Synergistic Memory Optimisations: Precision Tuning in Heterogeneous Memory Hierarchies

# Synergistic memory optimisations: precision tuning in heterogeneous memory hierarchies

Gabriele Magnani, Daniele Cattaneo, Lev Denisov,
Giuseppe Tagliavini *Member, IEEE*, Giovanni Agosta, Stefano Cherubin

***Index Terms*—computer architecture, RISC-V, approximate computing, precision tuning**

***Abstract*—Balancing energy efficiency and high performance in embedded systems requires fine-tuning hardware and software components to co-optimize their interaction. In this work, we address the automated optimization of memory usage through a compiler toolchain that leverages DMA-aware precision tuning and mathematical function memorization. The proposed solution extends the LLVM infrastructure, employing the TAFFO plugins for precision tuning, with the SETHET extension for DMA-aware precision tuning and LUTHET for automated, DMA-aware mathematical function memorization. We performed an experimental assessment on HERO, a heterogeneous platform employing RISC-V cores as a parallel accelerator. Our solution enables speedups ranging from $1.5\times$ to $51.1\times$ on AxBench benchmarks that employ trigonometrical functions and $4.23-48.4\times$ on Polybench benchmarks over the baseline HERO platform.**

## I. INTRODUCTION

THE current trends in computer architectures feature the resurgence of custom processing units, from DSPs employed in cloud-based media content processing to GPGPUs to xPUs employed in AI tasks [1]–[4]. These compute units feature significant heterogeneity [5] with respect to traditional CPUs, particularly in terms of data representations, such as wide-range, reduced size floating point types (`bfloat16`, `bfloat8`). Smaller processing units, such as microcontrollers employed in battery-powered devices for a range of embedded applications, may not even have a floating point unit, relying on the programmer's skill to optimise the code using fixed point arithmetics. *Precision tuning* is an Approximate Computing technique [6] that aims at reducing the reliance on manual code optimisation when trading-off computation accuracy for performance on architectures that do not support highly efficient IEEE 754 floating point computation.

At the same time, parallel heterogeneous systems face additional challenges, as the increasing number of parallel execution units is not supported by an equal scaling of memory systems. Indeed, shared memory systems have become an unsustainable performance bottleneck in many applications [7], as the memory unit is subject to excessive contention, but distributed memory systems burden the CPU with overheads due to the need to control the data transfers. Efficient data movement mechanisms, such as DMA engines, are critical in such systems to achieve sustained performance [8]. 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) at the cost of increased programming complexity.

Both issues lead to a so-called "programmability wall" [1], as the complexity of handling at the software level a variety of different architectural solutions, from both the compute unit and memory system perspectives, grows massively. While dedicated tools that can address one specific issue, e.g., Precision Tuning tools to automatically tune the data size and representation within an application [6], [9], there is always the risk that, managing such issues independently, optimisation opportunities are lost, or even worse, de-optimisation happens as a result of unforeseen interactions between different tools. Specifically, precision tuning can positively interfere with DMA, as 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.

*a) Main contribution:* In this paper, we explore the compiler-based automation of the interaction between Direct Memory Access (DMA) and approximate computing techniques, with a particular emphasis on Precision Tuning and the memoization of mathematical functions. Our approach harnesses the capabilities of the compiler to effectively and efficiently balance memory usage — especially the critical L1 memory resource — and select appropriate data types, particularly for fixed-point representations. We demonstrate that the orchestration of DMA and Precision Tuning can synergistically yield consistent performance enhancements across diverse micro-architectural frameworks. This condition holds for platforms with or without hardware support for single or double-precision floating-point arithmetic. While each technique proves beneficial independently, their isolated application results in less consistent performance, necessitating greater caution from designers and adversely affecting performance portability. Our findings reveal that automatic memoization can realize significant speedups in mathematically intensive kernels. Central to our contribution is the development of SETHET and LUTHET, two innovative extensions of the TAFFO compiler plugins for LLVM. These

G. Magnani is with DEIB, Politecnico di Milano, Milan, Italy.
D. Cattaneo is with DEIB, Politecnico di Milano, Milan, Italy.
L. Denisov is with DEIB, Politecnico di Milano, Milan, Italy.
G. Tagliavini is with DISI, University of Bologna, Bologna, Italy.
G. Agosta is with DEIB, Politecnico di Milano, Milan, Italy.
S. Cherubin is with IDI, NTNU, Trondheim, Norway.

extensions enable DMA-aware precision tuning and the automated memoization of mathematical functions, respectively, and target the HERO [10] platform—an exemplary parallel, ultra-low-power RISC-V-based accelerator that heavily relies on DMA for optimized performance. We evaluated SETHET on Polybench, achieving an impressive speedup ranging from 3.5-48.4× over the baseline HERO system when employing both DMA and precision tuning. In contrast, DMA alone resulted in performance improvements ranging from 1.2-42.7×. LUTHET was evaluated on all the AxBench tests that utilize the C standard library (`libm`) and compared with SETHET. It peaks with a remarkable 44.8× speedup in one benchmark and scores between 1.4-3.6× on the rest.

*b) Organization of the paper:* The rest of this paper is organized as follows. Section II briefly surveys the state of the art in DMA and Precision Tuning, while Section III provides an overview of HERO, the RISC-V platform targeted by SETHET and LUTHET. Section IV describes the SETHET and LUTHET solutions, which respectively extend the TAFFO [11] tool towards DMA-aware precision tuning (SETHET) and automated memoization of mathematical functions (LUTHET). Section V assesses the impact of SETHET and LUTHET through an experimental campaign of the HERO platform, while in Section VI, we draw some conclusions and highlight future research directions.

## II. Related Work

The concept of precision tuning for memory systems is just as old as the IEEE-754 floating point standard [12], which defined in its 2008 revision data types for binary interchange and recommended the use of smaller data types – such as BINARY16 – for efficient storage but not for computing purposes. The shift in paradigm from homogeneous to heterogeneous computing and the advent of accelerators created the possibility to improve storage efficiency on memory-constrained accelerators. By focusing on GPU register pressure, [13] proposed a software-defined data extraction solution to improve the memory layout by trading off memory access costs. Modern GPUs are no longer bound by register and storage space, but the data transfer cost is now the main bottleneck to their performance. In our work, we leverage the same precision tuning principles, and we extend them to minimise the data transfer cost.

More recent precision tuning works on GPU aim at a programming language extension replacing the keywords representing data types in the accelerator code [14]–[17]. However, these approaches cannot cross the boundaries between the host and accelerator code, and the potential impact is only evaluated on the GPU code. Despite this limitation, these techniques have been extensively applied to high-performance scientific computation at the source code level [17]–[20], and at the compiler level [11], [21]–[23], and to embedded and low-power workloads [24]. Most of these works aim to minimise the number of bits used in each variable without investigating the actual impact on memory access performances.

The field of high-performance computing has also received attention from the broader approximate computing community, yet the focus remains on execution rather than on memory access. [25], [26] In modern heterogeneous architectures, DMA represents an enabling technology to achieve performance improvements and is particularly helpful to interconnect memory hierarchies requiring explicit transfer management [27]. In such systems, SETHET leverages low-level software APIs to schedule DMA transfers more efficiently.

While precision tuning based on floating point types typically relies on hardware units that downscale their size to accommodate truncated – e.g., BFLOAT16 – or scaled – IEEE-754 BINARY16, BINARY8, etc. – types, works on fixed point types and logarithmic number systems usually rely on optimisations based on LUT to approximate efficiently operations that do not find a direct correspondence in the integer units [28]. Our approach leverages the LUT-based approach to instrument both the host and the accelerator code to exploit LUTs whenever the computation requires it.

Cherubini *et al.* [9] provides an in-depth analysis of precision tuning research. Our work aims to optimise memory management to be as close as possible to the hardware, taking inspiration from compiler-level and embedded systems frameworks, the most relevant of which is TAFFO [29], [30].

TAFFO is composed of compiler plug-ins for the LLVM framework, and it operates on its intermediate representation (LLVM-IR). The programmer annotates the source code with hints about the input values, and the compiler passes propagate them accordingly. This structure allows us to adapt TAFFO for diverse LLVM-based compilation toolchains. TAFFO is composed of five LLVM analysis and transformation passes that manipulate directly the LLVM-IR, namely: *Initializer*, *Value Range Analysis* (VRA), *Data Type Allocation* (DTA), *Conversion*, and *Feedback Estimator* (FE). Each pass exchanges information with the others by exploiting LLVM metadata; apart from that, the passes are independent. In the compilation pipeline, the *Initializer* reads annotations inserted by the user and generates the internal metadata structure required by the other passes. Annotations are employed to specify initial value range information for the variables used by the piece of code to be transformed, and appear as clang *annotate* attributes. Then, the *VRA* conservatively derives the numerical intervals both for the annotated variables and for all other variables that depend on them. *DTA* decides which fixed-point to use based on the value calculated by VRA, using a peephole-like algorithm where each variable is assigned a fixed-point data type with the highest valid point position. *Conversion* modifies the LLVM-IR following the data type picked by the previous passes. Finally, *FE* statically analyses the error using state-of-the-art estimation methods [11]. TAFFO provides some correctness guarantees that make it suitable for our purpose. More specifically, it provides hard guarantees in the absence of overflow errors and soft guarantees in error boundaries. TAFFO has a dynamic version that automatically computes ranges through dynamic analysis [31]; however, this version was not used in this paper. TAFFO already provides partial support for parallel and distributed computing via OpenMP acceleration [32]; however, it is limited to homogeneous platforms. The extension to support heterogeneous accelerators is non-trivial, and we describe it in Section IV.

## III. RISC-V HETEROGENEOUS PLATFORM

Heterogeneous computing systems combine multiple processing elements characterized by distinct architectures, capabilities, and purposes. A common approach to heterogeneity involves pairing general-purpose processors with parallel accelerators to guarantee performance for diverse workloads. The open-standard RISC-V instruction set architecture (ISA) [33] provides a solid baseline for the design of heterogeneous systems. Its modular design based on ISA extensions allows customization based on specific application needs.

The heterogeneous platform used for the experimental assessment is HERO [10], which combines a 64-bit host processor (ARMv8 or RISC-V) executing a Linux environment alongside a bare-metal programmable many-core accelerator. The accelerator features a parametric number of clusters, including a configurable number of OpenHW CV32E40P [34] cores with support for the RV32IMA ISA and the XpulpV2 ISA extension [35]. System designers can configure the clusters to include between 4 and 16 cores, offering a flexible platform for a wide range of computing tasks.

HERO employs a hierarchical memory architecture, with a hybrid memory management unit (MMU) to translate virtual to physical addresses on the accelerator side. At the highest level, the platform includes a shared main memory (L3) accessible to both the host and accelerator. The accelerator can access: a 128 KiB level-1 (L1) data scratchpad memory (SPM) (one per cluster); a 4 KiB L1 instruction cache (one per cluster); a 512 KiB level-2 (L2) SPM (shared). Each cluster has a dedicated direct memory access (DMA) engine, which allows high-throughput data transfers between L1 and L2/L3 memory tiers, supporting multiple concurrent data streams.

The HERO software stack includes an OpenMP 4.5 runtime [36] and an LLVM 12 heterogeneous compiler toolchain. OpenMP has already been proposed in the literature for programming high-end embedded systems (e.g., MPSoCs) [37], [38]. While SYCL is emerging as a strong competitor, it is still not as widely supported on embedded systems, except for heterogeneous solutions based on edge GPGPUs (e.g., NVIDIA Jetson) [39]. At the language level, executing code on the accelerator requires a `#pragma omp target` directive defined at the block or function level. This directive outlines an *accelerator kernel* that the compilation flow compiles for both targets (64-bit host and 32-bit accelerator). The *Clang RV32 frontend* assigns different address spaces to the host and accelerator code by means of the `addrspace` standard attribute. Notably, pointers in the host code are 64-bit wide *host address space* while the ones in the accelerator kernels are associated with the 32-bit *native address space*. The hybrid MMU enables address translation using an internal table called a translation look-aside buffer (TLB). A dedicated compiler pass instruments the code to guarantee that the TLB always contains the entries to access the used data structures.

At the end of the compilation flow, the Clang driver embeds the accelerator binary object files inside the host binaries, creating a so-called *fat binary*. This file adheres to the Executable and Linkable Format (ELF) standard and simplifies the deployment of applications across heterogeneous platforms.

## IV. SETHET & LUTHET SOLUTIONS

This section introduces SETHET, a framework that performs precision tuning on heterogeneous codebases through static code analysis and the generation of optimized binaries. Additionally, we present its extension, LUTHET, which can memoise mathematical functions via lookup tables and make them accessible through the fastest available memory in the heterogeneous platform. Figure 1 depicts a block diagram of the software components our pipeline consists of.

### A. The SETHET Solution

SETHET builds upon TAFFO, a precision tuning framework that adds dedicated semantics to a restricted set of OpenMP constructs [32]. However, this support is insufficient for heterogeneous devices due to the lack of cross-device task mapping and memory management. SETHET enhances TAFFO by introducing code offloading by means of the `target` construct. This construct manages device selection through the `device` clause and data mapping via the `map` clause. The `map` clause specifies which data should be copied to the target device before execution and whether that data should be transferred back to the host afterward. When the compiler identifies a target region, it wraps its code inside a function and moves its definition to the heterogeneous module (*function outlining*). In the host code, it replaces the region with a call to the OpenMP runtime function `__tgt_target_mapper`, which takes as arguments the device identifier, a pointer to the outlined function, and an array of void pointers for the offload arguments. This process creates artificial boundaries in the data flow, which demands more refined analyses by the precision tuning framework, such as inter-procedural and inter-module techniques. The latter becomes necessary because OpenMP generates distinct translation units for each device.

*Revisiting the Precision Tuning Workflow:* To address these challenges, we propose an OpenMP-aware approach. SETHET encapsulates the core logic of TAFFO, avoiding ad hoc specialization at each tuning step. It introduces a `Transformation` pass that translates the intermediate representation fragments from both the host and device code into a single LLVM-IR module, and an `Anti-Transformation` pass to restore the tuned fragments. These steps are performed as the entry and exit points of the precision tuning process, as illustrated in Figure 1. We revised the precision tuning step to keep track of the original data layout and address space for each LLVM-IR value, as these can differ between the device and the host. In the following paragraphs, we provide an in-depth analysis of the key changes introduced to the precision-tuning process.

*1) Transformation:* This pass serves as a crucial preprocessing stage to reshape code and associated metadata for precision tuning. Such reshaping effectively allows subsequent stages of SETHET to utilize information otherwise obfuscated by the OpenMP offloading patterns. It addresses three key challenges: the exchange of precision-tuning analysis results between distinct LLVM modules across different architectures; the OpenMP *type-erasure*; and the handling of indirect calls with arguments encapsulated in an array.

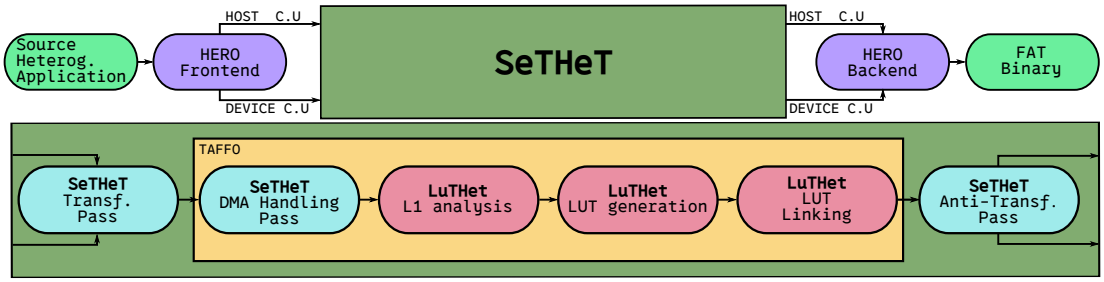



Fig. 1: Block diagram of the SETHET + LUTHET toolchain. The compilation process consists of two pipelines for each single-source heterogeneous application: one for the host (Host C.U.) and another for the accelerator device (Device C.U.). The light green elements represent the observed inputs and outputs of the toolchain. The purple elements signify HERO components, the dark green elements denote SETHET components, the red elements highlight LUTHET components, and the yellow box encloses the precision tuning steps.

To tackle the first challenge, the pass searches for each function call to `__tgt_target_mapper` within the host module. Indexing by the host function pointer provided as a parameter to this call site, it is possible to locate the corresponding external function definitions in the accelerator module, which are then cloned back into the host module. Before cloning, each function is analyzed to determine any dependencies on global values or other functions that are also required to be cloned. All cloned elements are prefixed with unique prefixes to prevent name conflicts and will later be used to simplify the precision tuning analyses.

OpenMP implements type erasure techniques to manage memory transfers across devices. The most direct implication for us is the use – by the OpenMP internals – of a `void*` array to store arguments that the precision tuning analyses must consider. Each argument and their *Def-Use* chains are analyzed for each heterogeneous region. Since LLVM-IR is a strongly typed language, the inference of data types from previous uses or definitions is often feasible. However, retrieving the original type by only looking at the callee side is not always possible, and it represents a problem in the case of data passed via OpenMP internals. The same happens for arguments from standard functions like `malloc`. In such a case, the data-flow inference is expanded to cover the callee function and its prologue. Figure 2 illustrates the code generated by Listing 1, where red arrows indicate the *Def-Use* chain analyzed by SETHET to deduce data types. The caller analysis identifies the first element as `i8**` while the callee analysis establishes it as `float addrspace(1)**`. Any discrepancies between caller and callee results for the same argument are reconciled according to the strict type aliasing rules of C/C++.

The final challenge is addressed by temporarily removing indirection and substituting it with a direct call to the cloned function version using arguments derived from the type-erasure analysis. TAFFO is able to recognize this direct function call pattern, applying inter-procedural precision tuning analyses and optimizations. Following the precision tuning process, the indirect call is reinstated to comply with the constraints of the reference OpenMP implementation, as detailed in Subsection IV-A3.

*2) DMA-Aware Precision Tuning:* Enabling precision tuning across memory transfer functions requires tailored support

```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 erasure resolution works in Figure 2.

```
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)**
```

Fig. 2: LLVM-IR generated by Listing 1, simplified to illustrate the type-erasure. The red arrows display the *Def-Use* chain traversed by SETHET to restore the information on types.

for DMA Application Programming Interfaces (APIs). The HERO compiler infrastructure provides intrinsics mirroring the syntax and arguments of the *memcpy* function in the C standard library, available in synchronous and asynchronous variants Adopting double-buffering techniques is beneficial for heterogeneous systems with complex memory hierarchies because they enable overlap between computations and memory transfers in iterative code regions; the use of the asynchronous API allows the program to continue processing the current data chunk while the DMA engine transfers the next one, with a wait primitive at the beginning of the next iteration. Within TAFFO, the Value Range Analysis (VRA) pass is responsible for computing and propagating the value ranges

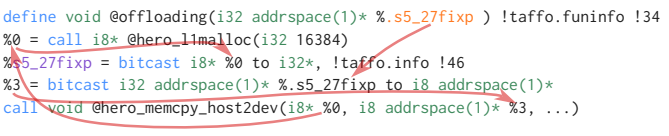



Fig. 3: LLVM-IR analysed when DMA resolutions happen. The red arrows display the Def-Use chain traversed by SETHET to restore the information on types. The ranges are propagated from the orange arguments .s5_27fixp to the purple local variable s5_27fixp through the DMA memory transfer function *hero_memcpy_host2dev*.

of annotated variables. This pass consults a catalog of known functions—such as mathematical functions, memory allocation routines, and memory movement operations—to adjust the range of output values and arguments according to their semantics. This technique provides an additional knob to handle a relevant subset of function calls otherwise corresponding to undefined symbols in the current compilation module. Building on this catalog-based methodology, SETHET extends the capabilities of the TAFFO VRA by incorporating semantics information for the HERO DMA intrinsics. As illustrated in Figure 3, the VRA pass propagates the range from the source to the destination pointer of the DMA call. Furthermore, introducing additional constraints on numeric ranges into the precision tuning algorithm ensures consistency with the maximum range of representable numeric values on each domain. This adjustment accounts for architectural differences between the host and device, such as the 64-bit integer registers on the host and 32-bit integer registers on the device.

*3) Anti-Transformation:* Following the Conversion pass of TAFFO, which is responsible for transforming annotated floating point representations into fixed point equivalents, the Anti-Transformation pass is subsequently executed. This stage is critical for reverting the output produced during the Conversion pass to its original structure before the SETHET transformations. This transformation is required because the HERO toolchain is incompatible with the output generated post-Conversion pass of TAFFO. To restore the compatibility with the HERO toolchain, the output of Conversion is split into two distinct modules: one that encapsulates the host code and another for the accelerator code. Furthermore, it is necessary to restore OpenMP calls to their original form, which mandates the use of indirect calls. In light of these requirements, SETHET schedules the Anti-Transformation pass to occur following the code manipulation phase within the precision tuning framework. During this pass, SETHET systematically moves all newly converted functions and global variables that were previously imported from the accelerator module back to it. Additionally, all OpenMP indirect calls are restored and modified to point to the newly generated regions, which have been exported to heterogeneous modules, as per the specifications outlined in Subsection IV-A1. Ultimately, any references about the heterogeneous module are expunged from the host module, ensuring compliance with the operational expectations of the HERO toolchain.

### B. *The* LUTHET *Solution*

LUTHET expands SETHET with the ability to memoize mathematical functions by leveraging the unique characteristics of the HERO environment. This new expansion can be logically subdivided into four methodological steps. The first step estimates the maximum available L1 memory. As a second step, for each `libm` mathematical function called in the cluster code, LUTHET computes a `LUT` and stores it in the main memory of the host. The third step modifies each heterogeneous call site and injects the required `LUT` pointers as new arguments. Lastly, the fourth step adds the code required to transfer the memory from the host to the L1 memory.

*1) L1 usage analysis:* LUTHET estimates the available L1 memory for the `LUT` allocation at each math function invocation site. To achieve this, a weighted reversed call graph (RCG) is reconstructed for each math function declared in the heterogeneous module. For each pair of nodes in the RCG, if the child node uses L1 memory, the net memory usage is used as the edge weight. This analysis requires a loop-free call graph. In HERO, L1 memory can be allocated independently of the code path execution. This means that global variables can be designated to reside in L1 or L2 memory at compile time through section annotation.

The construction of the weighted RCG can be formally specified as follows. Given two functions $Q$ and $G$, we state that $Q$ call $G$ using the $Q \prec G$ notation if one of the instructions of $Q$ is a call to $G$. Referring to $a(i)$ as the amount of *allocated* bytes and $r(i)$ the amount of *released* bytes in the L1 memory by the $i$ instruction, we define the net amount of L1 memory of a function $Q$ up to instruction $y$ as

$$\delta(Q,y) = \sum_{i \in Q \wedge i < y} a(i) - r(i)$$

This formula holds if the analyzed code does not include any loop. Otherwise, LLVM's scalar evolution analysis is used to estimate the iteration count, and the $\delta$ analysis is performed around the loop boundary and multiplied by this value.

Finally, the RCG for a function $F$ can be constructed by defining the set of reachable state $\gamma_F$:

$$\gamma_F = \{F \mid F \in \gamma_F \wedge \forall Q \forall G((Q \prec G \wedge G \in \gamma_F) \Rightarrow Q \in \gamma_F)\}$$

The set $\gamma_F$ is initialized with $F$ and then until reaching a fixed point for each pair of functions $Q$ and $G$ where $Q$ call $G$ and $G$ is part of the $\gamma_F$ set $Q$ is added to the set.

The weighted edge between two $\gamma_F$ elements is defined as

$$G \xrightarrow{x} Q \stackrel{\text{def}}{=} Q \prec G \wedge x = \delta(Q,y) \wedge G \in \gamma_F \wedge Q \in \gamma_F$$

where $x$ is the weight of the arc and $y$ is the last call instruction from $Q$ to $G$. The constructed weighted RCG can determine the amount of L1 memory from a math function call place to one of the leaves (heterogeneous entry points). It follows that this relationship captures the longest path between the two nodes. By assuming that no L1 de-allocation happens because of math function calls in this graph, we observe that the graph is monotone, and the analysis converges to a fixed point. This technique can be used only when all L1 allocations are data-independent, allowing for a comprehensive statistical analysis

```
def generalLUT( mathFunction, fixedPointArgType, ↘
     fixedPointRetType, LUTSize ):
    Nl = log2(LUTSize)
    Na = getBits(fixedPointArgType)
    decimalPosition = getDecimalPoint(fixedPointArgType)
    neededShift = Na−Nl
    # Na >= Nl as Na is the storage size of an integers 32 or 64
    # bits and Nl is the size in bits of the LUT
    internalFixedType = fixedPointArgType >> neededShift
    newDecimalPosition = decimalPosition − neededShift
    lastBitPower = pow2(−newDecimalPosition)
    LUT = []
    for i in [0 to LUTSize]:
      argMath = (float)(i)*lastBitPower
      retMath = mathFunction(argMath)
      LUT[i] = convertFloatToFixed(retMath, ↘
           fixedPointRetType)
    return LUT
}
```

Listing 2: Pseudocode of the generalLUT algorithm.

of L1 management. Fortunately, this is usually not an issue, as most kernels in machine learning and digital signal processing typically fall into this category.

*2) LUT generation:* In the second pass, each `libm` call site is analyzed to create a `LUT`. The `LUT` size is determined as the smaller value between the user-requested size and the available L1 memory at the call site (from the previous pass). This value is then adjusted to the smaller nearest power of two. The `LUT` generator uses information from the Data Type Analysis of SETHET to determine the fixed point data type that will be used as the argument and return type for the new function. With this piece of information, LUTHET can generate the appropriate `LUT` using a specialized algorithm for trigonometric functions such as sin, cos, arcsin, arccos, or generic generator algorithms for the other supported functions. This distinction stems from preprocessing the argument of the trigonometric function to a well-defined range.

*a) General algorithm:* This algorithm is useful for creating a `LUT` for a function that takes exactly one input and provides one output. The rationale consists of using a fixed-point type with the same number of bits as the base-two logarithm of the `LUT` size ($N_l$) to index the `LUT`. The `LUT` will be populated with fixed-point values of the type recommended by the SETHET data type analysis for the return value. Listing 2 shows the pseudocode for the general algorithm.

To create the new internal fixed-point type from the argument type, we need to right-shift the argument by a value equal to the difference between the argument size in bits ($N_a$) and $N_l$. This shift operation generates a new fixed-point type where the same number of shifts are made to the decimal place. If the decimal part of the argument has less bits than $N_l$, then the shift will move the decimal point outside the representation boundary. In such cases, all integers are extended logically to have an infinite number of trailing zeros to maintain the decimal place outside the boundary of the integer. This is used to calculate the scaling factor for the least significant bit. To populate the `LUT`, we can instantiate an iterator of the same type as the newly generated fixed-point type. This iterator is used to compute the requested mathematical function at each step. During each iteration, the iterator is converted to a double type, the mathematical function is calculated, and the return value is transformed to the required fixed-point value as determined by the SETHET data type analysis.

*b) Specialized sin & cos algorithm:* For periodic functions, such as in the case of sin and cos, it is possible to optimize the `LUT` generation with a range reduction algorithm [40]. Hence, LUTHET implements a specialised `LUT`-generation algorithm. This algorithm uses trigonometric identities to reduce the range to $[0,\ \frac{\pi}{2}]$, making it easier to index the `LUT`. The `LUT` also takes advantage of the well-known output range in the interval $[-1,\ 1]$ to populate with a predefined fixed point type for maximum precision.

By utilizing these identities

$$\sin(\theta \pm \tfrac{\pi}{2}) = \pm\cos(\theta)$$
$$\cos(\theta \pm \tfrac{\pi}{2}) = \mp\sin(\theta)$$

any angle can be transformed to an equivalent angle in the range of $[0,\ \frac{\pi}{2}]$, which allows for a precise `LUT` because it reduces the possible large range of values for the argument of the sin and cos functions to a small, well-known radius that makes it easier to index the `LUT`. It also allows for the use of the same `LUT` for all versions of the fixed point argument, as they are all mapped to the same range and have the same output. Internally, the `LUT` uses a fixed-point type with 1 bit for the sign, 1 bit for the integer part, and the remaining bits for the fractional part. The index generation follows the same rule as the general algorithm. Therefore, the argument is shifted after the reduction in order to obtain a fixed point with the same number of bits as $N_l$. Additionally, by using the identity $\cos(\theta) = \sin(\frac{\pi}{2} - \theta)$, we can further reduce memory usage by using the sin `LUT` for the cos. This streamlines the process of populating the lookup table with precise values, thereby enhancing the accuracy of calculations.

*c) Specialized asin and acos algorithm:* The algorithms for the arcsin and arccos functions offer similar advantages to the sin and cos functions, but they do not require the input value to be restricted to a specific range as the domain of the arcsin and arccos functions is already in the range of $[-1,1]$. This range is divided into a number of spaces equal to the `LUT` size, and a fixed point argument with the same number of bits as $N_l$ is used to index it. The output of arcsin and arccos is also within a precomputed range, allowing for the use of a predetermined fixed point type inside the `LUT` to maximize precision; it is not possible to use the same `LUT` for both functions, as the outputs have different ranges.

*3) Bridging the heterogeneous gap:* At this stage, LUTHET has successfully constructed the reversed weight graph and instantiated the `LUT` inside the host memory space. The next task is to ensure the transfer of `LUT` to each heterogeneous region while minimizing the number of `LUT` pointers passed at each one as new arguments. To do so, we can analyze the RCG of each `libm` function and notice that the leaves of each tree correspond to different heterogeneous starting points. In Figure 4, we can see an example of two different RCGs: one in red and the other in light blue. Their overlap is depicted in green. We can use this graph to discover which `libm` functions are required at the leaves.

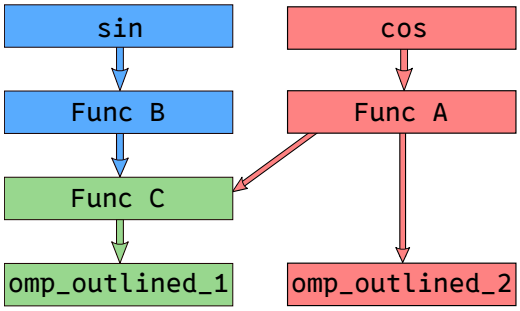



Fig. 4: RCG linking diverse entry points with different LUT. The red portion represents the RCG for cos, while the light blue section for sin. Their overlap is depicted in green.

After identifying the necessary LUT for each region, we can update the __tgt_target_mapper call by including their pointers in the argument array. Additionally, we need to adjust the signature of each heterogeneous region's entry function to account for the new argument. This function is also adjusted to store the $l3$ memory pointer in its private $l1$ global memory. This adjustment prevents the propagation of the LUT pointers down the call graph to the memory transfer location.

*4) Final step:* Each function that previously used the libm in the cluster code region can now be replaced with a function that utilizes the LUT allocated in L1. When using the general algorithm for the math functions, as well as for the arcsin and arccos functions, the fixed point argument is shifted by $N_a$ - $N_l$ and then used as an index. In the case of sin and cos functions, the argument is reduced to the $[0, \frac{\pi}{2}]$ range before shifting. Additionally, for special trigonometric functions, the output of the LUT needs to be adjusted to the requested fixed-point type through additional shifting.

After the function is no longer needed, the L1 memory can be freed. As a last step, the dominance tree is examined to identify the first basic block (BB) where the instantiated LUT is no longer in use and is dominated by its allocation. In such BB, the LUT is de-allocated from the L1 memory.

## V. Experimental Campaign

Our experimental campaign is designed to rigorously assess the advantages that SETHET and LUTHET provide in conjunction with established optimization techniques. Specifically, we aim to demonstrate that SETHET enhances performance relative to its constituent components, namely DMA and precision tuning. Furthermore, for LUTHET, we seek to evaluate its efficacy in complex benchmark scenarios.

### A. Experimental Setup

*1) SETHET Experimental Setup:*

*a) Benchmark Suite:* We have chosen the PolyBench-ACC [41] benchmark suite to evaluate our work. This suite encompasses a variety of numerical kernels with static control flow from multiple application domains. The HERO architecture currently supports:

**2mm** Multiplication of two matrices
**3mm** Multiplication of three matrices
**Atax** Matrix transpose and vector multiplication
**Bicg** Sub-kernel of BICGSTAB
**Covariance** Computation of covariance
**Gemm** Matrix multiplication
**Convolution-2d** Two-dimensional convolution computation

*b) Benchmark Configurations:* The benchmarks have been modified to offload kernel workloads onto heterogeneous devices utilizing DMA, as delineated in Section III. We have devised an alternate iteration of these benchmarks to maintain structural consistency without using the DMA. To this end, we replaced DMA memory transfer primitives with functionally analogous CPU-managed memory transfer operations, and we conducted a comparative analysis of both data transfer modalities, which we have designated as DMA and noDMA, respectively. Each version was further augmented with the standard annotation syntax requisite for the TAFFO precision tuning framework. Since the accelerator is constrained to 32-bit integer support, we established a parameter limit within the precision tuner to regulate the maximum fixed-point bit-width. This design choice leads to the creation of two distinct benchmark configurations, 32 and 64, which cap the bit-width of fixed-point computation at 32-bit and 64-bit, respectively. An additional parameter arises from the HERO capability to activate or deactivate the accelerator floating-point unit at design time. Accordingly, we tested each benchmark variant with hardware floating-point support (HARD) and software emulation (SOFT). For each benchmark, we have tested several versions provided by the Cartesian product of $\{\texttt{DMA}, \texttt{noDMA}\} \times \{32, 64\} \times \{\texttt{SOFT}, \texttt{HARD}\}$.

*2) LUTHET Experimental Setup:*

*a) Benchmark Suite:* The benchmarks used to evaluate LUTHET are part of the AxBench 1.0 [42] suite. Axbench comprises applications from different domains exploring various aspects of approximate computing. We filtered out benchmarks that do not use libm functions, resulting in:

**Black-Scholes** A mathematical model for the dynamics of a financial market that contains derivative investment instruments. Functions used: log, exp, sqrt.
**FFT** The Cooley–Tukey algorithm converts a signal from its original domain (often time or space) to a representation in the frequency domain. Functions used: sin and cos.
**Inversek2j** Inverse kinematics for a 2-joint arm. Functions used: sin, cos, asin, acos.
**K-means** Partition n observations into k clusters, with each observation belonging to the cluster with the nearest mean. Function used: sqrt.
**Sobel** Edge detection algorithm that creates an image emphasizing edges. Function used: sqrt.

Additionally, AxBench provides its input data, divided into various datasets categorized by size. We utilized the original input data to conduct our experimental evaluation.

*b) Benchmark Configurations:* We have ported the selected benchmarks to the HERO codebase, modifying the code to offload the workload to the accelerator. Additionally, we have added the TAFFO annotations. We executed the benchmarks with a SOFT 64 configuration, which implies floating point emulation on the accelerator, while fixed points can

grow up to 64 bits. We compiled different LUTHET versions, each with different LUT sizes ranging from 512 to 4096, and executed each version on HERO, SETHET, and LUTHET.

*3) Common Experimental Setup:*

*a) Hardware & System Setup:* We employed a Xilinx ZCU102, a Xilinx Zynq UltraScale+ MPSoC with a 64-bit ARMv8 quad-core Arm® Cortex®-A53 processor as host and a single cluster of 8 cores mapped on the programmable logic (PL) as accelerator. PetaLinux Tools (v.2019.2) enable loading a Linux OS on the host side and a binary image to the PL.

*b) Metrics:* To evaluate the efficacy of our proposed solution, we compute one performance metric alongside one quality metric across all experimental configurations. In this study, we prioritize enhancements in time-to-solution as the principal performance metric, while the numerical relative error serves as the metric for computational quality. To calculate these metrics, both the BASELINE and the SETHET & LUTHET versions were compiled using *clang-12* with the default optimization level (-O2). To address the potential interference of background noise during time measurements —primarily attributable to the operating system— we implement a strategy of averaging the timing data over 100 iterations for each benchmark configuration. The timing assessments utilize the OpenMP function `omp_get_wtime()`, which captures the elapsed wall clock time in seconds. This metric is thread-specific and serves as an established standard in benchmarking practices. The time improvements are expressed as speedup, adhering to the standard formula $S = \frac{T_{ref}}{T_{entry}}$. Speedup values are referenced against the time-to-solution of a BASELINE configuration $T_{ref}$, which does not incorporate precision tuning and operates under the `noDMA` data transfer mode. To evaluate the quality of the results, we calculated the relative error w.r.t. a golden reference. The reference is derived from identical code compiled within the same HERO architecture environment without applying precision tuning. Denoting the golden reference as $B$ and the outcomes from precision tuning as $P$, the relative error is computed using the formula $\frac{|P-B|}{B}$.

In alignment with prior research on approximate computing, we set an error threshold capped at 10% [43]. This threshold is essential for determining the viability of the explored precision tuning configurations. In accordance with the original definitions of the benchmarks, our golden reference relies on IEEE-754 *binary32* standard [44].

## B. SETHET *Evaluation*

We evaluate the individual effects of DMA and precision tuning using the BASELINE configuration for comparison. As a preliminary hypothesis regarding the potential synergistic impact of these two variables, we operate under the assumption that no correlation exists between their respective contributions. Subsequently, we will critically discuss this hypothesis and assess its validity through further empirical analysis.

*a) DMA Contribution:* We first adapt the benchmarks to isolate the DMA contribution to create a BASELINE version that includes memory transfer costs, as detailed in Section V-A1b. Table I illustrates the speedup achieved with DMA enabled, which varies significantly between SOFT and HARD floating-point configurations. The SOFT configuration has more computational overhead due to emulating floating-point support, while the HARD configuration benefits more from reduced memory transfer times. Benchmarks like **atax**, **bicg**, and **convolution** show particularly higher speedups, thanks to asynchronous DMA functions that allow significant overlap between computation and memory transfer through double buffering.

TABLE I: Speedup achieved by DMA w.r.t. BASELINE. Overlaid colours represent the speedup magnitude. GeoMean is the geometric mean.

| Benchmark | Configuration | DMA |
|---|---|---|
| **2mm** | HARD | 6.23 |
| | SOFT | 1.48 |
| **3mm** | HARD | 3.79 |
| | SOFT | 1.26 |
| **atax** | HARD | 42.70 |
| | SOFT | 8.75 |
| **bicg** | HARD | 28.90 |
| | SOFT | 6.32 |
| **convolution** | HARD | 20.90 |
| | SOFT | 2.49 |
| **covariance** | HARD | 6.08 |
| | SOFT | 1.52 |
| **gemm** | HARD | 3.82 |
| | SOFT | 1.22 |
| **GeoMean** | HARD | 10.51 |
| | SOFT | 2.41 |

*b) Precision Tuning Contribution:* We consider the contribution of precision tuning as the improvement in time-to-solution achieved by the precision tuning component, as described in Section IV, without using DMA primitives. Table II reports the speedup, showing a clear distinction between SOFT and HARD configurations. The results show that the SOFT configurations derive greater benefits from this optimization. While precision tuning can enhance both memory transfer and computation costs, fixed-point arithmetic significantly simplifies handling SOFT floating-point computations.

Figure 5 presents two separate subplots displaying the relative error. The left plot illustrates the relative error for the 32 configurations, which ranges from $[1 \cdot 10^{-5};\ 8.8 \cdot 10^{-2}]$. The right plot for the 64 configurations shows a relative error range from $[1 \cdot 10^{-7};\ 1 \cdot 10^{-5}]$. The specific floating-point format does not affect the results of SETHET, as the error remains consistent across all configurations. The plots also highlight that reducing the maximum integer size significantly impacts the **atax** and **gemm** computations. This effect is primarily due to the range of output and operations these two functions perform. **atax** has an impressive output range of $[0;\ 3.40 \cdot 10^{6}]$ while **gemm**'s range spans from $[-5.47 \cdot 10^{4};\ 5.34 \cdot 10^{4}]$.

*c) A Naïve Prediction:* A naïve prediction is calculated by multiplying the speedup of DMA and precision tuning together. This prediction assumes the independence of the two contributions. So it's valid if and only if DMA primitives do not interact with precision tuning, and precision tuning does not interact with DMA data transfer.

*d) Testing the Prediction:* The interaction between DMA and precision tuning depends on the structure of the bench-

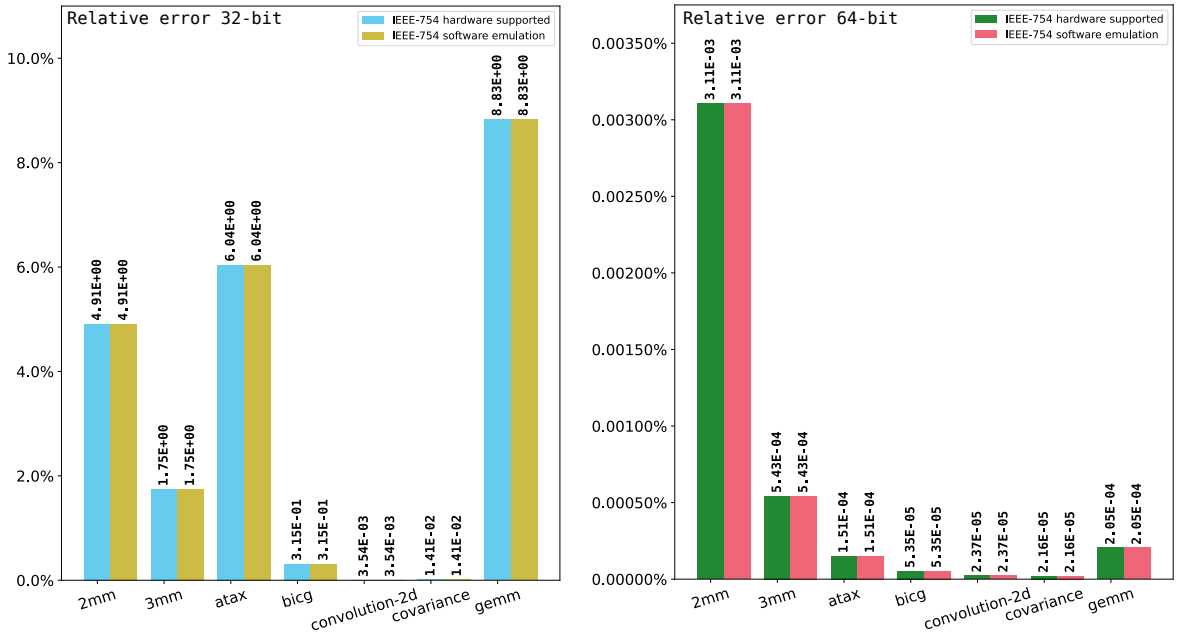



Fig. 5: Relative error of SETHET solution compared to BASELINE. The graph on the left compares SETHET employing only 32-bit fixed-point numbers, while the graph on the right shows SETHET employing only 64-bit fixed-point numbers. The reference version for functional correctness is defined using floating point IEEE-754 binary32.

mark. Figure 6 illustrates how the DMA primitive can be rearranged within a loop to achieve different properties. A serial layout can be obtained using either the synchronous or asynchronous DMA API. In this scenario, precision tuning will affect both the transfer time (if the size of the stored data changes) and the loop body execution time, while DMA will influence only the transfer time. In contrast, the double buffering layout allows computation and memory transfer to occur in parallel using the asynchronous DMA API. In this case, both precision tuning and DMA will impact transfer and loop body times. Here, the interaction between DMA and the precision tuning process is characterized by the competition between the newly optimized loop body time and the potentially modified memory transfer time.

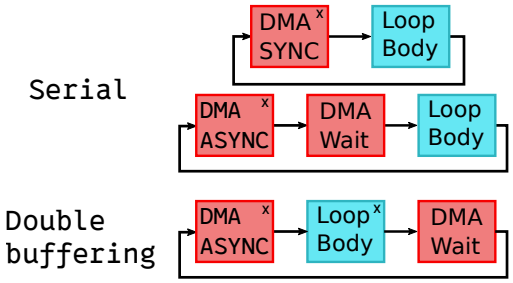



Fig. 6: Common DMA layouts used in a kernel loop. In the Serial layout (top), memory transfers happen at the beginning of the loop. In the Double buffering layout (bottom), memory transfers run parallel to the loop body. The blocks marked with **X** are influenced by both DMA and precision tuning.

Table III compares th configuration speedup achieved by SETHET using DMA against that of a naive prediction approach. Notably, in the HARD configurations, the naive prediction demonstrates satisfactory performance. In these scenarios, optimizing the loop body does not effectively diminish the running time to parallel the memory transfer time, suggesting that the optimizations are largely orthogonal.

```
define void @omp_offloading(float* %0/i16  %.s4_12_fixp,  ...)
  #Alloca double the slice size for double buffering
  %.flt = tail call i8* @hero_l1malloc(i32 32768/16384)
  %2 = bitcast float* %0/i16* %.s4_12_fixp to i8*
  #Transfer of first slice
  tail call void @hero_memcpy_host2dev(i8* %.flt, i8* %2, i32 16384/8192
...
#Kernel Body
  #Executed 4096 per slice
  %c = fmul float %92, 2.000000e+00
  %c = shl i32 %s17_15_fixp13, 1
...
 #Executed 262144 per slice
  %a = fmul float 3.000000e+00, %126
  %a = mul nsw i32 %116, 3
```

Fig. 7: LLVM-IR part of **gemm** benchmark compiled with HARD-32 configuration. The instructions highlighted in blue belong to the BASELINE implementation, while those in orange represent the instructions after the SETHET optimization. This optimization effectively reduces the amount of data transferred by half for one of the three arrays. Additionally, it simplifies some multiplications within the kernel body, leading to decreased transfer time and improved kernel execution efficiency.

An unexpected outcome is observed in the HARD-64 configuration: the speedup may even be less favorable than when utilizing the benchmark without SETHET. This outcome can be attributed to the increased memory transfer requirements associated with converting from binary32 to 64-bit fixed point representation. Conversely, the predictions fail to accurately estimate the achieved speedup in the SOFT configuration.

TABLE II: Speedup achieved by SETHET precision tuning w.r.t. BASELINE. Overlaid colours represent the speedup magnitude. GeoMean is the geometric mean.

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

TABLE III: Speedup achieved by SETHET w.r.t. BASELINE. Overlaid colours represent the speedup magnitude. GeoMean is the geometric mean.

| Benchmark | Configuration | Prediction | SETHET |
|---|---|---|---|
| **2mm** | HARD 32 | 6.42 | 6.69 |
| | HARD 64 | 6.35 | 6.68 |
| | SOFT 32 | 12.74 | 17.80 |
| | SOFT 64 | 12.43 | 17.30 |
| **3mm** | HARD 32 | 3.94 | 4.23 |
| | HARD 64 | 3.90 | 3.56 |
| | SOFT 32 | 11.16 | 15.00 |
| | SOFT 64 | 10.36 | 12.26 |
| **atax** | HARD 32 | 44.41 | 43.90 |
| | HARD 64 | 43.98 | 41.40 |
| | SOFT 32 | 47.22 | 48.40 |
| | SOFT 64 | 44.80 | 45.60 |
| **bicg** | HARD 32 | 29.77 | 29.50 |
| | HARD 64 | 29.77 | 28.70 |
| | SOFT 32 | 29.47 | 33.90 |
| | SOFT 64 | 27.62 | 33.00 |
| **convolution** | HARD 32 | 21.95 | 20.50 |
| | HARD 64 | 21.74 | 17.70 |
| | SOFT 32 | 19.62 | 32.50 |
| | SOFT 64 | 14.59 | 28.00 |
| **covariance** | HARD 32 | 6.32 | 6.60 |
| | HARD 64 | 6.26 | 6.11 |
| | SOFT 32 | 11.52 | 16.20 |
| | SOFT 64 | 10.67 | 15.00 |
| **gemm** | HARD 32 | 4.01 | 4.85 |
| | HARD 64 | 3.90 | 4.60 |
| | SOFT 32 | 11.91 | 20.70 |
| | SOFT 64 | 10.38 | 16.30 |
| **GeoMean** | HARD 32 | 10.93 | 11.34 |
| | HARD 64 | 10.81 | 10.50 |
| | SOFT 32 | 17.65 | 24.14 |
| | SOFT 64 | 15.91 | 21.59 |

In these instances, the measured performance is consistently underestimated, indicating a lack of independence between the two optimization strategies in this context. Upon cross-referencing the **"GeoMean"** values presented in Tables III and I, it becomes clear that precision tuning plays a vital role in the effectiveness of the SETHET solution. This aspect is particularly significant, as SETHET outperforms DMA in three out of four configurations analyzed.

*e) Performance Analysis:* As the HERO platform supports only 32-bit integer data types, the $<$ HARD, 64 $>$ configuration is expected to be the most challenging. This is because HERO executes a software-emulated version of all the sixty-four data operations. Despite this limitation, SETHET performs similarly to HERO DMA in this configuration. On the other hand, SETHET demonstrates exceptional efficiency in all the hardware designs lacking a Floating Point Unit (FPU) (SOFT configurations), attaining a performance enhancement of up to 20$\times$ in the **gemm** $<$ SOFT, 32 $>$ benchmark.

Analysis of Table III reveals distinct behaviours between **atax** and **gemm** compared to the other benchmarks, prompting further investigation into their performance characteristics. **atax** is particularly sensitive to DMA due to its structure, which contains two minimal loop bodies operating on a substantial amount of data, creating a memory-bound scenario significantly improved by DMA. In contrast, **gemm** experiences a dramatic increase in performance, improving from 1.22$\times$ $<$ SOFT, 64 $-$ 32 $>$ in HERO DMA to an impressive 20.7$\times$ $<$ SOFT, 32 $>$ when utilizing SETHET, representing a relative performance boost of 16$\times$. The structure of **gemm** consists of a single, expensive loop body, avoiding the **atax** limitations. Figure 7 illustrates part of the generated LLVM-IR of the **gemm** benchmark compiled with HARD-32, which serves as a prime example of how SETHET can reduce the amount of data transferred, specifically, by half for one of the three arrays. Additionally, it highlights an instance where a multiplication operation has been completely eliminated, as well as another instance where the operation has been converted to its fixed-point equivalent. Similar optimizations where the compiler can optimize away the fixed-point multiplications using shifts and additions are applied to multiplications and divisions in **covariance** and **convolution**.

Figure 5 illustrates the quality metric of SETHET, akin to the effects of precision tuning without DMA, as outlined in Section V-B0b. This figure highlights that the maximum relative error for **covariance** and **convolution** remains below 0.004%. This finding suggests potential for further improvements in time to solution by reducing the current data size limit from 32 bits to 16 bits or even lower.

### C. LUTHET *evaluation*

This Section presents the LUTHET's results from Axbench.

*a) Performance Metric:* Table IV displays the speedup achieved by SETHET and LUTHET, along with their comparison. We experimented with multiple entries per LUT ranging

TABLE IV: Speedup comparison between LUTHET and SETHET w.r.t. BASELINE. Fixed-point values are capped at 64-bit. Speedups are based on the benchmark without fixed-point support. `Comparison` is the speedup of LUTHET relative to SETHET. Overlaid colours represent the speedup magnitude.

| | blackscholes | fft | inversek2j | kmeans | sobel |
|---|---|---|---|---|---|
| SETHET | 1.1 | 1.5 | 1.1 | 1.0 | 17.6 |
| LUTHET | 1.5 | 5.3 | 51.1 | 2.0 | 30.8 |
| Comparison | 1.4 | 3.6 | 44.8 | 2.0 | 1.8 |

TABLE V: Relative error between SETHET and LUTHET using different `LUT` size (reported in the first column). The reference is IEEE-754 binary32. Fixed points are capped at 64 bits. Overlaid colours represent the error magnitude.

| | blackscholes | fft | inversek2j | kmeans | sobel |
|---|---|---|---|---|---|
| SETHET | $5.6e{-3}$ | $1.4e{-5}$ | $8.9e{-3}$ | $2.2e{-1}$ | $1.5e{-2}$ |
| LUTHET 512 | $3.8e{+1}$ | $1.6e{-5}$ | $5.3e{+0}$ | $6.0e{+0}$ | $1.2e{+1}$ |
| LUTHET 1024 | $2.4e{+1}$ | $1.6e{-5}$ | $2.7e{+0}$ | $4.4e{+0}$ | $8.4e{+0}$ |
| LUTHET 2048 | $1.2e{+1}$ | $1.6e{-5}$ | $1.4e{+0}$ | $3.2e{+0}$ | $5.7e{+0}$ |
| LUTHET 4096 | $6.8e{+0}$ | $1.6e{-5}$ | $6.5e{-1}$ | $2.3e{+0}$ | $3.8e{+0}$ |

from 512 to 4096, but we only report the LUTHET speedup for 1024 entries, as all the other sizes yield similar results. This behavior is attributed to LUTHET's capability to hoist the memory transfer out of loops. SETHET ranges from $[1\times;17\times]$, with a geometric mean of the speedup of approximately $1.9\times$, as most benchmarks achieve values between $[1\times;2\times]$. The outlier is **Sobel**, which achieves exceptional speed due to performing a single mathematical function operation (sqrt) and many other independent floating point operations, especially FMA, that SETHET optimizes. On the other hand, LUTHET achieves speedup in the range of $[1.5\times;51.1\times]$ and is consistently faster than SETHET alone. Indeed, the LUTHET geometric mean is approximately $7.5\times$. The notable speedup of **FFT** and **inversek2j** is due to the memoization of trigonometric functions used repeatedly inside a loop. The Comparison row reports the speedup of LUTHET relative to SETHET, and its geometric mean is approximately $3.8\times$. It also highlights that **inversek2j** is significantly impacted by the LUTHET optimization. Its kernel consists of 27 arithmetic operations and eight calls to trigonometric functions, which entirely dominate the running time.

*b) Accuracy Metric:* Table V shows the relative error of SETHET and LUTHET compared to the golden model. For the image-based benchmarks, the relative error is calculated as the Root Mean Squared Error (RMSE). SETHET achieves a relative error ranging from $[1.4\cdot10^{-5};\ 2.2\cdot10^{-1}]$. LUTHET's relative error depends on the `LUT` size, with a larger size resulting in better results. The only exception is **FFT**, where the relative error remains constant. This effect is due to the high precision of sin and cos approximations. No single threshold exists to determine an acceptable error for all benchmarks. For benchmarks like **Sobel** and **Kmeans**, an error of up to 10% can be considered acceptable, according to [43]. Figure 8 compares LUTHET applied to **Sobel** with different `LUT` sizes.

TABLE VI: Difference of binary size in bytes between SETHET and LUTHET using different `LUT` size (reported in the first column). Overlaid colours represent the value magnitude. Absolute and relative size increase is reported interleaved with the LUTHET results

| | blackscholes | fft | inversek2j | kmeans | sobel |
|---|---|---|---|---|---|
| BASELINE | 451023 | 440397 | 608011 | 421061 | 394885 |
| SETHET | 467350 | 443062 | 618217 | 450044 | 443476 |
| LUTHET 512 | 446142 | 456502 | 636185 | 404364 | 397972 |
| | +8184 +1.8% | +2048 +0.4% | +6144 +0.9% | +2048 +0.5% | +2048 +0.5% |
| LUTHET 1024 | 454326 | 458550 | 642329 | 406412 | 400020 |
| | +16376 3.6% | +4096 +0.9% | +12288 +1.9% | +4096 +1.0% | +4096 +1.0% |
| LUTHET 2048 | 470702 | 462646 | 654617 | 410508 | 404116 |
| | +32768 +7.0% | +8192 +1.8% | +24560 +3.8% | +8192 +2.0% | +8192 +2.0% |
| LUTHET 4096 | 503470 | 470838 | 679177 | 418700 | 412308 |

*c) Size Metric:* Table VI provides the size for each benchmark compiled using BASELINE, SETHET, and all the tested `LUT` sizes for LUTHET. It also shows the differences between each `LUT` increment. This analysis demonstrates how the size steadily increases with the `LUT` size. In some benchmarks (**blackholes**, **kmeans**, **sobel**), the `LUT` binary size is smaller than the BASELINE one. The most extreme case is **kmeans**, where LUTHET 4096 generates a binary smaller than BASELINE. This difference is due to the loop unrolling enabled by subsequent optimizations and the absence of `libm` symbols.

## VI. CONCLUSION AND PROSPECTIVE

*a) Conclusion:* This work explored the performance impact of combined DMA and approximate computing techniques, particularly Precision Tuning with memoization of mathematical functions. We developed SETHET and LUTHET, extensions of the TAFFO compiler plugins for LLVM, to handle DMA-aware precision tuning and automated memoization.

On the HERO platform, the proposed toolchain, leveraging the combined effect of Precision Tuning and DMA (i.e., SETHET), can provide significant speedups, ranging from $3.56\times$ to $48.4\times$, whereas the two techniques, employed in isolation, often fail at achieving more than marginal speedups. Particularly, DMA alone is very efficient on memory-bound benchmarks but provides an average speedup of 2.41 in the challenging scenario where HERO does not have an FPU. On the other hand, SETHET boosts the average speedup by an order of magnitude, up to 24.14. This result is because SETHET achieves speedups over $10\times$ in 10 configurations out of 14, whereas DMA alone only achieves such speedups in 3 cases, and never without the availability of an FPU.

Considering more complex applications, the computation of mathematical functions (in particular trigonometric functions) dominates the execution time. In this case, LUTHET boosts the performance massively in applications such as **inversek2j** ($44.8\times$ speedup over SETHET), and significantly in all other **AxBench** applications where `libm` is used ($1.4-3.6\times$).

*b) Future research directions:* Future research should consider the effectiveness of the proposed techniques when FPUs are unavailable. Low-power alternatives to conventional FPUs include reduced precision floating point formats

(a) Sobel with LUTHET 512
12.5% Relative error

(b) Sobel BASELINE

(c) Sobel with LUTHET 4096
3.4% Relative error

Fig. 8: Example of Sobel using LUTHET with different LUT size.

(bfloat16 [45]) and Posits [46]. The investigation could be also extended toward emerging low-power architectures, e.g., processing-in-memory ones, which often do not have FPUs. Allows the compiler to determine the LUT size and whether to convert certain mathematical functions into LUTs or retain them based on output quality constraints without user input. To implement this, we must not only modify SETHET, but also adjust the TAFFO framework to track the introduced errors at each compilation step instead of calculating them in the final Feedback Estimator pass.

## REFERENCES


[1] T. Hoberg, "The programmability wall," in *HiPEAC Vision 2023*, M. Duranton *et al.*, Eds., Jan 2023, pp. 120–126.

[2] S. Wang *et al.*, "Optimizing CNN computation using RISC-V custom instruction sets for edge platforms," *IEEE Transactions on Computers*, vol. 73, no. 5, pp. 1371–1384, 2024.

[3] J. Fornt *et al.*, "Mix-GEMM: Extending RISC-V CPUs for energy-efficient mixed-precision DNN inference using binary segmentation," *IEEE Transactions on Computers*, pp. 1–14, 2024.

[4] H. B. Amor *et al.*, "A RISC-V ISA extension for ultra-low power iot wireless signal processing," *IEEE Transactions on Computers*, vol. 71, no. 4, pp. 766–778, 2022.

[5] A. Krishnakumar *et al.*, "Domain-specific architectures: Research problems and promising approaches," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 2, jan 2023.

[6] Stanley-Marbell *et al.*, "Exploiting errors for efficiency," *ACM Computing Surveys*, vol. 53, pp. 1–39, 7 2020.

[7] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*, 2004, p. 162.

[8] P. Shantharama *et al.*, "Hardware-accelerated platforms and infrastructures for network functions: A survey of enabling technologies and research studies," *IEEE Access*, vol. 8, pp. 132 021–132 085, 2020.

[9] S. Cherubin *et al.*, "Tools for reduced precision computation: a survey," *ACM Computing Surveys*, vol. 53, no. 2, Apr 2020.

[10] A. Kurth *et al.*, "HEROv2: Full-Stack Open-Source Research Platform for Heterogeneous Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4368–4382, 2022.

[11] S. Cherubin *et al.*, "Dynamic precision autotuning with TAFFO," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 2, may 2020.

[12] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[13] A. Angerd *et al.*, "A framework for automated and controlled floating-point accuracy reduction in graphics applications on GPUs," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, dec 2017.

[14] R. Nobre *et al.*, "Aspect-driven mixed-precision tuning targeting GPUs," in *PARMA-DITAM workshop 2018*, 2018, pp. 26–31.

[15] I. Laguna *et al.*, "GPUMixer: Performance-driven floating-point tuning for GPU scientific applications," in *High Performance Computing*. Springer, 2019, pp. 227–246.

[16] R. Gu *et al.*, "GPU-FPtuner: Mixed-precision auto-tuning for floating-point applications on gpu," in *27th Int'l Conf on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2020, pp. 294–304.

[17] P. V. Kotipalli *et al.*, "AMPT-GA: Automatic mixed precision floating point tuning for GPU applications," in *Proceedings of the ACM International Conference on Supercomputing*, 2019.

[18] W.-F. Chiang *et al.*, "Rigorous floating-point mixed-precision tuning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '17. ACM, 2017, p. 300–315.

[19] E. Darulova *et al.*, "Sound mixed-precision optimization with rewriting," in *ACM/IEEE 9th Int'l Conf. on Cyber-Physical Systems (ICCPS)*, 2018.

[20] D. Ben Khalifa *et al.*, "Pop: A tuning assistant for mixed-precision floating-point computations," in *Formal Techniques for Safety-Critical Systems*, O. Hasan and F. Mallet, Eds. Cham: Springer International Publishing, 2020, pp. 77–94.

[21] S. Cherubin *et al.*, "Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error," in *Parallel Computing is Everywhere*, Mar 2018, pp. 297 – 306.

[22] M. O. Lam *et al.*, "Tool integration for source-level mixed precision," in *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, 2019, pp. 27–35.

[23] H. Guo *et al.*, "Exploiting community structure for floating-point precision tuning," in *Proc. 27th ACM SIGSOFT Int'l Symposium on Software Testing and Analysis, ISSTA*. ACM, 2018, pp. 333–343.

[24] D. Cattaneo *et al.*, "Embedded operating system optimization through floating to fixed point compiler transformation," in *21st Euromicro Conference on Digital System Design (DSD)*, 2018, pp. 172–176.

[25] K. Parasyris *et al.*, "HPAC: Evaluating approximate computing techniques on HPC OpenMP applications," in *Proc. ACM/IEEE Supercomputing*. ACM, 2021.

[26] A. K. Mishra *et al.*, "iACT: A software-hardware framework for understanding the scope of approximate computing," in *Workshop on Approximate Computing Across the System Stack*, ser. WACAS, 2014.

[27] L. Valente *et al.*, "HULK-V: a Heterogeneous Ultra-low-power Linux capable RISC-V SoC," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[28] D. Cattaneo *et al.*, "FixM: Code generation of fixed point mathematical functions," *Sustain. Comput. Inform. Syst.*, vol. 29, March 2021.

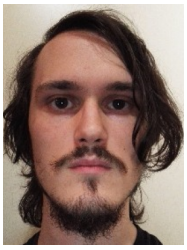
[29] ——, "TAFFO: The compiler-based precision tuner," *SoftwareX*, vol. 20, 2022.

[30] ——, "Architecture-aware precision tuning with multiple number representation systems," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 673–678.

[31] L. Denisov *et al.*, "The impact of profiling versus static analysis in precision tuning," *IEEE Access*, vol. 12, pp. 69 475–69 487, 2024.

[32] G. Magnani *et al.*, "Precision tuning in parallel applications," in *PARMA-DITAM workshop 2022*, 2022.

[33] RISC-V International, "RISC-V Specifications." [Online]. Available: https://riscv.org/technical/specifications/
[34] O. Group, "CORE-V CV32E40P User Manual." [Online]. Available: https://cv32e40p.readthedocs.io/en/latest/
[35] M. Gautschi *et al.*, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.
[36] OpenMP Architecture Review Board, "OpenMP 4.5 Complete Specifications." [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
[37] A. Marongiu *et al.*, "Supporting openmp on a multi-cluster embedded mpsoc," *Microprocessors and Microsystems*, vol. 35, no. 8, pp. 668–682, 2011, design and Verification of Complex Digital Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933111001001
[38] S. N. Agathos *et al.*, "Deploying openmp on an embedded multicore accelerator," in *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2013, pp. 180–187.
[39] Faqir-Rhazoui *et al.*, "Sycl in the edge: performance and energy evaluation for heterogeneous acceleration," *J. Supercomput.*, vol. 80, no. 10, p. 14203–14223, Mar. 2024. [Online]. Available: https://doi.org/10.1007/s11227-024-05957-6
[40] J. P. Lim, M. Aanjaneya, J. Gustafson, and S. Nagarakatte, "An approach to generate correctly rounded math libraries for new floating point variants," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3434310
[41] S. Grauer-Gray *et al.*, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, 2012.
[42] Yazdanbakhsh *et al.*, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, vol. 34, no. 2, 2017.
[43] M. Samadi *et al.*, "Paraprox: Pattern-based approximation for data parallel applications," *SIGPLAN Not.*, vol. 49, no. 4, p. 35–50, feb 2014. [Online]. Available: https://doi.org/10.1145/2644865.2541948
[44] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.
[45] Intel Corporation, "Bfloat16—hardware numerics definition," *White paper*, 2018.
[46] J. Gustafson *et al.*, "Beating floating point at its own game: Posit arithmetic," *Supercomput. Front. Innov.*, vol. 4, no. 2, 2017.



**Gabriele Magnani** Gabriele Magnani received his MSc in Computer Science and Engineering in 2021 at Politecnico di Milano, with a thesis on precision tuning of mathematically intensive programs. Currently he is PhD student, still at Politecnico di Milano, and his research interests involve the application of precision tuning in a concrete applicative context, computer security and real-time systems.

**Daniele Cattaneo** is a PhD candidate at Politecnico di Milano. He has earned his M.Sc. graduation cum laude in Computer Science and Engineering in December 2018, with a thesis work based on the architecture and implementation of a mixed precision compiler which exploits the fixed point numerical representation. His research interests involve embedded systems, compiler technology and software-hardware codesign.

**Lev Denisov** is a PhD candidate at Politecnico di Milano, Italy. He received his M.Sc. degree in Big Data Management and Analytics from TU Berlin, Germany in 2019. His research interests include approximate computing, big data and compilers.

**Giuseppe Tagliavini,** PhD, is a Tenure-Track Assistant Professor at the University of Bologna. His research interests are focused on programming models, orchestration tools, and compiler optimizations for AI-enabled resource-constrained computing platforms, with a strong emphasis on parallel architectures and accelerators in the context of ultra-low-power IoT end nodes. He is a member of the IEEE and ACM societies.

**Giovanni Agosta,** PhD, is Associate Professor at Politecnico di Milano, where he received his Ph.D. in 2004 and the MS degree (Laurea) in 2000. His research interests focus on compiler technologies for the enforcement of extra functional properties. He has published over 100 papers in international journals and conferences. He is a senior member of the ACM, a member of the HiPEAC NoE, and an associate editor of SoftwareX.

**Stefano Cherubin** Stefano Cherubin is an Associate Professor at NTNU (Norway). He received his Ph.D. in Information Engineering at Politecnico di Milano (Italy) in 2019 with a thesis on Compiler-Assisted Dynamic Precision Tuning. His research interests include compilers, and approximate computing.