the high dynamic range of the results. Therefore the usage of small fixed point types needs to be carefully evaluated in a case-by-case basis.

Again, this data shows that our quantitative goals are met for the majority of the benchmarks evaluated, confirming the validity of our approach.

5 Conclusions

This document discussed the status of the mixed precision tool-chain of the TEXTAROSSA project. In particular the TAFFO precision tuning toolchain was discussed, both in general to present the main features of the framework of interest to TEXTAROSSA, and more specifically about the modifications performed as part of the project itself in order to implement support for GPGPU applications and multi-processor CPU parallel algorithms. Support for these use-cases was achieved by supporting specific programming APIs, namely OpenMP, OpenCL and CUDA. Synthetic examples were provided about how to integrate the TAFFO component in the applications required by the rest of the overarching project. The workability and effectiveness of the modifications and updates performed to the TAFFO software was shown by performing an experimental evaluation on the well-known PolyBench/ACC benchmark suite. We show data demonstrating significant speedups both for GPU-based applications and CPU-based applications, meeting and exceeding our quantitative goal of 3% error and performance above the baseline.

As part of future deliverables (more specifically, D4.7) the development roadmap for TAFFO involves support for HLS workflows, including specific support for Xilinx Vitis and/or other open-source solutions. Additionally we will provide continued support for the features outlined in this document when required for development of demonstrators as part of WP2.

Appendix A. Software Metadata

Version	0.5
Tag	v0.5.0
Repository Link	https://github.com/TAFFO-org/TAFFO
Code License	MIT License
Versioning system	git
Programming languages used	C++
Dependencies	CMake, LLVM, Google ORTools

6 References

[Fossati2020] Nicola Fossati, Daniele Cattaneo, Michele Chiari, Stefano Cherubin, Giovanni Agosta: *Automated Precision Tuning in Activity Classification Systems: A Case Study*. Proceedings of PARMA-DITAM 2020, January 2020. https://doi.org/10.1145/3381427.3381432

[Magnani2021] Gabriele Magnani, Daniele Cattaneo, Michele Chiari, Giovanni Agosta: The *Impact of Precision Tuning on Embedded Systems Performance: A Case Study on Field-Oriented Control*. Proceedings of PARMA-DITAM 2021, January 2021. https://drops.dagstuhl.de/opus/volltexte/2021/13639/

[Cherubin2020] Stefano Cherubin, Giovanni Agosta: *Tools for Reduced Precision Computation: a Survey*. ACM Computing Surveys, Volume 53, Issue 2, April 2020. https://doi.org/10.1145/3381039

[Cattaneo2021] Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, Giovanni Agosta: *Architecture-aware Precision Tuning with Multiple Number Representation Systems*. 2021 58th ACM/IEEE Design Automation Conference (DAC), https://doi.org/10.1109/DAC18074.2021.9586303

[Magnani2022] Gabriele Magnani, Lev Denisov, Daniele Cattaneo, Giovanni Agosta: *Precision Tuning in Parallel Applications*. Proceedings of PARMA-DITAM 2022, June 2022. https://drops.dagstuhl.de/opus/volltexte/2022/16121

[Sampson2011] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, D. Grossman: *Enerj: Approximate data types for safe and general low-power computation.* Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design Implementation, PLDI '11, ACM, New York, NY, USA, 2011, p. 164–174. https://doi.org/10.1145/1993498.1993518

[Mittal2016] S. Mittal: *A survey of techniques for approximate computing.* ACM Computing Surveys 48~(4) (2016) 62:1—62:33. https://doi.org/10.1145/2893356

[Cattaneo2021FixM] D. Cattaneo, M. Chiari, G. Magnani, N. Fossati, S. Cherubin, G. Agosta: *FixM: Code generation of fixed point mathematical functions.* Sustainable Computing: Informatics and Systems 29 (March 2021). https://doi.org/10.1016/j.suscom.2020.100478

[Cherubin2020Dynamic] S. Cherubin, D. Cattaneo, M. Chiari, A. Giovanni: *Dynamic precision autotuning with TAFFO.* ACM Transaction on Architecture and Code Optimization 17 (2) (May 2020). https://doi.org/10.1145/3388785

[YAML] Ben-Kiki, O., et al.: *YAML ain't markup language.* (September 2009). http://www.yaml.org/spec/1.2/spec.html

[Laguna2019] Laguna, I., et al.: *GPUmixer: Performance-driven floating-point tuning for GPU scientific applications.* High Performance Computing. pp. 227—246. Springer, Cham (2019)

[Rojek2019] Rojek, K.: *Machine learning method for energy reduction by utilizing dynamic mixed precision on gpu-based supercomputers.* Concurrency and Computation: Practice and Experience 31 (6), e4644 (2019). https://doi.org/10.1002/cpe.4644

[Nobre2018] Nobre, R., et al.: Aspect-driven mixed-precision tuning targeting gpus. Proc. 9th PARMA-DITAM. p. 26–31. PARMA-DITAM '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3183767.3183776

[Angerd2017] Angerd, A., Sintorn, E., Stenström, P.: *A framework for automated and controlled floating-point accuracy reduction in graphics applications on GPUs.* ACM Trans. Archit. Code Optim. 14 (4) (Dec. 2017) https://doi.org/10.1145/3151032

[Pouchet2016] Louis-Noël Pouchet and Tomofumi Yuki: *PolyBench/C 4.2.1, 2016.* https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[GrauerGray2012] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos: *Auto-tuning a high-level language targeted to GPU codes.* Innovative Parallel Computing ({InPar)}, 2012 (May 2012).