electronics

MDPI

Article

Vectorization Programming Based on HR DSP Using SIMD

Chunhu Xie [1,*], Huachun Wu [1,2,*] and Jian Zhou [1]

[1] School of Mechanical and Electronic Engineering, Wuhan University of Technology, Wuhan 430070, China
[2] Hubei Provincial Engineering Technology Research Center for Magnetic Suspension, Wuhan 430070, China
* Correspondence: xiechunhu@whut.edu.cn (C.X.); whc@whut.edu.cn (H.W.)

Abstract: Single instruction multiple data (SIMD) vector extension has become an essential feature of high-performance processors. Architectures such as x86, ARM, MIPS, and PowerPC have specific vector extension instruction sets and SIMD micro-architectures. Using SIMD vectorization programming can significantly improve the performance of application algorithms while keeping the hardware overhead low. In addition, other methods can enhance algorithm performance, such as selecting the best SIMD vectorization model for algorithms, ensuring sufficient instruction streams, implementing reasonable and effective cache data prefetching, and aligning data access and storage addresses according to instruction characteristics. The goal of this paper is three-fold. First, we introduce the basic structural characteristics of a general RISC processor, Hua Rui (HR) DSP, with a custom vector instruction set based on compatibility with an MIPS64 fixed-point and floating-point instruction set, as well as a Fei Teng (FT) processor compatible with an ARMv8 instruction set. Second, we summarize the fundamental principles of SIMD vectorization programming design for the HR DSP, which provides ideas for other scholars or engineering and technical personnel to study the algorithm performance using SIMD vectorization optimization. Third, we implement representative typical algorithms based on the HR and FT platforms and obtain experimental results that show improvement in algorithm SIMD vectorization optimization according to the vector programming design principles summarized in this article can improve the single-core performance of scalar implementation without vectorization, instruction streams, and cache data prefetching by 4–22 times for mean filter, accumulation, and matrix–matrix multiplication, which is significantly better than the performance improvement of 3–13 times for the FT platform. Moreover, the performance of matrix–matrix multiplication using the best vectorization model on the HR platform is about 84% higher than that of the common SIMD vectorization model.

Keywords: DSP; SIMD; algorithm; cache; instruction stream

Citation: Xie, C.; Wu, H.; Zhou, J. Vectorization Programming Based on HR DSP Using SIMD. *Electronics* **2023**, *12*, 2922. https://doi.org/10.3390/electronics12132922

Academic Editor: Javid Taheri

Received: 17 May 2023
Revised: 30 June 2023
Accepted: 30 June 2023
Published: 3 July 2023

1. Introduction

With the continuous improvement in integrated circuits, computer architecture, and chip design and manufacturing technology, parallel computing based on SIMD [1] has gradually become an indispensable key technology for high-performance general purpose processors (GPPs). This has also driven research institutions and engineering technicians to focus on improving the SIMD vectorization performance.

Since the 1990s, Intel has been the first to integrate multi-media extensions (MMX) [2,3] on a Pentium processor, and vector calculation based on SIMD has gradually become an important functional component of microprocessors. Subsequently, several chip vendors have also integrated SIMD vector units on their processors, such as AMD's 3dNow! [4]; Alpha's MVI extension [5]; PowerPC's AltiVec extension [6]; ARM's NEON [7]; MIPS's ASE extension [8]; and Intel's subsequent development of 128-bit wide SIMD extensions (SSE, SSE2, SSE3, SSSE3, SSE4.1, and SSE4.2) [9], 256-bit wide advanced vector extensions (AVX, AVX2) [10], and 512-bit wide advanced vector extensions (AVX3) [1].

In this paper, we use the HR DSP, which is customized with a vector instruction set extension based on the compatible MIPS64 instruction set. The processor integrates 64 256-bit wide vector registers, which is conducive to improving the degree of parallel computing. More registers can improve the cache line data utilization and cache hit rate, and reduce additional data movement to the cache overhead. To demonstrate the advantages of implementing vectorization on the HR DSP, we compared it with the FT2000+ processor, which is compatible with the ARMv8 instruction set. The FT2000+ processor integrates 64 FTC662 processor cores, providing 32 128-bit SIMD and floating-point registers; each core has 32 KB L1 data cache, 32 KB L1 instruction cache, and a four-core shared 2 MB L2 cache. It adopts a four-issue out-of-order superscalar five-stage pipeline structure.

In this work, we illustrate the development of SIMD technology. We first summarize the SIMD vector extensions provided by current processor manufacturers in the market, and then outline the basic structure and SIMD vector extensions provided by HR DSP, as well as the characteristics of the vector extension instruction set. Finally, we summarize the design principles of HR DSP vectorization programming and implement three typical algorithms for experimental verification from the perspective of performance improvement.

The organization of this paper is as follows. Section 2 describes the basic structure and characteristics of HR DSP. The background information on the basic principles of SIMD technology is introduced in Section 3. Section 4 outlines the basic principles of HR DSP SIMD vectorization programming design. To verify the design principles of HR DSP SIMD vectorization programming, Section 5 explains the SIMD vectorization of typical algorithms on HR DSP and the implementation of different optimization methods. Section 6 compares and analyzes the performance impact of HR DSP vectorization programming design principles on different algorithms, demonstrating the advantages of HR vectorization compared with the FT platform. Finally, the experimental results are analyzed and the conclusion is drawn in Section 7.

2. HR DSP Overview

This section introduces the basic architecture of HR DSP and vector processing microarchitecture, provides an overview of the instruction set of the MIPS64 architecture compatible with HR DSP, focuses on the description of custom extended vector instruction set and some vector instructions, and introduces the structural characteristics of the basic instruction pipeline stages and the out-of-order four-issue superscalar used to improve pipeline efficiency.

2.1. Basic Structure

HR DSP (a self-developed product based on the MIPS64 architecture) is a heterogeneous multi-core processor designed for high-performance computing and digital signal processing applications, which combines general-purpose CPU and DSP fusion technology. As shown in Figure 1, HR DSP includes four HR DSP vector cores (each core has 32 KB L1Icache, 32 KB L1Dcache, and 1 MB L2cache), four configurable dedicated processing cores (RASP, which can be configured to achieve specialized signal processing algorithm calculations), four channels of general DMA (responsible for data transfer and matrix transposition), three 72-bit DDR3-1600 memory controllers, one PCIE, two RapidIO, three gigabit Ethernet, Low-speed IO (including UART, SPI, I2C, Nor/Nand Flash, LPC, CAN, GPIO, etc.), ACE (Cache coherence), IO_ACE (IO coherence), AXI Crossbar (interconnecting devices that do not support Cache coherence), AXI to APB (AXI to APB protocol conversion bridge, connecting low-speed interfaces), LODMA (low-speed device and memory data exchange DMA controller), etc.

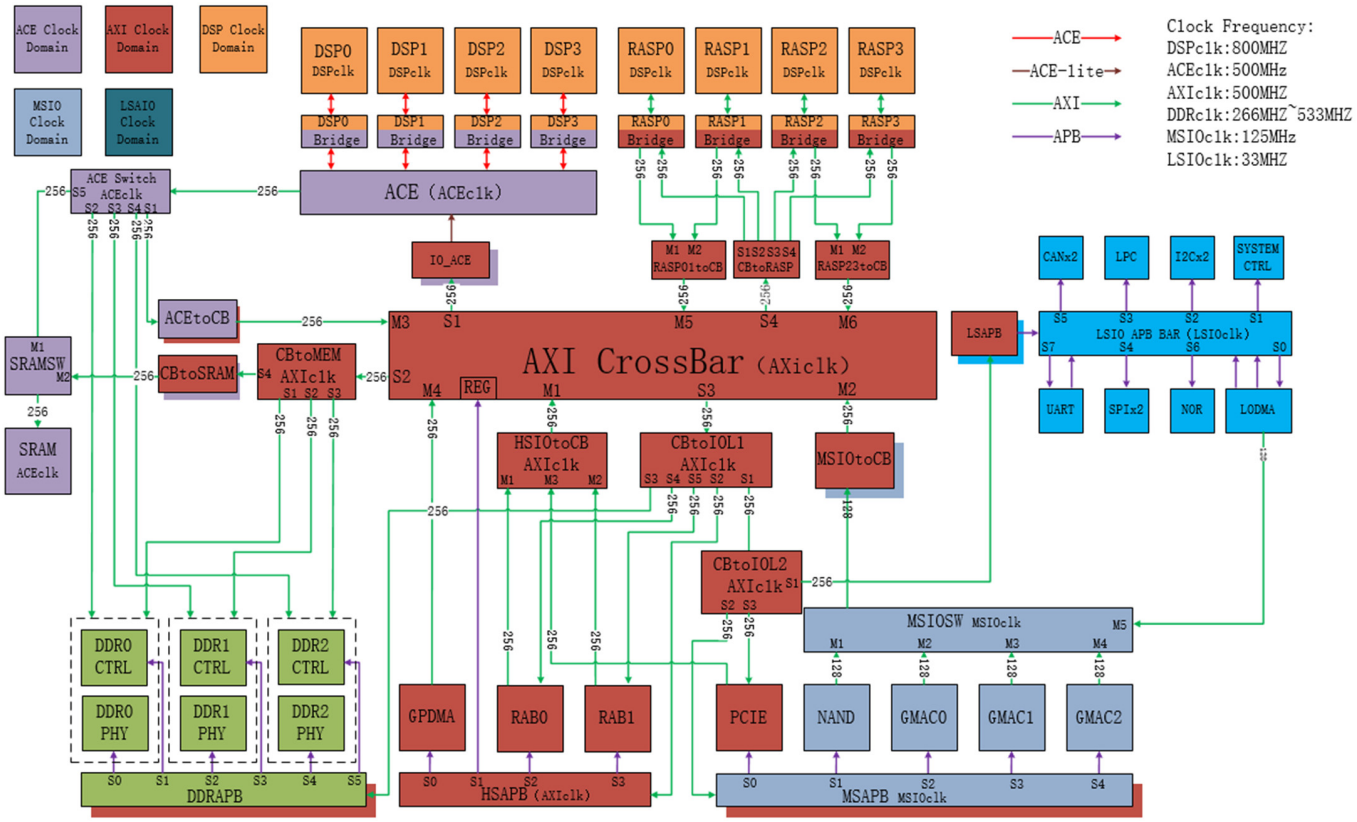

Figure 1. Structure diagram of HR DSP.

Figure 2 illustrates that the HR DSP vector core features two fixed-point ALUs, one fixed-point multiply-add unit with fixed-point ALU functionality, one branch queue unit, and one memory access queue unit that can dispatch two instructions simultaneously. It also includes two vector and floating-point ALUs, a fixed-point register file with 32 64-bit fixed-point registers, a floating-point register file with 32 64-bit floating-point registers, a renaming register file with 32 64-bit registers shared between fixed-point and floating-point registers, and a vector register file with 64 256-bit registers. Among the vector register file, 32 registers are dedicated to vector register renaming.

The HR DSP vector core's vector expansion is based on a vector register file, two vector operation units (VALU0 and VALU1), and two reservation stations that correspond to the two vector operation units. The vector register file contains a total of 64 items, each of which is 256 bits and provides operands for vector components. Of these items, 32 are used as renaming registers for the temporary storage of operation results. The vector operator reads up to three source operands from the vector register and produces a single result that is written back. In addition to the floating-point and fixed-point vector operations described earlier, the vector operator also supports flexible shift operations.

Furthermore, the memory access unit of the HR DSP vector core is extended from the MIPS64 instruction set by adding vector memory access instructions. These instructions support 256, 64, and 32-bit memory access operations and can initiate two memory access operations simultaneously per cycle.

2.2. Instruction Set

The HR DSP core comprises three coprocessors: Coprocessor 0 (CP0), Coprocessor 1 (CP1), and Coprocessor 2 (CP2). CP1 and CP2 are Vector Floating Point Units (VFPU). CP0 (system processor) manages memory and handles exceptions through the CP0 register. CP1 (floating-point coprocessor) instructions include floating-point, multimedia, and extended fixed-point computation instructions that operate on floating-point registers. CP2 (vector

coprocessor) instructions are vector instructions, including vector fixed-point and vector floating-point instructions, which operate on vector registers. Vector access instructions are executed by access tokens, but they can only be executed if CP2 is enabled and only support 32-byte aligned address access.

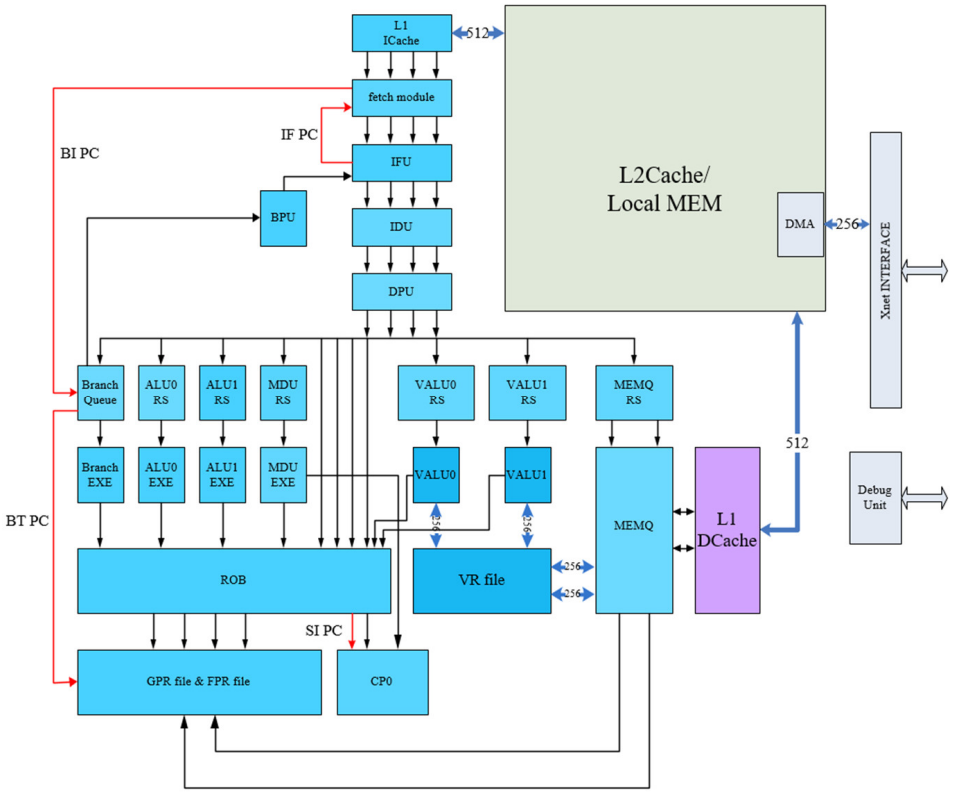

Figure 2. Basic structure of the HR DSP vector core.

The HR DSP core is compatible with the MIPS64 architecture and provides a complete set of instructions defined by it. The instruction encoding is 32 bits, including three instruction formats: immediate number instruction (I-type), jump instruction (J-type), and register instruction (R-type), as shown in Figure 3 (where OP is a 6-bit operation code, RS is a 5-bit source operation register field, RT is a 5-bit target (source/destination) operation register or jump condition, Immediate indicates the 16-bit immediate number, Target is the 26-bit jump target address, RD is a 5-bit destination operation register field, SA is a 5-bit shift, and FUNCT is a 6-bit functional domain).

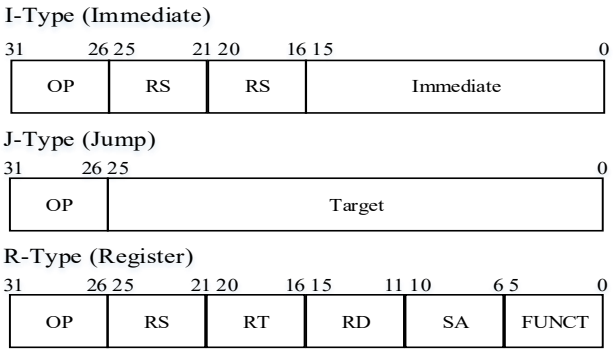

Figure 3. Instruction format.

The fixed-point instruction set in HR DSP, which is compatible with MIPS64, includes 172 instructions for logic operations, control operations, memory access, and more. The

floating-point instruction set includes 70 instructions for floating-point operations, floating-point control, floating-point memory access, and more. The vector instruction set includes 455 instructions for vector fixed-point operations, vector floating-point operations, vector mixing, vector control, vector access, and more.

The 256-bit vector floating-point SIMD operation includes instructions such as floating-point addition (subtraction), floating-point multiplication (subtraction), floating-point division, floating-point square root, floating-point inverse, conversion between floating-point and fixed-point formats, floating-point precision conversion, floating-point comparison, transfer judgment, vector trigonometric functions, and other simple logic. Additionally, instructions to support complex multiplication and Fast Fourier Transform (FFT) are also added, as shown in Table 1.

Table 1. Partial fixed-point vector instructions and descriptions.

OpCode	Description
VADDPS	Single precision floating-point addition
VMULPS	Single precision floating-point multiplication
VMULADDPS	Single precision floating-point multiplication–addition
VSQRTPS	Single precision floating-point square root
VRCPPS	Single precision floating-point reciprocal
VCMPLTPS	Single-precision floating-point comparison, less than
VCSMUL1	Single precision complex multiplication first step
VCSMUL2	Single precision complex multiplication step two
VCSFFTS1L	FFT first low-level operation
VCSFFTS1H	FFT first level-high operation
VCSFFTS2E	FFT second-level even 64-bit operation
VCSFFTS2O	FFT Level 2 odd 64-bit operation
VSINPS	sine
VCOSPS	cosine
VLOG2PS	Two base log function

The 256-bit vector fixed-point operation includes instructions such as fixed-point addition and subtraction, parallel addition and subtraction, taking maximum or minimum values, taking average or absolute values, fixed-point multiplication, fixed-point conversion, displacement and mixing, mixing, packaging, shift, extraction, insertion, and more, as shown in Table 2.

Table 2. Partial fixed-point vector instructions and descriptions.

OpCode	Description
VPADDW	Vector word addition
VPSUBW	Vector word subtraction
VPHADDW	Vector parallel words addition
VPMINSW	The signed word takes the minimum value
VPMULLW	Word multiplication, store low, sign
VPMULHW	Word multiplication, high storage, signed
VPBLENDB	Byte mixing
VPACKSDSWS	Signed double words packed into words, saturated
VPALIGNRI	Take the result according to the immediate number join right shift right alignment
VPINSRDI	Implement double—word insertion as required

2.3. Instruction Stream

The HR DSP's basic instruction stream consists of nine stages, including fetching, pre-decoding, decoding, sending, transmitting, reading register, executing, submitting. This nine-stage stream, which is based on a data-driven principle, is quite different from the five-stage stream based on the instruction cycle alignment used in MIPS. The HR

DSP's instruction stream decodes four instructions in each clock cycle and dynamically sends them to seven functional units. The hardware analyzes the correlation between the instructions and controls the execution of the instruction by determining whether the instruction operand is ready. Although instructions are executed out of order according to their dependencies, they are delivered in the program's original order to ensure precise interrupt and retrieval order.

The VFPU provides an instruction stream parallel to the CPU instruction stream. It shares the same nine-stage stream architecture as the CPU. However, depending on the vector operation, it may require multiple beats for some more complex instruction execution phases. Each vector instruction is executed by one of the two functional units (VALU0 or VALU1).

Each VALU unit receives one instruction per cycle and sends one result to a floating-point register file. The floating-point addition and subtraction, floating-point multiplication, and floating-point multiplication (subtraction) take three cycles per VALU cell. The format conversion operation between fixed-point and floating-point takes three execution cycles. Floating-point division takes 20/31 cycles depending on the operand, while floating-point square roots take 20/31 cycles depending on the operand. Trigonometric functions take 39 cycles, and other floating-point operations take one cycle.

If two instructions with different execution cycles output results at the same beat in each VALU unit, the instruction with higher priority outputs results in the bus in the order of precedence: multiplication and addition instruction > division root instruction > trigonometric instruction > one cycle instruction > four cycle instruction > three cycle instruction.

2.4. Other Characteristics

In addition, HR DSP employs an out-of-order execution and innovative storage system design to enhance stream efficiency and address the issue of instruction and data correlation arising from the four-issue super-scalar architecture. The out-of-order execution techniques include register renaming, dynamic scheduling, and branch prediction. Register renaming resolves the WAR (write-after-read) and WAW (write-after-write) dependencies to ensure accurate exception recovery and error cancellation. Dynamic scheduling executes instructions based on operand preparation rather than program order, reducing the blocking caused by RAW (read-after-write). Branch prediction reduces blocking due to control dependence by predicting whether a transition instruction will skip.

3. Single Instruction Multiple Data (SIMD)

This section briefly introduces the basic principles of SIMD and elaborates on the commonly used vector data type definitions, vector built-in instructions, macro definitions, and inline assembly functions in the HR DSP vector programming process.

3.1. Basic Principles of SIMD Technology

SIMD is an essential technical feature of high-performance GPPs where one instruction processes multiple data [11]. The parallel processing of multiple units has made SIMD technology a critical aspect of parallel technology. SIMD is a technique that involves bundling and storing the same data type into SIMD vector registers for parallel data processing. The SIMD vector registers can be separated into several distinct units, with the length of the unit being determined by the specific data type stored in the SIMD register. SIMD instructions execute the same operation simultaneously on all independent units in the SIMD register, enabling parallel data processing. The SIMD architecture plays a crucial role in enhancing the application performance by incorporating SIMD extensions onto various processors, providing SIMD vector arithmetic components, vector registers, and the SIMD instruction set. The SIMD vector operation unit organizes multiple logical operation units in a specific manner to achieve parallel processing of multiple data groups through the execution of a single SIMD instruction, effectively leveraging data parallelism in the application program. SIMD extensions are widely adopted as they can substantially

increase the vector computing capability of a processor using only many transistors and are designed to be more cost-effective than dedicated vector computers.

3.2. HR DSP SIMD Programming

Figure 4 demonstrates that HR DSP offers various SIMD data type definitions and pre-built instruction programming models. Employing these vector instructions for SIMD vectorization programming, combined with macro definitions and embedded assembly shown in Figure 5, can significantly enhance the convenience of program vectorization programming. Moreover, it can achieve a better performance acceleration ratio, which is the proportional relationship between the performance after vectorization optimization and the performance of scalar implementation.

```
//Data type definition
typedef uint32_t uint32x8_t __attribute__((vector_size (32)));
typedef int32_t int32x8_t __attribute__((vector_size (32)));
typedef float floatx8_t __attribute__((vector_size (32)));
typedef double doublex4_t __attribute__((vector_size (32)));

typedef floatx8_t vector_float_t;
typedef doublex4_t vector_double_t;
typedef int32x8_t vector_int_t;
typedef uint32x8_t vector_unsigned_int_t;

//Built-in instruction programming model
//Single-precision floating-point multiply-add IPM
__extension__ static __inline floatx8_t __attribute__ ((__always_inline__))
   vmuladdps (floatx8_t s, floatx8_t t, floatx8_t r)
{
 return __builtin_hr_vmuladdps (s, t, r);
}
//Single-precision floating-point multiply-sub IPM
__extension__ static __inline floatx8_t __attribute__ ((__always_inline__))
   vmulsubps (floatx8_t s, floatx8_t t, floatx8_t r)
{
 return __builtin_hr_vmulsubps (s, t, r);
}
//Signed integer data shifted to the right IPM
__extension__ static __inline int32x8_t __attribute__ ((__always_inline__))
   vpalignrw (int32x8_t s, int32x8_t t, int32x8_t r)
{
 return __builtin_hr_vpalignrw (s, t, r);
}
```

Figure 4. SIMD data type definition and built-in instruction programming model.

```
//Macro definition
#define vec_madd_f(A, B, C)   vmuladdps(A, B, C)  //floating-point multiply-add
#define vec_splat0_f(A)       vec_splat_f(A, 0)   //floating-point broadcast 1th element
#define vec_splat1_f(A)       vec_splat_f(A, 1)   //floating-point broadcast 2th element
.....

//Inline assembly
//Signed integer data vector load
INLINE(vector_int_t) vec_lda_i(signed int *data_ptr)
{
  vector_int_t _result;
  __asm__ __volatile__ ("vlddq %0,%1\n\t"
                        : "=Z"(_result)
                        : "m"(*data_ptr));
  return(_result);
}
//Signed integer data vector store
INLINE(void) vec_sta_i(signed int *data_ptr, vector_int_t value)
{
  __asm__ __volatile__ ("vstdq %0,%1\n\t"
                        : /* no output */
                        : "Z"(value), "m"(*data_ptr)
                        : "memory");
  return;
}
```

Figure 5. Macro definition and inline assembly.

Furthermore, HR DSP's vector load/store exclusively supports 32 bytes of address alignment operation. For unaligned addresses, programmers must use instructions such as

broadcast and shift to rearrange vector register data to enable vectographic programming optimization and enhance program performance when addresses cannot be aligned.

4. Basic Principles of SIMD Vectorization Programming

This section introduces the basic principles of vector programming design based on HR DSP, including four parts: optimal SIMD vectorization model, address alignment access characteristics, fully pipelined instructions, and data prefetch. Among them, the section on fully pipelined instructions provides a detailed description of several situations where the instruction pipeline is interrupted.

4.1. Optimal SIMD Vectorization Model

Regarding algorithm ontology, the algorithm vectorization model can be categorized into two types: explicit vectorization model and implicit vectorization model. The explicit vectorization model is illustrated in Figure 6, where the commonly used "for" loop in an application is depicted in Figure 6a. For this type of data, there are no dependencies between the data before and after processing. After optimization, this is shown in Figure 6b. As HR DSP's vector instructions have a width of 256 bits, the example assumes a vectorized loop increment of 4 if the data width is 64 bits. If the data width is 32 bits, then the vectorized loop increment would be 8.

```
for(i=0;i<N;i++){
    Z[i]=X[i]-Y[i];
}
```

(**a**)

```
for(i=0;i<N;i+=4){
    Simd_load(V1,X[i]);
    Simd_load(V2,Y[i]);
    Simd_sub(V1,V2);
}
```

(**b**)

Figure 6. Explicit vectorization model: (**a**) scalar and (**b**) SIMD.

The implicit vectorization model is illustrated in Figure 7a. It appears that the bivariate accumulation algorithm (BSWA) b[i] = a[i] + a[i + 1] cannot be vectorized due to the dependency relationship between the data in the front and back. However, the implementation process of the analysis algorithm is shown in Figure 7b (ignoring the tail corner data). The difference between the adjacent elements of array b is the difference between the adjacent elements of array a, which can be calculated by the vectorization difference to obtain V3. Then, it can be combined with b [0] to form new vectorization data newV3, followed by cumulative vectorization. This process requires programming personnel to convert to achieve vectorization. Furthermore, as shown in Figure 7b, the vectorization is not the optimal model. The two "for" loops in Figure 7b can be merged to reduce one newV3 data load operation, thereby achieving the best algorithm model.

```
for(i=0;i<N-1;i++){
    b[i]=a[i]+a[i+1];
}
```

(**a**)

```
b[0]=a[0]+a[1];
//ignoring the tail corner data
for(i=0;i<(N-2)-(N-2)%4;i+=4){
    Simd_load(V1,a[i+1]);
    Simd_load(V2,a[i]);
    V3=Simd_sub(V1,V2);
}
for(i=0;i<(N-1)-(N-1)%4;i+=4){
    Simd_load(newV3);
    Simd_cumsum(newV3);
}
```

(**b**)

Figure 7. Implicit vectorization model: (**a**) scalar and (**b**) SIMD.

4.2. Address Alignment

The vector register of HR DSP is 256 bits wide and only supports vector load/store operations with 32-byte address alignment (address alignment refers to 32-byte address alignment, otherwise it is address misalignment). Therefore, to achieve the best performance, input and output vector data addresses should be aligned. For row-dominated matrix structures, if the calculation type is independent of the row, the matrix can be treated as a vector. If the calculation type is related to rows, the starting address of each row needs to be aligned with 32 bytes. Therefore, in this type of calculation, not only does the data's starting address need to be aligned with 32 bytes, but the line length must also be a multiple of 8 (for single-precision real floating-point data type) or a multiple of 4 (for single-precision complex floating-point data type) to ensure that the starting address of each line from the second line also satisfies 32-byte alignment.

In addition, for address misalignment, programmers need to use instructions such as broadcast and shift for SIMD vectographic programming design. Address misalignment will result in a slightly reduced performance compared with the address alignment, but it is still significantly better than the non-vectographic optimization performance.

4.3. Instruction Stream

To maximize the instruction stream and improve CPU usage, stream vacuoles should be avoided as much as possible to reduce the CPU idle wait time. The HR DSP instruction stream is interrupted mainly in the following cases:

- The extension of the ALU operation;

As demonstrated in Table 3, some arithmetic logic operations instructions require multiple clock cycles to complete, including most floating-point computation instructions in the vector instructions.

Table 3. Execution cycles of some instructions.

Type of Single-Precision Instructions	Execution Cycles
Floating-point addition	3
Floating-point multiplication addition	3
Floating-point parallel addition	3
Floating-point division	20
Floating-point square root	20
Floating-point sine	39

Instructions with an execution period of 3 cycles in the table still hold the stream, but the dependency degree of instructions before and after is longer, requiring an extension back by 2 clock cycles. N + 2 continuous instructions (assuming that instructions with a single clock cycle need N) are required to maintain the normal flow, while the last instructions in the table, such as division, square root, and sine, interrupt the stream. When the current instruction occupies the ALU unit, the ALU operation can only be performed after the subsequent instruction is calculated. However, the HR DSP vector unit has two VALU units that can be used simultaneously, allowing two instructions to run at the same time.

- The extension of the MEM operation.

If read/write memory variables are not in the level 1 data cache, the MEM operation phase is prolonged, resulting in a broken stream. The prefetch instruction can be used to prefetch data. The prefetch instruction fetches one cache line to the data cache at a time (the size of a cache row in HR DSP is 128 bytes). Therefore, it is required to process 32 data points in a single cycle (the data type determines the size unit of data; for example, 1 point is a real number float/int, 1 complex point is a complex number float/int, and so on). As the internal data access is much slower than the CPU calculation speed, the data after several loops must be prefetched in advance, and the data need to be adjusted according to the calculation time.

- Branch Judgment.

The second half of the clock cycle of the execution of a branch instruction determines the address of the instruction after the next. If the instruction is not in the instruction cache, it will cause the IF operation to be prolonged, interrupting the instruction stream. Therefore, in the code design, we should reduce the large address segment jump operation, reduce the instruction cache miss rate, and avoid long code inside the loop body.

Based on the above considerations, taking floating-point data type as an example, 64 points can be selected to avoid stream interruption and process data in a single cycle. The tail of the vector can be used to complete the remaining data calculation operation. For matrix calculation related to the row, 32 points can be selected, which sacrifices a small amount of flow and reduces the discrete data operation at the end of each row of data.

4.4. Pref Instruction

The data to be computed needed to be prefetched from the DDR to the cache in advance by using the prefetch instructions. The specific number of levels of data to be prefetched in advance needed to be adjusted based on the amount of data processed in a single cycle, computational complexity, and processor core characteristics. As shown in Table 4, the experimental tests showed that the prefetch offsets corresponding to different input parameters and data types were also different. The table shows the single vector, bidirectional quantity, and reference offsets corresponding to different data types, which can be appropriately adjusted in the design of the actual algorithm. Additionally, the prefetch offset in the table is given under the condition that the amount of data processed in a single loop is 32 points. The basic law of data prefetch is that the more data are processed, the larger the prefetch offset is relative to the base address. The more complex the calculation, the smaller the prefetch offset relative to the base.

Table 4. Prefetch offsets.

Source Operand	Data Type	Offset (byte)
Single Vector	Real Vector	768
	Complex Vector	1024, 1152
Double Vector	Real Vector 1	512
	Real Vector 2	640
	Complex Vector 1	768, 896
	Complex Vector 2	1024, 1152

5. Implementation of Typical SIMD Algorithms

To compare and analyze the performance improvement of the SIMD algorithm before and after vectorization and to verify the effectiveness and universality of HR DSP SIMD vectorization programming design principles, three algorithms, matrix–matrix multiplication, mean filter, and accumulation, were selected, as shown in Table 5. The reasons for choosing these three algorithms were as follows. Matrix–matrix multiplication is a typical representative of linear algebra computation and is a computationally intensive algorithm. It is widely used in mathematical libraries for linear algebra, such as GotoBLAS [12] and ATLAS [13], and has been the subject of research for automatic tuning based on Intel AVX-512 extension instructions [14]. It also has different vectorization models, making it conducive to comparing and analyzing the performance differences between ordinary vectorization models and optimal vectorization models. Mean filter is a typical algorithm in the field of image processing, which can better reflect the impact of cache and SIMD vectorization on the performance by reading discontinuous data for calculation and storing it in continuous memory. Accumulation algorithm is a typical example of implicit vectorization, which can effectively avoid repeated calculations to improve algorithm performance.

Table 5. Typical algorithms.

Algorithm Name	Scalar Realization

Matrix Multiplication	for (i = 0; i < n; i++){
	for (j = 0; j < m; j++){
	R[i][j] = 0.0;
	for (k = 0; k < p; j++)
	R[i][j] += A[i][k] * B[k][j];
	}
	}

Mean Filter	for (i = 1; i < m − 1; i++){
	for (j = 1; j < n − 1; j++){
	R[i][j] = (A[I − 1][j] + A[i][j − 1] + A[i][j] + A[i][j + 1] + A[i + 1][j])/5;
	}
	}

Accumulation	for (i = 0; i < len; i++){
	R[i] = A[i + 0] + A[i + 1] + A[i + 2] + A[i + 3] + A[i + 4] + A[i + 5] + A[i + 6] + A[i + 7];
	}

5.1. Matrix–Matrix Multiplication

As shown in Figure 8, taking 8 × 8 matrix–matrix multiplication as an example (other scale matrices take 8 × 8 or 4 × 4 a block matrix for cyclic processing), the function of matrix–matrix multiplication R = A × B is to multiply in a row of matrix A and a column of corresponding elements of matrix B, and finally add the result into the position of the first element of matrix R, and so on, to obtain the whole result matrix R.

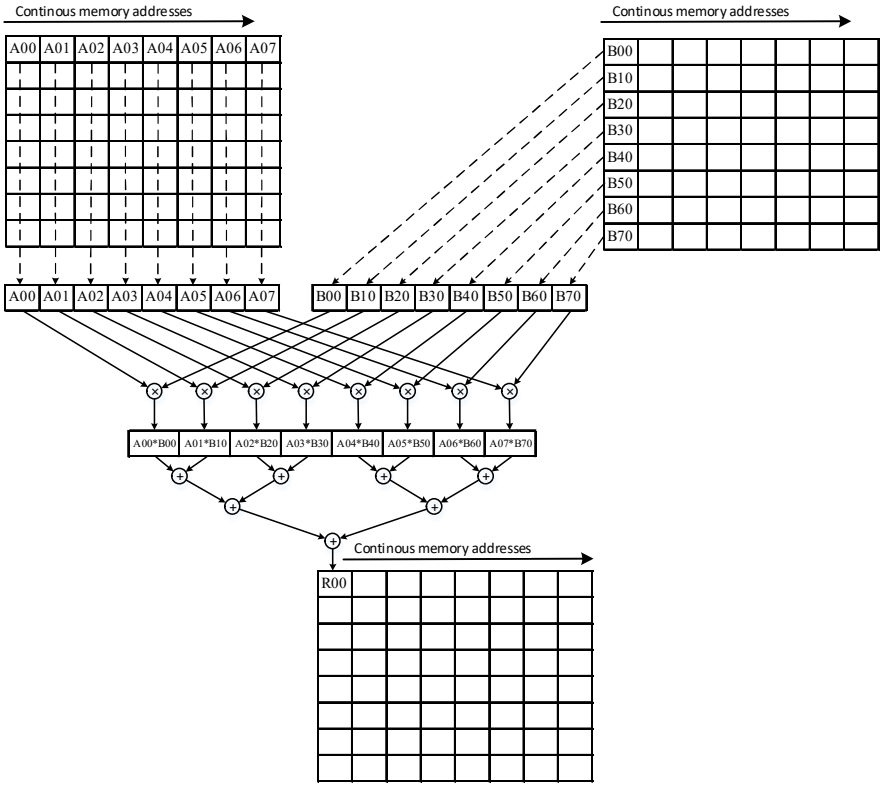

Figure 8. Matrix–matrix multiplication.

There are two common vectorization models for realizing matrix–matrix multiplication. The first is the ordinary vectorization model, which first transposes matrix B, and then multiplies it with the corresponding element of matrix A. Finally, the result is obtained through summation, as shown in Figure 9. The second is the optimal vectorization model, which multiplies every element in each row of matrix A with the corresponding row of matrix B, and then adds it and stores it to the resulting matrix R, as shown in Figure 10.

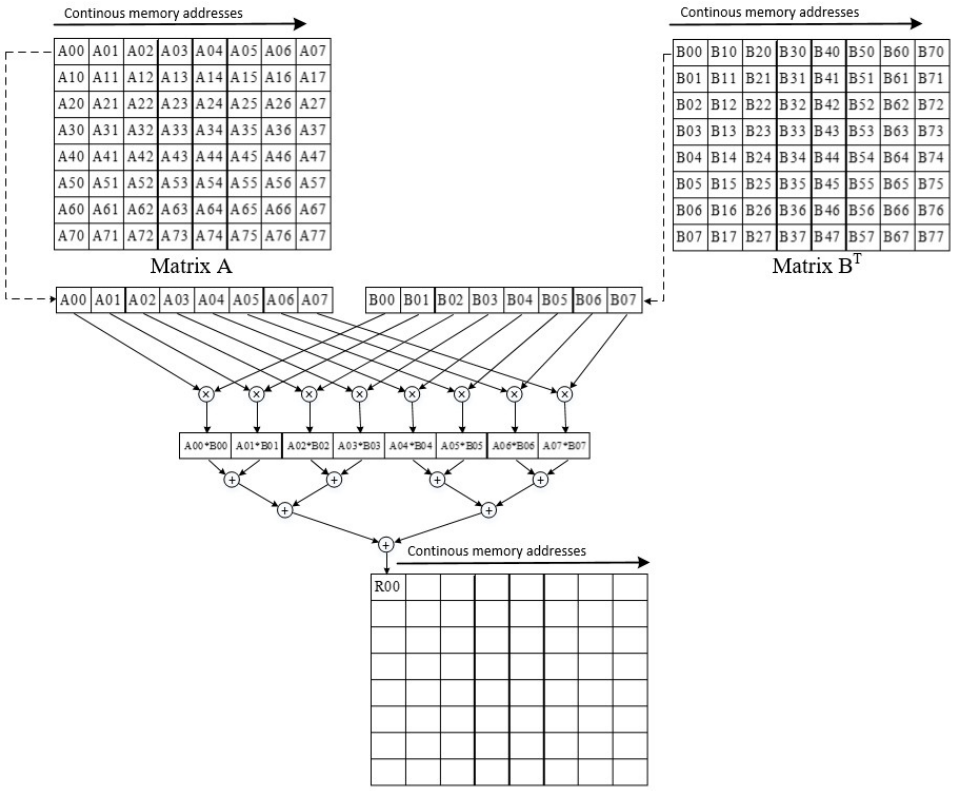

Figure 9. Common vectorization model.

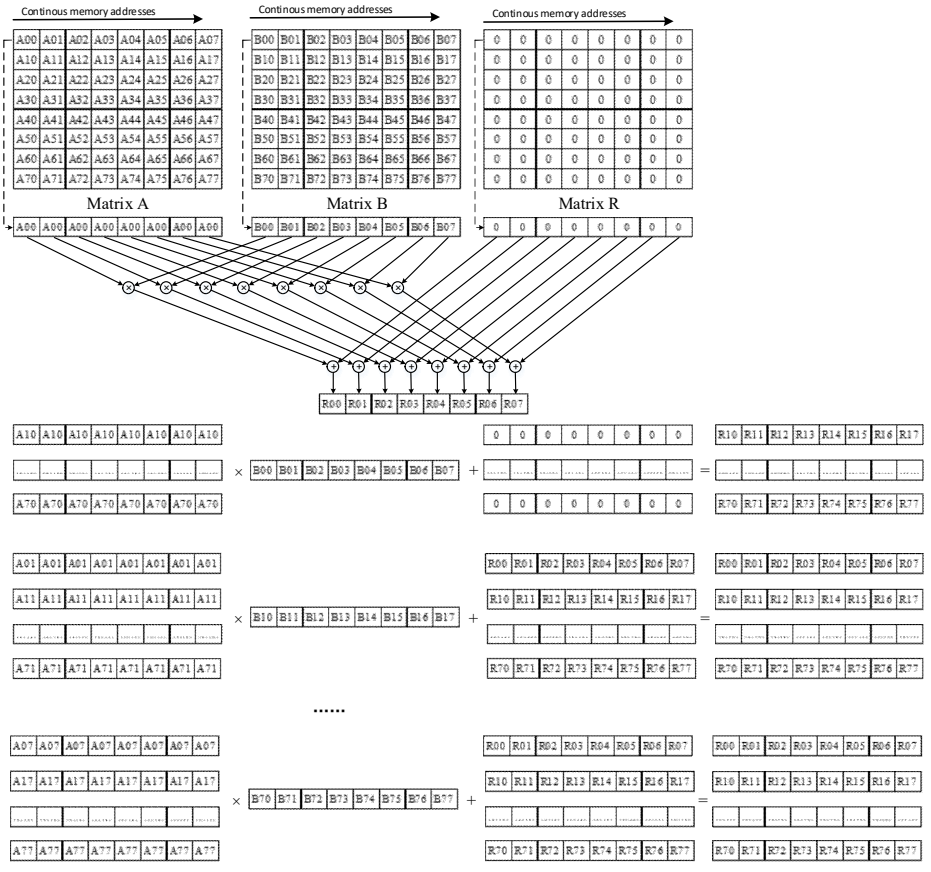

Figure 10. Optimal vectorization model.

Compared with the ordinary vectorization model, the advantage of the optimal vectorization model is that it does not require transpose matrix B. Instead, it multiplies the first element of each row of matrix A with the first row of matrix B and adds the corresponding element value of the current resulting matrix R (the initialized elements are all zero). Then, it multiplies the second element of each row of matrix A by the second element of the second row of matrix B and adds the corresponding element value of the current result matrix R, and so on, until the last element of each row of matrix A is multiplied by the last element of the last row of matrix B, and it adds the corresponding element value of the current result matrix R to obtain the final result matrix R. The partial implementation of the core vectorization code is shown in Figure 11. The purpose of the "multilayer pie" approach is to avoid reading matrix B from memory, transposing it, and then storing it in memory.

```
void mprod_f(float *A, float *B, float *R, int m, int n, int p){
    ……
    vec0R0 = vec_madd_f(vec_splat0_f(vec0A0), vec0B0, vec0R0);
    vec1R0 = vec_madd_f(vec_splat0_f(vec1A0), vec0B0, vec1R0);
    vec2R0 = vec_madd_f(vec_splat0_f(vec2A0), vec0B0, vec2R0);
    vec3R0 = vec_madd_f(vec_splat0_f(vec3A0), vec0B0, vec3R0);
    vec4R0 = vec_madd_f(vec_splat0_f(vec4A0), vec0B0, vec4R0);
    vec5R0 = vec_madd_f(vec_splat0_f(vec5A0), vec0B0, vec5R0);
    vec6R0 = vec_madd_f(vec_splat0_f(vec6A0), vec0B0, vec6R0);
    vec7R0 = vec_madd_f(vec_splat0_f(vec7A0), vec0B0, vec7R0);
    ……
    vec0R0 = vec_madd_f(vec_splat7_f(vec0A0), vec7B0, vec0R0);
    vec1R0 = vec_madd_f(vec_splat7_f(vec1A0), vec7B0, vec1R0);
    vec2R0 = vec_madd_f(vec_splat7_f(vec2A0), vec7B0, vec2R0);
    vec3R0 = vec_madd_f(vec_splat7_f(vec3A0), vec7B0, vec3R0);
    vec4R0 = vec_madd_f(vec_splat7_f(vec4A0), vec7B0, vec4R0);
    vec5R0 = vec_madd_f(vec_splat7_f(vec5A0), vec7B0, vec5R0);
    vec6R0 = vec_madd_f(vec_splat7_f(vec6A0), vec7B0, vec6R0);
    vec7R0 = vec_madd_f(vec_splat7_f(vec7A0), vec7B0, vec7R0);
    ……
}
/*************************************************
 *  Note:
 *  vec_madd_f：floating-point multiply-addition.
 *  vec_splat0_f：broadcast 1th element of the vector.
 *  ……
 *  vec_splat7_f：broadcast 8th element of the vector.
 *************************************************/
```

Figure 11. Partial implementation of the optimal vectorization model core code.

5.2. Mean Filter

In the field of image processing, the mean filter algorithm is widely used. A common 5-point mean filter algorithm is introduced below. As shown in Figure 12, the function of the 5-point mean filter algorithm is to reduce one layer inward of the m × n matrix (i.e., the second row to the (m − 1) row, the second column to the (n − 1) column) of all the elements corresponding to the upper, lower, left, right, and the element itself as the result of the mean value stored in the location of the element.

During the calculation process, the algorithm requires reading discontinuous data for calculation and storing it in continuous memory. Compared with single-point scalar processing, SIMD vectorization can efficiently utilize the parallel data processing capability of HR DSP. The core vectorization code is partially implemented as shown in Figure 13.

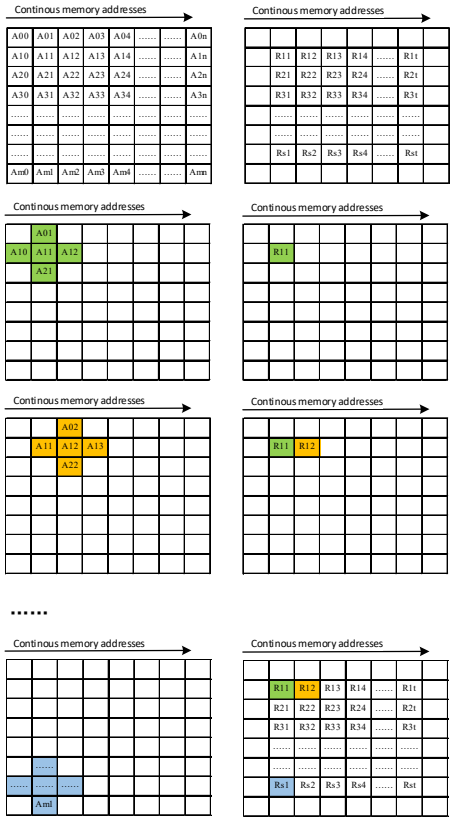

Figure 12. Implementation of the 5-point mean filter (scalar).

```
void vmeanfilter_i(int *A, int *B, int *C, int *R, float coe, int n){
    vector_float_t veccoe = vec_vldw_f(&coe);
    .....
    tmpvecA1 = vec_lda_i(vA0 + offsetA);
    tmpvecB1 = vec_lda_i(vB0 + offsetA);
    tmpvecB11 = vec_lda_i(vB01 + offsetA);
    tmpvecB12 = vec_lda_i(vB02 + offsetA);
    tmpvecC1 = vec_lda_i(vC0 + offsetA);

    vecA0 = vpalignrw(tmpvecA1,tmpvecA0,(vector_int_t)vec_offset_A);
    vecB0 = vpalignrw(tmpvecB1,tmpvecB0,(vector_int_t)vec_offset_B);
    vecB01 = vpalignrw(tmpvecB11,tmpvecB01,(vector_int_t)vec_offset_B1);
    vecB02 = vpalignrw(tmpvecB12,tmpvecB02,(vector_int_t)vec_offset_B2);
    vecC0 = vpalignrw(tmpvecC1,tmpvecC0,(vector_int_t)vec_offset_C);

    vecA0 = vec_add_i(vecA0,vecB0);
    vecB0 = vec_add_i(vecB01,vecB02);
    vecA0 = vec_add_i(vecA0,vecB0);
    vecC0 = vec_add_i(vecA0,vecC0);
    tmp0 = vec_i_f(vecC0);
    tmp0 = vec_mul_f(tmp0,veccoe);
    tmpvecR1 = vec_f_i(tmp0);
    vecR0 = vpalignrw(tmpvecR1,tmpvecR0,(vector_int_t)vec_offset_R);

    vec_sta_i(voutR0 + offset, vecR0);
    .....
}
/*************************************************************
 *  Note:
 *  vec_vldw_f：read a 32-bit data broadcast to a floating-point vector
register.
 *  vec_lda_i：read a 256-bit data from memory to a interge vector register, a
256-bit alignment.
 *  vpalignrw：get the result according to register join right shift right
alignment.
 *  vec_add_i：integer vector addition output result.
 *  vec_i_f：integer vector to floating-point vector.
 *  vec_mul_f：floating-point vector multiplication output result.
 *  vec_f_i：floating-point vector to integer vector.
 *  vec_sta_i：save a 256-bit data into memory, a 256-bit alignment.
 *************************************************************/
```

Figure 13. Partial implementation of the 5-point mean filter SIMD vectorization core code.

5.3. Accumulation

The accumulation algorithm is one of the commonly used algorithms in applications. An 8-point accumulation algorithm is introduced below, as shown in Figure 14. Its function is to obtain the result by summing every eight points from left to right.

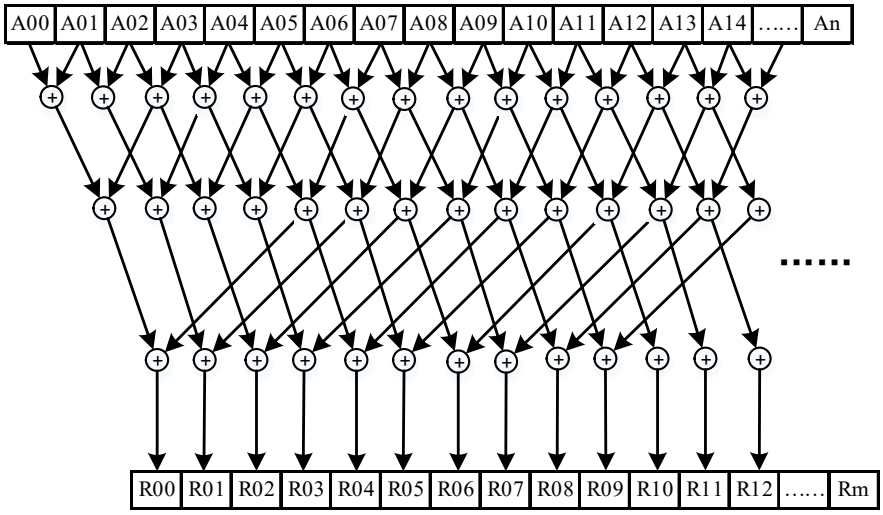

Figure 14. Implementation of the 8-point accumulation algorithm (scalar).

On the surface, the accumulation algorithm R[i] = A[i] + A[i + 1] + … + A[i + 7] seems to imply that the data before and after were associated and could not be vectorized. However, the accumulation algorithm has a hidden feature: the next accumulation result is the previous accumulation result plus the source data sliding into the new data and minus the old data sliding out, namely R[i + 1] = R[i] + A[i + 8] − A[i]. This is a typical implicit vectorization algorithm.

Based on this feature, the 8-point sum can be calculated and stored in the result address. The data source is staggered by 8 points to calculate the accumulated difference value and stored from the second point of the result address. Finally, the accumulation and summation from the first address of the result address can obtain the accumulated result. The partial implementation of the core vectorization code is shown in Figure 15.

```
void vacumsum(int *A,int *R,int n){
    tmp = 0;
    for(j=0;j<8;j++){
        tmp += A[j];
    }
    R[0] = tmp;

    vsub_i(A+8, A, R+1, n-8);
    vcumsum_i(R, n-8+1);
}

/*****************************
 *  Note:
 *  vec_lda_i：read a 256-bit data from
memory to a interge vector register, a
256-bit alignment.
 *  vec_cumsum_i：integer vector
element cumulative sum.
 *  vec_sta_i：save a 256-bit data into
memory, a 256-bit alignment.
 *****************************/
```

```
void vcumsum_i(int *R, int n){
    ……
    vecA0 = vec_lda_i(vA0 + offset);
    vecA1 = vec_lda_i(vA1 + offset);
    ……
    vecA6 = vec_lda_i(vA6 + offset);
    vecA7 = vec_lda_i(vA7 + offset);

    vec_cumsum_i(vecA0,sum_val);
    vecR0 = sum_val;
    vec_cumsum_i(vecA1,sum_val);
    vecR1 = sum_val;
    vec_cumsum_i(vecA2,sum_val);
    vecR2 = sum_val;
    ……
    vec_cumsum_i(vecA6,sum_val);
    vecR6 = sum_val;
    vec_cumsum_i(vecA7,sum_val);
    vecR7 = sum_val;

    vec_sta_i(voutR0 + offset, vecR0);
    vec_sta_i(voutR1 + offset, vecR1);
    ……
    vec_sta_i(voutR6 + offset, vecR6);
    vec_sta_i(voutR7 + offset, vecR7);
    ……
}
```

Figure 15. Partial implementation of the 8-point accumulation algorithm SIMD vectorization core code.

The above vectorization model is relatively easy to implement. Additionally, there is another vectorization model that merges vsub and vcumsum, which can reduce the process of one-time data load and one-time store, and better improve the vectorization level and performance. However, it also introduces the problem of address misalignment. Therefore, it is necessary to accurately shift the data during design and avoid stepping on the data (step out the legitimate data in memory when storing data after calculation).

6. Experimental Evaluation

All of the experiments presented in this paper were performed using the HR DSP and FT platform, and the specifications of the experimental environment are listed in Table 6 (if the following charts and explanations are not explicitly stated, they are assumed to be HR by default).

Table 6. Platform specification.

Type	Specification	
DSP/CPU	HR (800 MHz)	FT2000+ (2.2 GHz)
DDR	800 MHz	2666 MHz
Vector register width	256 bits	128 bits
Cache line size	128 B	64 B
L1 cache size	L1 Icache-32 KB L1 Dcache-32 KB	L1 Icache-32 KB L1 Dcache-32 KB
L2 cache size	1 MB	2 MB (4 cores sharing)
Compiler	GCC 4.7.2	GCC 9.3.0
Optimization	O2	O2
Architecture	MIPS	ARM

The experimental platform's demo board is displayed in Figure 16, which includes power supply, serial port, network port, DDR, HR DSP, and other components.

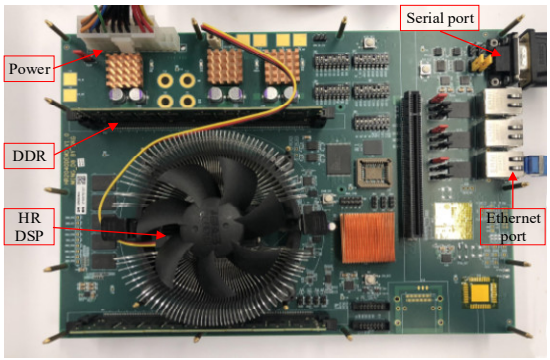

Figure 16. Experimental platform.

6.1. Matrix–Matrix Multiplication

Figure 17 displays the performance bar charts of different models, from which it is evident that the performance time of the matrix–matrix multiplication algorithm followed the order of the scalar > ordinary vectorization model > optimal vectorization model. Furthermore, when the matrix size was 256 × 256, the performance time of scalar implementation was much longer than that of the vectorization model.

Figure 18 depicts the time consumption ratio between scalar implementation of the matrix–matrix multiplication algorithm and the ordinary vectorization mode. It can be observed that when the matrix size was 4 K points (the matrix size was 64 × 64) or below, the time consumption of scalar implementation was approximately nine times that of the ordinary vectorization model. However, when the matrix size was 16 K points (the matrix size was 128 × 128) or above, the time consumption of scalar implementation was approximately 13 times that of the ordinary vectorization model.

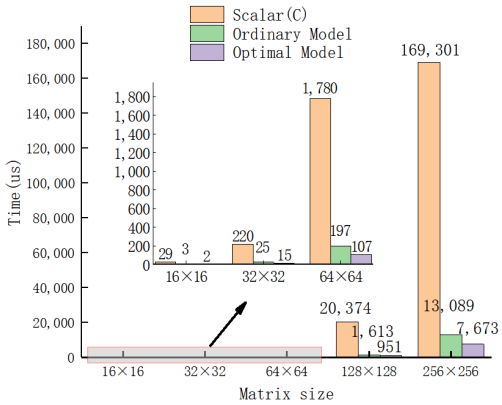

Figure 17. Time-consumption of different models.

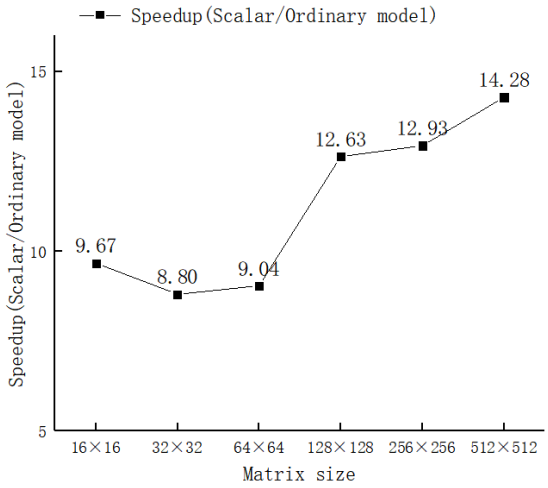

Figure 18. Time consumption ratio between the scalar implementation and common vectorization models.

This phenomenon can be attributed to the following: when the matrix size was 4 K or below, the data amount calculated by the matrix–matrix multiplication algorithm was 4(K) × 3(number of matrices) × 4(Byte) = 48 KB. In this case, partial cache replacement occurred between L1Dcache (32 KB) and L2cache (1 MB), and the prefetch operation started to take effect. As the matrix size increased, cache replacement became more frequent, and the data prefetch function became more effective. Consequently, the maximum performance time ratio was increased by approximately 14 times. However, when the matrix size decreased and became even smaller than the L1Dcache, no cache replacement occurred, and the prefetch instruction had no effect, but increased the instruction execution overhead. As a result, the performance time ratio decreased to about 8.8 times.

Figure 19 presents a dot plot, where it was evident that the scalar performance was approximately 14–22 times that of the optimal vectorization model. The overall trend of the dot plot was similar to that of the ordinary vectorization model, but when the matrix size was 512 × 512, the time consumption ratio decreased. This phenomenon can be attributed to the fact that when the matrix size reached a certain scale, the distance between matrix rows and elements between rows increased, resulting in an increased cache replacement frequency, which increased the cache miss rate and decreased the performance ratio. As shown in Figure 19, by comparing the speedup of the optimal vectorization model of HR and FT matrix multiplication, it can be seen that under the same conditions, the vectorization acceleration ratio of HR DSP was 14–22, while that of FT was 6–13, indicating that HR DSP has a significant advantage.

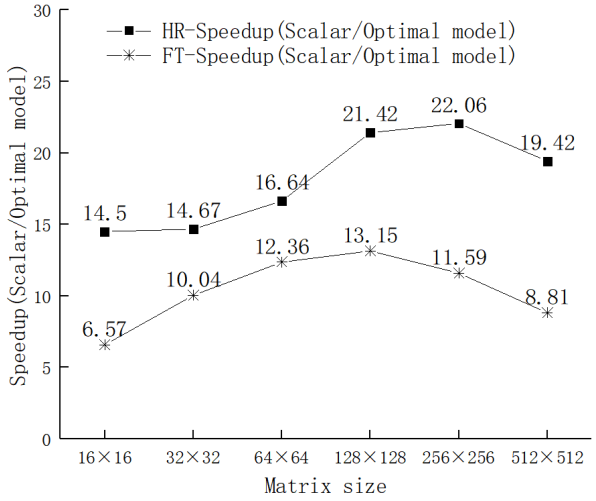

Figure 19. Time consumption ratio of the scalar realization and optimal vectorization models.

Figure 20 displays that the performance of the ordinary vectorization model was approximately 1.3–1.8 times that of the optimal vectorization model, and the performance of the time consuming ratio trend chart was consistent with that shown in Figures 18 and 19.

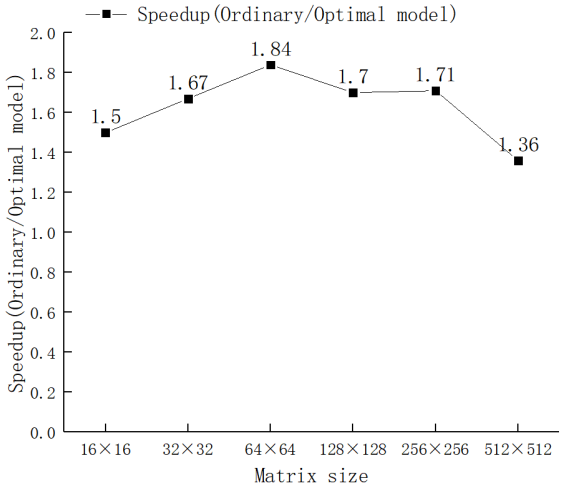

Figure 20. Time consumption of the common vectorization model and optimal vectorization model.

6.2. Mean Filter

Figure 21 illustrates a bar chart where it was evident that the implementation performance of the scalar model took much longer than that of the vectorization model, and both of them changed linearly with the increase in data amount.

Figure 22 presents a dot plot where it can be observed that when the matrix size was 64 K (the matrix size was 256 × 256) and below, the performance time showed a linear change rule, and there was a turning point between 64 K points (the matrix size was 256 × 256) and 256 K (the matrix size was 512 × 512). This was because cache replacement did not occur when the calculated data amount was less than L1Dcache (32 KB) at 64 K points and below, and partial cache replacement started to occur when the calculated data amount was between L1Dcache (32 KB) and L2cache (1 MB) at 256 K and above, and the prefetch function started to take effect. As a result, the time consumption ratio increased significantly. As shown in Figure 22, by comparing the speedup of the optimal vectorization model of HR and FT mean filtering, it can be seen that under the same conditions, FT had a certain advantage in vectorization speedup for sizes of 256 × 256 or less, while HR DSP had a significant advantage in vectorization speedup for sizes of 512 × 512 or more.

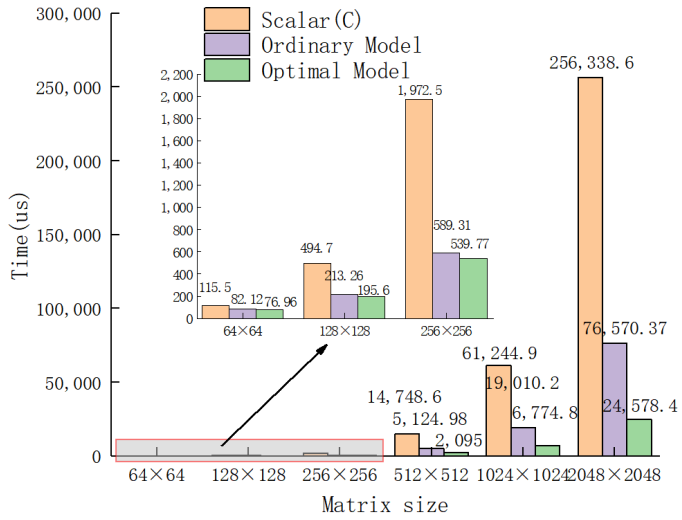

Figure 21. Time consumption of the different models in addressing.

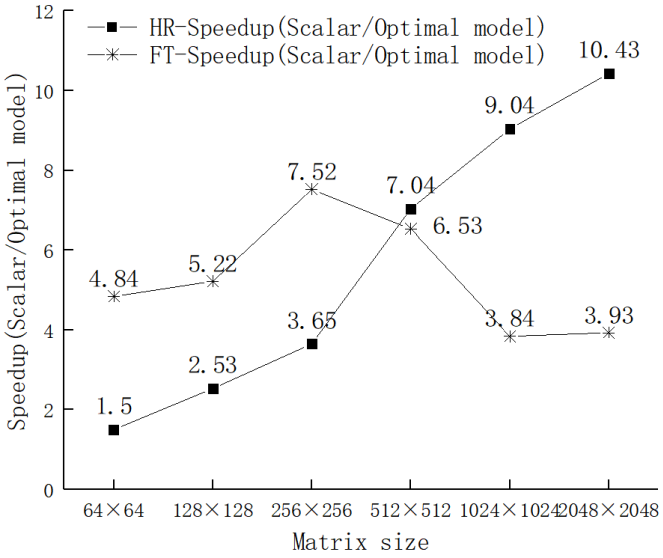

Figure 22. Time consumption ratio of the scalar realization and optimal vectorization models in addressing.

6.3. Accumulation

Figure 23 shows a bar chart where it was evident that the performance of the vectorization model at 64 K and below was equivalent to that of the instruction stream and was not affected by the instruction to prefetch or not. The performance time of the vectorization model at 128 K and above, without instruction prefetch and with instruction prefetch, was significantly longer than that of the quantization model with instruction stream and instruction prefetch.

Figure 24 displays a dot plot where it can be observed that when the data length was 32 K or above, the performance time ratio of the scalar implementation and vectorization model was slightly greater than that below 32 K. As shown in Figure 24, by comparing the acceleration ratio of the optimal vectorization model of the HR and FT accumulation algorithm, it can be seen that under the same conditions, FT had a significant advantage in vectorization acceleration ratio for sizes of 128 K or less, while HR DSP had a certain advantage in vectorization acceleration ratio for sizes of 512 K.

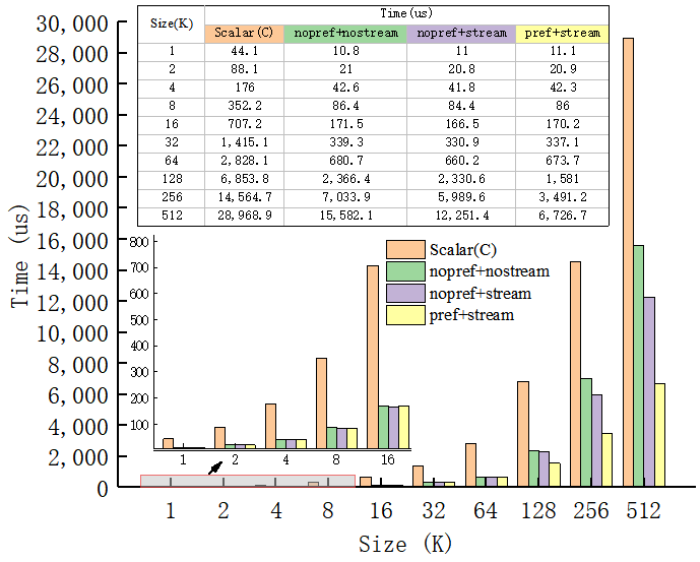

Figure 23. Time consumption of the different models in addressing.

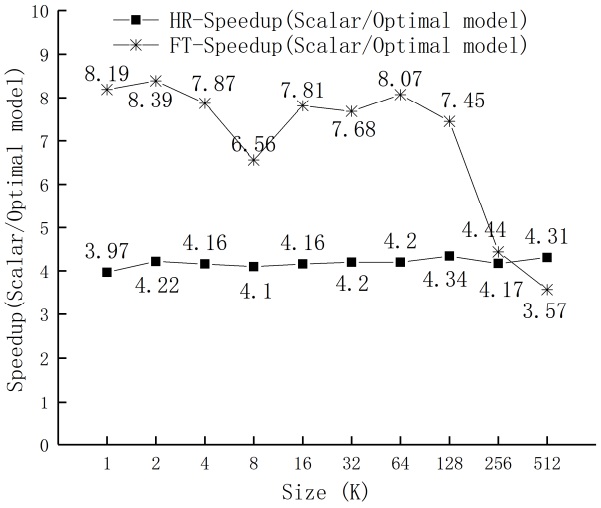

Figure 24. Time consumption ratio between the scalar realization and optimal vectorization models in addressing.

7. Conclusions

This paper provided an overview of SIMD stream technology and discussed the development of the extended instruction set. It also provided a detailed description of the structure characteristics and instruction stream of HR DSP. Furthermore, the paper introduced the basic principles of vectorization optimization design based on HR DSP. Finally, the vectorization performance comparison of the three typical algorithms on different platforms demonstrated the advantages of the HR DSP vectorization process.

The experimental results demonstrated that the performance of the three algorithms optimized according to SIMD vectorization programming was improved by about 4–22 times compared with the scalar without vectorization, instruction pipelining, and data prefetch. Additionally, the optimal vectorization model of matrix–matrix multiplication was improved by approximately 84% compared with the ordinary vectorization model after optimization.

Moreover, the paper suggests that for small point algorithms (the data size smaller than L1Dcache size), it was better to avoid using the pref instruction for data prefetch. This is because the data amount was already in the cache and no additional instruction

was needed to operate the data. Improper use of the data prefetch instructions would destroy the automatic cache and reduce the cache hit ratio, thereby affecting the algorithm performance.

Author Contributions: Conceptualization, H.W. and C.X.; methodology, C.X.; software, C.X.; validation, C.X., H.W. and J.Z.; formal analysis, C.X.; investigation, H.W.; resources, H.W.; data curation, C.X.; writing—original draft preparation, C.X.; writing—review and editing, H.W.; visualization, J.Z.; supervision, H.W.; project administration, H.W.; funding acquisition, H.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

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

References

1. Amiri, H.; Shahbahrami, A. SIMD programming using Intel vector extensions. *J. Parallel Distrib. Comput.* **2020**, *135*, 83–100. [CrossRef]
2. Peleg, A.; Weiser, U. MMX technology extension to the intel architecture. *IEEE Micro* **1996**, *16*, 42–50. [CrossRef]
3. Slingerland, N.T.; Smith, A.J. *Multimedia Instruction Sets for General Purpose Microprocessors: A Survey*; Report No. UCB/CSD-00-1124; Computer Science Division, University of California: Berkeley, CA, USA, 2000.
4. Oberman, S.; Favor, G.; Weber, F. AMD 3DNow! technology: Architecture and implementations. *IEEE Micro* **1999**, *19*, 37–48. [CrossRef]
5. Carlson, D.A.; Castelino, R.W.; Mueller, R.O. Mueller, Multimedia Extensions for a 550-MHz RISC Microprocessor. *IEEE J. Solid-State Circuits* **1997**, *32*, 1618–1624. [CrossRef]
6. Diefendorff, K.; Dubey, P.; Hochsprung, R.; Scale, H. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro* **2000**, *20*, 85–95. [CrossRef]
7. Lindoso, A.; Valderas, M.G.; Entrena, L. Analysis of neutron sensitivity and data-flow error detection in ARM microprocessors using NEON SIMD extensions. *Microelectron. Reliab.* **2019**, *100–101*, 113346. [CrossRef]
8. Effective DSP Programming Using MIPS® DSP Application Specific Extensions, Document No. MD00475. June 2008. Available online: https://s3-eu-west-1.amazonaws.com/downloads-mips/mipsdocumentation/loginrequired/eEffective_programming_of_the_24ke_and_the_34k_core_families_for_dsp_code.pdf (accessed on 16 May 2021).
9. Intel Corporation. *Intel SSE4 Programming Reference (D91561-003)*; Intel Corporation: Santa Clara, CA, USA, 2007; Available online: https://www.intel.com/content/dam/develop/external/us/en/documents/18187-d9156103.pdf (accessed on 7 October 2022).
10. Intel Corporation. *Intel Advanced Vector Extensions Programming Reference (319433-011)*; Intel Corporation: Santa Clara, CA, USA, June 2011.
11. Arslan, M.A.; Gruian, F.; Kuchcinski, K.; Karlsson, A. Code generation for a SIMD architecture with custom memory organization. In Proceedings of the 2016 Conference Design and Architectures for Signal and Image Processing (DASIP), Rennes, France, 12–14 October 2016; IEEE: New York, NY, USA, 2016; pp. 90–97. [CrossRef]
12. Goto, K. Goto Blas. Available online: http://www.tacc.utexas.edu/resources/software/ (accessed on 21 June 2022).
13. Whaley, R.C.; Petitet, A.; Dongarra, J.J. Automated empirical optimizations of software and the atlas project. *Parallel Comput.* **2001**, *27*, 3–35. [CrossRef]
14. Kim, R.; Choi, J.; Lee, M. Optimizing parallel GEMM routines using auto-tuning with Intel AVX-512. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2019), Guangzhou, China, 14–16 January 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 101–110. [CrossRef]