Mixed Precision in Heterogeneous Parallel Computing Platforms via Delayed Code Analysis

Daniele Cattaneo[1 [0000−0003−1453−3257]], Alberto Maggioli[1], Gabriele Magnani[1 [0000−0001−9729−5826]], Lev Denisov[1 [0000−0003−3540−4235]], Shufan Yang[2 [0000−0003−0531−2903]], Giovanni Agosta[1 [0000−0002−0255−4475]], and Stefano Cherubin[3 [0000−0002−5579−5942]]

[1] DEIB – Politecnico di Milano, name.surname@polimi.it
[2] Edinburgh Napier University, n.surname@napier.ac.uk
[3] Norges Teknisk-Naturvitenskapelige Universitet, name.surname@ntnu.no

Abstract. Mixed Precision techniques have been successfully applied to improve the performance and energy efficiency of computation in embedded and high performance systems. However, few solutions have been proposed that address precision tuning of both GPGPU code and its corresponding CPU code, limiting the gains achievable by mixed precision. We propose an extension to the TAFFO precision tuning toolset that enables Mixed Precision across the space of floating and fixed point data types on GPGPUs, leveraging static analysis and providing seamless interface adaptation between host and GPGPU kernel code. The proposed tool achieves speedups exceeding 2× by exploiting the optimization of both kernel and host code.

Keywords: compilers · precision tuning · heterogeneous systems · gpgpu

1 Introduction

General-Purpose Graphics Processing Units (GPGPUs) are nowadays the most popular class of accelerators for a variety of computationally intensive tasks in both High Performance Computing (HPC) and high-end embedded systems. There has been a steady increase in the GPGPU hardware support for low-precision data types, with modern GPGPUs offering "short float" formats such as BF16 (bfloat16) and half-precision (binary16), which can be used to achieve greater speedups in error-tolerant kernels, through the Mixed Precision computing approach. Mixed Precision is a branch of a more general class of techniques, known as Approximate Computing, which aim at trading off computation accuracy for other quality metrics, including performance and energy. Recent surveys [4,9] show that a significant number of tools have been developed to automatically analyze and transform codebases to exploit Approximate Computing.

However, such tools need help to cross the host/GPGPU barrier, as many of them cannot analyze GPGPU code directly due to restrictions of the input source code. Some specialized only in tuning GPGPU code [8]. This is particularly true for tools that automatically detect the region of code to be affected by

the precision tuning transformation, as the analysis of such a program requires the tool to understand the implementation details of the heterogeneity-aware programming paradigm (CUDA, OpenCL).

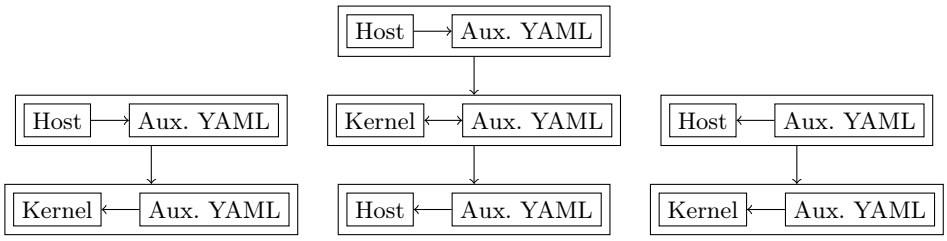
Our contribution In this work, we address the research question of what is the most effective way to obtain a mixed precision application in an heterogeneity-aware context, either automatically or semi-automatically. More specifically, we:

1. prove that compiler-based automatic and semi-automatic approaches to precision tuning can be applied in multi-source file applications;
2. analyse the benefits of applying precision tuning to the accelerator code with respect to the joint combination of host code and accelerator code;
3. discuss the impact of precision tuning on the data transfer overhead and on the data processing costs;
4. provide a proof of concept implementation of semi-automatic multi-source precision tuning framework for accelerator-aware programming paradigms;
5. assess our solution on two different accelerator-aware programming paradigms, using two different runtime environments.

To this end we introduce a new methodology based on TAFFO, called *Delayed Analysis* (DA). This new methodology allows TAFFO to perform precision tuning in a heterogeneous context where GPGPUs are involved. The choice of TAFFO, in contrast with existing tools, allows both the GPGPU and host code to be converted to exploit mixed precision, easing the programmer's workload. Using TAFFO and the Polybench/ACC [7] benchmark suite, we evaluate the time-to-solution and error figures of mixed-precision applications, comparing optimization of the entire program and optimization of the kernel alone. By optimizing the entire program we achieve speedups exceeding 2×, with a minimal impact on the error for most benchmarks, while optimizing the kernel alone limits the speedup to at most 1.19×.

2 Methodology for GPGPU Precision Tuning

Our solution for precision tuning with GPGPUs exploits the well-established TAFFO framework [3,6]. The five pipeline stages of the TAFFO architecture are called *Initializer* (INIT), *Value Range Analysis* (VRA) *Data Type Allocation* (DTA), *Conversion* (CONV), and *Feedback Estimator* (FE). 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 ILP-based technique [1]. 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 [2]. FE statically analyses the error using state-of-the-art estimation methods [5]. The design of TAFFO makes it independent from the source language as well as easy to expand.



(a) Open Loop Mode (b) Closed Loop Mode (c) User Controlled Mode

Fig. 1: Illustration of the different use cases for Delayed Analysis in TAFFO. 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 file used by DA.

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, 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 compilation unit with other variables or buffers present in another compilation unit. If the relationships between compilation 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 compilation 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 compilation units.

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 file which is then read and updated by subsequent passes of TAFFO. 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 [1], 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. These three modes are illustrated in Figure 1. Other modifications were required 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.

2.1 Comparison with the state-of-the-art

Our approach to GPGPU automated mixed-precision tuning bears the most resemblance to the one presented in *GPUMixer*[8]. 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.

3 Experimental Evaluation

In order to demonstrate the effectiveness of our approach for automated mixed precision computation in a GPGPU environment, we evaluate our solution on the Polybench/ACC benchmark suite [7]. Polybench/ACC provides implementations of the same set of kernels for both OpenCL and for CUDA which exploit the best programming practices for both APIs. It also provides CPU-based implementations which were disabled for the purpose of this evaluation. All comparisons are performed between GPGPU-based implementations.

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

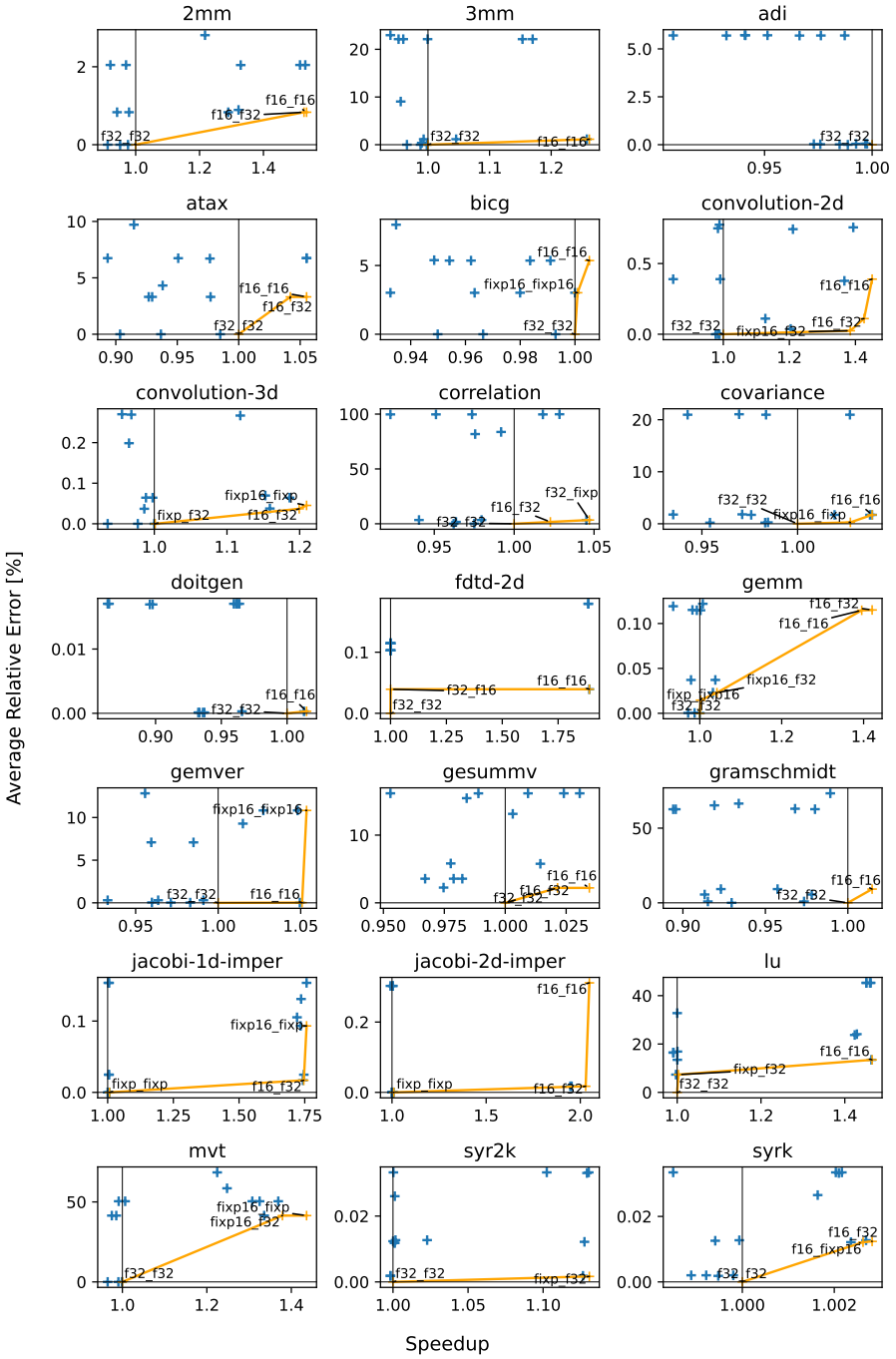


Fig. 2: Error/Speedup graphs of the benchmarks in Polybench/ACC when optimized by TAFFO in different configurations on the OpenCL machine. Each configuration is shown as a blue cross. Configurations which are Pareto-optimal are shown in orange and are connected by lines.

Table 1: Comparison of the speedup and the error between exposing the *float* type or the *half* type to the host. The kernel always performs the computation in *half* precision.

Benchmark	ARE		OpenCL Speedup		CUDA Speedup	
	float	half	float	half	float	half
2mm	0.8 %	0.8 %	0.98	1.54	1.02	1.88
3mm	1.1 %	1.1 %	0.99	1.26	1.01	1.21
adi	0.0 %	0.1 %	0.99	1.00	0.96	1.01
atax	3.3 %	3.3 %	0.98	1.06	1.00	0.99
bicg	5.4 %	5.4 %	0.98	1.01	0.99	0.98
convolution-2d	0.4 %	0.4 %	0.85	1.45	1.00	1.36
convolution-3d	0.1 %	0.1 %	1.00	1.19	1.00	0.87
correlation	3.4 %	3.4 %	0.98	1.05	0.99	1.04
covariance	1.7 %	1.7 %	0.98	1.04	0.98	1.06
doitgen	0.0 %	0.0 %	0.97	1.01	1.01	1.02
fdtd-2d	0.0 %	0.0 %	1.00	1.89	1.00	1.76
gemm	0.1 %	0.1 %	1.00	1.40	1.19	1.36
gemver	0.0 %	0.0 %	0.98	1.05	0.99	1.06
gesummv	2.3 %	2.2 %	0.97	1.03	0.95	0.98
gramschmidt	9.2 %	9.2 %	0.96	1.01	0.98	1.13
jacobi-1d-imper	0.0 %	0.0 %	1.01	1.75	1.04	1.06
jacobi-2d-imper	0.3 %	0.3 %	1.00	2.05	1.10	1.82
lu	13.5 %	13.5 %	1.00	1.47	0.99	1.32
mvt	50.4 %	50.4 %	0.99	1.32	1.00	1.01
syr2k	0.0 %	0.0 %	1.02	1.00	1.00	1.00
syrk	0.0 %	0.0 %	1.00	1.00	1.00	1.00

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, 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 time-to-solution is measured by the test fixtures included in Polybench/ACC, which also measure 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.

3.1 Analysis of the results

We show the results of our experiments in Figure 2, and in Table 1. In particular, Figure 2 shows the speedup and Average Relative Error (ARE) in percentage of all the configurations and all the benchmarks in the Polybench/ACC suite. For brevity, the figure only shows the data for the OpenCL machine, but the data for CUDA is similar. The figure highlights and labels the configurations that are Pareto-optimal with respect to the speedup and the error.

Table 1 compares the speedup and the error when the buffers exposed to the host code are reduced precision (specifically employing the *half* data type) or not (employing the *float* data type). In both cases the kernel performs all computation in the *half* data type. 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*. This is particularly evident in Figure 2 as the progression of the Pareto frontier follows the decrease in size of the data types selected. 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.

It can also be observed that smaller 16-bit data types also provide a speedup when they are employed simply for data storage. This can be seen in the case of *convolution-3d*, which by exploiting 16-bit-sized buffers achieves a speedup of about 1.2× with a very small error. This is conclusively demonstrated in Table 1, where it is evident that most of the speedup would not appear if the kernels also converted the data from reduced precision to single precision (*float*). We believe that most of the speedup is due to time saved when transferring the benchmark data to and from GPGPU-exclusive memory, as the amount of bytes to transfer is reduced to a similar ratio to the speedup. The variation amongst different benchmarks is due to the fact that some benchmarks cannot use reduced precision data types for some or all of the buffers because the range of the values contained in the buffers are not representable with those types.

With respect to the error, amongst 16-bit data types, half-precision floating point appears to behave better than 16-bit fixed point, as it delivers similar speedups with a lesser impact on the error. This is again visible from Figure 2, as the highest errors are always observed when exploiting 16-bit fixed point, the most extreme case being the *lu* benchmark 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 amplification of the quantization error due to the high dynamic range of the results and the long chains of multiplications performed. This is not a solvable issue without changing the algorithm employed by the benchmark and shows the intrinsic limitations of a compiler-based tool. 32-bit fixed point types appear to not provide any significant benefits over any 16-bit data type for most of the benchmarks, except for *convolution-3d*, *jacobi-1d-imper* and *mvt*.

4 Conclusion

We introduced the Delayed Analysis methodology for the TAFFO precision tuning LLVM plugins, which enables TAFFO to address the problem of automated Mixed Precision in GPGPU architectures. With the proposed approach, TAFFO supports the tuning of host and kernel codes written in both OpenCL and CUDA,

enabling precision tuning to be employed to leverage half-precision floating point data types as well as integers. Speedups can be obtained from both reduced computation and data transfer times, at a limited cost in accuracy. Through the utilization of TAFFO we also analyze the benefits of applying precision tuning on accelerator code, finding that the data transfer overhead is greatly reduced by applying reduced precision. Future works include the extension of TAFFO to support bfloat16 types, which currently are not fully implemented in LLVM backends for GPGPUs. Beyond GPGPUs, there is space for Mixed Precision computing in a variety of accelerator architectures, such as application specific accelerators for AI as well as reconfigurable architectures. The ability to customize the architecture can open interesting opportunities for a co-design approach, where the computation precision is tuned with greater freedom, while minimizing hardware area by entirely removing support for unused (wide) data types.

Acknowledgements This work is partially supported by the European Commission and the Italian Ministry of Economic Development (MISE) under the EuroHPC TEXTAROSSA project (G.A. 956831). The authors gratefully acknowledge funding from European Union's Horizon 2020 Research and Innovation programme under the Marie Skłodowska Curie grant agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimisation, http://www.apropos.eu/).

References

1. Cattaneo, D., et al.: Architecture-aware precision tuning with multiple number representation systems. In: 2021 58th ACM/IEEE Design Automation Conference (DAC). pp. 673–678 (2021). https://doi.org/10.1109/DAC18074.2021.9586303
2. Cattaneo, D., et al.: 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
3. Cattaneo, D., et al.: Taffo: The compiler-based precision tuner. SoftwareX **20**, 101238 (2022). https://doi.org/10.1016/j.softx.2022.101238
4. Cherubin, S., Agosta, G.: Tools for reduced precision computation: a survey. ACM Computing Surveys **53**(2) (Apr 2020). https://doi.org/10.1145/3381039
5. Cherubin, S., et al.: Dynamic precision autotuning with TAFFO. ACM Transaction on Architecture and Code Optimization **17**(2) (may 2020). https://doi.org/10.1145/3388785
6. Cherubin, S., et al.: TAFFO: Tuning assistant for floating to fixed point optimization. IEEE Embedded Systems Letters **12**(1), 5–8 (March 2020). https://doi.org/10.1109/LES.2019.2913774
7. Grauer-Gray, S., et al.: Auto-tuning a high-level language targeted to gpu codes. In: 2012 Innovative Parallel Computing (InPar). pp. 1–10 (2012). https://doi.org/10.1109/InPar.2012.6339595
8. Laguna, I., et al.: Gpumixer: Performance-driven floating-point tuning for gpu scientific applications. In: High Performance Computing. pp. 227–246. Springer, Cham (2019)
9. Stanley-Marbell, P., et al.: Exploiting errors for efficiency: A survey from circuits to applications. ACM Computing Surveys (CSUR) **53**(3), 1–39 (2020)