electronics

MDPI

Article

LUAEMA: A Loop Unrolling Approach Extending Memory Accessing for Vector Very-Long-Instruction-Word Digital Signal Processor with Multiple Register Files

Yonghua Hu [1,2], Aobo Cheng [1,2], Zhuoyou Tang [1,2], Peng Liu [1,2] and Wei Liang [1,2,*]

[1] School of Computer Science and Engineering, Hunan University of Science and Technology, Xiangtan 411201, China; yonghuahu@hnust.edu.cn (Y.H.); aobocheng99@163.com (A.C.); 17394946573@163.com (Z.T.); pengliu1112@163.com (P.L.)
[2] Hunan Key Laboratory for Service Computing and Novel Software Technology, Hunan University of Science and Technology, Xiangtan 411201, China
* Correspondence: wliang@hnust.edu.cn

Abstract: Loop unrolling can provide more instruction-level parallelism opportunities for code and enables a greater range of instruction pipeline scheduling. In high-performance very-long-instruction-word (VLIW) digital signal processors (DSPs), there are special registers to address. To further improve the instruction-level parallelism of code for such DSPs by making full use of these registers, in this paper, we propose a more effective loop unrolling approach through extending memory accessing (LUAEMA). In this approach, the final unrolling factor is computed by a model in which every register kind and every memory accessing operation are considered. For basic digital signal processing algorithms, the unrolling factor under the LUAEMA is larger than that under the conventional loop unrolling approach. We also provide the opportunity to reduce the number of instructions in a loop during the code transformation of loop unrolling. The experimental results show that the loop unrolling approach proposed in this paper can achieve an average speedup ratio ranging from 1.14 to 1.81 compared with the conventional loop unrolling approach. For some algorithms, the peak speedup ratio is up to 2.11.

Keywords: loop unrolling; loop optimization; VLIW; DSP; register resources

Citation: Hu, Y.; Cheng, A.; Tang, Z.; Liu, P.; Liang, W. LUAEMA: A Loop Unrolling Approach Extending Memory Accessing for Vector Very-Long-Instruction-Word Digital Signal Processor with Multiple Register Files. *Electronics* **2024**, *13*, 1425. https://doi.org/10.3390/electronics13081425

Academic Editor: Shinichi Yamagiwa

Received: 15 March 2024
Revised: 3 April 2024
Accepted: 7 April 2024
Published: 10 April 2024

1. Introduction

Nowadays, digital signal processors (DSPs) are widely used in many fields such as digital communications, video monitoring, machine control, unmanned aerial vehicles, instrumentation, and biomedicine [1–5]. Compared with general-purpose microprocessors, DSPs adopt the Harvard structure that separates the program and data, and their instruction scheduling adopts pipeline operation, which can quickly complete a variety of digital signal processing algorithms [6]. A typical DSP system is shown in Figure 1. The growing demand in the DSP market has seen the emergence of many high-performance DSPs in recent years such as AnySP [7], Tensilica ConnX BBE64EP [8], TI TMS320C6000 Series DSP [9], YHFT-DSP [10], SW processor [11], and ARM Cortex-M4 family [12]. As the study of computing parallelism becomes more and more deepened, the VLIW architecture together with the single-instruction-multiple-data (SIMD) vector unit has gradually become a typical representative of high-performance DSP architectures [10,13–16]. The advantage of the vector unit over the scalar unit is that it can compute multiple data with one instruction, resulting in a significant reduction in the time required to execute corresponding programs [17]. Vector DSPs are rich in hardware resources, while their architecture and instruction set differ from one DSP to another [18]. To take full advantage of the hardware resources of each vector DSP, the compilation optimization according to the architecture characteristics of the vector DSP is very necessary [19].

Figure 1. A typical DSP system.

Compilation optimization for vector DSPs takes into account the hardware resources of the DSP, especially the register resources [20]. Vector DSPs often have a lot of general-purpose registers in both scalar and vector data processing units, in order to adapt to the cases in which the computing of an algorithm can be vectorized or not. In particular, the configuration with both base address registers and offset registers provides more freedom in the optimization of loops. For example, YHFT's vector DSP FT-7002 provides scalar data registers and vector data registers [21]. Texas Instruments' TMS320C6000 Series DSPs provide rich register resources such as accumulation register ACC, data register Dx, status register SR, and control register CR [22]. An important part of compilation optimization is how to make the best use of multiple register resources [23,24].

Algorithm optimization has been one of the hot topics in current research [25–29]. Scholars' research on algorithm optimization can be divided into process optimization for the specific algorithms and general optimization for all algorithms [30]. Because the compiler does not know the purpose of the program being compiled, the compilation optimization is mainly the generic optimization for all algorithms. Loop structures play a large part in the running time in many algorithms (e.g., digital signal processing algorithms, deep learning algorithms, neural network algorithms) [31–36], so loop optimization is an essential part of compilation optimization. The main approach of loop optimization is loop transformation. Loop transformation means that the compiler performs multiple loop reconstruction transformations to help improve the parallelism of loops in a program. Loop transformation mainly includes loop tiling [37], loop skewing, loop unrolling [38], and software pipelining. Loop unrolling is one of the most direct and effective approaches in loop transformation [39], which improves the overall performance of the loop by increasing the efficiency of the parallel processing [38]. Loop unrolling copies multiple loop bodies into one iteration and adjusts the array index and loop increment accordingly to ensure that the program runs correctly, which has the significant effect of reducing the loop overhead [40]. Moreover, loop unrolling can increase the opportunities for other optimizations, such as common subexpression elimination, induction variable optimization, software pipelining, etc. [41,42]. And it is also an important part of optimizations such as vectorization and data prefetching [43]. Overall, the benefits of loop unrolling include instruction-level parallelism, register locality, and hierarchical storage locality [38].

The application of loop unrolling for vector architecture instruction sequences in the compilation optimization has been studied, but these studies are not flexible enough to deal with the allocation of register resources for vector DSPs, and there are cases where register resources for addressing are not fully used. In this paper, we use the conventional loop unrolling approach as a basis for extending the unrolling factor and increasing the scheduling range of instruction-level parallelism by making further use of the free register resources before register spilling happens. Based on the above ideas, this paper proposes a loop unrolling approach extending memory accessing (LUAEMA) that fully considers the free resources of registers and thus increases the unrolling factor.

The rest of this paper is as follows: Section 2 provides related work on loop unrolling. Section 3 illustrates the limitations of the conventional loop unrolling approach for vector DSPs and the model of LUAEMA. Section 4 describes the algorithms in the LUAEMA, including the preparation algorithm for the unrolling factor, the unrolling factor algorithm, and the description of the code transformation for the original loop. Section 5 analyzes the experimental results between the LUAEMA and the conventional loop unrolling approach. Finally, a few conclusions and directions for future studies are provided in Section 6.

2. Related Work

2.1. The Loop Unrolling Approach

The key issue of loop unrolling is the determination of the unrolling factor. It is not the case that the larger the unrolling factor, the greater the gain. Too large an unrolling factor will result in a surge in the number of instructions in the loop. If the instruction buffer of the target platform is not large enough, it will cause buffer overflow. Too many instructions in the loop can also cause an increase in the number of registers allocated to the loop after the loop unrolling, which can lead to register spilling [37,44,45]. On the other hand, if the unrolling factor is too small, the program will not be able to make full use of the hardware resources. So, how to determine the suitable unrolling factor is one of the main directions of loop unrolling research. Research on the unrolling factor can be broadly divided into two categories at present. One is to use machine learning algorithms to predict the unrolling factor, and the other is to build the cost model to calculate the unrolling factor automatically.

The main reason for using machine learning algorithms to predict the unrolling factor is that they automatically extract features and make predictions by being trained on a large dataset. Monsifrot et al. used a mapping mechanism based on BDT learning techniques to determine which loops should be unrolled [46]. Wang Dong et al. processed the conventional random forest model with weighted and unbalanced datasets and then used the improved model to predict the unrolling factor [47].

There are many studies on building cost models to automatically compute the unrolling factor. The common approaches are Simple Counting, Path Cover, and Data Dependency. Simple Counting calculates the unrolling factor by dividing the loop iteration times by the number of statements in the loop. Path Cover calculates the unrolling factor by estimating the number of paths to be executed in the algorithm. Data Dependency determines the unrolling factor by analyzing the relationship between variable definition and variable use within the loop. There are many examples of in-depth studies based on the above approaches. For example, UTBPC (Unrolling-Times-Based Program Characteristics) can better calculate the unrolling factor and apply it to the software pipeline and then consider the effect of loop unrolling on data prefetching to optimize the original data prefetching process [48]. Liu X et al. propose an approach that combines loop unrolling optimization with register pressure for estimating the unrolling factor based on the GCC-5.3.0 compiler [49]. VALU (Vectorization-aware loop unrolling) forwards the vectorizable code to SLP, allowing it to bypass its greedy search for vectorizable seed instructions, exposing more vectorization opportunities [40].

2.2. Relationship to Prior Work

In this paper, the unrolling factor is calculated by building a cost model. The approach proposed in this paper uses a paper that estimates the unrolling factor after combining loop unrolling with register pressure as the base theory [49] and the implementation of its unrolling factor algorithm on the vector DSP as the conventional loop unrolling approach. While considering the register pressure, this approach makes full use of the two characteristics of the vector DSP's multiple register files and multiple addressing methods so that the unrolling factor increases compared to the conventional loop unrolling approach, and reduces the number of instructions in the loop.

3. Framework and Model

3.1. Limitations of the Conventional Loop Unrolling Approach

Differences in the form of the loops can lead to different identified induction variables, even if the loops have the same arithmetic logic. Take Listing 1 and Listing 2, for example. Since both the conventional loop unrolling approach and the loop unrolling approach proposed in this paper are based on the consideration of the register resources in the hardware, the high-level language cannot fully explain the utilization of the register resources. Therefore, we take the instruction-level intermediate code as input. To illustrate

the limitations of the conventional loop unrolling approach in register usage, the loops of Listing 1 and Listing 2 use instruction sequences internally, while the loop control section retains the high-level language format.

There are two types of variables that are special for addressing in a loop, address variables and offset variables. These two types of variables refer to address registers (R_x) and offset registers (R_y) in instruction-level intermediate code. The conventional loop unrolling approach determines whether the address registers or the offset registers need to be unrolled in the intermediate code based on the identification of induction variables.

The arithmetic logic of the two loops shown in Listing 1 and Listing 2 is to take out each element of *ARa* in turn, add 1, and put them into *ARb*. The induction variables identified in the conventional loop unrolling approach are different depending on the different forms of a loop. The induction variables of the loop in Listing 1 are *Ri* and *Ork*. And the induction variables of the loop in Listing 2 can be identified as *Ri*, *Ara*, and *ARb*.

Listing 1. The loop whose inductive variables are *Ri* and O*Rk*.

```
Mov 0, ORk
do Ri = 1, Rn
   Load ARb[ORk], Rb
   Add 1, Rb, Ra
   Store Ra, ARa[ORk]
   Add 1, ORk, ORk
enddo
```

Listing 2. The loop whose inductive variables are *Ri*, *ARa*, and *ARb*.

```
Mov 0, ORk
do Ri = 1, Rn
   Load ARb[ORk], Rb
   Add 1, Rb, Ra
   Store Ra, ARa[ORk]
   Add 1, ARa, ARa
   Add 1, ARb, ARb
enddo
```

Before unrolling the loop, it is necessary to identify whether the instruction operands within the loop need to be created unrolling copies. The variables that require creating unrolling copies identified by the conventional loop unrolling approach are made up of inductive variables and loop variations. In Listing 1, they are *Ri*, *ORk*, *Ra*, and *Rb*, while in Listing 2, these variables are *Ri*, *ARa*, *ARb*, *Ra*, and *Rb*. Taking *Ra* as an example, its copies are written as *Ra_1*, *Ra_2*, and so on.

For the purpose of showing the limits of the conventional loop unrolling approach when using address registers and offset registers, the following loop unrolling example assumes that there are no restrictions on other types of register resources. This means that the need for using other types of register resources during loop unrolling will be fully satisfied and it is only guaranteed to consider the full use of the free resources in address registers or offset registers.

Let the total number of address registers be e and the total number of offset registers be f on the DSP hardware. Before the loop unrolling process, both Listing 1 and Listing 2 have used two address registers (*ARa* and *ARb*) and one offset register (*ORk*). Because Listing 1's set of variables that need to be created unrolling copies contains *ORk*, the unrolling factor (*UF*) of the loop shown in Listing 1 is set to f in order to fully use the offset registers. Similarly, the unrolling factor of the loop shown in Listing 2 is set to $e/2$. Listing 1 and Listing 2 are translated into Listing 3 and Listing 4, respectively, after being processed by the conventional loop unrolling approach.

Listing 3. Listing 1 after being processed by the conventional loop unrolling approach.

```
Mov f, UF
Mov 0, ORk
Add 1, ORk, ORk_1
. . .. . .
do Ri = 1, Rn/UF
   Load ARb[ORk], Rb
   Load ARb[ORk_1], Rb_1
   . . .. . .
   Add 1, Rb, Ra
   Add 1, Rb_1, Ra_1
   . . .. . .
   Store Ra, ARa[ORk]
   Store Ra_1, ARa[ORk_1]
   . . .. . .
   Add UF, ORk, ORk
   Add UF, ORk_1, ORk_1
   . . .. . .
enddo
```

Listing 4. Listing 2 after being processed by the conventional loop unrolling approach.

```
Mov e/2, UF
Mov 0, ORk
Add 1, ARa, ARa_1
. . .. . .
Add 1, ARb, ARb_1
. . .. . .
do Ri = 1, Rn/UF
   Load ARb[ORk], Rb
   Load ARb_1[ORk], Rb_1
   . . .. . .
   Add 1, Rb, Ra
   Add 1, Rb_1, Ra_1
   . . .. . .
   Store Ra, ARa[ORk]
   Store Ra_1, ARa_1[ORk]
   . . .. . .
   Add UF, ARa, ARa
   Add UF, ARa_1, ARa_1
   . . .. . .
   Add UF, ARb, ARb
   Add UF, ARb_1, ARb_1
   . . .. . .
enddo
```

When the *UF* of Listing 3 is equal to the *UF* of Listing 4, the number of self-incrementing instructions in Listing 4 is twice that in Listing 3. This means that although there are different ways to write the same loop logic, the conventional loop unrolling approach does not have the ability to change the form of the loop, which may lead to redundant code lines within the loop. When the loop is designed in the form of Listing 2, the conventional loop unrolling approach can only process it into the form of Listing 4, ignoring the possibility of further analyzing the types of the different variables to obtain a more concise loop form, as shown in Listing 3.

The hardware resources provided by the processor should be fully considered during loop unrolling in the vector DSP. The induction variables of the loop shown in Listing 3 include *ORk*, and the address registers *ARa* and *ARb* are used as loop invariants, so the free resources of R_x are not fully used in loop unrolling. This will result in the resources in

R_x appearing to be free when the loop is running. Similarly, the loop shown in Listing 4 suffers from the problem of under-use of free resources in R_y.

In order to solve the two problems of neglecting the number of instructions in the loop which can be reduced and not fully using the free resources of a certain register type in the conventional loop unrolling approach, this paper proposes an improved instruction-level loop unrolling approach that unrolls both address registers and offset registers at the same time to further improve loop optimization and increase the scheduling range of instruction-level parallelism. This approach makes full use of the two kinds of register resources, R_x and R_y, to further increase the unrolling factor when the general registers are sufficient. And it can also be used flexibly to reduce the number of instructions in the loop according to the unrolling situation of R_x and R_y.

3.2. The Mathematical Model for LUAEMA

ORk belonging to R_y in the loop shown in Listing 3 is the loop variation (the variable that is needed to create the unrolling copies in loop unrolling) in the loop unrolling process. *ARa* and *ARb* belonging to R_x are the loop invariants. This is the opposite case for the loop shown in Listing 4, where *ARa* and *ARb* are the loop variations in the loop unrolling process and *ORk* is the loop invariant. The two loop forms with the same arithmetic logic can only make use of the free resources of one register in R_x or R_y after loop unrolling, which will lead to the situation that the resources of the other register cannot be fully used. Therefore, this section considers whether it is possible to have a loop unrolling approach that can use the free resources of both types of registers, R_x and R_y, at the same time and proposes a loop unrolling processing model extending memory accessing that unrolls both R_x and R_y. The basic principle of the model is as follows.

3.2.1. Obtain the Set of Variables A_{ur}

Loop unrolling needs to obtain the set of variables A_{ur} that need to create unrolling copies in the loop before determining the unrolling factor. A_{ur} consists mainly of induction variables and loop variations in the conventional loop unrolling approach. The induction variables for the loop shown in Listing 1 are *Ri* and *ORk*, and the induction variables for the loop shown in Listing 2 are *Ri*, *ARa*, and *ARb*. The induction variables for both loops include *Ri*, since *Ri* is the loop control induction variable, and its main role is to guarantee the loop times. The main purpose of the induction variables other than *Ri* is to ensure that it is correct when accessing memory by address, except that the loop shown in Listing 1 uses R_y to access the memory correctly, and the loop shown in Listing 2 uses R_x. This would ignore the fact that both R_x and R_y can be unrolled. In order to satisfy that R_x and R_y are used in loop unrolling, this model adds the loop invariants belonging to R_x or R_y in the original loop to A_{ur}.

3.2.2. Calculate the Unrolling Factor

The calculation of the unrolling factor can be conducted in three steps: the classification of A_{ur}; calculation of UF_x, UF_y, UF_{xy}, and UF_R; and acquirement of the final unrolling factor *UF*. This is performed as follows:

(1) Analyze A_{ur} and classify all the variables in A_{ur} into two—A_{urXY} and $A_{urOther}$—according to whether they belong to R_x or R_y:

$$A_{ur} = A_{urXY} \bigcup A_{urOther}, \tag{1}$$

A_{urXY} contains variables related to the R_x and R_y of A_{ur}:

$$A_{urXY} = \{A_{urX},\ A_{urY}\}\ , \tag{2}$$

$A_{urOther}$ contains other variables in A_{ur} that are not related to R_x and R_y, with scalar general registers (*SRs*), vector general registers (*VRs*), and so on:

$$A_{urOther} = \{A_{urSR},\ A_{urVR}, \ldots\}. \tag{3}$$

A_{urX} and A_{urY} in Equation (2) represent the sets of variables that need to create unrolling copies of R_x and R_y in the original loop. Similarly, A_{urSR} and A_{urVR} in Equation (3) denote the sets of variables belonging to scalar general registers and vector general registers that need to create unrolling copies in the original loop.

(2) Calculate unrolling factors for R_x and R_y. The unrolling factor of R_x is set to UF_x, and the unrolling factor of R_y is set to UF_y. The equations for the unrolling factors are as follows:

The initial value of UF_x is given by the following:

$$UF_x = {}^{numX-|A_{usingX}|}/_{|A_{urX}|}, \tag{4}$$

where $|...|$ refers to the total number of variables in the set, $numX$ represents the total number of R_x in the vector DSP, and A_{usingX} refers to the R_x variables that are active before or after the original loop. Similarly, the initial value of UF_x is given by the following:

$$UF_y = {}^{numY-|A_{usingY}|}/_{|A_{urY}|}. \tag{5}$$

After obtaining the initial values of UF_x and UF_y, multiply them together to obtain UF_{xy} that R_x and R_y can provide for loop unrolling.

$$UF_{xy} = UF_x * UF_y. \tag{6}$$

According to Equations (4) and (5), calculate the maximum unrolling times that can be provided by each register type that does not belong to R_x and R_y such as scalar general registers and vector general registers. Compare them and select the minimum value as UF_R:

$$UF_R = min\{UF_{SR},\ UF_{VR}, \ldots\}. \tag{7}$$

(3) Compare UF_{xy} with UF_R. When UF_{xy} is less than UF_R, it means that the maximum unrolling times that can be provided by A_{urXY} are not limited by the other registers. This situation achieves the effect that the free resources of R_x and R_y are fully used. Therefore, the final unrolling factor UF is given by the following:

$$UF = UF_{xy}. \tag{8}$$

Conversely, when UF_{xy} is greater than UF_R, it indicates that the full use of R_x and R_y leads to passive measures of register protection for other types of registers, which is not in line with the original intention of loop unrolling in this paper. Therefore, it is necessary to round down the smaller values of UF_x or UF_y. The update in UF_x or UF_y is given by the following:

$$min\{UF_x,\ UF_y\} = {}^{UF_R}/_{max\{UF_x,\ UF_y\}} + 1. \tag{9}$$

Update UF_{xy} using Equation (8) after processing UF_x or UF_y according to Equation (9). The updated UF_{xy} must be within the range of

$$\mathrm{UF}_{xy} \in \{UF_{xy} | UF_R \le UF_{xy} < (UF_R + max\{UF_x, UF_y\})\}. \tag{10}$$

Then, the UF is given by the following:

$$UF = UF_R. \tag{11}$$

When UF_{xy} is equal to UF_R, it indicates that the full use of the free resources of R_x and R_y is accompanied by the full use of the free resources from one of the other register types. The final unrolling factor *UF* in this case can be derived by Equation (8) or Equation (11).

3.2.3. Update the Identified Set of Induction Variables

After calculating UF_x and UF_y, judge whether the identified set of induction variables needs to be updated. Calculate the total number of new instructions that need to be generated for loop unrolling when either R_x or R_y is used as the induction variable and compare to find out which one generates fewer new instructions. If the register type resulting from the comparison is the same as the register type identified in the origin loop as the induction variable, there is no need to update the set of induction variables. If different, it is necessary to replace the variables belonging to R_x or R_y in the set of induction variables with the register type derived from the comparison and replace the corresponding code in the loop body during code transformation.

3.2.4. Initialize Unrolling Copies for R_x and R_y

In code transformation, initialize $(UF_x - 1)$ copies for R_x that need to be unrolled in the loop and initialize $(UF_y - 1)$ copies for R_y that need to be unrolled in the loop.

Some rules should be followed when initializing unrolling copies of R_x and R_y. When initializing copies of R_x, the following rule must be obeyed:

$$X\_1 = X \pm L, \tag{12}$$

where *X* refers to a certain variable belonging to R_x that needs to create unrolling copies; *X_1* is the next unrolling copy that needs to be created for *X*; and *L* is the data volume that can be computed in one iteration of the origin loop, which is typically 1 in scalar processors but depends on the architecture of the processor in vector processors.

When initializing copies of R_y, the following rule must be obeyed:

$$Y_1 = Y \pm \left(L * UF_y\right), \tag{13}$$

where *Y* refers to a certain variable belonging to R_y that needs to create unrolling copies; *Y_1* is the next unrolling copy that needs to be created for *Y*.

The code example of Listing 2 after the LUAEMA is shown in Listing 5.

The LUAEMA includes the initialization of unrolling copies about *ARa* and *ARb* and also the initialization of unrolling copies about *ORk*. Compared to the loop shown in Listing 4, which performed the conventional loop unrolling approach of Listing 2, the LUAEMA replaces self-increasing instructions for *ARa* and *ARb* with self-increasing instructions for *ORk* in the loop, which means that the induction variables are updated—the set of induction variables consisting of *Ri*, *Ara*, and *ARb* is updated to the set of induction variables consisting of *Ri* and *ORk*. This is due to the fact that the total number of instructions generated by R_x and R_y as induction variables was compared after the computation of the unrolling factor was completed, and the register type with the smaller total number between them was selected as the register type with self-increasing instructions for the induction variables within the loop.

The output of the LUAEMA is the optimized instruction-level intermediate code. This intermediate code has a large number of instructions in the loop, so the scheduling domains available for VLIW to choose from during code scheduling are large. One major advantage of the LUAEMA is its ability to increase the parallel scheduling domains of instructions, which is reflected in VLIW. The very long instruction word (VLIW) of the vector processor can be filled with both scalar instructions and vector instructions simultaneously. For a loop, the instructions that are involved in its computation (such as the memory access instructions in Listing 5) are vector instructions. In contrast, the self-increasing instructions that control the loop iterations and correct memory access in the loop (such as the self-increasing instruction of *ORk* in Listing 5) are oriented towards data, so they can only

be used with scalar instructions. Loop unrolling increases the number of instructions for scalars and vectors. The increased number of instructions can provide VLIW with more instruction sequences to choose from when filling instruction words, thereby ensuring that the instruction words are fully filled multiple times during the execution of the loop, improving the efficiency of the instruction sequences.

Listing 5. Listing 2 after being processed by the LUAEMA.

```
Mov e/2, UF_x
Mov f, UF_y
Mul UF_x, UF_y, UF
Mov 0, ORk
Add 1, ORk, ORk_1
. . .. . .
Add UF_y, ARa, ARa_1
. . .. . .
Add UF_y, ARb, ARb_1
. . .. . .
do Ri = 1, Rn/UF
   Load ARb[ORk], Rb
   Load ARb[ORk_1], Rb_1
   . . .. . .
   Load ARb_1[ORk], Rb_f
   . . .. . .
   Add 1, Rb, Ra
   Add 1, Rb_1, Ra_1
   . . .. . .
   Store Ra, ARa[ORk]
   Store Ra_1, ARa [ORk_1]
   . . .. . .
   Store Ra_f, ARa_1[ORk]
   . . .. . .
   Add UF, ORk, ORk
   Add UF, ORk_1, ORk_1
   . . .. . .
enddo
```

4. Algorithms

4.1. Identification Algorithm of Variables That Need to Create Unrolling Copies

Each register in a loop can be classified as a loop variant or a loop invariant based on whether or not its value is changed in the loop. When performing the conventional loop unrolling approach, the values of loop invariants are unchanged, and the loop variations need to create unrolling copies based on the unrolling factor. Therefore, finding out the variables that need to create unrolling copies in a loop is essential for loop unrolling processing.

The conventional loop unrolling approach identifies R_x or R_y when identifying the variables that need to create unrolling copies. In order to satisfy the condition of creating unrolling copies for both R_x and R_y proposed in this paper, it is necessary to judge R_x and R_y again based on the identification of the set of variables that needs to create unrolling copies by the conventional loop unrolling approach, to arrive at the final set of variables A_{ur} that needs to create unrolling copies.

A detailed flow chart is shown in the dashed box in Figure 2.

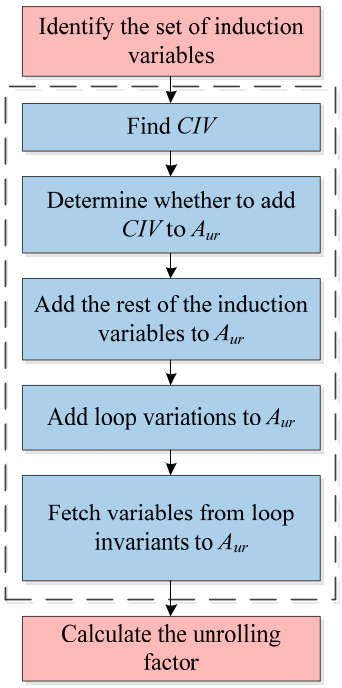

Figure 2. Flow chart for identifying the set of variables that needs to create unrolling copies.

The implementation can be found in Algorithm 1.

Algorithm 1. Identification algorithm of variables that need to create unrolling copies

Input: the set of induction variables(*IVs*); *LCVs*; *LINVs*; definition-use chain derived from data
Output: A_{ur}; *Type*

```
(1)  CIVTimes = getCIVTimes()
(2)  if CIVTimes > 2 then
(3)      A_ur.insert(CIV)
(4)  endif
(5)  for each IV in IVs do
(6)      if IV != CIV then
(7)          A_ur.insert(IV)
(8)      endif
(9)  2ndfor
(10) LCVs = getLCVs()
(11) A_ur.insert(LCVs)
(12) if A_ur.include(R_x) then
(13)     Type = R_y
(14) else
(15)     Type = R_x
(16) endif
(17) for each LINV in LINVs do
(18)     if LINV.type = Type then
(19)         A_ur.insert(LINV)
(20)         LINVs.clear(LINV)
(21)     endif
(22) endfor
```

The content note for Algorithm 1:

(1) *LCVs* is the set of loop variants in the loop.
(2) *LINVs* is the set of loop invariants in the loop.
(3) *getCIVTimes*() is to analyze the times that the loop control induction variable (*CIV*, i.e., *Ri* in Listing 1) has been used during the loop.

(4) A_{ur} is the set of variables that needs to create unrolling copies in the loop.
(5) *getLCVs*() is to obtain the set of loop variations.
(6) *Type* is R_x or R_y that is not included in the induction variables of the origin loop but can become induction variables in loop unrolling. Induction variables in the conventional loop unrolling approach contain R_x or R_y, depending on the form of the loop. Instead of determining which one of R_x or R_y is the induction variable based on the loop form, the LUAEMA compares the total number of new instructions that need to be generated after R_x or R_y is a part of the induction variables.

Unlike the identification algorithm of variables that need to create unrolling copies in the conventional loop unrolling approach, the algorithm proposed in this section takes a certain type of register that would otherwise be part of the loop invariants out of the set of loop invariants and puts it into A_{ur} and provides the *Type* for subsequent updates to the set of induction variables.

4.2. Algorithm for Calculating Unrolling Factor

After determining which variables in the loop need to create unrolling copies, the conventional loop unrolling approach needs to analyze the register types of variables in A_{ur}, calculate the total number of free registers in each register type in the loop, determining the unrolling factor for each register type, and then derive the unrolling factor of the loop.

The unrolling factor algorithm for the LUAEMA divides the registers into two groups, registers for memory access (R_x and R_y) and other registers (*VR*, *SR*, etc.), and sets the unrolling factors UF_x and UF_y for R_x and R_y, respectively. The product of the two unrolling factors is the maximum unrolling factor UF_{xy} that memory access registers can provide.

Other registers are categorized by the register type, and the unrolling factors that the different register types can provide are calculated from the total number of free resources in the different register types. The unrolling factors that are available for each register type are compared and the smallest value is derived as the unrolling factor UF_R that other registers can provide.

The UF_{xy} and UF_R are compared and calculated to obtain the final unrolling factor UF.

A flow chart of the unrolling factor algorithm for the LUAEMA is shown in the dashed box in Figure 3.

The process of realization is described in Algorithm 2.

Algorithm 2. The algorithm for calculating the loop unrolling factor

Input: *LINVs*; A_{ur}; *LiveVs*; *numReg*
Output: UF_x; UF_y; UF; *newIVs*

(1) **for** each *Regtype* in *RegTypes* do
(2) *numUnroll* = A_{ur}(*RegType*).size
(3) *numFree* = *numReg*(*RegType*)- *LINVs*(*RegType*).size- *LiveVs*(*RegType*).size
(4) *factor*(*RegType*) = *numFree* / *numUnroll*
(5) **endfor**
(6) UF_x = *factor*(R_x)
(7) delete *factor*(R_x)
(8) UF_y = *factor*(R_y)
(9) delete *factor*(R_y)
(10) $UF_{xy} = UF_x \times UF_y$
(11) UF_R = min{*factor*(*SR*), *factor*(*VR*); . . .}
(12) **if** $UF_{xy} > UF_R$ **then**
(13) min{UF_x, UF_y} = UF_R / max{UF_x, UF_y} + 1
(14) $UF = UF_R$
(15) **else**
(16) $UF = UF_{xy}$
(17) **endif**
(18) *getNewIVs*()

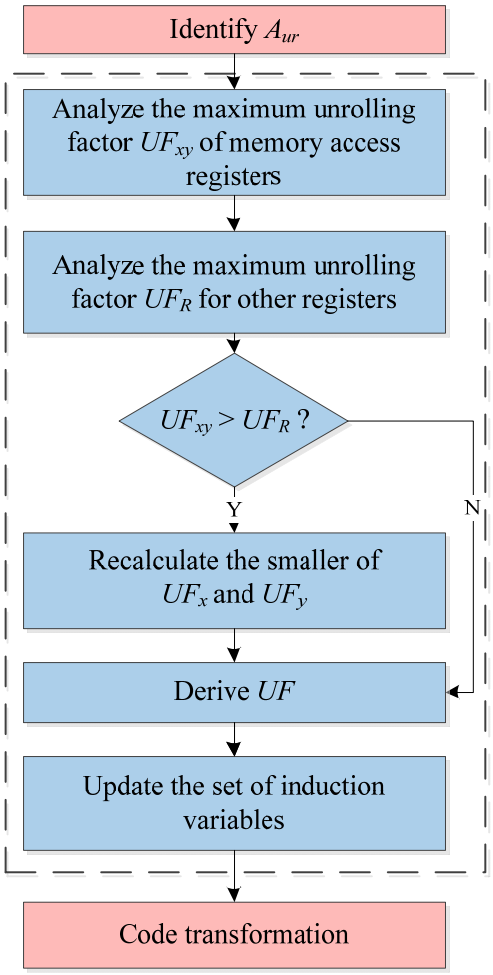

Figure 3. Flow chart of the unrolling factor algorithm.

This algorithm contains the following main elements:

(1) *numUnroll* is the number of variables that need to create unrolling copies in each register type.
(2) A_{ur} is the set of variables that needs to create unrolling copies in the loop.
(3) *numFree* is the number of free resources for each register type. This value is obtained by taking the total number of resources of the register type, *numReg*, and subtracting the number of loop invariants of this class in the loop, *LINVs*(*RegType*).size, and subtracting the number of this register type that are active outside the loop, *LiveVs*(*RegType*).size.
(4) *LINVs* is the set of loop invariants in the loop.
(5) *LiveVs* is the set of registers that are not used in the loop but are still active outside the loop.
(6) *factor* is the function to obtain the maximum unrolling factor that each register type can provide for loop unrolling processing. This value is obtained by dividing *numFree* by *numUnroll*.
(7) UF_x is the maximum unrolling factor determined by R_x.
(8) UF_y is the maximum unrolling factor determined by R_y.
(9) UF_R is the minimum of the maximum unrolling factors for each register type except R_x and R_y.
(10) *UF* is the final unrolling factor.
(11) *getNewIVs*() is used to update the set of *IVs*. The process is mainly performed to obtain the smaller value of the product of UF_x and $A_{ur}(R_x)$.size and the product of UF_y and $A_{ur}(R_y)$.size, then determine whether the register type that the smaller value belongs

to is the same as the register type recorded by the *Type*. If yes, there is no need to update the *IVs*. If no, the registers belonging to the *Type* in the set of registers that need to create unrolling copies are put into the *IVs*, and the R_x or R_y in the *IVs* that do not belong to the *Type* are moved out.

Unlike the conventional algorithm for calculating the unrolling factor, the unrolling factor algorithm in this section fully analyzes the register resources of vector DSP and its characteristics and targets to improve the conventional loop unrolling approach to make it more suitable for the hardware architecture of vector DSP. More specifically, the unrolling factor algorithm of the conventional loop unrolling approach restricted by both R_x and R_y is improved to the algorithm that treats R_x and R_y as the same register type, and the flexible updating of the self-increasing instructions for R_x or R_y as induction variables in the loop is taken into account prior to code transformation.

4.3. Code Transformation

Code transformation of the origin loop is required after determining the unrolling factor. The LUAEMA takes instruction-level intermediate code as input, and the code has been divided into basic blocks. The flow graph for the loop shown in Listing 1 is shown in Figure 4.

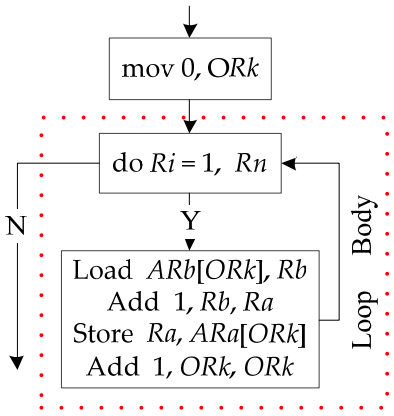

Figure 4. Flow graph of the loop shown in Listing 1.

The three basic blocks from top to bottom in Figure 4 are named according to the function of the different basic blocks: the first basic block is the initialization basic block; the second basic block is the loop control basic block; and the third basic block is the loop calculation basic block. The number of basic blocks involved in the loop will be five after the loop has been processed by the LUAEMA, including the three basic blocks of the original loop, as shown in Figure 4, and the two new basic blocks added to ensure the correctness of the program after the loop unrolling process—the recovery basic block and the tail processing basic block. Therefore, this section divides the code transformation into five parts in terms of the different basic blocks involved in loop unrolling:

The first part is the initialization basic block, which is generally located in the last basic block before entering the loop. The code transformation for this basic block is used to add initialization instructions for UF_x, UF_y, and the final unrolling factor UF, to add an initialization instruction for the total number that the loop can calculate by iterating once, and to add the instructions that initialize the unrolling copies of R_x and R_y according to the unrolling factors of UF_x and UF_y (following the initialization rules of Equations (12) and (13)).

The second part is the loop control basic block. The code transformation in this basic block mainly replaces the variables that represent the data volume that can be calculated in each iteration of the loop.

The third part is the loop calculation basic block. The code transformation of this basic block needs to traverse each of the instructions in turn. If one of the variables in the

instruction is in the set of variables that needs to create unrolling copies, the instructions for its unrolling copies are added after this instruction. The instructions in this basic block, other than the calculation-related instructions, are the self-increasing instructions for the induction variables. The code transformation of these instructions needs to judge and update the self-increasing instructions of the induction variables belonging to R_x and R_y—firstly, according to *getNewIVs*() in the unrolling factor algorithm to determine which one of R_x and R_y determines the self-increasing instructions, and then adding the self-incrementing instructions for the unrolling copies of the induction variables using either UF_x or UF_y.

The fourth part is the recovery basic block for R_x and R_y. The loop calculation basic block may have substitutions for the induction variables, e.g., the induction variables in the loop shown in Listing 2 contain R_x, but after being processing by the LUAEMA (i.e., Listing 5), the induction variables for addressing are changed to R_y. To ensure that the code runs correctly, the values of the induction variables belonging to R_x and R_y need to be modified in accordance with the loop form of the origin loop to fit the subsequent tail processing basic block.

The fifth part is the tail processing basic block. The data volume that can be calculated in one iteration of the unrolled loop is the multiplication result of the unrolling factor and the data volume calculated in one iteration of the original loop. If the total data volume of the loop cannot be evenly divisible by the data volume in one iteration after the LUAEMA, there are two ways to handle it. One is that the total data volume is increased to the point where it can be exactly divided by the data volume in one iteration of the loop, which can lead to the generation of invalid data. The other is to split the total data volume into the largest number that can be divided by the data volume calculated in one iteration and the tail number; the tail number part is calculated by adding the tail processing basic block. The second processing method is used here—the original loop code is added as the tail processing basic block at the end of the unrolling loop.

5. Case Study

In order to demonstrate the processing effect of the LUAEMA, we built a vector DSP processor model based on multiple register types. The model is an SIMD architecture, where vectors are set to calculate eight pieces of data at a time. The register resources can be divided into the following:

a. Sixty-four vector registers (*VRs*), each of them can calculate eight pieces of data and have vector processing units and instructions to support this register type;
b. Sixty-four scalar registers (*Rs*), each of them can calculate one piece of data;
c. Eight address registers (R_x) to hold the contents of the base addresses;
d. Eight offset registers (R_y) to hold the offset of the base addresses.

The vector instruction-level intermediate code of Listing 1 is processed on the basis of this model using the approach proposed in this paper for loop unrolling. Table 1 shows the instruction-level intermediate code of Listing 1.

The instruction-level intermediate code in Table 1 differs from the scalar loop in Listing 1 in that the instruction-level intermediate code uses vector instructions. However, the control instructions for the vector loop are still determined by the SR.

According to the definition of inductive variables, the inductive variables (*IVs*) for Table 1 are *Ri* (derived by S13) and *ORi* (derived by S14), where *Ri* controls the loop times. The analysis of BasicBlock2 yields *Ri* as the loop control induction variable (*CIV*).

First, the variables that need to create unrolling copies are identified. The first step is to analyze the times that the *CIV* (*Ri*) has been used in BasicBlock2 and BasicBlock3. *Ri* is the source operand in S3 and S13, so *Ri* has been used two times, and there is no need to create its unrolling copies. The other inductive variable (*ORi*) is placed into the A_{ur}. The loop variations of BasicBlock3 are *VRx*, *VRz*, *Ri*, and *ORi*, so *VRx* and *VRz* are placed into the A_{ur}. The loop invariants are *ARx*, *ARz*, and *VRy*, and since they contain R_x (*ARx*, *ARz*) that need to be unrolled, *ARx* and *ARz* are cleared from the loop invariants and placed into

the A_{ur}. The updated loop invariant is *VRy*. Finally, the A_{ur} contains *ORi*, *VRx*, *VRz*, *ARx*, and *ARz*. And the value of the *Type* is R_x.

Table 1. The instruction-level intermediate code of Listing 1.

Instruction-Level Intermediate Code		
BasicBlock1:		
S1	SR_To_VR	1, *VRy*
S2	MOV_SR	8, *Rstep*
S3	MOV_SR	0, *Ri*
S4	MOV_SR	0, *ORi*
BasicBlock2:		
S5	ADDI_SR	*Ri*, *Rstep*, *Ra*
S6	LT_SR	*Ra*, *Rsum*, *Rb*
S7	EQ_SR	*Ra*, *Rsum*, *Rc*
S8	OR_SR	*Rb*, *Rc*, *Rc*
S9	[!*Rc*]SBR	BasicBlock4
BasicBlock3:		
S10	LD_VR	*ARx*[*ORi*], *VRx*
S11	ADDI_VR	*VRx*, *VRy*, *VRz*
S12	ST_VR	*VRz*, *ARz*[*ORi*]
S13	ADDI_SR	*Rstep*, *Ri*, *Ri*
S14	ADDI_SR	*Rstep*, *ORi*, *ORi*
S15	SBR	BasicBlock2
BasicBlock4:		
.....		

The intermediate code in Table 1 is processed in the algorithm for calculating the unrolling factor: Grouping the A_{ur} by the register types, the number of *numUnroll* corresponds to 2, 0, 2, and 1 for *VR*, *R*, R_x, and R_y, respectively. The *numAll* for *VR*, *R*, R_x, and R_y is 64, 64, 8, and 8, respectively. The number of loop invariants (*LINVs*) for *VR*, *R*, R_x, and R_y is 1, 0, 0, and 0. *LiveVs* need to be determined based on the number of each register type used when running to the loop head (S5) and loop tail (runs to BasicBlock4); the number of *LiveVs* for *VR*, *R*, R_x, and R_y can be set to 20, 30, 3, and 0.

From the above information, the number of *numFree* for *VR*, *R*, R_x, and R_y is 43, 34, 5, and 8, respectively, so their *factor* is 21, 30, 2, and 8. Then, the value of UF_x is 2 and the value of UF_y is 8. UF_x and UF_y are removed from *factors*, which now only contains the *factors* of *VR* and *R*. The minimum value in all *factors* is compared with the product of UF_x and UF_y, i.e., $21 > 2 \times 8$. The final unrolling factor *UF* is 16. Finally, we compare the product of UF_x and the number of variables (*ARx*, *ARz*) needed to create the unrolling copies for R_x (2×2) with the product of UF_y and the number of variables needed to create the unrolling copies for R_y (*ORi*) (8×1), to obtain the *Type*. R_x is determined as the register type to be self-incremented in the loop.

After obtaining UF_x, UF_y, and *UF*, the code transformation of the original code is started. The following is the description of the transformation of each basic block:

(1) Firstly, there is the initialization basic block, which is BasicBlock1 in Table 1. The definitions of UF_x, UF_y and *UF* are added, the data volume in one iteration of the loop is updated, and instructions for initializing the unrolling copies that need to be newly created in R_x and R_y are added. The BasicBlock1 processed by the LUAEMA is shown in Table 2.

In Table 2, S1 is the definition of UF_x, S2 is the definition of UF_y, and S3 is the definition of *UF*; S6 is the update instruction for the data volume in one iterate of the loop. S9 and S10 are the instructions for initializing the unrolling copies that need to be newly created in R_x. S12 to S18 are the instructions for initializing the unrolling copies that need to be newly created in R_y.

Table 2. BasicBlock1 processed by the LUAEMA.

Instruction-Level Intermediate Code		
BasicBlock1:		
S1	MOV_SR	2, *Rfactor_AR*
S2	MOV_SR	8, *Rfactor_OR*
S3	MOV_SR	16, *Rfacter*
S4	SR_To_VR	1, *VRy*
S5	MOV_SR	8, *Rstep*
S6	MUL_SR	*Rstep*, *Rfactor*, *RunrollingStep*
S7	MOV_SR	0, *Ri*
S8	MUL_SR	*Rfactor_OR*, *Rfactor*, *Rstep_AR*
S9	ADD_SR	*Rstep_AR*, *ARx*, *ARx_1*
S10	ADD_SR	*Rstep_AR*, *ARz*, *ARz_1*
S11	MOV_SR	0, *ORi*
S12	ADD_SR	*Rstep*, *ORi*, *ORi_1*
. . .. . .		
S18	ADD_SR	*Rstep*, *ORi_6*, *ORi_7*

(2) The loop control basic block, i.e., BasicBlock2. *Rstep* is updated to the data volume in one iterate of the loop after the LUAEMA, *RunrollingStep*. BasicBlock2 processed by the LUAEMA is shown in Table 3.

Table 3. BasicBlock2 processed by the LUAEMA.

Instruction-Level Intermediate Code		
BasicBlock2:		
S1	ADDI_SR	*Ri*, *RunrollingStep*, *Ra*
S2	LT_SR	*Ra*, *Rsum*, *Rb*
S3	EQ_SR	*Ra*, *Rsum*, *Rc*
S4	OR_SR	*Rb*, *Rc*, *Rc*
S5	[!*Rc*] SBR	BasicBlock4

(3) The loop calculation basic block, i.e., BasicBlock3. It is divided into two parts: the first is to add the calculation instructions in the loop by A_{ur} and the unrolling factor; the second is to add or update the self-increment instructions of the inductive variables. BasicBlock3 processed by the LUAEMA is shown in Table 4.

Table 4. BasicBlock3 processed by the LUAEMA.

Instruction-Level Intermediate Code		
BasicBlock3:		
S1	LD_VR	*ARx*[*ORi*], *VRx*
. . .. . .		
S8	LD_VR	*ARx*[*ORi_7*], *VRx_7*
S9	LD_VR	*ARx_1*[*ORi*], *VRx_8*
. . .. . .		
S16	LD_VR	*ARx_1*[*ORi_7*], *VRx_15*
S17	ADDI_VR	*VRx*, *VRy*, *VRz*
. . .. . .		
S32	ADDI_VR	*VRx_15*, *VRy*, *VRz_15*
S33	ST_VR	*VRz*, *ARz*[*ORi*]
. . .. . .		
S40	ST_VR	*VRz_7*, *ARz*[*ORi_7*]
S41	ST_VR	*VRz_8*, *ARz_1*[*ORi*]
. . .. . .		
S48	ST_VR	*VRz_15*, *ARz_1*[*ORi_7*]

Table 4. *Cont.*

Instruction-Level Intermediate Code		
S49	ADDI_SR	*Rstep*, *Ri*, *Ri*
S50	ADDI_SR	*RunrollingStep*, *ARx*, *ARx*
S51	ADDI_SR	*RunrollingStep*, *ARx_1*, *ARx_1*
S52	ADDI_SR	*RunrollingStep*, *ARz*, *ARz*
S53	ADDI_SR	*RunrollingStep*, *ARz_1*, *ARz_1*
S54	SBR	BasicBlock2

S1 to S48 are the additions of memory access and calculation instructions in the loop. S50 to S53 are the update instructions for the inductive variables. In the original loop, *ORi* is the inductive variable, but it is updated here to be the self-increment instructions for *ARx* and *ARz*.

(4) The recovery basic block for R_x and R_y. This basic block does not exist in the original code. Before the loop unrolling process, the R_x of the loop in Table 1 are the loop invariants and the R_y are the inductive variables, whereas after the loop unrolling to further improve the performance, the R_x are changed to the inductive variables and the R_y are the loop invariants. To handle data in *Rsum* that are not evenly divided by *RunrollingStep*, R_x and R_y used in the loop need to be changed to the values of the original loop format, so this basic block needs to be added. The basic block is placed between BasicBlock3 and BasicBlock4, as shown in Table 5.

Table 5. The recovery basic block for R_x and R_y.

Instruction-Level Intermediate Code		
BasicBlock_Recover:		
S1	MOV_SR	*Ri*, *ORi*
S2	SUB_SR	*Ri*, *ARx*, *ARx*
S3	SUB_SR	*Ri*, *ARz*, *ARz*

(5) The tail processing basic block. The basic block of Table 5 has restored R_x and R_y used in the loop to their values at the corresponding runtime in the intermediate code of Table 1. The instructions for BasicBlock2 and BasicBlock3 in Table 1 are placed after the basic block of Table 5.

The LUAEMA has completed the processing of the intermediate code in Table 1. Unlike the conventional loop expansion approach, the unrolling factor of the loop is greatly improved, and the number of instructions in the loop is reduced. The conventional loop unrolling approach for the intermediate code in Table 1 yields a value of 8 in the calculation of the unrolling factor, whereas the LUAEMA's unrolling factor is 16. And the LUAEMA reduces the number of instructions in the loop by replacing the self-increment instructions of the inductive variables in the loop under the same unrolling factor, which further improves the performance of the loop.

6. Performance Analysis

In order to test the effect of the LUAEMA in this paper, the YHFT high-performance DSP FT-M7002 was selected as the hardware platform for testing.

The FT-M7002 is a DSP-based 40 nm process chip with a main frequency of 1 GHz, which has two DSP cores and one CPU core. Each DSP core has 64 KB L1D Cache and 32 KB L1P Cache with 32 KB of in-chip scalar space and 768 KB of in-chip vector space. The global shared Cache is 2 MB, and there is up to 32 GB of synchronous Dynamic Random Access Memory (DDR) outside the core [50].

The DSP core of the FT-M7002 is based on the VLIW structure, which contains a five-outflow scalar process unit (SPU) and a six-outflow vector process unit (VPU), and

the two processing units work in a tightly coupled manner, as shown in Figure 5. The SPU contains only one processing unit, which is mainly responsible for serial task processing and program control. The VPU consists of 16 vector process engines (VPEs) that support up to 16 vector operations on 32-bit data, providing parallel processing for intensive computation. DMA (direct memory access) provides a high-speed data transfer path for the core, enabling fast data exchange between out-of-core DDR and SM or AM. AM is a data memory exclusive to VPU, which can support two vectors for read/write operations and two DMAs for read/write operations at the same time when the accesses are not conflicting, for a total of four parallel requests. Through reasonable data arrangement, DMA transfer and vector memory access can be realized in parallel.

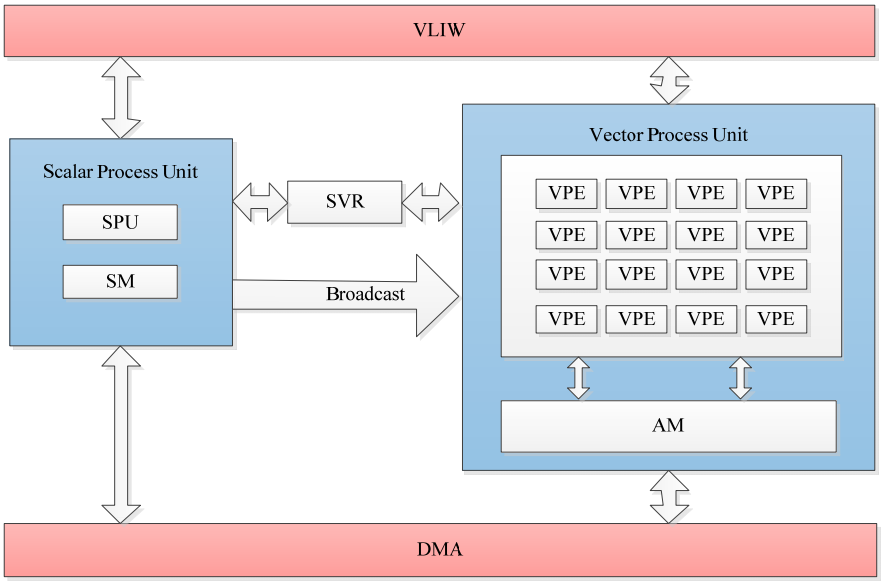

Figure 5. Structure of the FT-M7002 DSP core.

We note that both the output code of the LUAEMA and that of the conventional loop unrolling approach are optimized instructor-level intermediate codes. These intermediate codes need to be further processed by register allocation, code scheduling, and other back-end procedures to obtain the corresponding assembly codes of FT-M7002.

For the algorithms suitable for loop unrolling optimization, we notice that they often have a concise data association relationship, and the data dependency between loop iterations is weak. The calculations in these algorithms are mostly array-oriented. Therefore, we use the following representative digital signal algorithms to analyze the effect of the above-mentioned approach:

(1) The complex data conjugation algorithm. (2) The real data subtraction algorithm. (3) The real data summation algorithm. (4) The real data dot product algorithm. The core codes of the loops of them are "$y\_r[j] = x\_r[i]$, $y\_i[j] = -x\_i[i]$", "$c[i] = a[i] - b[i]$", "$sum += a[i]$", and "$sum += a[i] \times b[i]$", respectively.

These algorithms differ in the amount of R_x and R_y within their computing loops. The complex data conjugation algorithm needs two R_x and two R_y; the real data subtraction algorithm needs three R_x and one R_y; the real data summation algorithm needs one R_x and one R_y; and the real data dot product algorithm needs two R_x and one R_y. Therefore, their unrolling factors are different. In this section, we discuss the efficiency of the LUAEMA by analyzing the unrolling factor and speedup ratio.

6.1. Analysis of Timing Cycles for Algorithms

The input vector intermediate code of the algorithm is the same for both the conventional loop unrolling approach and the LUAEMA. For each algorithm, we consider several data volumes and obtain corresponding timing cycles by experiment. We use C, N, and V

to represent the timing cycle, data volume, and speedup ratio, respectively. The results are shown in Figures 6 and 7, where CLUA means that the corresponding values are obtained under the conventional loop unrolling approach, CnCon refers to the complex data conjugation algorithm, RnSub refers to the real data subtraction algorithm, RnSum refers to the real data summation algorithm, and RnDot refers to the real data dot product algorithm.

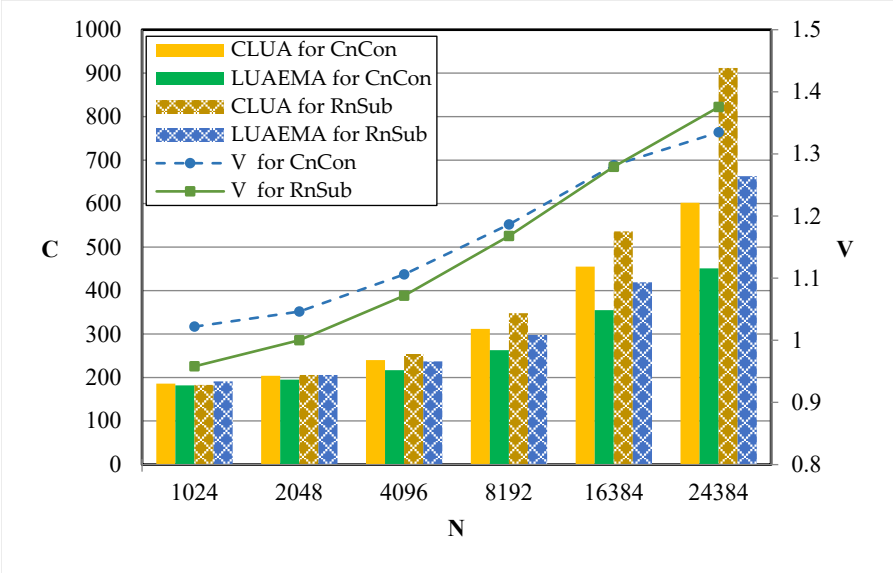

Figure 6. The comparisons of timing cycles and speedup ratios for CnCon and RnSub.

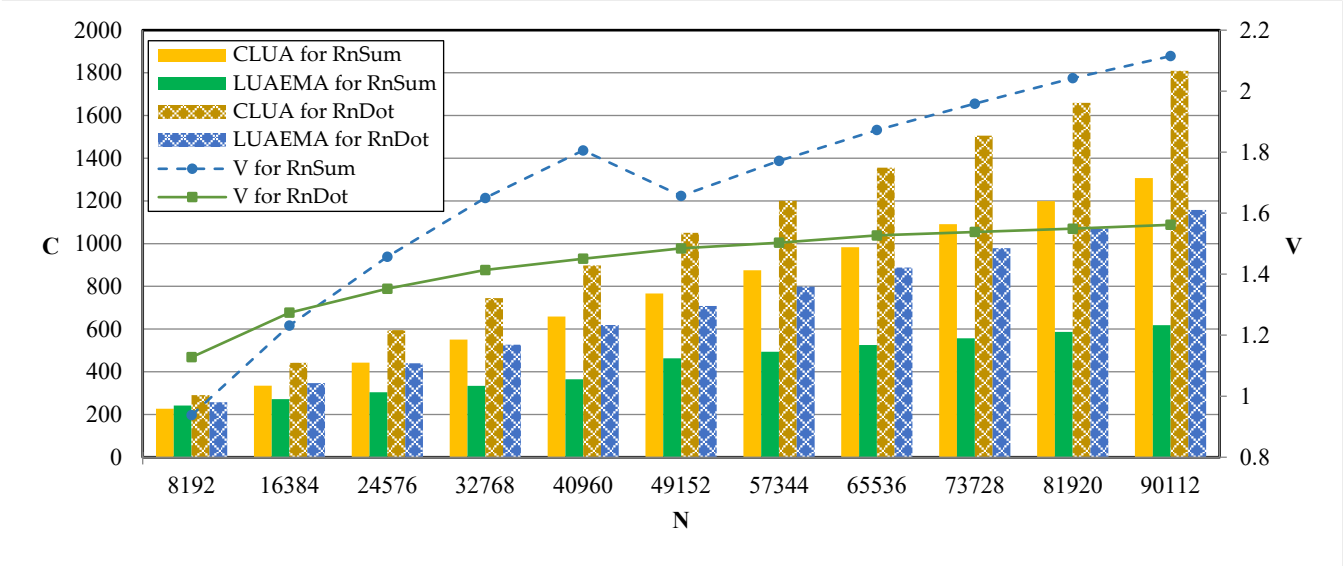

Figure 7. The comparisons of timing cycles and speedup ratios for RnSum and RnDot.

Figure 6 shows that for CnCon, when compared with the conventional loop unrolling approach, the speedup gradually increases with the increase in data volume, and the average speedup ratio is 1.16. When the data volume is 1024 or 2048, the timing cycles of RnSub under the LUAEMA are greater than those under the conventional loop unrolling approach. This is because the loop times are very small under our approach for small data volumes, and thus the profit of the decrease in loop times is smaller than the time cost of the auxiliary instructions introduced in this approach. As the data volume increases, the advantage of the LUAEMA becomes obvious. Compared with the conventional loop

unrolling approach, the maximum speedup ratio of the LUAEMA is 1.38, and the average speedup ratio is 1.14.

Figure 7 shows that for RnSum, when the data volume is larger, the loop unrolling approach in this paper can obtain higher speedup than the conventional loop unrolling. And when the data volume reaches 90,112, the speedup ratio can be 2.11, and the average speedup ratio is 1.81; for RnDot, the increase in V gradually slows down as the data volume increases. When the data volume is around 90,112, V becomes a fixed value, and the speedup ratio is about 1.60.

The overall comparisons show that the LUAEMA can obtain an average speedup ratio of 1.36 over the conventional loop unrolling approach with the same algorithm.

6.2. Analysis of Unrolling Factors for Algorithms

In addition to analyzing the speedup ratios of the different algorithms, the internal unrolling of the algorithms provides a more intuitive demonstration of how the LUAEMA improves the code performance compared to the conventional loop unrolling approach.

We use UF to represent the unrolling factor. Figure 8 shows that the unrolling factor under the conventional loop unrolling approach for CnCon is 4, while the LUAEMA unrolls Rx and Ry to obtain the unrolling factor with a value of 16, which is a triple increase over the value of the unrolling factor compared to the conventional loop unrolling approach; the unrolling factor for RnSub under the LUAEMA is four times higher than that under the conventional loop unrolling approach; the unrolling factor for RnSum under the LUAEMA is increased by a factor of 6.25; and the unrolling factor of RnDot under the LUAEMA is increased by a factor of 2.75.

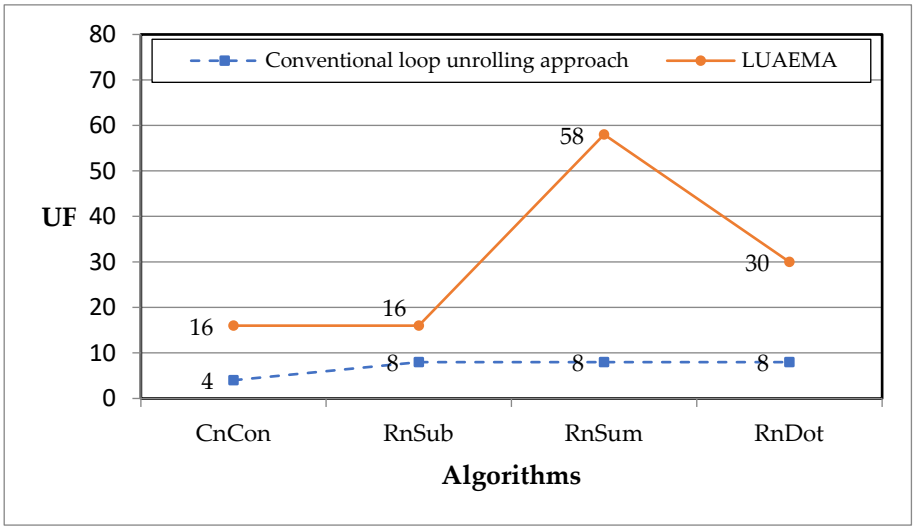

Figure 8. The comparison of UFs after processing four algorithms using two loop unrolling approaches.

The original loop of CnCon uses two R_x, which are used to determine the base addresses of the real and imaginary parts of the complex array, and two R_y, which are used to read or write the different elements of the complex array in order. Whether the array is accessed sequentially by incrementing R_x or incrementing R_y, the unrolling factor under the conventional loop unrolling approach is 4. However, the LUAEMA can obtain an unrolling factor of 16 (4×4).

RnSub uses three address registers in the original loop to hold the base addresses of the minuend, subtracted, and resultant arrays and one offset register to read or write the elements of the three arrays sequentially. The unrolling factor under the conventional loop unrolling approach is 8, whereas the LUAEMA can obtain an unrolling factor with a value of 16.

The original loop of RnSum uses one address register to read the base address of the array. And the number of offset registers used is one. The unrolling factor under the conventional loop unrolling approach for RnSum is 8. The LUAEMA, on the other hand, can obtain an unrolling factor of 64 while ensuring that the other register resources are sufficient (in practice, it can only unroll 58 times because the number of other register resources available is not sufficient to unroll 64 times).

When the input code is RnDot, the unrolling factor under the conventional loop unrolling approach is 8. However, the LUAEMA considers the full utilization of memory access registers and obtained an unrolling factor of 30. Before register spilling happens, the unrolling factor by R_x is 4 and that by R_y is 8. Fully utilizing R_x and R_y in the LUAEMA can increase the unrolling factor to 32, but the unrolling factor for the general register is 30. Therefore, the final unrolling factor is 30.

Combining the speedup ratios of the timing cycles derived from the comparisons and the increase in the unrolling factors of the different algorithms, it can be found that when the unrolling factor is larger, the advantage in timing cycles is greater. The conventional loop unrolling approach cannot take full advantage of the multi-register types of vector DSP, while the LUAEMA can achieve better performance before register spilling happens.

7. Conclusions and the Future Work

In this paper, we propose an instruction-level loop unrolling approach for a vector DSP that makes full use of different register types, the LUAEMA, in response to the limitation that the conventional loop unrolling approach leads to unused free registers in vector DSPs. Firstly, the authors analyze the limitations of the conventional loop unrolling approach implemented on a vector DSP, which in turn leads to its insufficient use of registers. Secondly, a mathematical model is constructed to elucidate the basic principles of the LUAEMA. Then, the implementation process of the LUAEMA is explained by introducing several algorithms for loop unrolling. In order to verify the effectiveness of this approach, the authors selected four algorithms on the FT-M7002 DSP. The experiments show that the LUAEMA's code optimization on the FT-M7002 DSP significantly outperforms the conventional loop unrolling approach, where the unrolling factor is increased by 3.25 on average, and the speedup ratio is up to 1.36 on average. The LUAEMA is a loop unrolling approach in compilation optimization, which is an automatic processing pass for its input code, and thus the output unrolled loop code is automatically generated. However, because its input is instruction-level intermediate code, we need to manually write the instruction-level intermediate code corresponding to the high-level language. The same applies to the conventional approach.

A vector DSP not only uses the address register and offset register to access memory but also uses the address register and immediate number to access memory. This indicates that the immediate value can be used in the instruction instead of the offset register, and the usable number of the offset register can be the sum of the actual total number and the number of immediate values. Therefore, our next work is to consider the memory access mode of the address register and immediate number to improve the LUAEMA. Besides, the unrolling factor is not restricted by the requirement of avoiding the generation of register spilling code. Therefore, we can try to study the method to further expand the unrolling factor, so that the corresponding code can run more effectively on vector DSPs.

Author Contributions: Conceptualization, Y.H.; methodology, Y.H.; software, A.C.; validation, A.C.; formal analysis, W.L.; investigation, Z.T. and P.L.; writing—original draft preparation, Y.H. and A.C.; writing—review and editing, Y.H. and A.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Science and Technology Major Project (No. 2022ZD0119003) and the Hunan Provincial Natural Science Foundation (No. 2023JJ50019).

Data Availability Statement: The authors approve that data used to support the findings of this study are included in the article.

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

References

1. Frantz, G. Digital signal processor trends. *IEEE Micro* **2000**, *20*, 52–59. [CrossRef]
2. Bariko, S.; Arsalane, A.; Klilou, A.; Abounada, A. Efficient parallel implementation of Gaussian Mixture Model background subtraction algorithm on an embedded multi-core Digital Signal Processor. *Comput. Electr. Eng.* **2023**, *110*, 108827. [CrossRef]
3. Khan, S.; Alzaabi, A.; Iqbal, Z.; Ratnarajah, T.; Arslan, T. A Novel Digital Twin (DT) Model Based on WiFi CSI, Signal Processing and Machine Learning for Patient Respiration Monitoring and Decision-Support. *IEEE Access* **2023**, *11*, 103554–103568. [CrossRef]
4. Mohammed, A.A.A.; Husam, A.W.; Ahmed, A.A. Design and Implementation of Communication Digital FIR Filter for Audio Signals on the FPGA Platform. *J. Commun.* **2023**, *18*, 89–96.
5. Gouveia, C.; Albuquerque, D.; Vieira, J.; Pinho, P. Dynamic Digital Signal Processing Algorithm for Vital Signs Extraction in Continuous-Wave Radars. *Remote Sens.* **2021**, *13*, 4079. [CrossRef]
6. Ibrahim, D.; Davies, A. The Evolution of Digital Signal Processors. In Proceedings of the 2019 6th IEEE History of Electrotechnology Conference (HISTELCON), Glasgow, UK, 18–19 September 2019.
7. Who, M.; Seo, S.; Mahlke, S.; Mudge, T.; Chakrabarti, C.; Flautner, K. AnySP: Anytime Anywhere Anyway Signal Processing. *IEEE Micro* **2010**, *30*, 81–91.
8. Rowen, C.; Nicolaescu, D.; Ravindran, R.; Heine, D.; Martin, G.; Kim, J.; Maydan, D.; Andrews, N.; Huffman, B.; Papaparaskeva, V.; et al. The world's fastest DSP core: Breaking the 100 GMAC/s barrier. In Proceedings of the 2011 IEEE Hot Chips 23 Symposium (HCS), Stanford, CA, USA, 17–19 August 2011.
9. Texas Instruments. *TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor, Data Manual*; Texas Instruments: Dallas, TX, USA, 2010.
10. Wang, Y.; Li, C.; Liu, C.; Liu, S.; Lei, Y.; Zhang, J.; Zhang, Y.; Guo, Y. Advancing DSP into HPC, AI, and beyond: Challenges, mechanisms, and future directions. *CCF Trans. High Perform. Comput.* **2021**, *3*, 114–125. [CrossRef]
11. Wang, Z.; Dong, X.; Kang, Y.; Chen, H. Parallel SHA-256 on SW26010 many-core processor for hashing of multiple messages. *J. Supercomput.* **2022**, *79*, 2332–2355. [CrossRef]
12. Reay, D.S. *Digital Signal Processing Using the ARM Cortex M4*; Wiley: Hoboken, NJ, USA; Blackwell: Hoboken, NJ, USA, 2015; pp. 2–7.
13. van Berkel, K.; Heinle, F.; Meuwissen, P.P.E.; Moerman, K.; Weiss, M. Vector Processing as an Enabler for Software-Defined Radio in Handheld Devices. *EURASIP J. Adv. Signal Process.* **2005**, *2005*, 906408. [CrossRef]
14. Damjancevic, S.A.; Matus, E.; Utyansky, D.; van der Wolf, P.; Fettweis, G.P. Channel Estimation for Advanced 5G/6G Use Cases on a Vector Digital Signal Processor. *IEEE Open J. Circuits Syst.* **2021**, *2*, 265–277. [CrossRef]
15. Synopsys, Inc. ASIP Designer Website. Available online: https://www.synopsys.com/asip (accessed on 9 August 2020).
16. Lu, K.; Wang, Y.; Guo, Y.; Huang, C.; Liu, S.; Wang, R.; Fang, J.; Tang, T.; Chen, Z.; Liu, B.; et al. MT-3000: A heterogeneous multi-zone processor for HPC. *CCF Trans. High Perform. Comput.* **2022**, *4*, 150–164. [CrossRef]
17. Razilov, V.; Matúš, E.; Fettweis, G. Communications Signal Processing Using RISC-V Vector Extension. In Proceedings of the 2022 International Wireless Communications and Mobile Computing (IWCMC), Dubrovnik, Croatia, 30 May–3 June 2022.
18. Jumana, M.; Mohamed, A.B.; Roel, J. Fast and Portable Vector DSP Simulation Through Automatic Vectorization. In Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES '18), Association for Computing Machinery, New York, NY, USA, 28 May 2018.
19. VanHattum, A.; Nigam, R.; Lee, V.T.; Bornholt, J.; Sampson, A. Vectorization for digital signal processors via equality saturation. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), Association for Computing Machinery, New York, NY, USA, 17 April 2021.
20. Hu, Y.H.; Zhang, X.; Wang, S.Y.; Liang, W.; Li, K.C. Research on global register allocation for code containing array-unit dual-usage register names. *Concurr. Comput. Pract. Exp.* **2023**, *35*, e7519. [CrossRef]
21. Zhao, X.L.; Chen, Z.Y.; Shi, Y.; Wen, M.; Zhang, C.Y. Kernel Code Automatic Generation Framework on FT-Matrix. *J. Comput. Res. Dev.* **2023**, *60*, 1232–1245.
22. Wang, C.W.; Zhao, D.S. The Design of TMS320 DSP Simulation and Testing System. In Proceedings of the 2010 Second World Congress on Software Engineering, Wuhan, China, 19–20 December 2010.
23. Ghassan, S.; Vahl, S.G.; Paul, M.; Theodore, D.; Austin, K. Register-Pressure-Aware Instruction Scheduling Using Ant Colony Optimization. *ACM Trans. Archit. Code Optim.* **2022**, *19*, 1–23.
24. Kim, M.; Park, J.; Moon, S.M. Irregular Register Allocation for Translation of Test-pattern Programs. *ACM Trans. Archit. Code Optim.* **2021**, *18*, 1–23. [CrossRef]
25. Xiong, N.; Han, W.; Vandenberg, A. Green cloud computing schemes based on networks: A survey. *IET Commun.* **2012**, *6*, 3294–3300. [CrossRef]
26. Wang, J.; Jin, C.; Tang, Q.; Xiong, N.; Srivastava, G. Intelligent ubiquitous network accessibility for wireless-powered MEC in UAV-assisted B5G. *IEEE Trans. Netw. Sci. Eng.* **2020**, *8*, 2801–2813. [CrossRef]
27. Wan, R.; Xiong, N.; Hu, Q.; Wang, H.; Shang, J. Similarity-aware data aggregation using fuzzy c-means approach for wireless sensor networks. *EURASIP J. Wirel. Commun. Netw.* **2019**, *2019*, 59. [CrossRef]

28. Zhang, S.; Hu, B.; Liang, W.; Li, K.C.; Gupta, B.B. A Caching-Based Dual K-Anonymous Location Privacy-Preserving Scheme for Edge Computing. *IEEE Internet Things J.* **2023**, *10*, 9768–9781. [CrossRef]
29. Diao, C.; Zhang, D.; Liang, W.; Li, K.C.; Hong, Y.; Gaudiot, J.L. A Novel Spatial-Temporal Multi-Scale Alignment Graph Neural Network Security Model for Vehicles Prediction. *IEEE Trans. Intell. Transp. Syst.* **2023**, *24*, 904–914. [CrossRef]
30. Kang, L.; Chen, R.S.; Xiong, N.; Chen, Y.C.; Hu, Y.X.; Chen, C.M. Selecting Hyper-Parameters of Gaussian Process Regression Based on Non-Inertial Particle Swarm Optimization in Internet of Things. *IEEE Access* **2019**, *7*, 59504–59513. [CrossRef]
31. Cai, J.; Liang, W.; Li, X.; Li, K.; Gui, Z.; Khan, M.K. GTxChain: A Secure IoT Smart Blockchain Architecture Based on Graph Neural Network. *IEEE Internet Things J.* **2023**, *10*, 21502–21514. [CrossRef]
32. Hu, W.J.; Fan, J.; Du, Y.X.; Li, B.S.; Xiong, N.; Bekkering, E. MDFC–ResNet: An Agricultural IoT System to Accurately Recognize Crop Diseases. *IEEE Access* **2020**, *8*, 115287–115298. [CrossRef]
33. Wang, Y.; Fang, W.; Ding, Y.; Xiong, N. Computation offloading optimization for UAV-assisted mobile edge computing: A deep deterministic policy gradient approach. *Wirel. Netw.* **2021**, *27*, 2991–3006. [CrossRef]
34. Shen, X.; Yi, B.; Liu, H.; Zhang, W.; Zhang, Z.; Liu, S.; Xiong, N. Deep variational matrix factorization with knowledge embedding for recommendation system. *IEEE Trans. Knowl. Data Eng.* **2019**, *33*, 1906–1918. [CrossRef]
35. Shen, Y.; Fang, Z.; Gao, Y.; Xiong, N.; Zhong, C.; Tang, X. Coronary Arteries Segmentation Based on 3D FCN With Attention Gate and Level Set Function. *IEEE Access* **2019**, *7*, 42826–42835. [CrossRef]
36. Li, Y.; Liang, W.; Xie, K.; Zhang, D.; Xie, S.; Li, K.C. LightNestle: Quick and Accurate Neural Sequential Tensor Completion via Meta Learning. In Proceedings of the IEEE INFOCOM 2023—IEEE Conference on Computer Communications, New York, NY, USA, 17–20 May 2023.
37. Liu, S.; Zhao, B.; Jiang, Q.; Wu, W.G. A Semi-Automatic Coarse-Grained Parallelization Approach for Loop Optimization and Irregular Code Sections. *Chin. J. Comput.* **2017**, *40*, 2127–2147.
38. Kumar, S.A. Enhancing the Scope for Automated Code Generation and Parallelism by Optimizing Loops through Loop Unrolling. In Proceedings of the 2020 Fourth International Conference on Inventive Systems and Control (ICISC), Coimbatore, India, 8–10 January 2020.
39. Gao, W.; Zhao, R.C.; Yu, H.N.; Zhang, Q.H. Loop Unrolling in Vectorized Programs. *Comput. Sci.* **2016**, *43*, 226–231, 245.
40. Rocha, R.C.O.; Porpodas, V.; Petoumenos, P.; Góes, L.F.W.; Wang, Z.; Cole, M.; Leather, H. Vectorization-aware loop unrolling with seed forwarding. In Proceedings of the 29th International Conference on Compiler Construction (CC 2020), Association for Computing Machinery, New York, NY, USA, 24 February 2020.
41. Yang, C.; Yang, X.; Xue, J. Improving the Performance of GCC by Exploiting IA-64 Architectural Features. In Proceedings of the 10th Asia-Pacific Conference on Advances in Computer Systems Architecture (ACSAC), Berlin, Germany, 24–26 October 2005.
42. Monniaux, D.; Six, C. Formally Verified Loop-Invariant Code Motion and Assorted Optimizations. *ACM Trans. Embed. Comput. Syst.* **2022**, *22*, 1–27. [CrossRef]
43. Dong, Y.; Li, C.; Xu, Y. Implementation and effects of loop-array-prefetching optimization in GCC. *Comput. Eng. Appl.* **2016**, *52*, 19–25.
44. Stephenson, M.; Amarasinghe, S. Predicting Unroll Factors Using Supervised Classification. In Proceedings of the International Symposium on Code Generation and Optimization (CGO '05). IEEE Computer Society, Washington, DC, USA, 20–23 March 2005.
45. Li, G.; Hu, Y.; Qiu, Y.; Huang, W. Investigation on the Optimization for Storage Space in Register-Spilling. In Proceedings of the Collaborate Computing: Networking, Applications and Worksharing, Berlin, Germany, 5 July 2017.
46. Monsifrot, A.; Bodin, F.; Quiniou, R. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA), Varna, Bulgaria, 4–6 September 2002.
47. Wang, D.; Zhao, R.C.; Gao, W.; Li, Y.B. Loop unrolling method based on random decision forest. *Comput. Eng. Des.* **2018**, *39*, 1–6.
48. Li, W.L.; Liu, L.; Tang, Z.Z. Loop unrolling optimization for software pipelining. *J. Beijing Univ. Aeronaut. Astronaut.* **2004**, *30*, 1111–1115.
49. Liu, X.; Ding, L.; Li, Y.; Chen, G.; Du, J. Research of Register Pressure Aware Loop Unrolling Optimizations for Compiler. In Proceedings of the 3 International Conference on Circuits and Systems (CAS 2018), Seoul, Republic of Korea, 14 November 2018.
50. Xie, A.; Hu, Y.; Cheng, A.; Tang, Z.; Liu, P.; Zhang, X. Advancing Matrix Decomposition Efficiency: A Study on FT-Matrix DSP Based SVD Optimization. In Proceedings of the 2023 IEEE 10th International Conference on Cyber Security and Cloud Computing (CSCloud)/2023 IEEE 9th International Conference on Edge Computing and Scalable Cloud (EdgeCom), Xiangtan, China, 1–3 July 2023.