SeTHet – Sending Tuned numbers over DMA onto Heterogeneous clusters: an automated precision tuning story

Gabriele Magnani
gabriele.magnani@polimi.it
Politecnico di Milano
Milan, Italy

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

Lev Denisov
lev.denisov@polimi.it
Politecnico di Milano
Milan, Italy

Giuseppe Tagliavini
giuseppe.tagliavini@unibo.it
University of Bologna
Bologna, Italy

Giovanni Agosta
giovanni.agosta@polimi.it
Politecnico di Milano
Milan, Italy

Stefano Cherubin
stefano.cherubin@ntnu.no
NTNU
Trondheim, Norway

ABSTRACT

Energy and performance optimization of embedded hardware and software is of critical importance to achieve the overall system goals. In this work, we study the optimization of memory access through a combination of hardware (Direct Memory Access, DMA) and software (Precision Tuning) techniques, and we propose a compiler toolchain for managing both in the context of heterogeneous RISC-V-based platforms. Our proposed toolchain, SeTHet, enables $3-48\times$ speedup over the baseline system when employing both DMA and precision tuning, regardless of the availability of floating point units in hardware. SeTHet also achieves up to $16\times$ speedup compared to DMA alone, thus proving that the combination of the two techniques provides a major improvement over either technique employed in isolation.

CCS CONCEPTS

• **Software and its engineering** → *Compilers*; *Software performance*; • **Computer systems organization** → *Heterogeneous (hybrid) systems*.

KEYWORDS

computer architecture, RISC-V, approximate computing, precision tuning

1 INTRODUCTION

The increasing compute power requirements of modern applications, coupled with the end of traditional technology scaling, i.e., Moore's Law and Dennard scaling, are pushing both high-performance and embedded systems towards higher degrees of heterogeneity [20]. Parallel heterogeneous systems have challenges of their own. In particular, as the number of processing elements increases in a shared memory system, the contention for memory access can lead to severe performance bottlenecks. When shared memory becomes unsustainable, an efficient data movement mechanism between different memory partitions becomes a critical factor in obtaining performance.

It is widely acknowledged that in the last decades the performance of memory systems did not scale as much as in computing systems, and memory systems became bottlenecks in more and more application domains [25]. To overcome this problem, countless approaches have been proposed. In this paper, we address the interaction between two of these approaches: bypassing the control unit for some data movements via Direct Memory Access (DMA) and changing the binary representation of real numbers via Precision Tuning. The first one aims at reducing the control flow overhead for data movement, while the second one aims at optimising data size to decrease data processing and data movement costs.

A DMA engine that manages autonomous data transfers without involving the CPU is fundamental in heterogeneous architectures, where memories are physically distributed into partitioned areas [30]. Moreover, using software-managed memories instead of data caches as last-level memories is a common design technique for embedded heterogeneous systems to improve energy efficiency and prevent the scalability limitations of coherency protocols. DMA engines enable high-bandwidth data transfers and reduce access latency (i.e., CPU cores simultaneously operate on local data).

Precision tuning is the process of maximising the efficiency of the computation while ensuring the quality of the output does not degrade below an application-dependent acceptance threshold. It is one of the main branches of Approximate Computing [31], and it mitigates the cost of data movement by reducing the number of bytes to be moved. However, this trade-off requires the coordination of precision tuning across different heterogeneous subsystems, which is non-trivial and mostly unexplored in literature.

In this work, we show how DMA orchestration and Precision Tuning can synergize to achieve consistent speedups across various micro-architectural solutions – i.e., when dealing with platforms that have (or lack) single or double precision floating-point arithmetic hardware support. While still beneficial on their own, the two techniques achieve less consistent results when used in isolation, requiring the designer to be more careful and reducing the performance portability.

To achieve this result, we introduce as a main contribution the design of SeTHet, a compiler toolchain for orchestrating both DMA and precision tuning at the host and at the accelerator side in heterogeneous parallel applications, thus bridging a gap in the state-of-the-art. We assessed our findings on a RISC-V-based open hardware platform, HERO [21], achieving 3.5-48.4$\times$ speedup over the baseline HERO system when employing both DMA and precision tuning, regardless of availability of hardware floating point units, against a range of 1.2-42.7× achieved with DMA alone.

The rest of this paper is organized as follows. Section 2 analyzes the state-of-the-art and highlights the current gap. Section 3 describes the background of our work. Section 4 introduces our solution. Section 5 describes the experimental campaign and its results. Finally, in Section 6, we draw some conclusions and highlight future research directions.

2 RELATED WORK

Notable related works oriented to memory systems include the exploration of different floating point data types for storage efficiency in GPU accelerators. In particular, [2] proposes a solution leading towards an improved memory layout at the cost of a slowdown in memory access, which is expected in the case of software-defined data extraction. In our work, we share the same approach by including the conversion within the compiler, yet we extend this approach to heterogeneous clusters where the architecture supports the reduced precision data type, and no slowdown is expected.

A higher-level approach to precision tuning on GPU code has been proposed before in the form of replacing the keywords representing data types in the accelerator code [15, 19, 22, 27]. These approaches, however, are limited by the toolchain support of such data types. Another limitation relates to the coherence between the host and kernel code, as they only operate on the accelerator side of the application. Other approximate computing techniques have been applied in HPC and embedded systems environments, yet their study has always been limited to the individual contribution provided by a single technique. In contrast, combining them with other memory-saving approaches such as DMA has not been implemented before [26, 28]. It is important to highlight that the DMA engine is a key component available in all modern heterogeneous architectures. In particular, it is widely used in systems that comprise memory hierarchies requiring explicit transfer management [32]. These systems expose a low-level software interface that SETHET can use to perform DMA transfers.

Precision tuning techniques have been extensively applied to high-performance scientific computation at the source code level [9, 11, 19], and at the compiler level [8, 16, 23], and to embedded and low-power systems workloads [6]. Most of these works aim to minimise the number of bits used in each software-defined variable without going forward to investigate the actual impact on memory access performances. A more in-depth analysis of recent works in precision tuning can be found in [7]. Our work aims to optimize memory management to be as close as possible to the hardware.

We take inspiration from compiler-level and embedded systems frameworks, the most relevant of which is the TAFFO framework [4, 5]. In particular, TAFFO is a precision tuning tool that manipulates the intermediate representation provided by the LLVM compiler framework (LLVM-IR). Conveniently, it is designed as a sequence of modular compiler plug-ins that only loosely depend on the surrounding compiler optimisations. In this framework, the programmer annotates the source code with hints about the input values, and the compiler passes propagate them accordingly. This structure enables us to adapt this framework for different and diverse LLVM-based compilation toolchains. An additional advantage of TAFFO is its preexisting partial support for parallel and distributed computing via OpenMP acceleration [24]. However, this support is currently limited to homogeneous platforms and requires non-trivial extensions to handle heterogeneous accelerators. We designed such extensions, and Section 4 describes them.

3 RISC-V HETEROGENEOUS PLATFORM

Heterogeneous computing systems employ multiple classes of processing elements, each with different architectures, capabilities, and purposes. Combining general-purpose processors and parallel accelerators is a well-established heterogeneous system design. In this context, RISC-V [18] is an open standard instruction set architecture (ISA) that offers several advantages to designing heterogeneous systems. These advantages include ISA modularity, customization capabilities by design, community support, reduced licensing costs, and its role in fostering innovation and research thanks to its open-standard nature.

The heterogeneous computing system that we target in this work is HERO [21], a platform combining a 64-bit host processor (ARMv8 or RISC-V) executing a Linux environment with a bare-metal programmable many-core accelerator that includes multiple clusters of 32-bit RISC-V cores. Each cluster can include a configurable number (from 4 to 16) of OpenHW CV32E40P [14] cores, supporting the RV32IMA instruction set architecture (ISA) and the XpulpV2 ISA extension [12].

The HERO memory hierarchy is partitioned between the host and accelerator, with a memory management unit (MMU) guaranteeing memory coherency. The accelerator side consists of a 128 KiB level-1 (L1) scratchpad memory (SPM) for data (one per cluster), a 4 KiB L1 instruction cache (one per cluster), and a 512 KiB shared level-2 (L2) SPM. The main memory is the last hierarchy level (L3) and is shared between the host and the accelerator. Each cluster has a dedicated DMA engine to move data between the L1 and the L2/L3 levels, supporting multiple concurrent operations with a bandwidth of up to 1024 bits per clock cycle.

The HERO software stack includes an OpenMP 4.5 runtime [3] and an LLVM 12 heterogeneous compiler toolchain. To offload code to the accelerator, programmers annotate a block with a `#pragma omp target` directive, defining an *accelerator kernel* that the compilation flow compiles for both targets (64-bit host and 32-bit accelerator). The *Clang RV32 frontend* has been extended to assign different address spaces based on OpenMP `target` annotations, using the standard `addrspace` attribute. Pointers passed to an accelerator kernel are 64-bit wide and are associated with the host address space. All the other pointers referenced by the accelerator kernel are associated with the 32-bit *native address space*. After completing the compiler stages and executing the assembler for both flows, the Clang driver embeds the accelerator ELF files inside the host one, creating a so-called *fat binary*.

4 THE SETHET SOLUTION

Our solution derives from the combination of distinctive software elements, adapted and tailored for the specific needs and challenges typically present in clusters of heterogeneous computing architectures. Figure 1 depicts a block diagram of the software components

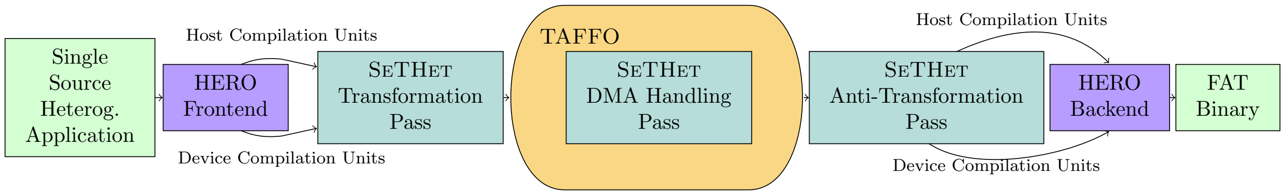


Figure 1: Block diagram of the SeTHet toolchain. The compilation process follows two pipelines for each single-source heterogeneous application, one for the host (top) and one for the accelerator device (bottom). The green elements represent the observed input and output of the toolchain. The purple elements represent HERO components, the azure elements represent SeTHet components, while the yellow element represents the precision tuning steps

our pipeline consists of. In this section, we first present these components, the reason why we selected them, and their role in the bigger picture of SeTHet.

4.1 Revisiting the Precision Tuning Workflow

The TAFFO framework already comes with partial support for OpenMP, including the *parallel*, *for*, *section*, and *critical* constructs [24]. However, this subset is insufficient for addressing heterogeneous devices since it lacks support for cross-device task mapping and related memory management. SeTHet introduces the support for code offloading using the `target` construct, with proper handling of the device selection (`device` clause) and the mapping between the device and target region (`map` clause).[1]

The *map* clause specifies which data should be copied to the target device before executing the offloading task and whether data must be copied back to the host after execution. Under the hood, the compiler extracts the target region, wraps it into a function, and moves its definition in the heterogeneous module; in the host code, the compiler replaces the extracted code region with a function call to the OpenMP runtime library `__tgt_target_mapper`. This runtime function requires the identifier of the device on which the code will run, a pointer to the function to offload, and a data structure containing the offloaded arguments. For the sake of generality, the compiler packs the arguments into an array of `void` pointers, and it generates – in the newly extracted function – a prologue fragment to restore the original types.

The code extraction and encapsulation pattern described above creates artificial boundaries in the data-flow and effectively requires the precision tuning framework to expand the scope of its analyses. To seamlessly integrate inter-procedural precision tuning analyses into the compilation pipeline, non-trivial extensions are required. However, these extensions are constrained by the scope of the translation unit they are contained in, while the OpenMP compiler effectively generates a distinct translation unit for each device that has code mapped onto it.

To tackle the challenges of this flow, we introduce a new approach to the precision tuning problem, which is OpenMP-aware.

As an intentional design choice, SeTHet wraps the core logic of an inter-procedural precision tuning framework – TAFFO– and avoids *ad hoc* specialization of each precision tuning step. More specifically, SeTHet introduces a *Transformation* pass in which the intermediate representation the HERO toolchain is based on is translated into a single LLVM-IR module, and a *Anti-Transformation* pass – run after the precision tuning stage – that is in charge of restoring the precision-tuned fragments in the toolchain flow. These two additional steps mark the entry and exit point of the precision tuning component in Figure 1. Furthermore, we extended the compiler support for precision tuning to preserve data layout and address space information in typed-pointer LLVM-IR values.

4.2 Transformation

The Transformation pass is a pre-processing stage aimed at reshaping the code and its metadata related to precision tuning such that the later stages of SeTHet could access information encapsulated by OpenMP offloading patterns. More specifically, the design of this pass aims to solve three problems: (a) exchange of precision-tuning analyses results between two different LLVM modules based on different architectures; (b) OpenMP type-punning due to storing all arguments inside a `void*` array; and (c) handling of an indirect call with arguments encapsulated within an array.

The problem (a) can be solved by searching for each function call to `__tgt_target_mapper` within the host module. We rely on the host function pointer parameter of this call to locate the external function definitions within the accelerator module, and then clone them back into the host module. Right before performing this cloning stage, we analyse each function and schedule for cloning any function and global values they might depend on. To avoid name collision with existing elements of the host module, appropriate unique prefixes are added to the identifiers of the cloned functions and global values.

The problem (b) is solved by analyzing each argument stored in the array of void pointer and their *Def-Use* chains for each heterogeneous region. Thanks to LLVM-IR being a strongly-typed language, it is often possible to infer the data type of a value from its previous uses or definitions. However, it is not always possible to infer the original type solely from the caller's side since the argument can be a `void`-typed allocation generated through a call to `malloc`. In such a case, the data-flow inference is expanded to cover the callee and its prologue inserted by the compiler to ensure all parameters have an associated data type for precision tuning purposes.

Figure 2 reports the code generated by Listing 1. The red arrows starting from the arguments of `__tgt_target_mapper` and of the offloaded function display the Def-Use chain traversed by SeTHet to infer the information on data types. In this example, the

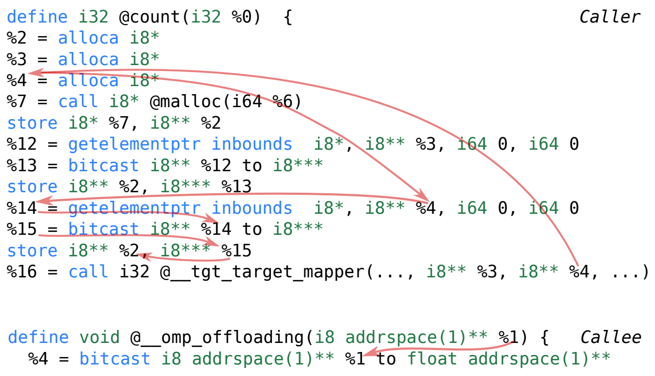
[1]Throughout this article, we assume explicit device mapping via `map` clauses. However, SeTHet also supports the selection of a default device via the standard `omp_set_default_device`. This difference has no impact on the methodology of our proposed solution.

```c
int count(int size) {
  void* value = malloc(sizeof(float) * size);
  #pragma omp target map(tofrom : value)
  {
    for (int i = 0; i < size; i++) {
      ((float*)value)[i] = (float)i;
    }
  }
  return 0;
}
```

Listing 1: Simple C code used as reference for demonstrate how type punning resolution works in Figure 2.

analysis on the caller side identifies the first element of the array as `i8**`, while the analysis on the callee side determines a `float addrspace(1)**`. Any conflict between the results of the caller and callee analyses on the same argument is resolved according to the strict type aliasing rules of C/C++.

Finally, problem (c) is handled by temporarily removing the indirection and replacing it with a direct call to the cloned version of the function. This change is applied after each corresponding `__tgt_target_mapper`, and its arguments are derived via the type-punning analysis from step (b). The direct function call pattern is transparently recognised by TAFFO and allows for inter-procedural precision tuning analyses and optimisations. To comply with the constraints of our reference OpenMP implementation, the indirect call is restored right after the precision tuning process – see Subsection 4.4 for details.



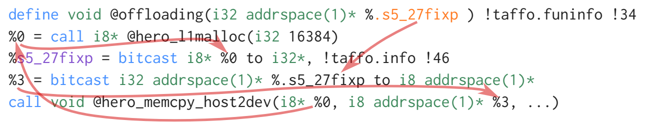
```
define i32 @count(i32 %0)  {                              Caller
%2 = alloca i8*
%3 = alloca i8*
%4 = alloca i8*
%7 = call i8* @malloc(i64 %6)
store i8* %7, i8** %2
%12 = getelementptr inbounds  i8*, i8** %3, i64 0, i64 0
%13 = bitcast i8** %12 to i8***
store i8** %2, i8*** %13
%14 = getelementptr inbounds  i8*, i8** %4, i64 0, i64 0
%15 = bitcast i8** %14 to i8***
store i8** %2, i8*** %15
%16 = call i32 @__tgt_target_mapper(..., i8** %3, i8** %4, ...)


define void @__omp_offloading(i8 addrspace(1)** %1) {    Callee
  %4 = bitcast i8 addrspace(1)** %1 to float addrspace(1)**
```

Figure 2: LLVM-IR generated by Listing 1, simplified to show only the relevant part to illustrate the type-punning resolution. The red arrows display the Def-Use chain traversed by SeTHet to restore the information on types.

4.3 DMA-Aware Precision Tuning

To enable the precision tuning across HERO's memory transfer functions, the precision tuning component requires special handling of the DMA APIs. DMA memory transfer within HERO is performed using compiler intrinsics that have the same syntax and arguments as the *memcpy* in C/C++. These intrinsics come in two flavours: synchronous and asynchronous. Both versions enable the user to transfer memory from a source pointer within the host or device to a destination within the device or host. The asynchronous variant facilitates double buffering algorithms that allow overlapping the computation of an iterative code region with the memory transfer needed for the next iteration.



```
define void @offloading(i32 addrspace(1)* %.s5_27fixp ) !taffo.funinfo !34
%0 = call i8* @hero_l1malloc(i32 16384)
%s5_27fixp = bitcast i8* %0 to i32*, !taffo.info !46
%3 = bitcast i32 addrspace(1)* %.s5_27fixp to i8 addrspace(1)*
call void @hero_memcpy_host2dev(i8* %0, i8 addrspace(1)* %3, ...)
```

Figure 3: LLVM-IR analysed when DMA resolutions happen. The red arrows display the Def-Use chain traversed by SeTHet to restore the information on types. The ranges are propagated from the orange arguments .s5_27fixp to the purple local variable s5_27fixp through the DMA memory transfer functions *hero_memcpy_host2dev*.

In TAFFO, a Value Range Analysis (VRA) pass is responsible for computing and propagating the information from the programmer's annotation on the variable through all intermediate values. When the VRA encounters a call to a function that is not defined within the module under analysis, it uses a catalog of well-known functions – e.g., mathematical functions, memory allocation, and memory copy functions – to modify the range of its output value and arguments. Consistently with the catalog-based approach, SeTHet provides the semantics of the HERO DMA intrinsics to extend the capabilities of the TAFFO VRA. In particular, the expected behaviour for the VRA when encountering these DMA APIs is to propagate the range from the source pointer to the destination pointer, as illustrated in the example in Figure 3. Additionally, new constraints are artificially created in the precision tuning algorithm to guarantee consistency with the maximum range of represented numeric values in each device. This is especially useful since the host and the device architectures have different size for the integer and floating point registers. In the case of HERO platform, the widest integer registers are 64-bit wide for the host and 32-bit wide for the device while the widest floating point registers are homogeneously implemented as 64-bit wide.

4.4 Anti-Transformation

After the Conversion pass of TAFFO, converting the annotated floating points to fixed points, the Anti-Transformation pass comes into play. Its role is to revert the output generated by the Conversion pass to the original structure before Transformation. This step is necessary because the HERO toolchain cannot work directly with the output generated by the Conversion pass of TAFFO. Two different modules are required to ensure compatibility with HERO. One module contains the code of the host, while the other one provides the code for the accelerator. Additionally, calls to the OpenMP API must be restored to their original form with indirect calls. For this reason, SeTHet schedules the Anti-Transformation pass after the code manipulation stage of the precision tuner framework. During this pass, SeTHet copies back all the newly converted functions and globals previously imported from the accelerator module. It restores all OpenMP indirect calls to newly generated regions exported to heterogeneous modules, as before the modification described in Subsection 4.2. Finally, any reference to the heterogeneous module is removed from the host module.

5 EXPERIMENTAL CAMPAIGN

Our experimental campaign is tailored to assess the benefits that SeTHet brings on top of well-known optimisation techniques. In particular, we aim to show that our solution improves performance portability with respect to any of its components – DMA and precision tuning – considered individually.

5.1 Experimental Setup

Benchmark Suite. We selected PolyBench-ACC [13] to test our work. This benchmark suite contains various numerical kernels with static control flow from different application domains. Our reference architecture – HERO– already supports several applications from the PolyBench-ACC suite. More specifically, the supported kernels are: **2mm**, **3mm** multiplication between 2 and 3 matrices; **atax** matrix transpose and vector multiplication; **bicg** sub Kernel of BiCGStab; **covariance** covariance computation; **gemm** matrix multiplication; **convolution-2d** convolution computation

Benchmark Configurations. These benchmarks have been adapted by HERO developers to offload the kernel workload onto the heterogeneous device using the DMA, as described in Section 3. We have developed a new version of the benchmarks that are structurally the same but without using DMA. More precisely, we have modified the benchmarks to replace DMA memory transfer primitives in the code with functionally equivalent CPU-managed memory transfers. In our campaign, we compare both data transfer modes, and we denote them as `noDMA` and `DMA`. We augmented these two versions with the standard annotation syntax required by the TAFFO precision tuning framework. Having the RISC-V accelerator device limited to 32-bit integers, we configure a limit within the precision tuner to control the maximum fixed-point bit-width, thus generating two benchmark configurations: `32` and `64`, respectively, capping the bit-width of fixed point computation to 32-bit and to 64-bit. An additional configuration parameter comes from the possibility in the HERO architecture to enable or disable the hardware implementation of a floating point unit in the accelerator device. As such, we tested each benchmark both with hardware floating point support (`HARD`) and with software emulation (`SOFT`). Therefore, for each benchmark, we have run several versions provided by the Cartesian product of $\{\texttt{DMA},\texttt{noDMA}\} \times \{\texttt{32},\texttt{64}\} \times \{\texttt{SOFT},\texttt{HARD}\}$.

Hardware & System Setup. We run all benchmarks on a Xilinx ZCU102, a Xilinx Zynq UltraScale+ MPSoC with a 64-bit ARMv8 quad-core Arm® Cortex®-A53 processor. In the programmable logic (PL), HERO instantiates a single cluster of 8 cores. PetaLinux version 2019.2 was used to configure binary bootable images loaded on the board and provide a compatible Linux OS to interact with.

Metrics. To assess our solution, we observe one performance metric and one quality metric across multiple configurations. In this work, we consider improvements in time-to-solution to be the most relevant performance metric and the numeric relative error to be the computational quality metric. We mitigate the effect of background noise in time measurements – mostly due to the presence of an OS – by averaging the time measurement readings over 100 runs for each benchmark configuration. The timing was measured using the OpenMP function `omp_get_wtime()`, which returns the elapsed wall clock time measured in seconds. The time returned is specific to each thread and is a standard metric for benchmarking. Time improvements are reported as speedup, following the well-known formula $S = \frac{T_{ref}}{T_{entry}}$. Each speedup entry is validated using the clock cycle count provided by the offloading runtime support. All the speedups are relative to the time-to-solution T_{ref} of the baseline benchmark configuration featuring no precision tuning and using `noDMA` data transfer mode. The relative error for each precision tuning experiment is the relative distance of the output from a golden reference result obtained by the same code when compiled on the same HERO architecture and with no precision tuning enabled. Calling B the golden reference value and P the precision tuning ones, their relative error is computed with the well-known formula $\frac{|P-B|}{B}$. As per previous research on approximate computing, we have set the acceptable error threshold to be as high as 10%. [29]. In accordance with the original definitions of the benchmarks, our golden reference relies on IEEE-754 *binary32* floating point standard [1].

5.2 Preliminary Analyses

At first, we experimentally assess the effects of `DMA` and precision tuning as individual improvements over the same baseline. Based on these analyses, we sketch a naïve prediction of their combined effort under the hypothesis of non-correlation between the two contributions. Later in this paper, we discuss such a hypothesis and comment on its validity.

DMA Contribution. To single out the contribution of `DMA` alone in the HERO platform, we start from the `DMA` benchmarks and adapt them to achieve a functionally equivalent `noDMA` version that still includes memory transfer costs as described in section 5.1.

Table 1 shows the speedup achieved when the `DMA` is enabled. The speedup varies significantly between `SOFT` and `HARD` floating point configurations because the former requires more compute time to emulate the floating point support. The savings in memory transfer time are proportionally larger on the time-to-solution for `HARD` floating point configurations.

It should be noted that **atax**, **bicg**, and **convolution** have a significantly higher speedup compared to other benchmarks. This is because they benefit the most from the use of *async DMA* memory functions that allow for significant overlap of computation and memory transfer via the double buffering approach, which is not possible with the serial *sync DMA*.

Since the memory transfer primitives have no impact on the functional behaviour of the code, the computation quality metric is not impacted, retaining the best level available – 0 distance from the golden reference – in every configuration.

Precision Tuning Contribution. We consider the precision tuning contribution as the improvement on the time-to-solution obtained by the precision tuning component, as described in Section 4, without the use of DMA memory transfer primitives. The speedup obtained is reported in Table 2, where there is a clear distinction in speedup values between `SOFT` and `HARD` floating point configurations. This time, `SOFT` ones find more benefits from this optimisation.

Indeed, while precision tuning may improve both memory transfer and computation cost, the use of fixed-point arithmetic largely simplifies the handling of `SOFT` floating-point computation.

We have also measured the quality metric of Precision Tuning. Figure 4 presents two separate subplots reporting the relative error,

Table 1: Speedup achieved by `DMA` with respect to the baseline solution with `noDMA` memory transfer and no precision tuning applied. A colour representing the magnitude of the value is overlaid to aid in visualizing the speedup difference. Average is the geometric mean of all the reported benchmark and partitioned by configuration.

Benchmark	Configuration	DMA
2mm	SOFT	1.48
	HARD	6.23
3mm	SOFT	1.26
	HARD	3.79
atax	SOFT	8.75
	HARD	42.7
bicg	SOFT	6.32
	HARD	28.9
convolution	SOFT	2.49
	HARD	20.9
covariance	SOFT	1.52
	HARD	6.08
gemm	SOFT	1.22
	HARD	3.82
Average	SOFT	2.41
	HARD	10.51

one for the `32` configurations and the other for the `64` configurations due to the difference in magnitude. The plot on the left shows the relative error for `32` configurations , which spans between $1 \cdot 10^{-5}$; $8.8 \cdot 10^{-2}$. On the right, the `64` configuration relative error goes from $1 \cdot 10^{-7}$; $1 \cdot 10^{-5}$. It is evident from the graph that the type of float supported by the hardware does not affect SETHET since the error generated is consistent across all configurations. The graph clearly illustrates that reducing the maximum integer size impacts on **atax** and **gemm** significantly. This effect is primarily because of the range of output and operations performed by these two. **atax** has an impressive output range of 0; $3.40 \cdot 10^6$, while **gemm**'s range spans from $-5.47 \cdot 10^4$; $5.34 \cdot 10^4$.

A Naïve Prediction. On the one hand, we have the `DMA` contribution that uniformly cuts data transfer time, most notably improving `HARD` floating point configurations. On the other hand, we have the precision tuning configuration, which significantly improves `SOFT` floating point configurations. Would it be possible to have the best of both? What speedup can we expect to achieve in each configuration? A naïve prediction can be obtained by multiplying the individual contributions of `DMA` and precision tuning. This prediction is valid under the assumption of independence of the two contributions. In our case, this hypothesis is true if: i) `DMA` primitives do not alter the scope and the behaviour of the precision tuning process; AND ii) precision tuning does not alter the `DMA` data transfer process.

5.3 Experimental Evaluation

In this Subsection, we demonstrate that the combination of both contributions produces dissimilar outcomes under a set of precise circumstances when compared to the previously described prediction, thus refuting the optimization-orthogonality hypothesis.

Table 2: Speedup achieved by SETHET leveraging only precision tuning with respect to the baseline solution with `noDMA` memory transfer and no precision tuning applied. A colour representing the magnitude of the value is overlaid to aid in visualizing the speedup difference. Average is the geometric mean of all the reported benchmarks and partitioned by configuration.

Benchmark	Configuration	Precision Tuning
2mm	SOFT 64	8.40
	HARD 64	1.02
	SOFT 32	8.61
	HARD 32	1.03
3mm	SOFT 64	8.22
	HARD 64	1.03
	SOFT 32	8.86
	HARD 32	1.04
atax	SOFT 64	5.12
	HARD 64	1.03
	SOFT 32	5.39
	HARD 32	1.04
bicg	SOFT 64	4.37
	HARD 64	1.03
	SOFT 32	4.67
	HARD 32	1.03
convolution	SOFT 64	5.86
	HARD 64	1.04
	SOFT 32	7.88
	HARD 32	1.05
covariance	SOFT 64	7.02
	HARD 64	1.03
	SOFT 32	7.60
	HARD 32	1.04
gemm	SOFT 64	8.51
	HARD 64	1.02
	SOFT 32	9.77
	HARD 32	1.05
Average	SOFT 64	6.59
	HARD 64	1.03
	SOFT 32	7.32
	HARD 32	1.04

Testing the Prediction. The relationship between the precision tuning process and the `DMA` depends on the layout of the `DMA` being used. Figure 5 illustrates two common layouts used within a kernel loop: the serial layout and the double buffering layout. In the case of the serial layout, the **DMA [A]SYNC** transfer is the only component affected by both precision tuning and `DMA`. In fact, the precision tuning process can impact the amount of data that needs to be transferred. For instance, if a binary64 is mapped in a 32-bit integer, the total amount of memory to be transferred will be reduced by half. On the other hand, the double buffering layout exhibits a similar kind of interaction within the **DMA ASYNC** transfer. Unlike the synchronous layout, the double buffering runs in parallel with the memory transfers for the next iteration with the loop body. In this case, the interaction between the `DMA` and the precision tuning

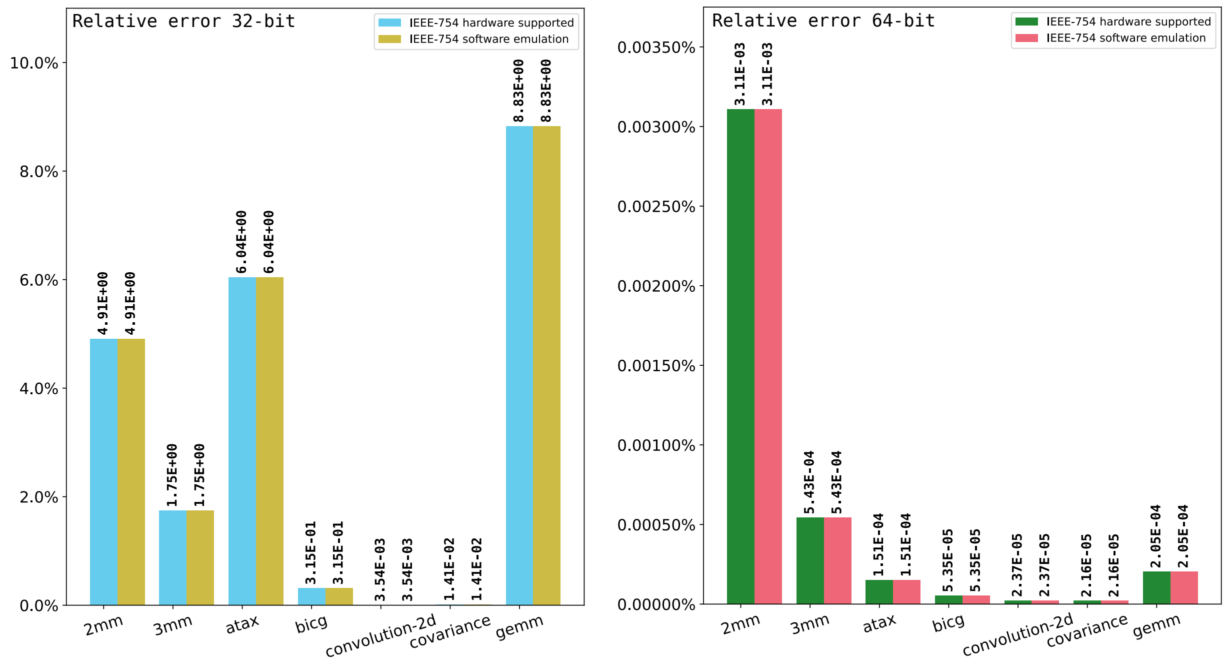


Figure 4: Relative error of SETHET solution compared to the baseline HERO. The graph on the left compares SETHET employing only 32-bit fixed-point numbers, while the graph on the right shows SETHET employing only 64-bit fixed-point numbers. The reference version for functional correctness is defined using floating point IEEE-754 binary32.

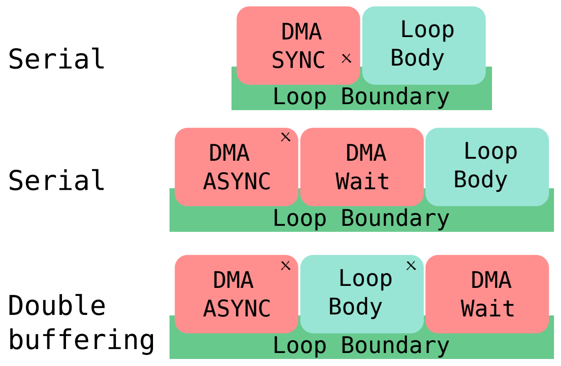


Figure 5: Common DMA layouts that are used in a kernel loop. The top layout is the Serial layout, where all the memory transfer happens at the beginning of the loop. The bottom layout is the Double buffering layout, where the memory transfer of the next iteration runs parallel to the loop body. The blocks that are marked with an X are the ones that are influenced by both the DMA and the precision tuning process.

process is given by the race between the new optimized time of the loop body and the potentially modified **DMA ASYNC** transfer. The best performance scenario occurs when the precision tuning optimization is capable of reducing the time spent in the loop body, so that its execution time is smaller than the memory transfer time. On the other hand, the worst-case scenario arises when the precision tuning increases the amount of memory to transfer, such as in the case of binary32 to 64-bit fixed point, but the loop optimization cannot compensate for the difference. All benchmarks are implemented using the asynchronous DMA API.

Table 3 displays a comparison between the speedup of SETHET with DMA and the naïve prediction. From the table, it is evident that the naïve prediction performs well in the HARD configurations. In these scenarios, optimizing the loop body doesn't adequately reduce its running time to match the memory transfer time, so the two optimizations are orthogonal. Furthermore, in the <HARD, 64 > configuration, the speedup is sometimes even slower than when using the same benchmark without SETHET. This is because of the increase in memory transfer when converting from binary32 to 64-bit fixed point. On the other hand, the predictions fail to estimate the correct speedup in the SOFT configuration. In these cases, the measured value is always underestimated, indicating that the two optimizations in these configurations are not entirely independent. Cross-checking the **"Average"** values in Table 3 and Table 1 confirms that the contribution of the precision tuning is crucial in the SETHET solution, which outperforms DMA in 3 out of 4 configurations.

Performance Analysis. The only configuration challenging for SETHET is $<$ HARD, 64 $>$ – the platform has hardware support only for 32-bit data types; thus, the 64 have to be emulated on smaller arithmetic and logic units – where SETHET delivers comparable results to HERO DMA. Overall, SETHET's efficiency is remarkable in hardware designs without FPU, which is one of the possible hardware configurations of HERO, achieving a speedup of up to 20× in **gemm** $<$ SOFT, 32 $>$.

From Table 3 we see that **atax** and **gemm** behave differently from the others, so we investigate further. **atax** is the benchmark most impacted by DMA and **gemm** is the one most affected by SETHET. **atax** is significantly affected by DMA due to the presence in its kernel of two separate, very small loop bodies that need to operate on a large amount of data. This pattern creates a memory-bound loop body, which DMA greatly impacts. On the other hand, **gemm**'s performance goes from 1.22 $<$ SOFT,3264 $>$ in HERO DMA to 20.7 $<$ SOFT,32 $>$ when using SETHET, with a relative increment of $16\times$. **gemm** is made up of a single loop body that is large enough not to be memory-bound. Upon analyzing the assembly produced by SETHET on **gemm**, we discovered that some multiplication was optimized out. The **gemm** kernel involves multiplying an array of floating points by two constant floats, alpha and beta. SETHET converts both the arrays and constants to a common fixed-point representation and inserts the appropriate integer multiplication. However, while optimizing the code, the compiler can replace multiplication by constants with a combination of shifts and additions when the constants have only a few bits set to one. In both **covariance** and **convolution**, multiplication and division are simplified in a similar way. The quality metric of SETHET is reported in Figure 4 and is the same as applying precision tuning without DMA as discussed in Section 5.2. Examining Figure 4, we can observe that the maximum relative error for **covariance** and **convolution** is less than 0.004% implying that there is still room for further improvements on the time to solution, which can be achieved by reducing the data size cap from the current 32 bits to 16 bits or even lower.

6 CONCLUSION AND PROSPECTIVE

In this paper, we demonstrated how coupling DMA with Precision Tuning techniques provides a valuable synergy. Where the two individual techniques achieve respectively a speedup of $1.22-42.7\times$ and $1.02-9.77\times$, the combination of the two techniques achieves an $4.6-48.4\times$ speedup in all configurations of the target hardware platform. Comparing the adoption of DMA in HERO with the combined effect of SETHET, we can see that DMA alone can achieve speedups over $10\times$ only in 3 configurations out of 14, all of them requiring hardware FPUs, whereas SETHET achieves this result in 10 configurations out of 14. This synergy is significantly more valuable than the linear combination of the two techniques taken individually.

Future directions include the extension of the proposed technique beyond the standard floating point formats to include reduced precision floating point formats – e.g., bfloat [10] – as well as alternative representations of real numbers, such as Posits [17], to better fit the application requirements.

ACKNOWLEDGMENTS

The authors gratefully acknowledge Xilinx for providing the ZCU102 board employed in this work.

This work is supported in part by the European Commission and the Italian MIMIT through the KDT JU TRISTAN (G.A. 101095947) and ISOLDE (G.A. 101112274).

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/).

Table 3: Speedup achieved by SETHET with respect to the baseline solution with noDMA memory transfer and no precision tuning applied. A colour representing the magnitude of the value is overlaid to aid in visualizing the speedup difference. Average is the geometric mean of all the reported benchmarks and partitioned by configuration.

Benchmark	Configuration	Prediction	SETHET
2mm	SOFT 64	12.43	17.3
	HARD 64	6.35	6.68
	SOFT 32	12.74	17.8
	HARD 32	6.42	6.69
3mm	SOFT 64	10.36	12.26
	HARD 64	3.90	3.56
	SOFT 32	11.16	15.0
	HARD 32	3.94	4.23
atax	SOFT 64	44.80	45.6
	HARD 64	43.98	41.4
	SOFT 32	47.22	48.4
	HARD 32	44.41	43.9
bicg	SOFT 64	27.62	33.0
	HARD 64	29.77	28.7
	SOFT 32	29.47	33.9
	HARD 32	29.77	29.5
convolution	SOFT 64	14.59	28.0
	HARD 64	21.74	17.7
	SOFT 32	19.62	32.5
	HARD 32	21.95	20.5
covariance	SOFT 64	10.67	15.0
	HARD 64	6.26	6.11
	SOFT 32	11.52	16.2
	HARD 32	6.32	6.6
gemm	SOFT 64	10.38	16.3
	HARD 64	3.90	4.6
	SOFT 32	11.91	20.7
	HARD 32	4.01	4.85
Average	SOFT 64	15.91	21.59
	HARD 64	10.81	10.50
	SOFT 32	17.65	24.14
	HARD 32	10.93	11.34

REFERENCES

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[2] Alexandra Angerd, Erik Sintorn, and Per Stenström. 2017. A Framework for Automated and Controlled Floating-Point Accuracy Reduction in Graphics Applications on GPUs. *ACM Trans. Archit. Code Optim.* 14, 4, Article 46 (dec 2017), 25 pages. https://doi.org/10.1145/3151032

[3] OpenMP Architecture Review Board. [n. d.]. OpenMP 4.5 Complete Specifications. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[4] Daniele Cattaneo, Michele Chiari, Giovanni Agosta, and Stefano Cherubin. 2022. TAFFO: The compiler-based precision tuner. *SoftwareX* 20 (2022). https://doi.org/10.1016/j.softx.2022.101238

[5] Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. 2021. Architecture-aware Precision Tuning with Multiple Number Representation Systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 673–678. https://doi.org/10.1109/DAC18074.2021.9586303

[6] Daniele Cattaneo, Michele Chiari, Gabriele Magnani, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. 2021. FixM: Code Generation of Fixed Point

Mathematical Functions. *Sustainable Computing: Informatics and Systems* 29 (March 2021), 17 pages. https://doi.org/10.1016/j.suscom.2020.100478
[7] Stefano Cherubin and Giovanni Agosta. 2020. Tools for Reduced Precision Computation: a Survey. *Comput. Surveys* 53, 2 (Apr 2020), 35 pages. https://doi.org/10.1145/3381039
[8] Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Agosta Giovanni. 2020. Dynamic Precision Autotuning with TAFFO. *ACM Transaction on Architecture and Code Optimization* 17, 2, Article 10 (may 2020), 26 pages. https://doi.org/10.1145/3388785
[9] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-Point Mixed-Precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. ACM, 300–315. https://doi.org/10.1145/3009837.3009846
[10] Intel Corporation. 2018. BFLOAT16—Hardware Numerics Definition. *White paper* (2018).
[11] Eva Darulova, Einar Horn, and Saksham Sharma. 2018. Sound Mixed-Precision Optimization with Rewriting. In *ACM/IEEE 9th Int'l Conf. on Cyber-Physical Systems (ICCPS)*. https://doi.org/10.1109/ICCPS.2018.00028
[12] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca Benini. 2017. Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (2017), 2700–2713. https://doi.org/10.1109/TVLSI.2017.2654506
[13] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. 1–10. https://doi.org/10.1109/InPar.2012.6339595
[14] OpenHW Group. [n. d.]. CORE-V CV32E40P User Manual. https://cv32e40p.readthedocs.io/en/latest/
[15] Ruidong Gu and Michela Becchi. 2020. GPU-FPtuner: Mixed-precision Auto-tuning for Floating-point Applications on GPU. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 294–304. https://doi.org/10.1109/TVLSI.2017.2654506
[16] Hui Guo and Cindy Rubio-González. 2018. Exploiting community structure for floating-point precision tuning. In *Proc. 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam (NL) July 16-21, 2018*. ACM, 333–343. https://doi.org/10.1145/3213846.3213862
[17] John Gustafson and Isaac Yonemoto. 2017. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations* 4, 2 (2017).
[18] RISC-V International. [n. d.]. RISC-V Specifications. https://riscv.org/technical/specifications/
[19] Pradeep V Kotipalli, Ranvijay Singh, Paul Wood, Ignacio Laguna, and Saurabh Bagchi. 2019. AMPT-GA: Automatic Mixed Precision Floating Point Tuning for GPU Applications. In *Proceedings of the ACM International Conference on Supercomputing*. https://doi.org/10.1145/3330345.3330360
[20] Anish Krishnakumar, Umit Ogras, Radu Marculescu, Mike Kishinevsky, and Trevor Mudge. 2023. Domain-Specific Architectures: Research Problems and Promising Approaches. *ACM Trans. Embed. Comput. Syst.* 22, 2, Article 28 (jan 2023), 26 pages. https://doi.org/10.1145/3563946
[21] Andreas Kurth, Björn Forsberg, and Luca Benini. 2022. HEROv2: Full-Stack Open-Source Research Platform for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4368–4382. https://doi.org/10.1109/TPDS.2022.3189390
[22] Ignacio Laguna, Paul C. Wood, Ranvijay Singh, and Saurabh Bagchi. 2019. GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications. In *High Performance Computing*. Springer, 227–246. https://doi.org/10.1007/978-3-030-20656-7_12
[23] Michael O. Lam, Tristan Vanderbruggen, Harshitha Menon, and Markus Schordan. 2019. Tool Integration for Source-Level Mixed Precision. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*. 27–35. https://doi.org/10.1109/Correctness49594.2019.00009
[24] Gabriele Magnani, Lev Denisov, Daniele Cattaneo, and Giovanni Agosta. 2022. Precision Tuning in Parallel Applications. In *13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASIcs.PARMA-DITAM.2022.5
[25] Sally A McKee. 2004. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*. 162.
[26] Asit K Mishra, Rajkishore Barik, and Somnath Paul. 2014. iACT: A software-hardware framework for understanding the scope of approximate computing. In *Workshop on Approximate Computing Across the System Stack (WACAS)*.
[27] Ricardo Nobre, Luís Reis, João Bispo, Tiago Carvalho, João M. P. Cardoso, Stefano Cherubin, and Giovanni Agosta. 2018. Aspect-Driven Mixed-Precision Tuning Targeting GPUs. In *PARMA-DITAM workshop 2018*. 26–31. https://doi.org/10.1145/3183767.3183776
[28] Konstantinos Parasyris, Giorgis Georgakoudis, Harshitha Menon, James Diffenderfer, Ignacio Laguna, Daniel Osei-Kuffuor, and Markus Schordan. 2021. HPAC: Evaluating Approximate Computing Techniques on HPC OpenMP Applications. In *Proc. of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM, Article 86, 14 pages. https://doi.org/10.1145/3458817.3476216
[29] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-Based Approximation for Data Parallel Applications. *SIGPLAN Not.* 49, 4 (feb 2014), 35–50. https://doi.org/10.1145/2644865.2541948
[30] Prateek Shantharama, Akhilesh S Thyagaturu, and Martin Reisslein. 2020. Hardware-accelerated platforms and infrastructures for network functions: A survey of enabling technologies and research studies. *IEEE Access* 8 (2020), 132021–132085. https://doi.org/10.1109/ACCESS.2020.3008250
[31] Phillip Stanley-Marbell, Armin Alaghi, Michael Carbin, Eva Darulova, Lara Dolecek, Andreas Gerstlauer, Ghayoor Gillani, Djordje Jevdjic, Thierry Moreau, Mattia Cacciotti, Alexandros Daglis, Natalie Enright Jerger, Babak Falsafi, Sasa Misailovic, Adrian Sampson, and Damien Zufferey. 2020. Exploiting Errors for Efficiency. *Comput. Surveys* 53 (7 2020), 1–39. Issue 3. https://doi.org/10.1145/3394898
[32] Luca Valente, Yvan Tortorella, Mattia Sinigaglia, Giuseppe Tagliavini, Alessandro Capotondi, Luca Benini, and Davide Rossi. 2023. HULK-V: a Heterogeneous Ultra-low-power Linux capable RISC-V SoC. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6. https://doi.org/10.23919/DATE56975.2023.10137252