Article

Self-Adaptive Run-Time Variable Floating-Point Precision for Iterative Algorithms: A Joint HW/SW Approach

Noureddine Ait Said *, Mounir Benabdenbi and Katell Morin-Allory

CNRS, Grenoble INP (Institute of Engineering University Grenoble Alpes), TIMA, University Grenoble Alpes, 38000 Grenoble, France; mounir.benabdenbi@univ-grenoble-alpes.fr (M.B.); katell.morin-allory@univ-grenoble-alpes.fr (K.M.-A.)
* Correspondence: noureddine.ait-said@univ-grenoble-alpes.fr

Abstract: Using standard Floating-Point (FP) formats for computation leads to significant hardware overhead since these formats are over-designed for error-resilient workloads such as iterative algorithms. Hence, hardware FP Unit (FPU) architectures need run-time variable precision capabilities. In this work, we propose a new method and an FPU architecture that enable designers to dynamically tune FP computations' precision automatically at run-time called Variable Precision in Time (VPT), leading to significant power consumption, execution time, and energy savings. In spite of its circuit area overhead, the proposed approach simplifies the integration of variable precision in existing software workloads at any level of the software stack (OS, RTOS, or application-level): it only requires lightweight software support and solely relies on traditional assembly instructions, without the need for a specialized compiler or custom instructions. We apply the technique on the Jacobi and the Gauss–Seidel iterative methods taking full advantage of the suggested FPU. For each algorithm, two modified versions are proposed: a conservative version and a relaxed one. Both algorithms are analyzed and compared statistically to understand the effects of VPT on iterative applications. The implementations demonstrate up to 70.67% power consumption saving, up to 59.80% execution time saving, and up to 88.20% total energy saving w.r.t the reference double precision implementation, and with no accuracy loss.

Keywords: transprecision computing; variable precision; floating-point; iterative methods; RISC-V

Citation: Ait Said, N.; Benabdenbi, M.; Morin-Allory, K. Self-Adaptive Run-Time Variable Floating-Point Precision for Iterative Algorithms: A Joint HW/SW Approach. *Electronics* **2021**, *10*, 2209. https://doi.org/10.3390/electronics10182209

Academic Editor: Alessandro Savino

Received: 30 July 2021
Accepted: 31 August 2021
Published: 9 September 2021

1. Introduction

Recently, many industrial applications have emerged in domains such as the Internet of Things (IoT), Artificial Intelligence (AI), Neural Networks (NNs), etc. with a common characteristic: inherent error-resilience. Thus, HW/SW designers could trade the precision of computations against cost, resource, and power savings for such a class of applications.

Transprecision Computing (TC) [1] is a paradigm that came to implement this vision by introducing efficient, flexible, and reconfigurable architectures adequate for such applications. TC especially targets Floating-Point (FP) arithmetic, since Floating-Point Units (FPUs) are present in most modern application classes and even embedded processors since they significantly boost computationally-intensive applications. However, an FPU is usually responsible for an extensive amount of power consumption and high memory bandwidth. Moreover, the energy consumption associated with FP arithmetic is known to be higher than that of its integer counterpart [2], making FPU optimization a priority.

A typical FP algorithm implementation flow is shown in Figure 1. It is a process that usually starts with the algorithm design and mathematical stability analysis [3–5]. The next step is often a naive implementation, where all variables are declared in high precision formats, e.g., using `double` or `long double` type variables in C/C++. Next, the process of Variable Type Optimization (VTO) [6–9] is overtaken. Its objective is to migrate as many variables as possible from high precisions to lower ones for a given constraint. For example, by carefully changing *double* variables to *float* or *long double* variables to *double*, either manually or automatically, one can still satisfy a given Quality of Result (QoR) constraint on the output. Then, designers can use arbitrary Reduced Precision (ARP) [10] for fine-grained bit-width optimization. It consists of reducing the exponent and/or mantissa bit-widths to either narrower standard bit-widths such as the 16-bit `binary16` format [11], or custom reduced arbitrary bit-widths such as Intel Nervana's Flexpoint [12], Microsoft Brainwave's 9-bit floats [13], Google TPU's 16-bit BFloats [14], and NVIDIA's 19-bit format [15].

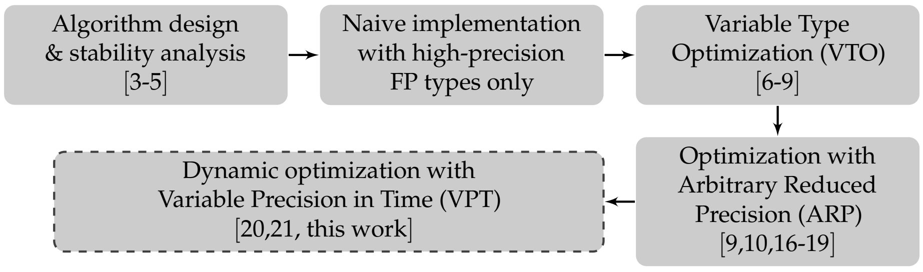
Algorithm design & stability analysis [3-5]
Naive implementation with high-precision FP types only
Variable Type Optimization (VTO) [6-9]
Optimization with Arbitrary Reduced Precision (ARP) [9,10,16-19]
Dynamic optimization with Variable Precision in Time (VPT) [20,21, this work]

Figure 1. Typical FP application implementation flow [3–10,16–21].

Many tools have been proposed to allow the functional simulation of such reduced precision operators [16–20]. However, when it comes to arbitrary reduced precision, most of these tools are only adequate for simulation without actually providing appropriate hardware architectures that leverage such concepts to quantify hardware-level gains. In this work, a new method that enables designers to automatically and dynamically tune FP computations' precision at run-time is proposed. The theoretical and statistical study performed in this work shows a drastic energy reduction for iterative algorithms. These algorithms are the cornerstone of Computer Vision applications which are widely used in the context of Artificial Intelligence. Reducing the power consumption of such applications is crucial to meet the current international trends.

This paper is an extension of our previous work [21] in which the concept of Variable Precision in Time (VPT) was introduced for the first time. VPT consists of allowing the programmer to configure computations' precision dynamically at run-time. The previous paper presented a preliminary feasibility study to evaluate the potential of the methodology. In the current paper, we focus on generalizing VPT to all stationary iterative algorithms such as Jacobi and Gauss–Seidel. For such applications, the requirements in terms of precision vary at run-time. This means that standard FP formats are over-designed for these workloads. Moreover, it is shown that even fixed reduced precision formats are over-designed for some parts of these applications. Hence, there is a need for FPU architectures that enable fine-grained VPT. A video presentation explaining our previous work is available at [22].

This paper first revisits the proposed hardware architecture and its lightweight software support in more detail (Section 4). Then the concept of VPT is pushed further by proposing two transformed iterative algorithms that self-adapt their computation precision automatically. These algorithms do not need intervention from the user or the programmer. The first is a conservative version and the second is a relaxed (approximate) version. Programmers can choose one of them depending on their application; the first provides more robust guarantees while achieving interesting energy gains, while the second has relaxed guarantees but with a higher power-efficiency. Moreover, the software implementation of these algorithms is discussed and a set of statistical studies is performed to evaluate the effectiveness of the two approaches across many inputs.

This paper proposes the following new contributions:

- Generalization of VPT to other iterative methods (Section 5) and proposition of two modified VPT-enabled algorithms with self-adaptive run-time precision (Section 6);
- An in-depth statistical verification study of the VPT's impact on Jacobi algorithm's behavior (Section 7); and

- An ASIC implementation of the proposed FPU on a 28 nm FD-SOI technology node and a post-synthesis evaluation of the proposed VPT algorithms' power consumption, execution time, and overall dissipated energy (Section 8).

Before diving into the details, some related works are presented in Section 2, and a set of definitions and the motivations behind this work are introduced in Section 3.1.

2. Related Works

The proposed approach takes advantage of non-standard arbitrary reduced operators [10,19] to achieve significant gains in terms of execution time, power consumption, and overall energy consumption. The following are some related works from the literature.

2.1. FP Variable Type Optimization (VTO)

For a given triplet {application, input dataset, QoR constraint}, tools such as [6–8] perform coarse-grained VTO using the delta-debugging search heuristic [23]. The authors' objective is to minimize the number of high-precision variables and maximize the number of low-precision variables. For some of these tools [7,8], the objective is to optimize for speed, whereas for others, such as Promise [6], the goal is to maximize the number of `float` variables.

2.2. Non-Standard/Arbitrary Precision Support

The mentioned tools support standard IEEE 754 [11] formats only, except Precimonious [8], which also supports Intel's 80-bit format implemented as `long double` in C. The authors of fpPrecisionTuning [9] proposed an arbitrary precision impact simulation methodology based on an automatic source code transformation tool. Libraries such as FlexFloat and FloatX [17,18] enable developers to simulate the impact of reduced precision on application-level Quality of Result (QoR). These tools support arbitrary precision only in simulation to help designers decide which precision is adequate for their application. However, they do not provide hardware-level support to leverage these decisions for power, execution time, or energy savings.

2.3. Mixed-Precision for Linear Algebra

There are two categories of linear system solvers: direct solvers (e.g., Gaussian Elimination, Cholesky decomposition) and iterative solvers. Carson and Higham [24,25] have proposed a general algorithm for solving linear systems based on iterative refinement [26] with three standard FP precisions. However, this method only uses standard precisions that traditional processors support. Moreover, the technique is mixed-precision in space but not in time, i.e., it contains mixed-precision instructions. Still, each instruction keeps its precision for all iterations. Authors of [27] have built on the previous work and proposed a solution with five different precisions. In contrast, new FP data types supported in NVIDIA GPUs were introduced in [28]. Finally, others [29] proposed an adaptive scheme to reduce communication overhead by selectively storing parts of the system preconditioner in different precision formats (half, single, or double). In this paper, we focus specifically on classical stationary iterative solvers such as Jacobi, Gauss–Seidel, Richardson, Successive Over-Relaxation, [30,31] etc.

2.4. Arbitrary Reduced Precision

In [10,19], we proposed a simulator (AxQEMU) (source code available at [32]) that simulates the impact of arbitrary reduced FP precision on applications in a non-intrusive way without modifying the source code. This approach is complementary to VTO since the tools mentioned before can be used as a starting point. Furthermore, using AxQEMU allows primary memory footprint optimization and enables fine-grained accuracy/energy trade-offs.

2.5. Variable Precision in Time (VPT)

VPREC [20] is a software back-end component that enables non-intrusive run-time variable precision simulation in the Verificarlo [33] software toolchain. This tool simulates run-time variable precision, specifically for iterative algorithms. However, VPREC is designed for impact simulation only in software and does not provide a hardware implementation for power reduction, whereas this work targets both software and hardware-level implementations. Moreover, VPREC evaluates the needed reduced precision for each iteration through off-line and data-dependent studies performed after the execution. In contrast, we propose a technique to automatically select the adequate precision for each iteration online at run-time in this work.

3. Background and Motivation

3.1. Definitions

A binary floating-point number can be written in the form $(-1)^s(1+m)2^e$, where s is the sign bit, m is the mantissa (also called significand or fraction), and e is the exponent. An FP number can be encoded either following standard IEEE 754 [11] formats, e.g., `binary32` (32-bit single-precision format), `binary64` (64-bit double precision format), or using a custom bit-width representation.

Definition 1. *A* **Floating-Point (FP) format** *is defined by the pair (E, M), where E is the bit-width of its* **exponent** *and M is the bit-width of its* **mantissa** *field.*

An FP format's **"precision"** refers to its mantissa bit-width M. The exponent bit-width E reflects its dynamic range. Custom non-standard (arbitrary) formats can be defined when some loss of precision is tolerated, or the numbers represented have a limited dynamic range. Hardware implementations contain an additional bit in the mantissa field, also called hidden bit [11]. In this work, the hidden bit is not counted in the mantissa bit-width M.

Definition 2. *The* **machine epsilon** *(or* **machine precision***) of an FP format (E, M) provides an upper bound on the relative error caused by rounding. It is defined as*

$$\epsilon_M = 2^{-M} \tag{1}$$

The **machine epsilon** ϵ_M also constitutes the distance from 1.0 to the next larger FP number [34].

For this work, only software Application Binary Interfaces (ABI) that support FP arithmetic in hardware (i.e., hard-float ABI) are considered. We also focus on CPU architectures and precisely on the 64-bit RISC-V [35] Instruction Set Architecture (ISA), with single-precision (F) and double precision (D) extensions. This architecture is referred to as "RV64FD" and will constitute our reference baseline FPU architecture.

In the rest of this paper, let P be the maximum number of precisions supported by the FPU and p an integer index such that $1 \leq p \leq P$. Then, let $(E_1, M_1), \ldots, (E_P, M_P)$ denote the list of reduced precision FP formats supported by the hardware.

3.2. Motivation

This section presents the motivations for this work. First, the fact that floating-point computations constitute an essential part of iterative workloads is demonstrated. Second, we show to which extent the standard FP double precision is over-designed for such applications using the AxQEMU Arbitrary Reduced Precision simulator [10,19].

3.2.1. FP Computation Usage in Jacobi and Gauss–Seidel

The first motivation behind this work is the fact that iterative algorithms such as Jacobi and Gauss–Seidel spend a lot of time performing floating-point (FP) computations. To evaluate the execution time associated with such computational operations, a cycle-accurate simulation is performed.

To do that, a cycle-accurate processor model (called CVA6 [36]) running at 200 MHz is used. The Jacobi and Gauss–Seidel algorithms have been executed on top of this model with a randomly generated input. Figure 2 depicts dynamic assembly instructions' breakdown for both Jacobi and Gauss–Seidel, i.e., how many instances of assembly instructions have been issued through the processor pipeline during the execution of the application.

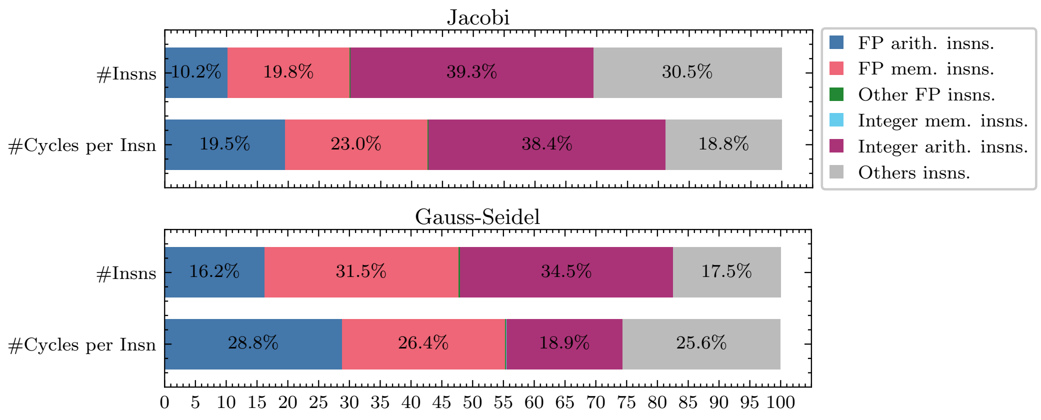

Figure 2. Dynamic instructions breakdown and number of cycles per instruction for Jacobi and Gauss–Seidel. Both applied to the same random input with a tolerance threshold $\texttt{TOL} = 10^{-12}$).

As depicted in Figure 2, for a 10^{-12} tolerance threshold, 10.2% (resp. 16.2%) of the total assembly instructions executed are FP arithmetic instructions, and 19.8% (resp. 31.5%) are FP memory instructions for Jacobi (resp. Gauss–Seidel). The remaining portions concern instructions manipulating integers (arithmetic and memory) as well as other instructions such as branch and control flow instructions, system instructions that manipulate CSRs, etc.

Although executed FP arithmetic and memory instructions are limited to 10.2% (resp. 16.2%) and 19.8% (resp. 31.5%), they constitute 19.5% (resp. 28.8%) and 23.0% (resp. 26.4%) in terms of how many cycles are actually spent on each instruction for Jacobi (resp. Gauss–Seidel).

These statistics might vary slightly from one input to another and depend on the SoC parameters (cache parameters, memory bus/interconnect, . . .). However, it is safe to say that targeting FP computation optimization for these applications is a good decision. Moreover, when the approach is combined with classic state-of-the-art techniques, designers can also benefit from low memory overhead in addition to the computation gains afforded by our technique.

3.2.2. The Limitation of Arbitrary Fixed Reduced Precision

The second motivation behind this work is the fact that even fixed ARP is over-designed for iterative algorithms. Figure 3 shows the impact of precision variation on the convergence profile (variation of the computed solution's accuracy at each iteration cf., Definition 3) of Jacobi when operating on different arbitrary reduced precisions. The reference application is simulated using the AxQEMU simulator [10,19] for multiple precisions (4, 8, . . . , 48, 52). Each of the colored lines shows the evolution of the error metric when the application is executed with arbitrary reduced precision. For each simulation, the precision is defined at launch time and stays constant at run-time.

This example demonstrates that it is unnecessary to compute all iterations with double precision. For instance, for Jacobi to reach a target 10^{-10} error threshold, the designer only needs an FP format with a 32-bit mantissa, i.e., the FP format (11, 32), which has a total bit-width of 44 bits. Similarly, for Gauss–Seidel, a 28-bit mantissa, i.e., a total bit-width of 40 bits, is sufficient to reach a target error threshold of 10^{-10}.

For Jacobi, even though the (11, 32) format will dissipate less power than the original `binary64` format, there is still room for significant improvement. For instance, the (11, 32) format is still over-designed for iterations 0 to 457. A similar case could be made for Gauss–Seidel.

As a conclusion, the standard FP formats are over-designed for these applications and ARP optimizes the energy consumption compared to the original reference precision. However, there is still room for improvement using an Arbitrary Reduced Precision variable at run-time. This would allow precision to be varied during the execution depending on the error threshold needed. In this paper, an FPU capable of computing different configurable ARPs at run-time is proposed with two modified self-adaptive algorithms that automatically vary the computation precision without user intervention.

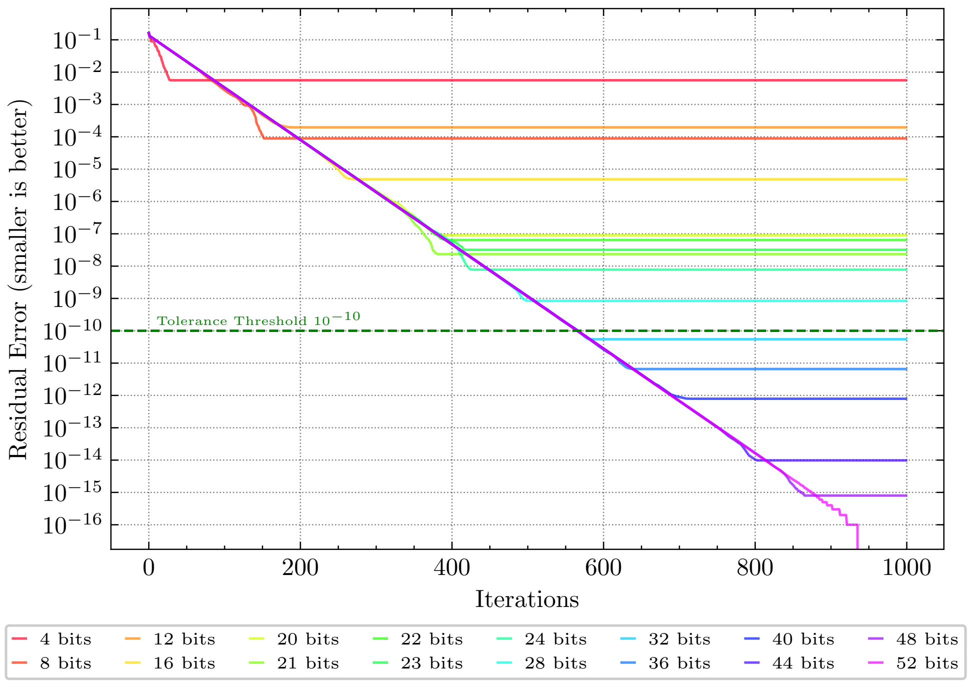

Figure 3. The variation of solution accuracy when Jacobi is executed for each fixed Arbitrary Reduced Precision.

4. Proposed Hardware Architecture

We propose a hardware FPU architecture named VPT-FPU. It enables dynamic variable precision at run-time. We also propose a lightweight software library that facilitates its integration in an existing CPU core.

4.1. Architecture Overview

VPT-FPU contains two types of data-paths: precise data-paths, that contain standard-precision arithmetic operators, and approximate data-paths, that contain reduced arbitrary precision operators. The proposed architecture (Figure 4) includes:

Ⓐ **Non-computational operations data-path:** performs non computational operations such as FP-to-integer and integer-to-FP conversion, comparison, input classification etc. In practice, the hardware/energy cost of these instructions is negligible when compared to computational instructions, so we chose to not optimize them.

Ⓑ **Multi-precision DIV/SQRT data-path** [37]: a block that computes division and square root operations using an iterative algorithm implemented in hardware [38].

Ⓒ **Precise Fused Multiply Add (FMA) data-path:** contains the precise FMA operators that perform addition, subtraction, multiplication, and multiplication-accumulation in `binary32` and `binary64` formats.

Ⓓ **Approximate FMA data-paths:** contains two sets of reduced arbitrary precision data-paths (D_F) and (D_D). Approximated `float` computations are executed in (D_F), whereas approximated *double precision* computations are executed in (D_D).

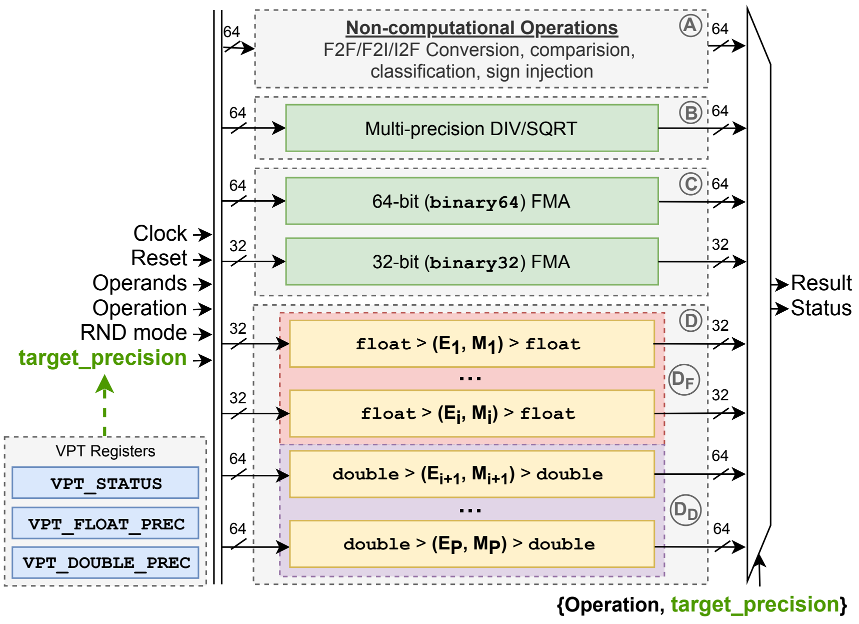

Figure 4. Hardware architecture of the Approximate Aware FPU.

For each supported non-standard FP format (E_p, M_p), $1 \leq p \leq P$, an approximate block is integrated inside either (D_F) or (D_D) at the RTL level at synthesis time. This means that the set of supported precisions is fixed, once synthesized. Each of these approximate blocks contains:

- A reduction block that converts the inputs from the original format (`binary32` or `binary64`) to the target reduced format (E_p, M_p).
- An FMA computational operator in (E_p, M_p) format.
- An extension block that converts the result from (E_p, M_p) back to the original format.

The blocks Ⓐ, Ⓑ, and Ⓒ constitute the original standard RV64FD FPU. The block Ⓑ containing DIV/SQRT operators do not need to be duplicated or surrounded by reduction and extension blocks since their precision can be adjusted via a precision selection input signal that varies at run-time. This is why only FMA blocks are duplicated.

Section 8 studies the area overhead introduced by block Ⓓ and the savings in terms of power/energy and execution time.

4.2. Custom VPT Registers

To simply and flexibly support VPT, three custom Control and Status Registers (CSR) have been added to the FPU (see Figure 4): the `VPT_STATUS` register, which is used to enable/disable the VPT mode, and the two registers `VPT_FLOAT_PREC` and `VPT_DOUBLE_PREC`, which, respectively, select the precision settings for `float` and `double` operations via software.

4.3. VPT Software Support

The added registers are memory-mapped when the FPU is integrated into a CPU core. This enables lightweight software support since read/write (R/W) operations can then be performed using the usual CSR assembly instructions, i.e., no special compiler modification is needed. The additional CSRs belong to the custom R/W user-level address space [35], which means that no particular machine- or supervisor-level privileges are needed to perform R/W operations.

A small set of Hardware Abstraction Layer (HAL) functions has been developed to enable the programmer to select `float` and `double` operations' precision via software. Of course, at the SW level, the programmer can only select a precision among the ones supported in hardware (E_p, M_p), $1 \leq p \leq P$. The proposed HAL functions are wrappers for the CSR read/write assembly instructions. For a seamless integration in existing applications, the functions are bundled as a header-only library. Listing 1 depicts a few examples. For instance, `vpt_set_prec_float` (resp. `vpt_set_prec_double`), sets the precision of `float` (resp. `double`) operations, i.e., by configuring the content of the register `VPT_FLOAT_PREC` (resp. `VPT_DOUBLE_PREC`) to `1 << M`, where M is the target arbitrary reduced precision.

Listing 1. VPT-related HAL function prototypes.

```
void    vpt_enable(void);                /* Enable  VPT mode */
void    vpt_disable(void);               /* Disable VPT mode */
uint8_t vpt_set_prec_float(uint8_t M); /* Sets the precision of float FP instructions
and returns the previous precision          */
uint8_t vpt_set_prec_double(uint8_t M);/* Sets the precision of double FP instructions
and returns the previous precision          */
```

5. Iterative Methods: Mathematical Foundations

The Jacobi and Gauss–Seidel iterative algorithms are used as case studies to evaluate the gains and the limitations of the proposed VPT-FPU. The original algorithms are presented as well as two VPT-enabled versions that benefit from the run-time variable precision capability of suggested hardware FPU.

5.1. Presentation of Jacobi and Gauss–Seidel Iterative Methods

Stationary iterative methods are algorithms that determine the solution of a square $n \times n$ system of linear equations, in the form:

$$\mathbf{A}\vec{x} = \vec{b} \tag{2}$$

Given the $n \times n$ real coefficient matrix $\mathbf{A}$ and the right-hand side n-vector $\vec{b}$, an iterative algorithm aims to find the unknown vector $\vec{x}$ satisfying Equation (2). This equation can be transformed to a fixed-point iteration [30]. The system is solved by computing, at each iteration $(k+1)$, the $(k+1)$th approximation of the vector $\vec{x}^{(k+1)}$ as a function of the previous result $\vec{x}^{(k)}$. This paper focuses on two main iterative algorithms to solve this problem based on the fixed-point method:

- The Jacobi method (published in 1845), whose formula can be written as follows:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right), \quad i = 1, 2, \ldots, n. \tag{3}$$

  where $x_i^{(k+1)}$ is the ith element of the vector $\vec{x}^{(k+1)}$.

- The Gauss–Seidel method (published in 1874), which uses a slightly different equation:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}\right) \tag{4}$$

These methods and others are well documented in the literature [30,34].

5.2. Convergence of Iterative Algorithms

To solve this problem iteratively, the convergence of the system should first be mathematically ensured. In the remainder of this paper, we assume $\mathbf{A}$ to be **strictly diagonally dominant**: the elements of matrix $\mathbf{A}$ satisfy the following condition $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$. For both Jacobi and Gauss–Seidel, this is a sufficient (but not necessary) condition to ensure the convergence for any initial guess vector $\vec{x}^{(0)}$ [30] (Theorem 4.5, p. 111).

The iterative algorithm reaches convergence when the computed approximation $\vec{x}^{(k)}$ is very close to the exact solution of the system $\vec{x}^*$, i.e., the *forward error* $\vec{e}^{(k)}$ defined below is small enough

$$\vec{e}^{(k)} = \vec{x}^{(k)} - \vec{x}^* \tag{5}$$

However, the exact solution $\vec{x}^*$ is unknown. This is why the error of the computed solution is estimated in an indirect way using a *convergence metric*. The latter is a scalar that represents the evolution of the solution accuracy at each iteration k. The idea is to compute the convergence metric at the end of each iteration and stop the iterative process when progress is no longer being made, i.e., when the computed metric is below a given user-defined threshold denoted `TOL`. The result of this comparison is called the *stopping criterion*.

For the sake of simplicity, the distance between two consecutive results is chosen as the metric. It is defined as follows:

$$||\vec{x}^{(k+1)} - \vec{x}^{(k)}|| = \sqrt{\sum_{i=1}^{n} (x_i^{(k+1)} - x_i^{(k)})^2} \tag{6}$$

where $||.||$ refers to the Euclidean norm. The metric is computed at the end of each iteration. Other metrics (e.g., residual error $||\vec{b} - \mathbf{A}\vec{x}||$) can also be used.

At the end of each iteration, the stopping criterion is evaluated to (1) identify when the *forward error* $\vec{e}^{(k)}$ is small enough to stop iterations [31] (p. 63), (2) detect when the error is no longer decreasing or decreasing too slowly, and (3) limit the maximum amount of time spent iterating.

The following definitions are provided for the remaining of this document:

Definition 3. *The* **Convergence Profile** *(CP) is the curve representing the variation of the convergence metric (e.g., $||\vec{x}^{(k+1)} - \vec{x}^{(k)}||$) through iterations k. An* **average** *CP is an average curve computed across many inputs.*

Definition 4. *The* **Precision Variation Profile** *(PVP) is the curve representing the variation of operating precision (M_p) through iterations k. An* **average** *PVP is an average curve computed across many inputs.*

6. Implementation of VPT-Enabled Iterative Methods

This part presents the implementation details of iterative algorithms.

6.1. The Original Algorithm

The following algorithm (Algorithm 1) depicts a typical implementation of an iterative method with only standard precision, (i.e., single-precision/double precision) with no custom variable precision involved.

Algorithm 1: general structure of iterative methods.

```
Inputs : $\mathbf{A}$ : a diagonally dominant matrix of size $n \times n$,
         $\vec{b}$ : the right-hand side vector of size $n \times 1$,
         $\vec{x}^{(0)}$ : an initial guess vector of size $n \times 1$,
         MAX_ITER : maximum number of iterations,
         TOL : original global error threshold.
Output: $\vec{x}^{(k+1)}$: the solution of the linear system.
1 $k \leftarrow 0$
2 repeat
3     Compute $\vec{x}^{(k+1)}$                                  ▷ Implements Equation (3) or (4)
4     metric$^{(k+1)}$ = compute_metric($\vec{x}^{(k)}$, $\vec{x}^{(k+1)}$)   ▷ Compute convergence metric (Equation (6))
5     $k \leftarrow k + 1$                                       ▷ Continue until convergence is reached
6 until metric$^{(k+1)} \leq$ TOL or $k \geq$ MAX_ITER;
7 return $\vec{x}^{(k+1)}$
```

The original algorithm contains a main loop (lines 2–6), where the elements $x_i^{(k+1)}$ of the solution vector $\vec{x}^{(k+1)}$ are computed in line 3, using either Equation (3) for Jacobi, or Equation (4) for Gauss–Seidel. Then, the convergence metric is computed according to Equation (6) (line 4). After that, the stopping criterion evaluation is performed by checking if the computed metric is lower than the tolerance threshold `TOL` or if the number of iterations k has reached its limit `MAX_ITER`. If the stopping criterion is satisfied, the algorithm stops and returns the last computed result. Otherwise, the algorithm continues until reaching convergence in the next iterations or potentially reaching the maximum iterations limit `MAX_ITER`.

6.2. The Transformed Algorithm

To take advantage of the VPT-FPU presented in Section 4, Jacobi and Gauss–Seidel algorithms should be manually transformed. Algorithm 2 depicts the general structure of an iterative method with VPT enabled. The regions added to the original baseline implementation (depicted in Algorithm 1) are colored in blue.

Our transformation methodology consists of starting the process with the lowest possible precision and increasing it gradually until convergence. In this example, both the original and the VPT-enabled algorithms are applied to the same input matrix **A** and vector $\vec{b}$ for comparison. These inputs are generated randomly with a randomization seed equal to zero. The steps of the transformation process are:

1. **Define a list of available precisions supported in hardware** $\{M_1, \ldots, M_P\}$ (input of Algorithm 2).
2. **Define a list of intermediate tolerance thresholds** $\texttt{TOL}_p$ (input of Algorithm 2).

   Since the objective is to increase precision gradually, the intermediate thresholds $\{\texttt{TOL}_0, \texttt{TOL}_1, \texttt{TOL}_2, \ldots\}$ should be defined for each one of the intermediate precisions $\{M_0, M_1, M_2 \ldots\}$.
3. **Enable VPT at the beginning of the algorithm** (line 2 of Algorithm 2).
4. **Iterate over the supported precisions to gradually improve the accuracy of the solution.**

   The outer loop (lines 3–11) iterates over the available precisions M_p. At the beginning of each outer loop iteration, the precision is set to M_p with the function `vpt_set_precision(`M_p`)` which configures the internal registers of the FPU to use the precision M_p.

   Inside the inner loop iteration (lines 6–10), the vector $\vec{x}^{(k+1)}$ as well as the metric are computed using the intermediate precision M_p:

(a) If the intermediate tolerance threshold $\texttt{TOL}_p$, corresponding to the precision M_p, is reached, the algorithm exits the inner loop and moves on to the next higher precision M_{p+1}.

(b) If the maximum number of iterations `MAX_ITER` is reached, the algorithm exits both loops and returns the last computed $\vec{x}^{(k+1)}$ vector.

(c) Otherwise, the inner loop continues with the same precision M_p until convergence or until `MAX_ITER` is reached, and the outer loop continues until scanning all available precisions or until reaching `MAX_ITER`.

This process is illustrated through the example depicted in Figure 5. It shows the evolution of the convergence profile (left axis), with iterations (horizontal axis) for the double precision reference original algorithm (continuous blue line) and the VPT-enabled algorithm (continuous red line) of Jacobi.

In this example, a set of 13 supported precisions $\{4, 8, \ldots, 52\}$ is considered (right axis) to which 13 tolerance thresholds $\{2^{-4}, 2^{-8}, \ldots, 2^{-52}\}$ are associated. The figure also shows how the computation precision M_p of the VPT-enabled algorithm increases at run-time (orange dashed line, right axis). The intermediate thresholds $\texttt{TOL}_p$ are marked in green.

As shown in Figure 5, the precision is increased from M_p to M_{p+1} when the convergence metric reaches the intermediate threshold $\texttt{TOL}_p$. Through this example, it is shown that the VPT-enabled algorithm follows the same trend and provides the same accuracy at the input while operating with much lower and auto-adaptive precision.

The presented methodology does not alter the convergence of the algorithm, and programmers can apply it to other applications, (e.g., Successive Over-Relaxation (SOR), Richardson method, etc. [30]) as long as its convergence is guaranteed mathematically. In strict logic, we verify that the input couple $(\mathbf{A}, \vec{b})$ remains strictly diagonally dominant for each reduced precision. The variation of the convergence profile for Gauss–Seidel is depicted in Figure A1 in the Appendix A.

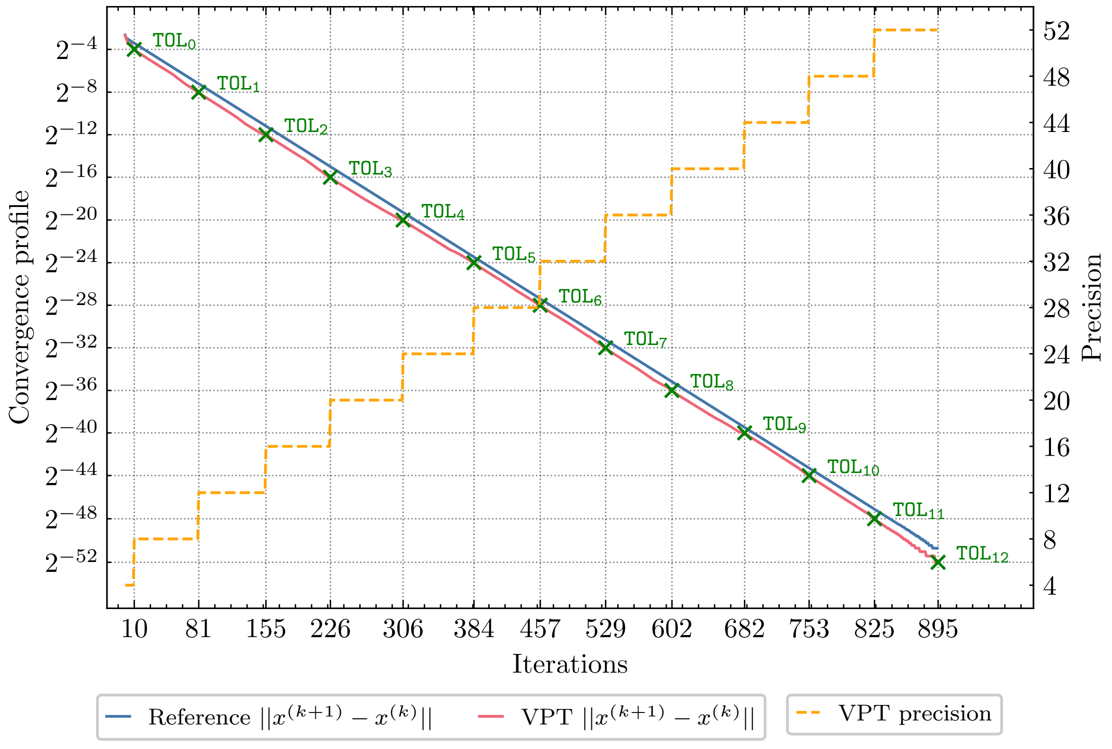

Figure 5. The convergence profile of the original and the VPT-enabled Jacobi applied to one randomly generated input.

The choice of intermediate thresholds is critical: the smaller they are, the harder they can be reached. On one hand, choosing thresholds that are easy to reach will lead to premature precision increment; hence more iterations will be spent on higher precisions, and power consumption will be increased. On the other hand, if the thresholds are very difficult to reach for a given precision, it can cause convergence stagnation. In this paper, two threshold policies are proposed for choosing these thresholds. The first one provides conservative thresholds and is explained in Section 6.2.1. The second generates smaller thresholds and handles stagnation cases. The latter is detailed in Section 6.2.2.

Algorithm 2: VPT-enabled iterative methods.

```
Inputs : A: a diagonally dominant matrix of size $n \times n$,
         $\vec{b}$ : the right-hand side vector of size $n \times 1$,
         $\vec{x}^{(0)}$: an initial guess vector of size $n \times 1$,
         MAX_ITER: maximum number of iterations,
         TOL: original error threshold.
         $M_1 \ldots M_P$: available precisions,
         $\mathtt{TOL}_1 \ldots \mathtt{TOL}_P$: intermediate error thresholds.
Output: $\vec{x}^{(k+1)}$: the solution of the linear system.
1  $k \leftarrow 0,\ p \leftarrow 0$
2  vpt_enable()                                        ▷ Enable VPT
3  repeat
4      $p \leftarrow p + 1$                            ▷ Increase precision index
5      vpt_set_precision( $M_p$ )                      ▷ Set precision to $M_p$
6      repeat
7          Compute $\vec{x}^{(k+1)}$                   ▷ Implements Equation (3) or (4)
8          metric$^{(k+1)}$ = compute_metric($\vec{x}^{(k)}$, $\vec{x}^{(k+1)}$)   ▷ Compute convergence metric (Equation (6))
9          $k \leftarrow k + 1$                        ▷ Continue until convergence is reached.
10     until metric$^{(k+1)} \leq \mathtt{TOL}_p$ or $k \geq$ MAX_ITER ;
11 until $p == P$ or $k \geq$ MAX_ITER;
12 return $\vec{x}^{(k+1)}$
```

6.2.1. Details of Threshold Policy (1): Conservative Thresholds

An intermediate threshold should be computed for each precision M_p. With this threshold policy, the computed thresholds are more conservative, i.e., they are sufficiently high, which makes them more reachable with a given precision. They are mathematically computed according to the smallest distance between two consecutive points in a given FP format.

In the case where the distance metric is used for convergence, an upper bound can be computed in the precision M_p, by assuming that the distances between all elements of $\vec{x}^{(k+1)}$ and $\vec{x}^{(k)}$ i.e., $x_i^{(k+1)} - x_i^{(k)}$ are as low as some small positive floating-point value u. This assumption provides an upper bound on the tolerance threshold equal to $u\sqrt{n}$, as explained below.

$$x_i^{(k+1)} - x_i^{(k)} \leq u \iff \sum_{i=1}^{n} (x_i^{(k+1)} - x_i^{(k)})^2 \leq n\, u^2 \tag{7}$$

$$\iff \sqrt{\sum_{i=1}^{n} (x_i^{(k+1)} - x_i^{(k)})^2} \leq u\sqrt{n} \tag{8}$$

This upper bound has to be computed and rounded to the precision M_p. For example, if u is set to ϵ_{M_p}, then $\mathtt{TOL}_p = \mathrm{round}(\sqrt{n}\,\epsilon_{M_p}, M_p)$, where $\mathrm{round}(X, M_p)$ is a function that rounds the result of a FP number X to M_p mantissa bits following one of the standard FP rounding modes, typically the round-to-nearest rounding mode.

As the convergence metric is computed in low precision, the designer should also take the rounding errors associated with the metric computation into consideration [31] (Section 4.2.5, p. 56), since the metric computation involves n multiplications, $(n-1)$ additions, plus a final square root operation (see Equation (6)). All these computations will contribute to the final computed $\texttt{TOL}_p$ values.

To compute an estimation of the thresholds, a proof assistant called Gappa [39] is used to compute the thresholds for all possible precisions $M_p \in \{1, 2, \ldots, 52\}$. Listing A1 of the Appendix A shows the Gappa script used to compute the thresholds. In this case, it is applied to a precision M_p equal to 40 bits. Furthermore, this script is run offline only once for each precision. This means that it does not add an overhead to the iterative application itself at run-time.

Example

Consider $n = 50$, $M_p = 40$, and $u = \epsilon_{40} = 2^{-40}$. If the threshold is computed ideally, the result would be $\texttt{TOL}_p = 2^{-40} \times \sqrt{50} = 6.43109\ldots10^{-12}$, which is similar to Gappa's result. However, for $M_p = 4$, the ideal result would be $\texttt{TOL}_p = 0.44194\ldots$, whereas using Gappa, it provides the result 0.5. The latter is more conservative and takes rounding error into account.

6.2.2. Details of Threshold Policy (2): Relaxed Thresholds with Stagnation Detection

This threshold policy is more "relaxed" in the sense that the chosen intermediate thresholds are as low as possible to maximize the number of iterations spent on lower precisions.

Definition of Convergence Stagnation

Choosing very low thresholds is riskier since there are no guarantees that the convergence metric can actually go as low as the specified intermediate $\texttt{TOL}_p$ thresholds, i.e., there is a chance that the convergence metric will stagnate at a specific value or oscillate around it. Hence, it is important to consider this effect when choosing to lower down the selected convergence thresholds. This process will be referred to as "stagnation detection". The stagnation behavior has been observed only for a small subset of the tested inputs, yet it is important to consider when the thresholds are selected.

Example of Convergence Stagnation

Figure 6 depicts the convergence profile and the precision variation profile for three separate cases 0, 1, and 2 (each one has a randomly generated matrix $\mathbf{A}$ and vector $\vec{b}$). In this case, $\texttt{TOL}_p$ is set to ϵ_{M_p} for each precision M_p and a discrete subset of precisions, i.e., $M_p \in \{4, 8, 12, \ldots, 48, 52\}$ is chosen so that the total bit-width $(1 + 11 + M_p)$ of the reduced VPT-FPU formats are multiples of 4 bits.

As shown in Figure 6, input matrices 0 and 1 converge normally without problems. However, the distance metric associated with input matrix 2 oscillates between two values 0.14312744140625 (which is equal to $2^{-6.1265}$) and 0.13732910156250 (which is equal to $2^{-6.1862}$) while operating in 8-bit precision. The residual error $\vec{r}^{\,(k)}$ (i.e., $||\vec{b}^{\,(k)} - \mathbf{A}\vec{x}^{\,(k)}||$) also stagnates at the value 0.0000889301300049. For an 8-bit mantissa, the expected convergence threshold is normally 2^{-8}, which seems to be difficult to reach for input matrix 2. Hence the importance of being able to detect such stagnation cases.

Figure A2 of the Appendix A shows the same phenomenon for the residual error metric.

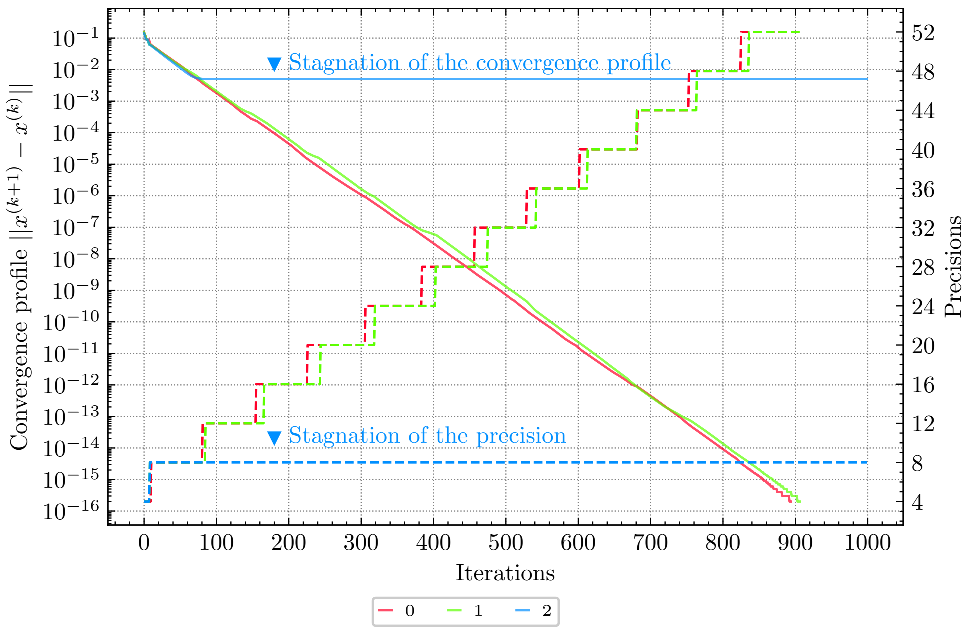

Figure 6. Convergence profiles (left axis, continuous lines) and their corresponding precision variation profiles (right axis, dotted lines) for three inputs (Matrix IDs 0, 1, and 2). The convergence profile here is in terms of the distance metric ($||\vec{x}^{(k+1)} - \vec{x}^{(k)}||$).

The Proposed Convergence Stagnation Detection Mechanism

To avoid the stagnation of the convergence profile, a condition to the stopping criterion should be added to detect this stagnation or oscillation phenomenon [31] (Section 4.2.4, p. 56). For that, we define the **Stagnation Maximum Iterations** (denoted `SMI`) as the maximum number of iterations for which stagnation can be tolerated. If the convergence metric stagnates at a fixed level or oscillates, the precision should be increased if higher precision is available. Otherwise, if there is no higher available precision, then the algorithm is stopped.

A comprehensive VPT-enabled iterative algorithm is proposed (Algorithm 3) to account for the convergence stagnation effect. In addition to the standard version of the algorithm (black-colored statements) and the original VPT statements (colored in blue), additional instructions (colored in red) have been added to implement stagnation detection.

From an implementation point of view, an additional integer variable should be added to keep stagnation iterations count, as well as one subtraction instruction and one comparison to check if two consecutive iterations have similar or very close distance metric ($||\vec{x}^{(k+1)} - \vec{x}^{(k)}||$) values (line 11). This overhead is negligible compared to the cost of the main iteration computations, which is confirmed by the performed energy consumption study.

Algorithm 3: VPT-enabled iterative algorithm with stagnation detection.

```
Inputs : A: a diagonally dominant matrix of size $n \times n$,
         $\vec{b}$: the right-hand side vector of size $n \times 1$,
         $\vec{x}^{(0)}$: an initial guess vector of size $n \times 1$,
         MAX_ITER: maximum number of iterations,
         TOL: original error threshold.
         $M_1 \dots M_P$: available precisions,
         $TOL_1 \dots TOL_P$: intermediate error thresholds.
         SMI: stagnation maximum iterations.
Output: $\vec{x}^{(k+1)}$: the solution of the linear system.
1  $k \leftarrow 0$, $p \leftarrow 0$
2  vpt_enable()                                              ▷ Enable VPT
3  repeat
4      $p \leftarrow p + 1$                                  ▷ Increase precision index
5      stag_counter $\leftarrow 0$                           ▷ Initialize stagnation counter
6      vpt_set_precision( $M_p$ )                            ▷ Set precision to $M_p$
7      repeat
8          Compute $\vec{x}^{(k+1)}$                         ▷ Implements Equation (3) or (4)
9          metric$^{(k+1)}$ = compute_metric($\vec{x}^{(k)}$, $\vec{x}^{(k+1)}$)   ▷ Compute convergence metric (Equation (6))
10         $k \leftarrow k + 1$                              ▷ Continue until convergence is reached
11         if |metric$^{(k+1)}$ − metric$^{(k)}$| $\leq$ DBL_MIN (The value of the macro DBL_MIN is the minimum normalized positive
             floating-point number that is representable in double precision format. It is provided by the header float.h from
             the standard C library) then
12             stag_counter $\leftarrow$ stag_counter + 1    ▷ Increase stagnation counter
13         end
14         else
15             stag_counter $\leftarrow 0$                   ▷ Reset stagnation counter
16         end
17     until metric$^{(k+1)} \leq TOL_p$ or $k \geq$ MAX_ITER or stag_counter == SMI ;
18 until $p == P$ or $k \geq$ MAX_ITER;
19 return $\vec{x}^{(k+1)}$
```

6.2.3. Comparing Different Threshold Policies

The different kinds of intermediate thresholds resulting from each threshold policy are studied and compared to understand how conservative or relaxed they are. Figure 7 depicts a comparative study of these thresholds.

For each precision $M_p \in \{1, 2, \dots, 52\}$ three versions of intermediate thresholds $\mathtt{TOL}_p$ are computed:

- **Relaxed thresholds**: by considering ϵ_{M_p} for each $M_p \in \{1, 2, \dots, 52\}$.
- **Theoretical conservative thresholds**: computed using the formula $\sqrt{n}\,\epsilon_{M_p}$ and rounded to the nearest.
- **Conservative thresholds computed with Gappa**: these are generated using the Gappa proof assistant as explained in Section 6.2.1 to take rounding errors into account.

Figure 7 shows no meaningful difference between the two conservative threshold versions, except for very low precisions ($M_p < 6$). Thus, in practice, using one or the other does not change the results since it only affects the threshold associated with the precision $M_p = 4$.

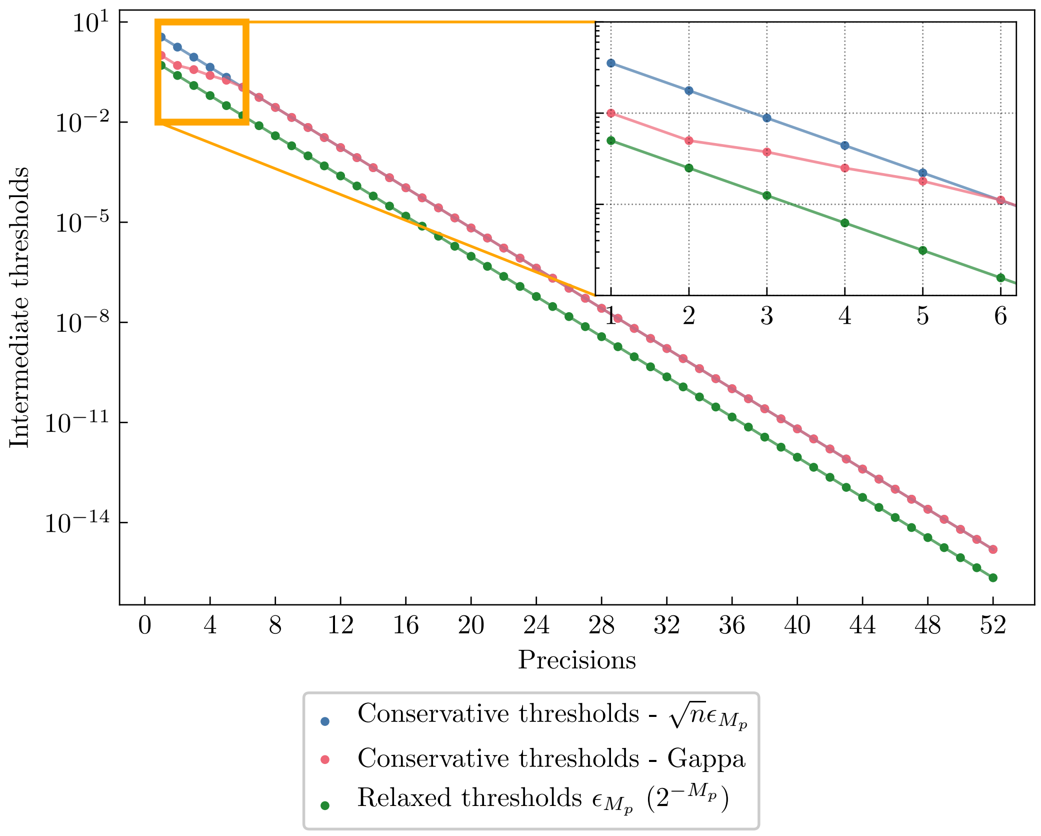

Figure 7. Thresholds comparison.

7. Statistical Analysis

Section 6.2, demonstrated the effect of VPT by applying it to a single randomly generated input. In this section, the behavior of VPT and its gains will be evaluated across a set of 1000 randomly generated input $(\mathbf{A}, \vec{b})$.

The first objective of this section is to statistically study the effects of VPT on the generated 1000-input set. The second is to study and compare the influence of each convergence threshold policy (conservative thresholds vs. relaxed thresholds) on the convergence profile, the precision variation profile, the total number of iterations, the distribution of iterations across the intermediate precisions.

To perform the statistical study, each input will be identified by an ID ranging from 0 to 999. Then, each input is generated using a randomization seed equal to its ID. This section only studies the Jacobi iterative method, but the conclusions also apply to Gauss–Seidel and similar iterative methods.

7.1. Software Implementation Aspects

In the following sections, an open-source Jacobi implementation [40] written in C is considered. It takes as an input a randomly generated 50×50 diagonally dominant matrix $\mathbf{A}$, with pseudo-random values between 0 and 1. The default target tolerance threshold is set to 10^{-10} unless otherwise is specified. Gauss–Seidel is also implemented following the same structure of the original Jacobi application.

The software applications have been cross-compiled for the Proxy Kernel execution environment (a lightweight bare-metal-like OS dedicated to RISC-V-based systems) [41], using the RISC-V GCC compiler in a similar fashion to the cycle-accurate study performed earlier in Section 3.2.1.

7.2. Effects of VPT on the Convergence Profile and Precision Variation Profile

Considering the two threshold policies and their respective parameters presented in Section 6.2, five different use cases are established to be evaluated and compared:

1. **Standard baseline results.**

   In this case, the original algorithm (Algorithm 1) double precision implementation is applied to the randomly generated 1000-input set. Figure 8 depicts the average convergence profile (continuous blue line). The area highlighted in light-blue covers the range of possible convergence profiles obtained for the 1000 inputs. This case is represented by the blue line (Figures 8–10).

2. **VPT-enabled results, with threshold policy (1), using $\sqrt{n}\,\epsilon_{M_p}$ conservative thresholds.**

   In this case, the VPT-enabled algorithm (Algorithm 1) is implemented. Here, the thresholds are computed with the formula below as explained in Section 6.2.1.

   $$\text{round}(\sqrt{n}\,\epsilon_{M_p}, M_p) \tag{9}$$

   The corresponding convergence profile is depicted in Figure 9, which overlaps with the original standard baseline convergence profile. This means that in this case, the convergence speed does not change compared to the reference even though lower precisions are used. This case is represented by the orange line (Figures 9 and 10).

3. **VPT-enabled results, with threshold policy (1), using conservative thresholds generated with Gappa.**

   In this case, the VPT-enabled algorithm (Algorithm 1) is implemented. Here the thresholds computed using the Gappa proof assistant are used (Section 6.2.1). Please note that these thresholds are similar to the latter case except for low precisions $1 \leq M_p \leq 5$. This case is represented by the green line (Figures 9 and 10).

4. **VPT-enabled, with threshold policy (2), along with ϵ_{M_p} thresholds and the Stagnation Maximum Iterations set to 2 (`SMI = 2`).**

   In this case, the VPT-enabled algorithm (Algorithm 2) is implemented. The convergence thresholds used here are more relaxed, and the stagnation detection mechanism presented in Section 6.2.2 is activated. For this experiment, stagnation is tolerated for at most two consecutive iterations before incrementing the precision. This case is represented by the red line (Figures 9 and 10).

5. **VPT-enabled, with threshold policy (2), along with ϵ_{M_p} thresholds and the Stagnation Maximum Iterations set to 4 (`SMI = 4`).**

   This case is represented by the purple line (Figures 9 and 10).

Figure 9 depicts the average convergence profiles for each one of the five use cases. The conservative thresholds (overlapping orange and green) produce a convergence profile similar to the standard baseline (blue). The relaxed thresholds (overlapping red and purple) also result in similar convergence profiles on average, although its speed slows down compared to the baseline starting from iteration 700.

Figure 10 shows the precision variation profile for the five cases. For the baseline reference, the operating precision is fixed at 52 bits, i.e., double precision format for all iterations. Both conservative threshold sets ($\sqrt{n}\,\epsilon_{M_p}$ and Gappa) produce a similar overlapping precision variation profile (green and orange). Relaxed thresholds also produce similar overlapping profiles (red and purple) indifferent to the value of `SMI`.

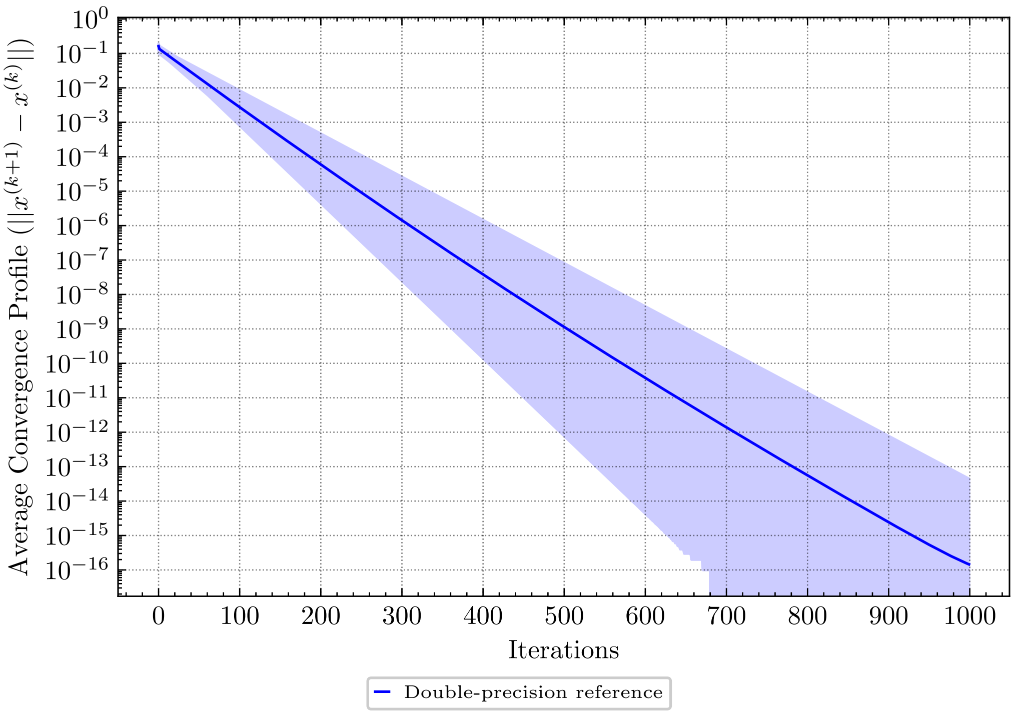

Figure 8. The average convergence profile of the reference double precision Jacobi.

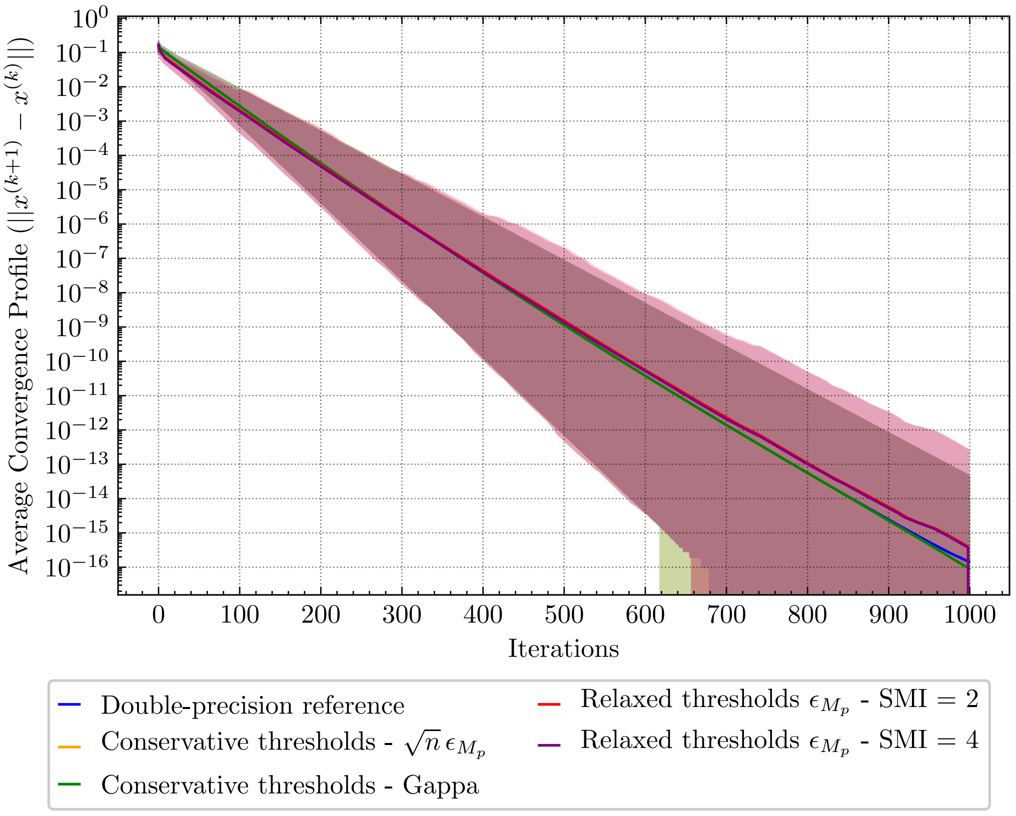

Figure 9. Comparison of average convergence profiles for each use case.

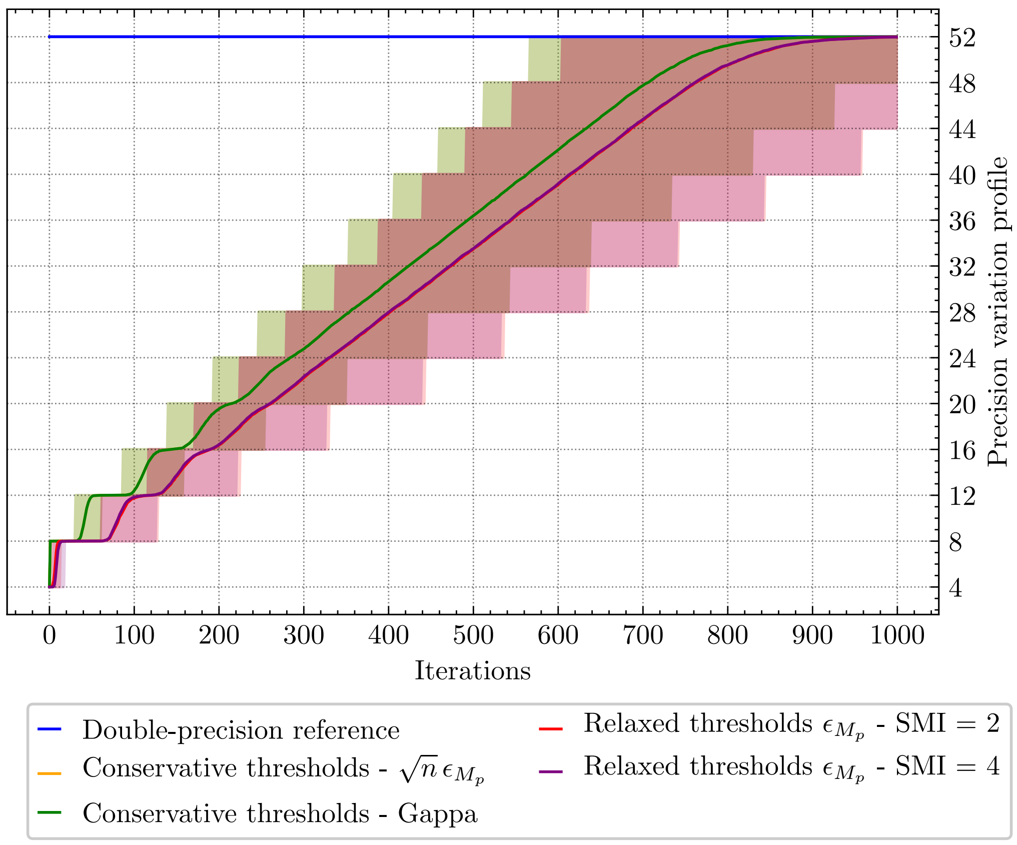

Figure 10. Comparison of average precision variation profiles for each use case.

7.3. Effects of VPT on the Total Number of Iterations

In this paragraph, the effect of VPT on the total number of iterations is studied by analyzing each one of the five use cases stated previously. Figures 11 and 12 show the distribution of the total number of iterations (vertical axis) for each use case (horizontal axis), for two target thresholds 10^{-4} and 10^{-12}, respectively. To get these data, each version of Jacobi has been executed on the 1000 inputs. The figure represents the medians (red lines in the middle of the box plot) and the means (black dot). The blue crosses (+) represent non-significant outliers.

For higher target thresholds, and specifically 10^{-4} in this case (Figure 11), there is a clear effect on iterations' number distribution: using the relaxed thresholds induces less time to converge. It is also clear that an `SMI` value of 4 achieves slightly less number of iterations compared to an `SMI` of 2. On the other hand, threshold policy (1) (conservative thresholds) shows no noticeable effect on the number of iterations.

When the target tolerance threshold is set to 10^{-12} (Figure 12), no significant effect is observed. However, the relaxed thresholds lead to slightly higher iteration numbers on average compared to the double precision reference.

There seems to be an effect on the outliers (+). Although this is not statistically significant, a worst-case and best-case HW-level comparison will be performed to evaluate to which extent this increase affects the power, energy, and execution time savings.

Figures 13 and 14 present the distribution of the overheads brought by each threshold policy with respect to the total number of iterations for the whole 1000-input set for a 10^{-04} and 10^{-12} tolerance threshold, respectively. Positive percentage values signify an increase in the number of iterations, whereas negative ones mean that there was a reduction in the number of iterations. Furthermore, the conservative policies do not alter the number of iterations much compared to the double precision reference: they achieve a bit less than $\pm 1.5\%$. However, the relaxed policies have a slightly important but scattered effect on iterations' count. For example, for a threshold of 10^{-4} (Figure 13), relaxed policies achieve a

-17% to -1% reduction in the number of iterations for more than 75% of inputs. At 10^{-12} (Figure 14), nearly half of the input dataset achieves a variation between -1% and $+2.5\%$.

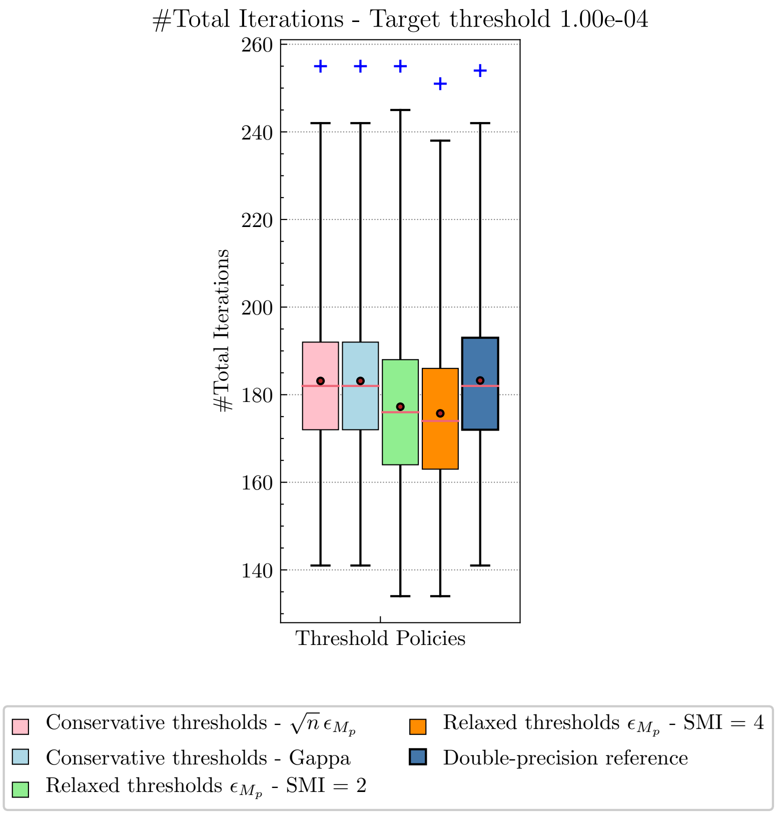

Figure 11. Total number of iterations for `TOL` $= 10^{-4}$.

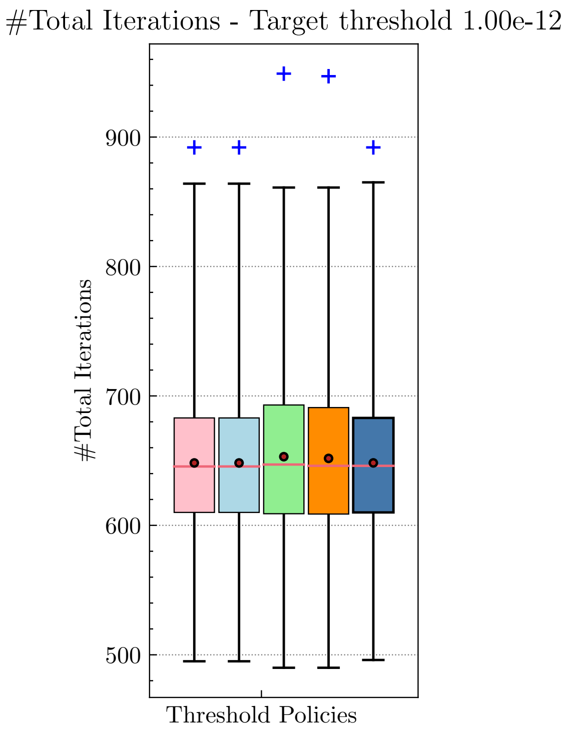

Figure 12. Total number of iterations for `TOL` $= 10^{-12}$.

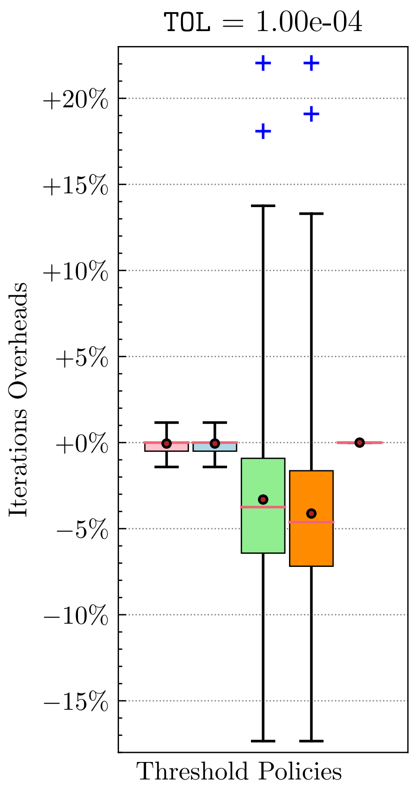

Figure 13. Total number of iterations' overhead w.r.t each policy for TOL $= 10^{-4}$.

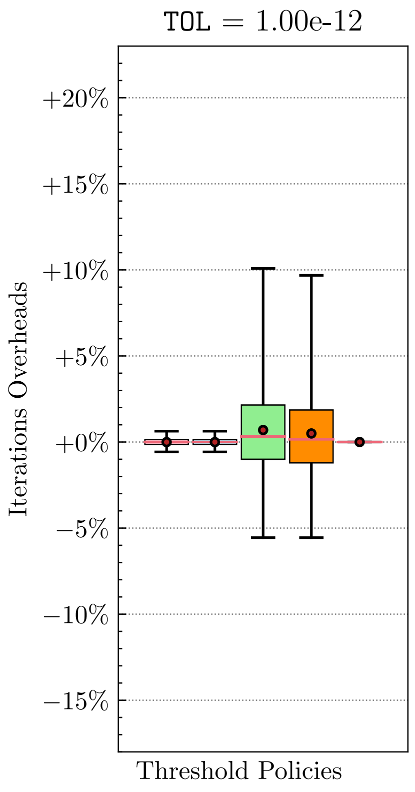

Figure 14. Total number of iterations' overhead w.r.t each policy for TOL $= 10^{-12}$.

7.4. Effects of VPT on Iterations' Distribution

Studying the effects on the total number of iterations alone is insufficient to understand the consequences of applying VPT to such algorithms fully. Hence the necessity of also studying the distribution of iterations across the intermediate precisions. The reason for that is that, even though two use cases have the same total number of iterations, some cases will tend to over-use lower precisions more than the higher ones or vice versa, hence leading to potentially different power/energy consumption.

Figures 15 and 16 illustrate a study similar to the one performed in the last section, this time by considering the distribution of the number of iterations per each precision for the 1000 inputs.

Figure 15 depicts the study when the target threshold is set to 10^{-4}. As you can see, threshold policy (2) (relaxed thresholds) tends to maximize iterations at the lower precisions, whereas threshold policy (1) (conservative thresholds) tend to maximize iterations at the higher precisions. Thus, if the two policies have the same total number of iterations, it is more likely that most of these iterations will be skewed towards lower precisions for threshold policy (2) and skewed to higher precisions for threshold policy (1). This will translate into an important difference in terms of power consumption.

Figure 16 provides a similar study with a target threshold of 10^{-12} to examine the previous trend in the long term. The figure shows that the earlier conclusion holds since threshold policy (1) seems to minimize iterations at lower precisions and maximize them at higher ones. For example, at $M_p = 44$, most cases already finished their execution when operating with threshold policy (2), whereas they have around 45 iterations left when working with threshold policy (1). Moreover, those 45 iterations are executed in higher precision, which means that there will be significant overhead in terms of energy compared to threshold policy (1).

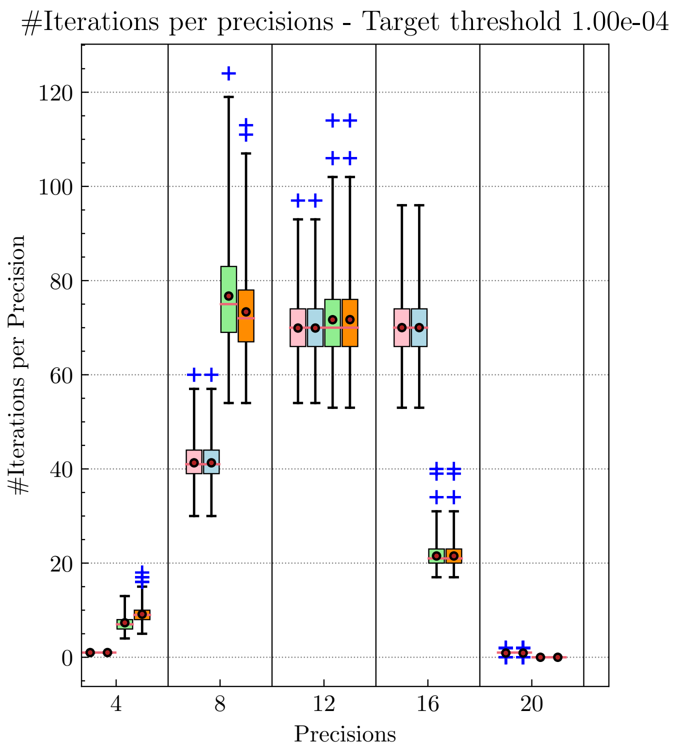

Figure 15. Iterations' distribution per precision for $\texttt{TOL} = 10^{-4}$.

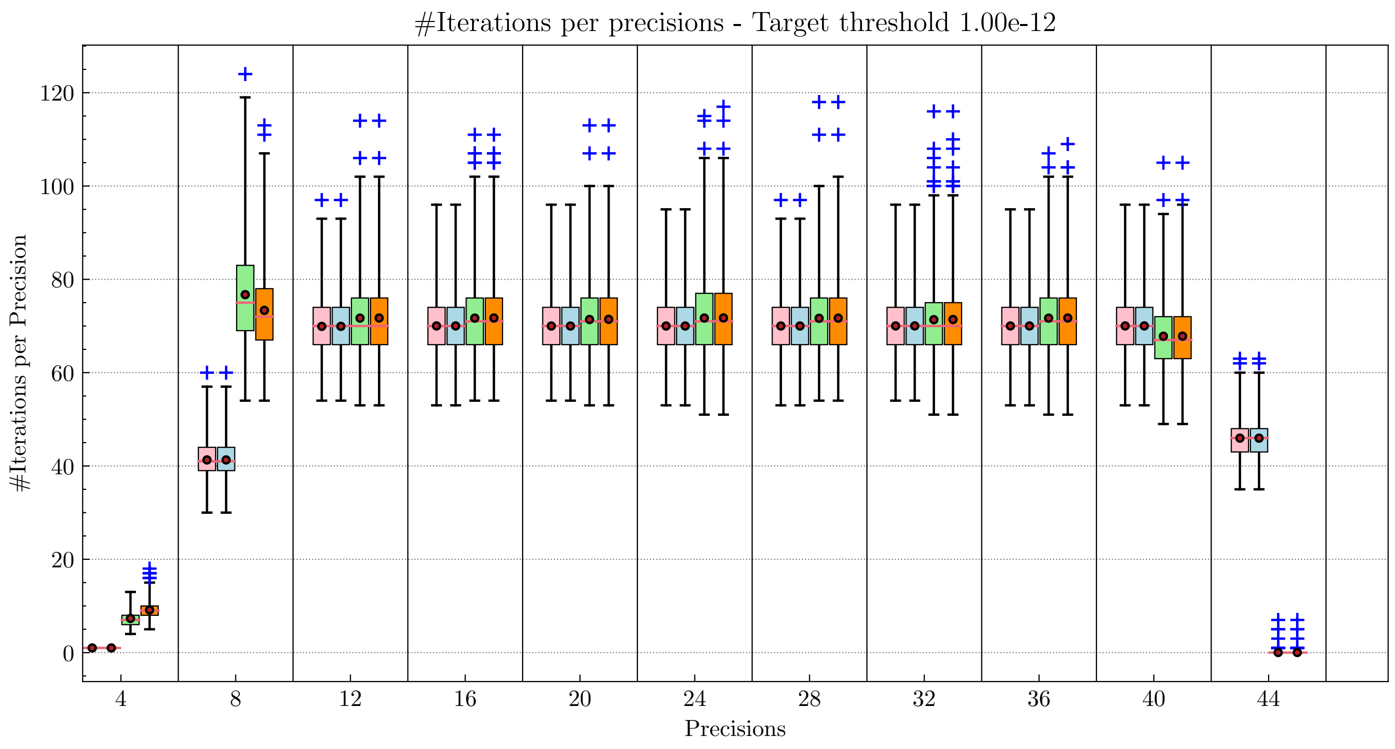

Figure 16. Iterations' distribution per precision for `TOL` $= 10^{-12}$.

Conclusions

The conservative thresholds provide a similar convergence profile to the baseline reference, whereas the relaxed thresholds tend to be slightly slower at the end of convergence. However, the relaxed thresholds tend to use less precision, which will translate to lower power consumption.

In the next section, a hardware-level power, execution time, and total computational energy evaluation will be performed for a single (typical) input and then on two cases among the 1000 random inputs.

8. Hardware-Level Evaluation & Discussion

After assessing the effectiveness of the proposed approach from a statistical and software point of view, this section presents a hardware-level evaluation of the power, execution time, and energy savings related to computations occurring inside the FPU.

8.1. Hardware Synthesis Conditions

The presented VPT-FPU was implemented in SystemVerilog, based on an open-source parametrized FPU [42]. The hardware design is globally parametric so that the list of supported formats $\{(E_1, M_1), \ldots, (E_P, M_P)\}$ in hardware are variable at design time.

The implementation was synthesized as an ASIC, on a 28-nm FD-SOI technology node, in the typical corner (Regular V_t, 1.00 V, 25 °C, No Body Biasing) for a 200-MHz frequency target. Synthesis has been performed on Synopsys Design Compiler® with automatic clock-gating enabled and default effort levels. In addition, Post-synthesis gate-level simulations were performed using Synopsys VCS®, and power consumption was estimated by considering both static power and dynamic switching activity associated with the application using Synopsys PrimeTime®.

8.2. HW-Level Evaluation with One Input and Relaxed Thresholds (Nominal Scenario)

This section will quantitatively evaluate the hardware savings and overheads introduced by VPT in terms of execution time, average power consumption, and overall energy. In this subsection, only one randomly generated matrix **A** and vector $\vec{b}$ is considered (the same one presented in Section 6.2).

8.2.1. Evaluation Methodology

Table 1 depicts the evaluation results for each error threshold `TOL` $\in \{10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}\}$. Experiments are performed for each {SW configuration, HW configuration} pair, defined as follows:

Definition 5. *A* **SW configuration** *specifies which algorithm is implemented and what variable types are used in the source code implementation.*

Definition 6. *A* **HW configuration** *specifies the set of supported reduced floating-point formats* $\{(E_0, M_0), (E_1, M_1), \ldots, (E_P, M_P)\}$ *and to which approximate data-path they belong* $\textcircled{D_D}$ *or* $\textcircled{D_F}$*, as explained in Section 4.1 and depicted in Figure 4.*

For each tolerance threshold, post-synthesis results are provided for three different SW configurations of each iterative algorithm:

1. *Ref double*: an implementation of the original algorithm (Algorithm 1) in double precision format. FP computations are executed in `binary64` (11, 52) format within the $\textcircled{C}$ data-path.
2. VPT *double*: an implementation of the VPT-enabled algorithm (Algorithm 2) in double precision format. FP computations are executed within the $\textcircled{D_D}$ data-path of the VPT-FPU, which is populated with the supported precisions (4, 8, 12, . . .). All corresponding exponents are 11-bit.
3. VPT *float*: an implementation of the VPT-enabled algorithm (Algorithm 2) in single-precision format. FP computations are executed within the $\textcircled{D_F}$ data-path of the VPT-FPU, which is populated with the supported precisions (3, 7, 11, . . .). All corresponding exponents are 8-bit. This SW configuration takes advantage of both VPT and the classic VTO technique.

Seven different HW configurations: VPT_A, VPT_B, . . . VPT_G are provided (Table 1).

Baseline Results

The baseline performance results are obtained by executing the reference SW configuration *Ref double* on the reference RV64FD hardware FPU. This experiment is repeated for each tolerance threshold. All upcoming results will be normalized w.r.t this one.

VPT Results

For each tolerance threshold and each {SW configuration, HW configuration} pair, a gate-level simulation is performed. The absolute values of execution times, average power, and the dissipated energy spent on FP computations are reported in Table 1 for Jacobi and Gauss–Seidel. The results were obtained by applying the algorithm on a single input matrix **A** and vector $\vec{b}$ auto-generated with a randomization seed equal to zero. Column-wise normalized values (%) are reported by dividing the estimated values by the reference ones for each target error threshold, i.e., there is a reference for each target threshold column.

Table 1. Post-synthesis and gate-level simulation results for Jacobi and Gauss–Seidel applied to 1 input with relaxed thresholds (ϵ_{M_p}).

Target Error Thresholds		**10^{-4}**		**10^{-6}**		**10^{-8}**	**10^{-10}**	**10^{-12}**
VPT SW Configurations		**VPT Float**	**VPT Double**	**VPT Float**	**VPT Double**	**VPT Double**	**VPT Double**	**VPT Double**
M_p precisions	D$_D$ ($E_p = 11$)	-	4, 8, 12, 16	-	4, 8, 12, 16, 20	4, 8, 12, 16, 20, 24, 28	4, 8, 12, 16, 20, 24, 28, 32, 36	4, 8, 12, 16, 20, 24, 28, 32, 36, 40
	D$_F$ ($E_p = 8$)	3, 7, 11, 15	-	3, 7, 11, 15, 19	-	-	-	-
VPT-FPU Config. name		VPT_A	VPT_B	VPT_C	VPT_D	VPT_E	VPT_F	VPT_G
Area overhead		**1.19×**	**1.27×**	**1.30×**	**1.39×**	**1.78×**	**2.30×**	**2.63×**
Jacobi Results								
Average Power (W)	Switching	2.44×10^{-4}	3.01×10^{-4}	2.95×10^{-4}	4.24×10^{-4}	4.61×10^{-4}	4.36×10^{-4}	4.77×10^{-4}
	Internal	4.97×10^{-4}	6.14×10^{-4}	5.83×10^{-4}	8.03×10^{-4}	8.74×10^{-4}	8.72×10^{-4}	9.39×10^{-4}
	Leakage	2.90×10^{-5}	3.10×10^{-5}	3.20×10^{-5}	3.40×10^{-5}	4.30×10^{-5}	5.60×10^{-5}	6.40×10^{-5}
	Total	7.70×10^{-4}	9.46×10^{-4}	9.10×10^{-4}	1.26×10^{-3}	1.38×10^{-3}	1.36×10^{-3}	1.48×10^{-3}
Total power savings (%)		**79.09%**	**74.31%**	**75.3%**	**65.77%**	**62.55%**	**62.9%**	**59.71%**
Execution time (ps)		5.96×10^{9}	5.35×10^{9}	1.00×10^{10}	1.00×10^{10}	1.56×10^{10}	1.32×10^{10}	1.65×10^{10}
Execution time savings (%)		**47.49%**	**52.85%**	**45.72%**	**45.72%**	**39.17%**	**59.77%**	**58.70%**
Energy (pJ)		4.59×10^{6}	5.06×10^{6}	9.12×10^{6}	1.26×10^{7}	2.15×10^{7}	1.80×10^{7}	2.45×10^{7}
Energy savings (%)		**89.02%**	**87.89%**	**86.59%**	**81.46%**	**77.21%**	**85.07%**	**83.35%**
Gauss–Seidel Results								
Average Power (W)	Switching	2.11×10^{-4}	2.20×10^{-4}	2.83×10^{-4}	3.26×10^{-4}	3.74×10^{-4}	4.50×10^{-4}	5.89×10^{-4}
	Internal	4.40×10^{-4}	4.93×10^{-4}	5.57×10^{-4}	6.59×10^{-4}	7.47×10^{-4}	8.81×10^{-4}	1.09×10^{-3}
	Leakage	2.90×10^{-5}	3.10×10^{-5}	3.20×10^{-5}	3.40×10^{-5}	4.30×10^{-5}	5.60×10^{-5}	6.40×10^{-5}
	Total	6.80×10^{-4}	7.44×10^{-4}	8.72×10^{-4}	1.02×10^{-3}	1.16×10^{-3}	1.39×10^{-3}	1.74×10^{-3}
Total power savings (%)		**80.06%**	**78.18%**	**74.97%**	**70.75%**	**66.92%**	**60.79%**	**50.83%**
Execution time (ps)		1.68×10^{8}	1.68×10^{8}	2.54×10^{8}	2.83×10^{8}	4.10×10^{8}	5.38×10^{8}	7.12×10^{8}
Execution time savings (%)		**51.61%**	**51.70%**	**51.19%**	**45.76%**	**41.00%**	**38.03%**	**31.70%**
Energy (pJ)		1.14×10^{5}	1.25×10^{5}	2.22×10^{5}	2.88×10^{5}	4.77×10^{5}	7.46×10^{5}	1.24×10^{6}
Energy savings (%)		**90.35%**	**89.46%**	**87.78%**	**84.15%**	**80.49%**	**75.72%**	**66.40%**

8.2.2. Discussion

VPT Double vs. Ref Double

Table 1 shows that the VPT-enabled implementations always achieve better performance with no accuracy loss compared to the reference. For example, the *VPT double* SW configuration of Jacobi achieves power consumption savings ranging from 59.71% up to 74.31%, execution time savings ranging from 39.17% up to 59.77%, and energy savings between 77.21% and 87.89%. Meanwhile, the same SW configuration for Gauss–Seidel application achieves power consumption savings ranging from 50.83% up to 78.18%, execution time savings ranging from 31.70% up to 51.70%, and energy savings between 66.40% and 89.46%.

VPT Float vs. Ref Double

Analyzing the two first columns (10^{-4} and 10^{-6}) reveals the importance of traditional Variable Type Optimization (VTO) techniques in floating-point. For example, in the case of Jacobi, when `TOL` equals 10^{-4}, the *VPT double* SW configuration saves 74.31% of the original total power and 87.89% of the energy consumed by the *Ref double* configuration. However, if VPT is applied along with VTO, i.e., if FP variables are migrated from `double` to `float` in the VPT-enabled algorithm, the power savings are further optimized up to 79.09%, and the energy savings up to 89.02%. Similarly, at 10^{-6} error threshold, optimizing variable types increases energy savings brought by VPT from 81.46% up to 86.59%. This observation also holds for Gauss–Seidel.

Conclusions

To sum up, starting from an already optimized standard software version (using VTO) instead of a double precision, one can achieve higher power and energy savings. Using the VPT approach allows to refine and enhance VTO savings one step further. Even though the additional VTO gains shown may seem limited to a few percentage points in terms of computations, starting from an already optimized version (using VTO) guarantees subsequent memory footprint savings too. However, this is only feasible for 10^{-4} and 10^{-6} target error thresholds which are reachable using single-precision.

8.3. Worst Case/Best Case HW-Level Evaluation

With everything said in Section 7 in mind, it is important to perform an empirical HW-level evaluation. Only edge cases where VPT affects either negatively or positively the energy consumption of the algorithm are considered since it would take months to perform the HW-level study for all the 1000 inputs. Moreover, for the sake of concision, only the conservative thresholds generated with Gappa in the case of threshold policy (1) are considered. Furthermore, for threshold policy (2), only the case where `SMI` is set to 4 is evaluated.

The edge cases have been chosen from the 1000-input dataset based on the software simulation presented in Section 7. They correspond to the input couples $(\mathbf{A}, \vec{b})$ that produce the lowest and the biggest iterations' overhead for each threshold in Figures 13 and 14. Table 2 lists the nomenclature that will be used in the remaining of this paper. For example, BC-04 refers to the input couple $(\mathbf{A}, \vec{b})$ that represents the best case for a 10^{-4} tolerance threshold.

To ensure a fair comparison from a hardware standing point, the following experiments are all run on top of a VPT-FPU hardware configuration that will be referred to as **VPT_H**. This configuration supports the following precisions $\{4, 8, \dots, 48\}$ within the $\textcircled{\text{D}_\text{D}}$ data-path (with an exponent bit-width maintained at 11). Post-synthesis power simulations are performed with the same synthesis conditions as explained in Section 8.1.

Table 2. Worst cases' and best cases' nomenclature per tolerance threshold and threshold generation policy—Jacobi.

Threshold Policy	10^{-4}		10^{-12}	
	Best Case	**Worst Case**	**Best Case**	**Worst Case**
Conservative thresholds—Gappa	BC-04	WC-04	BC-12	WC-12
Relaxed thresholds—SMI = 4	BR-04	WR-04	BR-12	WR-12

Baseline HW-Level Results for All Edge Cases

Table 3 shows the baseline average power, execution time, and energy consumption results for the standard double precision version of Jacobi applied to each of the selected cases and executed on top of the reference baseline RV64FD architecture. This data will constitute the baseline against which all the following studies will be compared.

Table 3. Power, execution time, and energy consumption of the standard double precision version of Jacobi executed on the RV64FD reference FPU.

Target Error Thresholds		10^{-4}				10^{-12}			
Input		**BC-04**	**WC-04**	**BR-04**	**WR-04**	**BC-12**	**WC-12**	**BR-12**	**WR-12**
VPT SW Configs	Type	Double	Double	Double	Double	Double	Double	Double	Double
Average Power (W)	Switching	1.42×10^{-3}	1.43×10^{-3}	1.43×10^{-3}	1.43×10^{-3}	1.42×10^{-3}	1.42×10^{-3}	1.43×10^{-3}	1.42×10^{-3}
	Internal	2.21×10^{-3}	2.22×10^{-3}	2.22×10^{-3}	2.22×10^{-3}	2.21×10^{-3}	2.21×10^{-3}	2.22×10^{-3}	2.20×10^{-3}
	Leakage	2.60×10^{-5}	2.60×10^{-5}	2.60×10^{-5}	2.60×10^{-5}	2.60×10^{-5}	2.60×10^{-5}	2.60×10^{-5}	2.60×10^{-5}
	Total	3.67×10^{-3}	3.68×10^{-3}	3.67×10^{-3}	3.68×10^{-3}	3.66×10^{-3}	3.66×10^{-3}	3.68×10^{-3}	3.65×10^{-3}
Execution Time (ps)		1.09×10^{10}	1.18×10^{10}	1.00×10^{10}	1.15×10^{10}	4.03×10^{10}	3.20×10^{10}	3.54×10^{10}	4.05×10^{10}
Energy (pJ)		4.03×10^{7}	4.35×10^{7}	3.70×10^{7}	4.26×10^{7}	1.47×10^{8}	1.17×10^{8}	1.30×10^{8}	1.48×10^{8}

Overhead of VPT-FPU When Operating in Full Precision

Table 4 depicts HW results when executing the application on the precise part of the VPT-FPU for each selected edge case. This evaluates the overhead brought by the static energy dissipated in the approximate part of the VPT-FPU circuitry when execution mode is fully precise.

As shown in the table, execution time is not affected. Only power increases due to static power (leakage) dissipated in the non-active approximate parts, leading to a 3.72% (input BC-12) up to 3.85% (input WR-12) energy increase, which is negligible compared to the savings exposed in the following sections.

Table 4. Power, execution time, and energy consumption **overheads** of the standard double precision version of Jacobi executed on the VPT_H configuration of the VPT-FPU.

Target Error Thresholds		**10^{-4}**				**10^{-12}**			
Input		**BC-04**	**WC-04**	**BR-04**	**WR-04**	**BC-12**	**WC-12**	**BR-12**	**WR-12**
VPT SW Configs	Type	Double	Double	Double	Double	Double	Double	Double	Double
Average Power (W)	Switching	1.39×10^{-3}	1.40×10^{-3}	1.40×10^{-3}	1.40×10^{-3}	1.39×10^{-3}	1.39×10^{-3}	1.40×10^{-3}	1.39×10^{-3}
	Internal	2.33×10^{-3}	2.33×10^{-3}	2.33×10^{-3}	2.33×10^{-3}	2.32×10^{-3}	2.32×10^{-3}	2.33×10^{-3}	2.31×10^{-3}
	Leakage	8.30×10^{-5}	8.30×10^{-5}	8.30×10^{-5}	8.30×10^{-5}	8.30×10^{-5}	8.30×10^{-5}	8.30×10^{-5}	8.30×10^{-5}
	Total	3.81×10^{-3}	3.82×10^{-3}	3.81×10^{-3}	3.82×10^{-3}	3.80×10^{-3}	3.80×10^{-3}	3.82×10^{-3}	3.79×10^{-3}
Total power overhead		**3.81%**	**3.77%**	**3.81%**	**3.77%**	**3.77%**	**3.79%**	**3.77%**	**3.81%**
Execution Time (ps)		1.09×10^{10}	1.18×10^{10}	1.00×10^{10}	1.15×10^{10}	4.03×10^{10}	3.20×10^{10}	3.54×10^{10}	4.05×10^{10}
Execution time overhead		**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
Energy (pJ)		4.19×10^{7}	4.51×10^{7}	3.84×10^{7}	4.42×10^{7}	1.53×10^{8}	1.22×10^{8}	1.35×10^{8}	1.53×10^{8}
Energy overhead		**3.81%**	**3.77%**	**3.83%**	**3.80%**	**3.72%**	**3.83%**	**3.83%**	**3.85%**

Gains Using the Conservative Thresholds

Table 5 depicts the HW-level gains for the conservative threshold policy (1). In addition, the table presents the results of the best case and worst case inputs at 10^{-4} and 10^{-12} target thresholds.

Table 5. Power, execution time, and energy consumption **gains** of VPT-enabled version of Jacobi (with theoretical conservative thresholds) executed on the VPT_H configuration of the VPT-FPU.

Target Error Thresholds		**10^{-4}**		**10^{-12}**	
Input		**Best Case (BC-04)**	**Worst Case (WC-04)**	**Best Case (BC-12)**	**Worst Case (WC-12)**
VPT SW Configs	Type	VPT double	VPT double	VPT double	VPT double
Average Power (W)	Switching	3.41×10^{-4}	3.51×10^{-4}	7.14×10^{-4}	7.23×10^{-4}
	Internal	7.89×10^{-4}	8.05×10^{-4}	1.31×10^{-3}	1.32×10^{-3}
	Leakage	8.10×10^{-5}	8.10×10^{-5}	8.20×10^{-5}	8.20×10^{-5}
	Total	1.21×10^{-3}	1.23×10^{-3}	2.10×10^{-3}	2.13×10^{-3}
Total power savings		**67.03%**	**66.43%**	**42.48%**	**41.90%**
Execution Time (ps)		5.31×10^{9}	5.82×10^{9}	2.58×10^{10}	2.07×10^{10}
Execution time savings		**51.64%**	**50.61%**	**35.88%**	**35.15%**
Energy (pJ)		6.43×10^{6}	7.20×10^{6}	5.44×10^{7}	4.42×10^{7}
Energy savings		**84.06%**	**83.43%**	**63.13%**	**62.33%**

As depicted in the table, there is not much difference between best-case and worst-case scenarios in terms of computational energy, i.e., 84.06% (best case for 10^{-4}) vs. 83.43%

(worst case for 10^{-4}) then 63.13% (best case for 10^{-12}) vs. 62.33% (worst case for 10^{-12}). This can be explained by the fact that using VPT threshold policy (1) does not affect the number of iterations executed.

Upon a closer look to Figures 13 and 14, we conclude that this is due to the negligible effect on the number of iterations when using conservative threshold policies.

Gains Using the Relaxed Thresholds

Results for this experiment are depicted in Table 6. For a target threshold of 10^{-4}, this threshold policy achieves energy savings between 82.95%(worst case) up to 88.20% (best case), mainly due to 42.20% up to 59.80% savings in terms of execution time. Similarly, for a target of 10^{-12}, the energy gain stands between 63.20% (worst case) and 68.97% (best case). These percentages are slightly more interesting than in threshold policy (1), especially in the best case.

Table 6. Power, execution time, and energy consumption **gains** of VPT-enabled version of Jacobi (with relaxed thresholds and SMI = 4) executed on the VPT_H configuration of the VPT-FPU.

Target Error Thresholds		**10^{-4}**		**10^{-12}**	
Input		**Best Case (BR-04)**	**Worst Case (WR-04)**	**Best Case (BR-12)**	**Worst Case (WR-12)**
VPT SW Configs	Type	VPT double	VPT double	VPT double	VPT double
Average Power (W)	Switching	2.90×10^{-4}	2.94×10^{-4}	6.42×10^{-4}	6.48×10^{-4}
	Internal	7.08×10^{-4}	7.13×10^{-4}	1.20×10^{-3}	1.21×10^{-3}
	Leakage	8.10×10^{-5}	8.10×10^{-5}	8.20×10^{-5}	8.20×10^{-5}
	Total	1.07×10^{-3}	1.08×10^{-3}	1.93×10^{-3}	1.94×10^{-3}
Total power savings		**70.67%**	**70.49%**	**47.58%**	**46.74%**
Execution Time (ps)		4.04×10^{9}	6.68×10^{9}	2.09×10^{10}	2.80×10^{10}
Execution time savings		**59.80%**	**42.20%**	**40.82%**	**30.89%**
Energy (pJ)		4.36×10^{6}	7.27×10^{6}	4.05×10^{7}	5.45×10^{7}
Energy savings		**88.20%**	**82.95%**	**68.97%**	**63.20%**

Please remember that, in this situation, the VPT relaxed threshold policy (2) has a more substantial impact on the number of iterations, as shown in Figures 13 and 14. For example, input BR-12 sees a decrease in the number of iterations from 612 to 578 (−5.5% variation), translating to a 68.97% energy gain. On the other hand, WR-12 sees an increase from 700 to 763 iterations (+9.0% increase) which translates to 63.20% total energy gain, which is still an important achievement despite the extra iterations' overhead.

8.4. Circuit Area Results

Table 1 reports the total cell area overhead for each HW configuration w.r.t the baseline standard RV64FD FPU.

The area overhead depends on the additional supported reduced precision formats. Overheads can range from 1.19× (VPT_A) up to 2.63× (VPT_G).

When VTO is also applied along with VPT, (i.e., when the *VPT float* SW configuration is considered), this technique can attenuate the area overhead. For example, overhead can be reduced from 1.27× (VPT_B) down to 1.19× (VPT_A) for a 10^{-4} threshold.

These ratios also depend on the number of intermediate pipeline registers inserted in each approximate datapath, configurable at design time. The area overhead is the only significant disadvantage of the proposed architecture.

Conclusions

These results demonstrate that the VPT-enabled software implementation is a one-size-fits-all solution, i.e., using the same software implementation along with our proposed VPT-FPU, the designer can drastically reduce consumed resources by using only the needed precision instead of an over-designed solution such as standard FPUs. The price to pay is in terms of circuit area overhead. Designers can also use standard VTO techniques jointly to enhance the energy savings of our technique and reduce its area overhead.

9. Conclusions

This work proposes a new method and an FPU architecture that enable designers to dynamically tune FP computations' precision automatically at run-time called Variable Precision in Time (VPT). In spite of its circuit area overhead, the proposed approach simplifies the integration of variable precision in existing software workloads at any level of the software stack (OS, RTOS, or application-level): it only requires lightweight software support and solely relies on traditional assembly instructions, without the need for a specialized compiler or custom instructions.

The technique was applied to the Jacobi and the Gauss–Seidel iterative methods taking full advantage of the suggested FPU. For each algorithm two threshold policies were proposed: a conservative policy (1) and a relaxed policy (2).

The last two sections presented a statistical study that explored the effects of each VPT threshold policy on many aspects of the application: impact on the total number of iterations, impact on the iterations' distribution across different precisions, impact on HW-level estimations such as execution time, power and the overall energy consumption.

The implementations demonstrate up to 70.67% power consumption saving, up to 59.80% execution time saving, and up to 88.20% total energy saving w.r.t the reference double precision implementation, and with no accuracy loss.

To conclude, the threshold policy (1) is a conservative approach that brings some predictability and safety along with all the run-time variable precision benefits. On the other hand, threshold policy (2) is a relaxed approach that further optimizes power and energy consumption. However, it tends to alter the total number of iterations, sometimes favorably and sometimes negatively. However, even when the total number of iterations is increased, there is still a very interesting energy gain. Thus, threshold policy (1) is the safest solution, and threshold policy (2) represents a risky but optimized solution.

9.1. Limitations

To choose between the two threshold policies for a given problem, the user/designer should use representative datasets and especially evaluate the memory-related aspects, especially for threshold policy (2). Furthermore, the memory-related aspects should be studied further to evaluate whether the cost of the potential added iterations (hence more load and store operations) is lower than the gain associated with computation optimization.

The proposed methodology is application-dependent, i.e., the designer should transform the algorithm manually, ensure convergence after modification, and select adequate intermediate tolerance thresholds depending on the convergence metric used in the application.

9.2. Future Works

Our future studies will focus on optimizing the circuit area by merging some components of the VPT-FPU. We also intend to explore the usability of the proposed VPT-FPU with other kinds of workloads such as Machine Learning and Computer Vision applications.

Author Contributions: N.A.S.: Conceptualization, Methodology, Formal analysis, Software, Validation, Investigation, Data curation, Visualization, Writing—Original Draft, Writing—Review & Editing. M.B.: Conceptualization, Writing—Review & Editing, Supervision, Project administration, Funding acquisition. K.M.-A.: Conceptualization, Formal analysis, Investigation, Writing—Original Draft, Writing—Review & Editing, Supervision, Project administration, Funding acquisition. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

FP	Floating-Point
FPU	Floating-Point Unit
TC	Transprecision Computing
VTO	Variable Type Optimization
ARP	Arbitrary Reduced Precision
VPT	Variable Precision in Time
HAL	Hardware Abstraction Layer

Appendix A

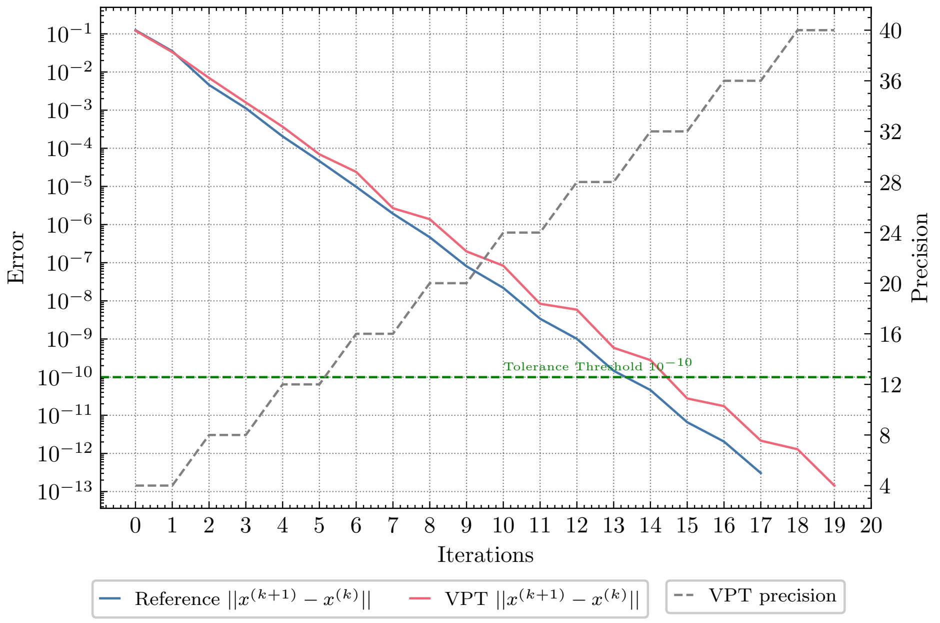

Figure A1. The convergence profile of the VPT-enabled Gauss–Seidel.

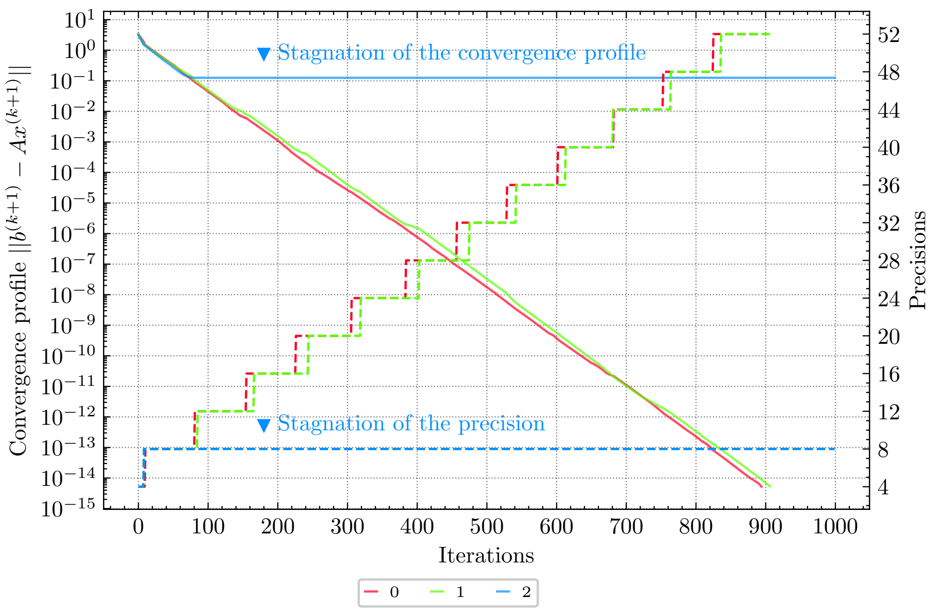

Figure A2. Convergence profiles (left axis, continuous lines) and their corresponding precision variation profiles (right axis, dotted lines) for three inputs (Matrix IDs 0, 1, and 2). The convergence profile here is in terms of the residual error metric ($||\vec{b}^{(k+1)} - \mathbf{A}\vec{x}^{(k+1)}||$).

Listing A1. Gappa script for the generation of conservative thresholds. It is applied on $n = 50, M_p = 40$. Here we use RNE (round-to-nearest tie to even) rounding for the round() function.

```
# Create a FP format with E = 11 (same as binary64) and M_p = 40
# Choose Round to Nearest, tie to even (ne)
@rnd = float<40, -1022, ne>;

# Declare a variable u, which is cast from a real value U
u = rnd(U);

# Compute expression with 40 bits of precision
square  rnd= u * u;

# Unrolled summation of squares, loops don't exist in Gappa
nsquare_1 rnd= square;
nsquare_2 rnd= nsquare_1 + square;
nsquare_3 rnd= nsquare_2 + square;
...
...
nsquare_49 rnd= nsquare_48 + square;
nsquare_50 rnd= nsquare_49 + square;

sqroot  rnd= sqrt( nsquare );
norm2      = rnd(sqroot);

{
# Hypotheses ---------------------------------------
# Given that n = 50, and U is in [0, 2^-40]
    n = 50  /\  U in [0 , 1b-40 ]
->
# The property to be proven ---------------------
    rnd(norm2) in ?
}

# Output of Gappa -----------------------------------------
Results:
float<40,-1022,ne>(norm2) in [0, 121480019999b-74 {6.431098710766519833e-12,
                                                   2^(-3.717810058593750000e+01)}]
# The lower bound of the variable <norm2> is 0
# The upper bound of the variable <norm2> is 6.43110e-12
```

References

1. Malossi, A.C.I.; Schaffner, M.; Molnos, A.; Gammaitoni, L.; Tagliavini, G.; Emerson, A.; Tomás, A.; Nikolopoulos, D.S.; Flamand, E.; Wehn, N. The transprecision computing paradigm: Concept, design, and applications. In Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 19–23 March 2018; pp. 1105–1110. [CrossRef]
2. Matoussi, O.; Durand, Y.; Sentieys, O.; Molnos, A. Error Analysis of the Square Root Operation for the Purpose of Precision Tuning: A Case Study on K-means. In Proceedings of the ASAP 2019—30th IEEE International Conference on Application-Specific Systems, Architectures and Processors, New York, NY, USA, 15–17 July 2019; IEEE: New York, NY, USA, 2019.
3. Panchekha, P.; Sanchez-Stern, A.; Wilcox, J.R.; Tatlock, Z. Automatically Improving Accuracy for Floating Point Expressions. *SIGPLAN Not.* **2015**, *50*, 1–11. [CrossRef]
4. Sanchez-Stern, A.; Panchekha, P.; Lerner, S.; Tatlock, Z. Finding Root Causes of Floating Point Error with Herbgrind. *arXiv* **2018**, arXiv:1705.10416.
5. Benz, F.; Hildebrandt, A.; Hack, S. A Dynamic Program Analysis to find Floating-Point Accuracy Problems. *ACM SIGPLAN Not.* **2012**, *47*, 453–462. [CrossRef]
6. Graillat, S.; Jézéquel, F.; Picot, R.; Févotte, F.; Lathuilière, B. Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic. *J. Comput. Sci.* **2019**, 36, 101017. [CrossRef]
7. Rubio-González, C.; Hough, D.; Nguyen, C.; Mehne, B.; Sen, K.; Demmel, J.; Kahan, W.; Iancu, C.; Lavrijsen, W.; Bailey, D.H. Floating-point precision tuning using blame analysis. In Proceedings of the 38th International Conference on Software Engineering—ICSE'16, Austin, TX, USA, 14–22 May 2016; ACM Press: Austin, TX, USA, 2016; pp. 1074–1085. [CrossRef]
8. Rubio-González, C.; Nguyen, C.; Nguyen, H.D.; Demmel, J.; Kahan, W.; Sen, K.; Bailey, D.H.; Iancu, C.; Hough, D. Precimonious: Tuning Assistant for Floating-point Precision. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, CO, USA, 17–22 November 2013.
9. Ho, N.; Manogaran, E.; Wong, W.; Anoosheh, A. Efficient floating point precision tuning for approximate computing. In Proceedings of the 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), Chiba, Japan, 16–19 January 2017; pp. 63–68. [CrossRef]
10. Ait Said, N.; Benabdenbi, M.; Morin-Allory, K. Arbitrary Reduced Precision for Fine-grained Accuracy and Energy Trade-offs. *Microelectron. Reliab.* **2021**, *120*, 114099. [CrossRef]
11. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008. 2008. pp. 1–70. Available online: https://ieeexplore.ieee.org/document/4610935 (accessed on 26 August 2021). [CrossRef]
12. Köster, U.; Webb, T.J.; Wang, X.; Nassar, M.; Bansal, A.K.; Constable, W.H.; Elibol, O.H.; Gray, S.; Hall, S.; Hornof, L.; et al. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17), Long Beach, CA, USA, 4–9 December 2017; Curran Associates Inc.: Red Hook, NY, USA, 2017; pp. 1740–1750.
13. Chung, E.; Fowers, J.; Ovtcharov, K.; Papamichael, M.; Caulfield, A.; Massengill, T.; Liu, M.; Lo, D.; Alkalay, S.; Haselman, M.; et al. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* **2018**, *38*, 8–20. [CrossRef]
14. Jouppi, N.P.; Young, C.; Patil, N.; Patterson, D.; Agrawal, G.; Bajwa, R.; Bates, S.; Bhatia, S.; Boden, N.; Borchers, A.; et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* **2017**, *45*, 1–12. [CrossRef]
15. Xie, S.; Davidson, S.; Magaki, I.; Khazraee, M.; Vega, L.; Zhang, L.; Taylor, M.B. Extreme Datacenter Specialization for Planet-Scale Computing: ASIC Clouds. *SIGOPS Oper. Syst. Rev.* **2018**, *52*, 96–108. [CrossRef]
16. Fousse, L.; Hanrot, G.; Lefèvre, V.; Pélissier, P.; Zimmermann, P. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.* **2007**, *33*, 13-es. [CrossRef]
17. Flegar, G.; Scheidegger, F.; Novaković, V.; Mariani, G.; Tomás, A.E.; Malossi, A.C.I.; Quintana-Ortí, E.S. FloatX: A C++ Library for Customized Floating-Point Arithmetic. *ACM Trans. Math. Softw.* **2019**, *45*, 40:1–40:23. [CrossRef]
18. Tagliavini, G.; Marongiu, A.; Benini, L. FlexFloat: A Software Library for Transprecision Computing. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2018**, *39*, 145–156. [CrossRef]
19. Ait Said, N.; Benabdenbi, M.; Morin-Allory, K. FPU Bit-Width Optimization for Approximate Computing: A Non-Intrusive Approach. In Proceedings of the 2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS), Marrakech, Morocco, 1–3 April 2020; pp. 1–6.
20. Chatelain, Y.; Petit, E.; de Oliveira Castro, P.; Lartigue, G.; Defour, D. *Automatic Exploration of Reduced Floating-Point Representations in Iterative Methods*; Euro-Par 2019: Parallel Processing; Yahyapour, R., Ed.; Springer International Publishing: Cham, Switzerland, 2019; pp. 481–494.
21. Ait Said, N.; Benabdenbi, M.; Morin-Allory, K. FPU Reduced Variable Precision in Time: Application to the Jacobi Iterative Method. In Proceedings of the 2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI) (ISVLSI-2021), Tampa, FL, USA, 7–9 July 2021.
22. Ait Said, N. FPU Reduced Variable Precision in Time: Application to the Jacobi Iterative Method—ISVLSI'21. Available online: https://www.youtube.com/watch?v=w8Hem9r0daU (accessed on 15 July 2021).
23. Zeller, A.; Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.* **2002**, *28*, 183–200. [CrossRef]
24. Carson, E.; Higham, N.J. A New Analysis of Iterative Refinement and Its Application to Accurate Solution of Ill-Conditioned Sparse Linear Systems. *SIAM J. Sci. Comput.* **2017**, *39*, A2834–A2856. [CrossRef]

25. Carson, E.; Higham, N.J. Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions. *SIAM J. Sci. Comput.* **2018**, *40*, A817–A847. [CrossRef]
26. Moler, C.B. Iterative Refinement in Floating Point. *J. ACM* **1967**, *14*, 316–321. [CrossRef]
27. Amestoy, P.; Buttari, A.; Higham, N.; L'Excellent, J.Y.; Mary, T.; Vieuble, B. Five-Precision GMRES-based iterative refinement. working paper or preprint.
28. Blanchard, P.; Higham, N.J.; Lopez, F.; Mary, T.; Pranesh, S. Mixed Precision Block Fused Multiply-Add: Error Analysis and Application to GPU Tensor Cores. *SIAM J. Sci. Comput.* **2020**, *42*, C124–C141. [CrossRef]
29. Anzt, H.; Dongarra, J.; Flegar, G.; Higham, N.J.; Quintana-Ortí, E.S. Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers: Adaptive precision in block-Jacobi preconditioning for iterative solvers. *Concurr. Comput. Pract. Exp.* **2019**, *31*, e4460. [CrossRef]
30. Saad, Y. *Iterative Methods for Sparse Linear Systems*; Other Titles in Applied Mathematics; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2003. [CrossRef]
31. Barrett, R.; Berry, M.; Chan, T.F.; Demmel, J.; Donato, J.; Dongarra, J.; Eijkhout, V.; Pozo, R.; Romine, C.; der Vorst, H.V. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. *Math. Comput.* **1995**, *64*, 1349. [CrossRef]
32. Ait Said, N. AxQEMU: A Floating-Point Approximation-Aware Emulator. 2021. Available online: https://github.com/noureddine-as/axqemu (accessed on 18 August 2021).
33. Denis, C.; de Oliveira Castro, P.; Petit, E. Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic. In Proceedings of the 23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, 10–13 July 2016; pp. 55–62. [CrossRef]
34. Higham, N.J. *Accuracy and Stability of Numerical Algorithms*, 2nd ed.; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2002.
35. Waterman, A.; Asanović, K. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121*. 2019. Available online: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf (accessed on 18 August 2021).
36. The Open-source Repository of CORE-V CVA6 CPU: An Application Class 6-Stage RISC-V CPU Capable of Booting Linux. 2021. Available online: https://github.com/openhwgroup/cva6 (accessed on 13 June 2021).
37. Li, L.; Gautschi, M.; Benini, L. Approximate DIV and SQRT instructions for the RISC-V ISA: An efficiency vs. accuracy analysis. In Proceedings of the 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), Thessaloniki, Greece, 25–27 September 2017; pp. 1–8. [CrossRef]
38. Jun, K.; Swartzlander, E.E. Modified non-restoring division algorithm with improved delay profile and error correction. In Proceedings of the 2012 Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR), Pacific Grove, CA, USA, 4–7 November 2012; pp. 1460–1464. [CrossRef]
39. de Dinechin, F.; Lauter, C.; Melquiond, G. Certifying the Floating-Point Implementation of an Elementary Function Using Gappa. *IEEE Trans. Comput.* **2011**, *60*, 242–253. [CrossRef]
40. The Open-Source Repository of the Jacobi C Implementation. 2020. Available online: https://github.com/UoB-HPC/intro-hpc-jacobi (accessed on 13 June 2021).
41. RISC-V Proxy Kernel. 2019. Available online: https://github.com/riscv/riscv-pk (accessed on 3 June 2019).
42. Mach, S.; Rossi, D.; Tagliavini, G.; Marongiu, A.; Benini, L. A Transprecision Floating-Point Architecture for Energy-Efficient Embedded Computing. In Proceedings of the 2018 IEEE International Symposium on Circuits and Systems (ISCAS), Florence, Italy, 27–30 May 2018; pp. 1–5.