CCF Transactions on High Performance Computing
https://doi.org/10.1007/s42514-023-00175-7

REGULAR PAPER

thSORT: an efficient parallel sorting algorithm on multi-core DSPs

Mouzhi Yang[1] · Peng Zhang[2] · Jianbin Fang[2] · Weifeng Liu[1] · Chun Huang[2]

Received: 24 May 2023 / Accepted: 24 November 2023

Abstract
Multi-core architecture has become the main trend in high performance computing (HPC) because of its powerful parallel computing capability. Due to energy efficiency constraints, energy-efficient multi-core digital signal processors (DSPs) have become an alternative architecture in HPC systems. FT-M7032 is a CPU-DSP heterogeneous processor that integrates 16 CPU cores for running operating systems and four multi-core general purpose DSP (GPDSP) clusters for providing high performance. Sorting is a fundamental operation in computer science with numerous applications and has been studied extensively, but high-performance parallel sorting algorithms are typically architecture-specific. To our knowledge, little attention has been paid to optimizing the sorting on the low-power multicore DSPs. In this paper, we propose thSORT, an efficient bitonic sorting algorithm for FT-M7032. Our algorithm consists of two parts: single-core DSP sorting and multi-core DSP sorting, both aiming to tap the features of FT-M7032. We implement a vector micro-kernel for bitonic sort and propose a multi-level algorithm to merge the results of the micro-kernel. When compared to the CPU baseline, our implementation is 1.43× faster than the parallel sorting of the Boost C++ Libraries, and is 2.15× faster than std::sort.

Keywords Bitonic sorting network · Multi-core DSPs · Parallel sorting

1 Introduction

Sorting algorithms play a fundamental role in numerous computer applications. In the era of big data, organizations deal with massive datasets that require efficient sorting techniques. Parallel sorting algorithms allow data analytics platforms to process and sort large volumes of data faster (Graefe 2006). Search engines deal with indexing and retrieving vast amounts of information from the web. Parallel sorting enhances the efficiency of indexing and searching operations. Parallel sorting in data mining accelerates tasks like clustering, association rule mining, and outlier detection by efficiently organizing and processing large datasets, enhancing the speed and scalability of data-driven insights. The efficiency of sorting algorithms directly affects the overall performance of applications. This has pushed academia to spend efforts to provide efficient sorting algorithms on new architectures.

Due to the prevalence of multi-core computers, parallel algorithms have experienced substantial improvements in terms of performance and scalability. Parallel sorting algorithms can effectively utilize the computing power of multiple cores or nodes to perform sorting operations in parallel. There are numerous high-performance parallel sorting methods that leverage a variety of hardware, including multi-core CPUs (Chhugani et al. 2008; Hou et al. 2018), GPUs (Stehle and Jacobsen 2017; Satish et al. 2009), and FPGAs (Jun et al. 2017). There are significant challenges in developing parallel algorithms to fully utilize the performance of the hardware.

Due to energy efficiency constraints, energy-efficient multi-core digital signal processors (DSPs) have become an alternative option in HPC systems. Unlike CPUs or

✉ Peng Zhang
zhangpeng13a@nudt.edu.cn

✉ Weifeng Liu
weifeng.liu@cup.edu.cn

Mouzhi Yang
mouzhi.yang@student.cup.edu.cn

Jianbin Fang
j.fang@nudt.edu.cn

Chun Huang
chunhuang@nudt.edu.cn

1 Super Scientific Software Laboratory, China University of Petroleum-Bejing, Beijing 102249, China

2 College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China

GPUs, DSPs typically feature Very Long Instruction Word (VLIW) or vector cores without out-of-order execution. Moreover, DSPs work on on-chip memory and integrate Direct Memory Access (DMA) engines for data transmission between on-chip memory and off-chip main memory. Therefore, many optimizations used for CPU and GPU may be inapplicable.

Traditional DSP processors, constrained by their small word lengths and low computational precision, are also incapable of effectively supporting scientific computations involving large datasets. To solve this problem, FT-M7032, one of the prototype CPU-DSPs heterogeneous processors explored by the National University of Defense Technology, is proposed to accomplish scientific computations by means of DSPs. FT-M7032 integrates 16 ARM CPU cores for running operating systems and four multi-core general purpose DSP (GPDSP) clusters for high computing performance (Yin et al. 2022), and it has a hierarchical software stack, including compilers, high-performance math libraries, `hthreads` (a heterogeneous threading model), and standard parallel programming interface as detailed in Sect. 3.

As far as we know, there is no study on how to best optimize sort on these emerging multi-core DSP processors.

To bridge this gap, we present thSORT, a first efficient parallel bitonic sorting algorithm for FT-M7032. Our algorithm consists of two parts: single-core DSP sorting and multi-core DSP sorting, both aiming to take advantage of the features of FT-M7032. The main contributions of this paper are as follows:

- We implement a vectorized bitonic sort micro-kernel to sort small arrays.
- We divide our algorithm into two parts: single-core DSP sorting and multi-core DSP sorting, which facilitates its portability and scalability.
- We propose a multi-level merge algorithm consisting of vectorized merge and merge and Merge Path (Green et al. 2012).
- We model the proposed algorithm's performance and use the performance model to guide the optimization.
- We achieve a 1.43× speedup against the parallel sort of Boost C++ Libraries, both of which use 8 cores (CPU or DSP), and a 2.15× speedup against std::sort.

The rest of the paper is organized as follow: Section 2 introduces bitonic sort and related work. Section 3 explains the architecture of the FT-M7032 platform. Section 4 details our thSORT algorithm. Section 5 presents the performance model. Section 6 gives the experimental results. Section 7 concludes the paper.

2 Background

2.1 Bitonic sort and sorting network

Parallel sorting algorithms are an important research area in sorting algorithms due to the ubiquity of multi-core processors. Bitonic sort is a widely used parallel sorting algorithm based on sorting networks and was first proposed by Batcher (1968). Since then, various modifications and extensions have been proposed by researchers to improve the efficiency and scalability (Guo et al. 2018; Nassimi 1979).

In computer science, a sorting network is an abstract description of how to sort a fixed-length array. The network can be represented graphically as a timeline in which the values of the elements are passed from left to right while being swapped vertically at the corresponding positions. Figure 1 shows a bitonic sorting network when 16 elements are sorted. The algorithm complexity of the sorting network

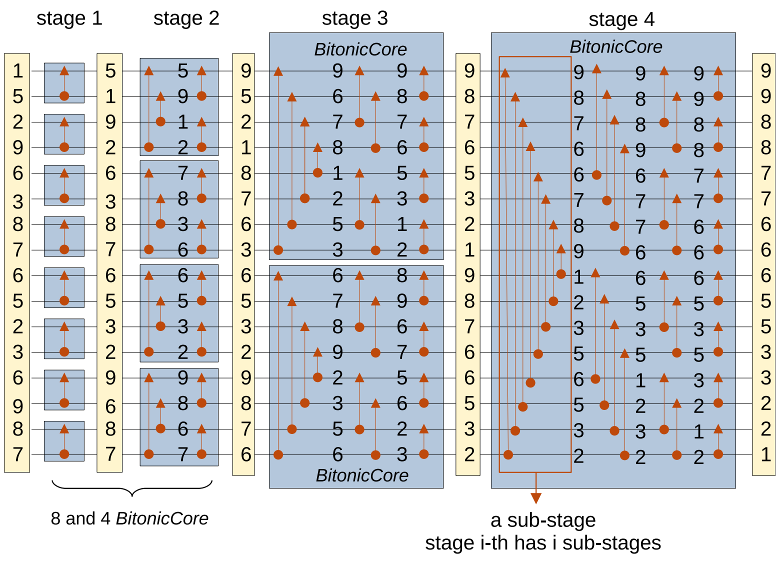


Fig. 1 Bitonic sorting network examples. Arrows represent test & exchange operation

is $O(nlog_2n)$. There are two strategies commonly used to implement sorting network. One is the hard-code, which is achieved by directly mapping the operations of the sorting network. Another one is using flexible algorithms to perform the same operations in the sorting network. We choose the first strategy, as the second is not convenient to achieve vectorization.

Algorithm 1 Original bitonic sort algorithm

```
Input: array: an array to sort. length: the size of array
Output: array: the array sorted.
Function bitonicSort(array, n)
    for $s = 2;\ s \leq n;\ s = s * 2$ do
        // Each value of $s$ corresponds to a stage, and there
        are log(n) stages in total.
        for $i = 0;\ i < n;\ i = i + s$ do
            $BitonicCore(subarray(array, i), s)$
    return array

Function BitonicCore(array, n)
    // Two pointers to the head and tail of the array.
    $leftPtr \rightarrow array[0]$
    $rightPtr \rightarrow array[n-1]$
    for $k = 0;\ k < n/2;\ k = k + 1$ do          // 1st sub-stage
        $test\&exchange(leftPtr, rightPtr)$
        // Compare two elements and swap them if necessary.
        $leftPtr += 1$
        $rightPtr -= 1$
    for $step = n/4;\ step > 0;\ step = step/2$ do // Other sub-stages
        for $i = 0;\ i < n;\ i = i + step \times 2$ do
            $leftPtr \rightarrow array[i]$
            $rightPtr \rightarrow array[i + step]$
            for $k = 0;\ k < step;\ k = k + 1$ do
                $test\&exchange(leftPtr, rightPtr)$
                $leftPtr += 1$
                $rightPtr += 1$
    return array
```

In Algorithm 1, we show a bitonic sorting algorithm implementation, and the function *BitonicCore* is used to complete the sorting of a bitonic sequence. The bitonic sequence is a sequence with $x_0 \leq \cdots \leq x_k \geq \cdots \geq x_{n-1}$ for some $k, 0 \leq k < n$, or a circular shift of such a sequence.

The procedure of sorting n elements is divided into log_2n stages, and stage $ith(i = 1, 2, ..., log_2n)$ needs to perform $n/2^i$ BitonicCore to sort bitonic sequences, and each includes 2^i elements. Two sorted bitonic sequences make up a new bitonic sequence, which belong a part of the input for next stage. The stage ith includes i sub-stages which performs test&exchange operation between two elements with $2^{(i-1)}$ strides. In Fig. 1, there are four stages, 10 sub-stages, and 15 BitonicCore. For a fixed-size bitonic sorting network, the task of the BitonicCore is determined. We will hard-code several sub-stages to complete these tasks in Sect. 4.1.1.

2.2 Related work

In this section, we focus on the bitonic sorting algorithm that utilizes SIMD and multicore processors, including CPUs, GPUs, and other parallel devices.

Chhugani et al. (2008) proposed a bitonic sorting optimization using Intel's SSE instructions. They implement the sorting micro-kernel with 128-bit SIMD. In addition, their algorithm performs an efficient multiway merge, and is not constrained by memory bandwidth. Yin et al. (2019) proposed a hybrid sorting method that takes advantage of wide vector registers and the high bandwidth memory of modern

AVX-512-based multi-core and many-core processors. There are numerous other studies on bitonic sorting on SIMD processors (Hou et al. 2018; Gueron and Krasnov 2016).

The high computational density and bandwidth of GPUs are exploited to accelerate the sorting algorithm. Guo et al. (2018) proposed a memory access reduced bitonic sort on multi-core GPUs to relieve pressure on memory bandwidth. They implement a multiway bitonic sorting network in which the warp-shuffle instructions are taken advantage of Peters et al. (2011) proposed a high-performance in-place implementation of bitonic sorting networks for CUDA-enabled GPUs. In their implementation, compare/exchange operations are assigned to threads in a way that decreases low-performance global-memory access and makes efficient use of high-performance shared memory.

There are also a number of sorting algorithms designed for specific architectures. Chen and Prasanna (2017) proposed a systematic methodology for mapping large-scale bitonic sorting networks onto FPGAs. By utilizing the proposed design for data permutation, they developed a hardware generator to automatically build bitonic sorting architectures on FPGAs.

The FT-M7032 is notably different from previous processors like the CPU and GPU in that it includes 16 CPU cores and four multi-core GPDSP clusters. Therefore, the FT-M7032 platform cannot use existing algorithms, and we present thSORT, a first efficient parallel bitonic sorting algorithm for FT-M7032.

3 FT-M7032 architecture

FT-M7032 is a high performance CPU-DSPs heterogeneous processor entirely designed and implemented by the National University of Defense Technology. The processor integrates 16 ARMv8 CPU cores for running operating systems and four multi-core general computing DSP (GPDSP) clusters for high computing performance (Yin et al. 2022). Its architecture is shown in Fig. 2. The multi-core CPU is responsible for process-level management and communication and has a peak single-precision floating point performance of 281.6 GFlops.

Each GPDSP cluster includes eight DSP cores. All eight DSP cores and GSM in each cluster can communicate via an on-chip crossbar network. For data coherency among them, it needs to be maintained by software developers. The DSP core in the GPDSP cluster is based on the VLIW architecture, including an instruction dispatch unit (IFU), a scalar processing unit (SPU), a vector processing unit (VPU), and a DMA engine, as shown in Fig. 3. IFU can emit up to 11 instructions per cycle, including five scalar instructions and six vector instructions. SPU is responsible for instruction flow control and scalar computation, and

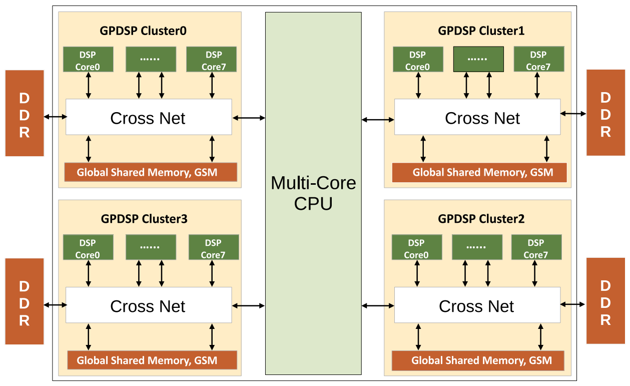


Fig. 2 Architecture of FT-M7032

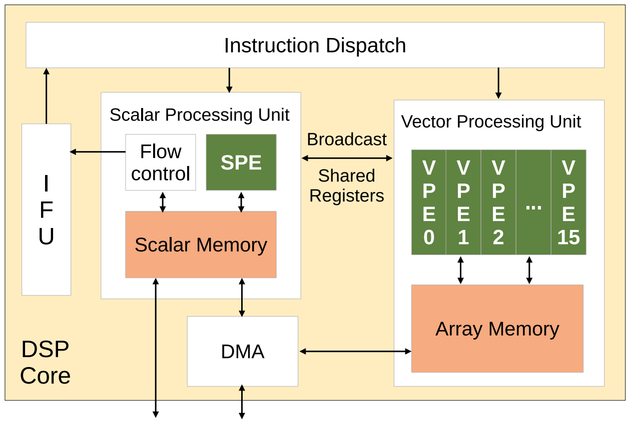


Fig. 3 Micro-architecture of DSP Core in FT-M7032

consists of Scalar Processing Element (SPE) and 64KB Scalar Memory (SM). VPU provides the main computing performance for each DSP core, including 768 KB Array Memory (AM) and 16 vector processing elements (VPE) working in an SIMD manner.

For better bandwidth performance, the FT-M7032 has a hybrid memory hierarchy. The multi-core CPU and four GPDSP clusters share the same main memory space. The multi-core CPU can access the whole main space, but each GPDSP cluster can only access its own corresponding part. The eight DSP cores in a GPDSP cluster share 6 MB on-chip Global Shared Memory (GSM). All DSP cores have their own SM of 64 KB and AM of 768 KB. Between SM in SPU and AM in VPU, data can be transferred through broadcast instructions and shared registers. The DMA engine is utilized to transfer data between different memory hierarchies, including DDR, GSM, and SM/AM.

On the software part, `hthreads` is provided to improve the programmability of multi-core DSPs (Fang et al. 2023). The `hthreads` runtime manages the interaction between CPU and GPDSP regions, with the CPU side serving as the host and each acceleration cluster as a computing device.

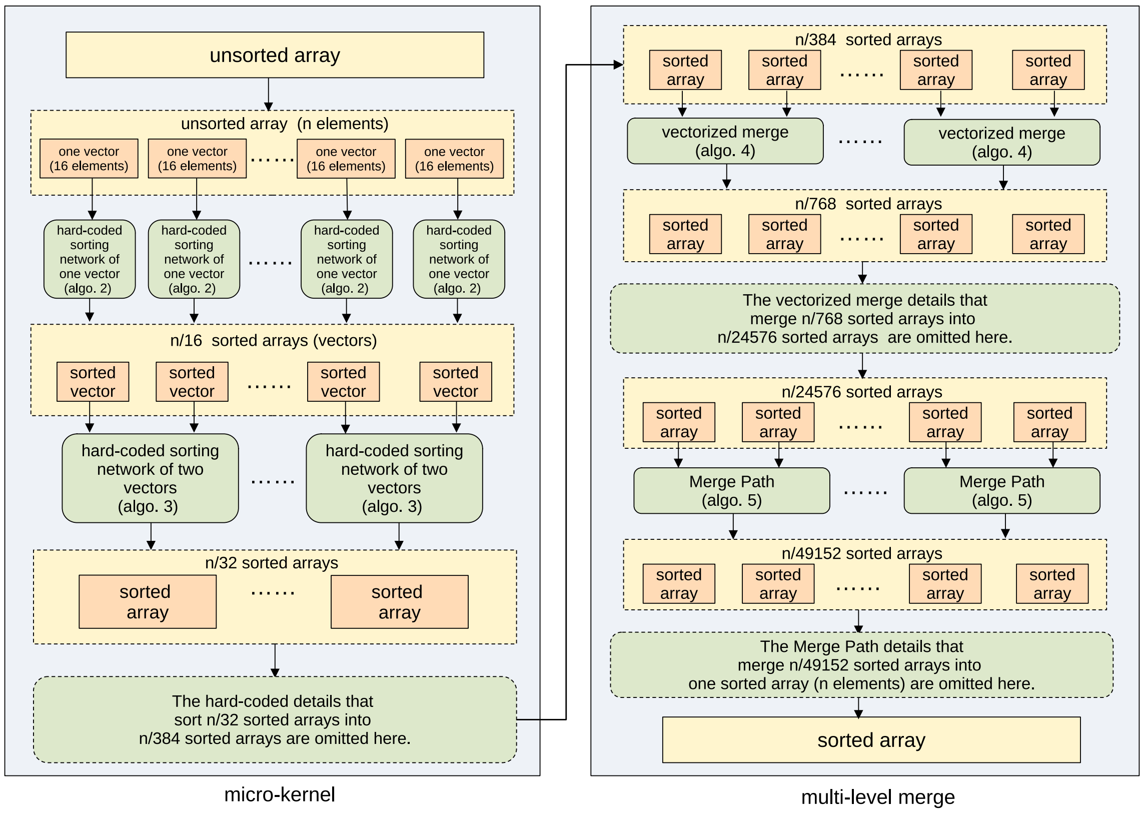

unsorted array
unsorted array (n elements)
one vector (16 elements)
hard-coded sorting network of one vector (algo. 2)
n/16 sorted arrays (vectors)
sorted vector
hard-coded sorting network of two vectors (algo. 3)
n/32 sorted arrays
sorted array
The hard-coded details that sort n/32 sorted arrays into n/384 sorted arrays are omitted here.
micro-kernel
n/384 sorted arrays
vectorized merge (algo. 4)
n/768 sorted arrays
The vectorized merge details that merge n/768 sorted arrays into n/24576 sorted arrays are omitted here.
n/24576 sorted arrays
Merge Path (algo. 5)
n/49152 sorted arrays
The Merge Path details that merge n/49152 sorted arrays into one sorted array (n elements) are omitted here.
sorted array
multi-level merge

Fig. 4 An overview of our sorting algorithm. The reason why the number of vectors in hard-coding does not exceed 24 is explained in Sect. 4.1.1. Due to the capacity limitations of AM, the result of vectorized merge is no more than 24,576 elements, as detailed in Sect. 4.1.2

We use `hthreads` to implement and optimize our sort algorithm.

4 thSORT

4.1 Single-core DSP sorting

Based on the architecture of FT-M7032, our algorithm is divided into two parts: single-core DSP sorting and multi-core DSP sorting. An overview of our sorting algorithm is shown in Fig. 4. The single-core DSP sorting which consists of micro-kernel and vectorized merge, and the multi-core DSP sorting consists of Merge Path. In this section, we propose the single-core DSP sorting, which takes advantage of the vector instructions supported by FT-M7032.

4.1.1 Hard-code sorting network

In this section, we hard-code a bitonic sorting network of 24 vectors called the micro-kernel. The bitonic sorting network consists of stages and each stage includes sub-stages,

Fig. 5 The procedure of a sub-stage

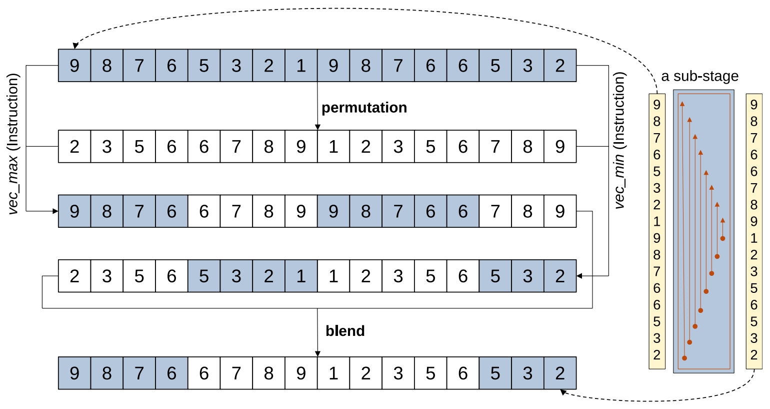


Table 1 The FT-M7032 instructions related to SVR

Instruction	Cycle	Function
VMVCGC	3	Configure SVR with a vector
SMVCCG	2	Configure a scalar register with SVRx
SMVCGC	3	Configure SVRx with a scalar
VMVCCG	2	Configure a vector register with SVR

as shown in Fig. 1. We hard-code several sub-stages and eventually formed sorting networks. The function of the sub-stage is to perform $n/2$ comparisons and decide whether to exchange elements based on the results of the comparisons. As the comparing operations followed are based on vectors, we need to load the data from DDR to AM before sorting. The vectors instructions *vec_min* and *vec_max* given by the FT-M7032 instruction set are used to implement comparisons in the sub-stage. However, the two instructions can only compare the corresponding positions of two vectors. To complete the comparison of elements in any position, permutation operation of the vector is necessary. After permutations and comparisons, the two result vectors are blended into a single vector. The procedure of a sub-stage in Fig. 1 is shown in Fig. 5.

Since there is no permutation instruction in the FT-M7032 instruction set, we implement the operation by

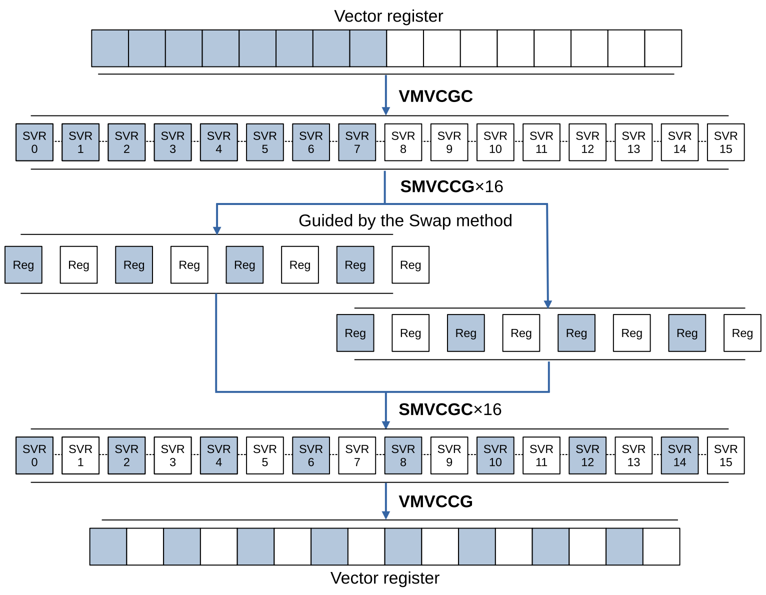
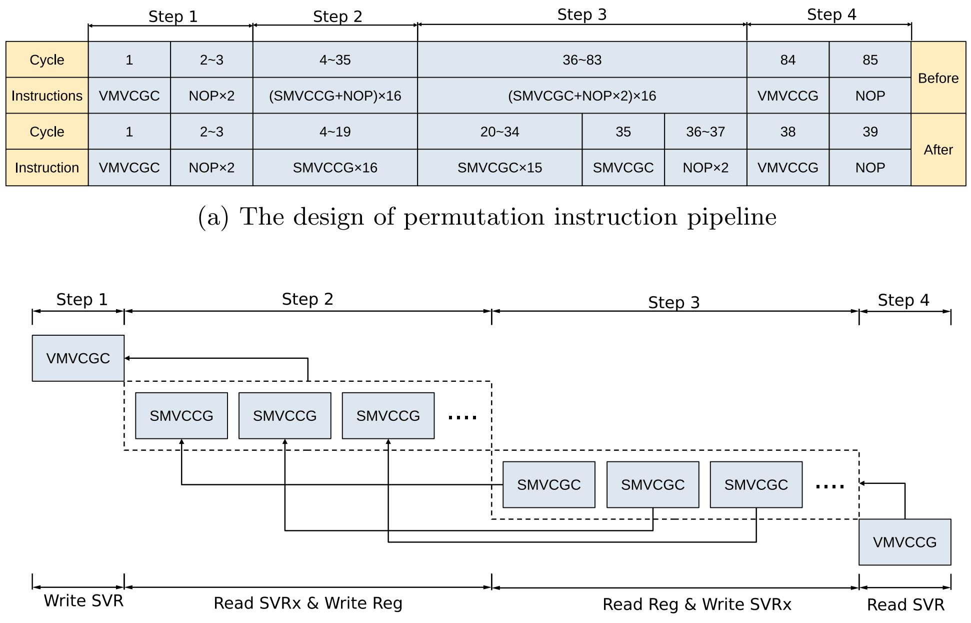


Fig. 6 The procedure of permutation using the instructions of FT-M7032



	Step 1		Step 2	Step 3			Step 4		
Cycle	1	2~3	4~35	36~83			84	85	Before
Instructions	VMVCGC	NOP×2	(SMVCCG+NOP)×16	(SMVCGC+NOP×2)×16			VMVCCG	NOP	
Cycle	1	2~3	4~19	20~34	35	36~37	38	39	After
Instruction	VMVCGC	NOP×2	SMVCCG×16	SMVCGC×15	SMVCGC	NOP×2	VMVCCG	NOP	

(a) The design of permutation instruction pipeline

(b) The dependencies between instructions

Fig. 7 The cycle of permutations is reduced from 85 to 39 by taking advantage of the pipeline to optimize instructions

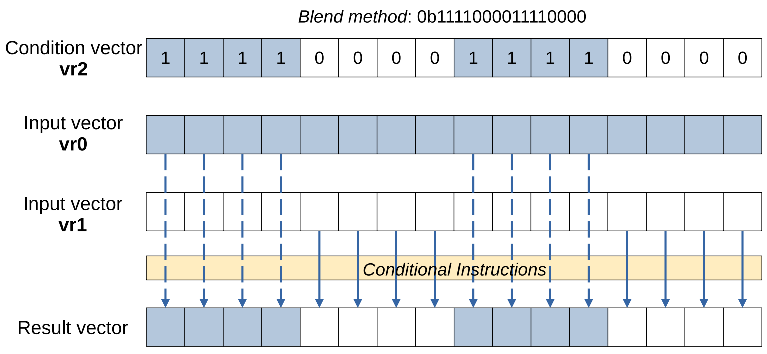


Fig. 8 The procedure of blend using the conditional instructions of FT-M7032

using the scalar vector register, which is a special register. This special register comes in two forms. SVR is regarded as a vector register and SVRx ($x = 0, 1, \dots, 14, 15$) is regarded as a scalar register in assembly code. The permutation needs four instructions: VMVCGC, SMVCCG, SMVCGC, and VMVCCG (for details, see Table 1). They have different execution cycles. The execution of VMVCGC and SMVCGC instructions needs three cycles, and of VMVCCG and SMVCCG instructions needs two cycles.

Figure 6 illustrates the procedure of a permutation. In step 1, moving a vector, which includes 16 elements, to SVR uses a VMVCGC instruction. In step 2, moving SVRx ($x = 1, 2, ..., 15, 16$) to 16 scalar registers uses 16 SMVCCG instructions. In step 3, moving the scalar registers to SVRx ($x = 1, 2, ..., 15, 16$) uses 16 SMVCGC instructions. In step 4, moving SVR to a vector uses a VMVCCG instruction. This procedure executes 1 VMVCGC, 16 SMVCCG, 16 SMVCGC, and 1 VMVCCG instructions. According to Table 1, NOP instructions are needed. After adding NOP instructions, the whole procedure takes 85 cycles as shown in Fig. 7a. Thanks to the pipeline design, we can optimize the instruction sequence by removing some NOP instructions. In step 2, 16 SMVCCG needs 16 NOP instructions. There is no dependency between the 16 instructions, so all 16 NOP instructions can be removed. In step 3, 16 SMVCGC need 32 NOP instructions. However, the VMVCCG in step 4 depends on the last SMVCGC in step 3. So there are only 30 NOP instructions that can be removed. The procedure needs 39 cycles after optimization. The design of the pipeline is shown in Fig. 7a, and the dependency between the instructions is shown in Fig. 7b.

The result of comparisons is gain after permutation and vector comparison (*vec_min* and *vec_max*). We blend the two result vectors into a vector by using the conditional instruction of FT-M7032 instruction set. The form of a conditional instruction is as follows:

$$[vr2]MOV\ vr0,\ vr1$$

Three vector registers are used in the instruction. In $vr2$, there are 16 elements that depend on the *Blend method*, a 16-bit integer, mentioned in Algorithm 2, and each element is used as a condition to determine whether the corresponding MOV instruction is executed. If the ith element of $vr2$ is zero, move the ith element of $vr1$ to the ith position of the result vector; otherwise, move the element of $vr0$ to that position. By taking advantage of this feature of the FT-M7032 instruction set, the blend operation can be performed efficiently. Suppose the results of *vec_min* and *vec_max* in Fig. 5 are vectors $vr0$ and $vr1$, respectively. With two conditional instructions, values from $vr0$ and $vr1$ can be blended to compose the result vector as shown in Fig. 8. For a given sub-stage, the two instructions costs 2 cycles if the condition vector $vr2$ is prepared in advance.

Through permutation and blend, we can implement hard-coding of sub-stages. In Algorithm 2, we hard-code 10 sub-stages to form a sorting network of 16 elements. The input of function *Substage* in Algorithm 2 includes a vector and two control parameters, and the meaning of the control parameters is as follows: *Swap method* provides guidance on how to perform permutation, and *Blend method* represents the conditional vector to perform blend.

Algorithm 2 Bitonic sort for one vector(16 elements)

```
Input: V: one vector to sort
Output: V: the vector sorted
Function vector_sort_16(V)
    // the sorting network of 16 elements has 10 sub-stages
    V←Sub-stage(V, 0-1 2-3 4-5 6-7 8-9 10-11 12-13 14-15, 0xaaaa)
    V←Sub-stage(V, 0-3 1-2 4-7 5-6 8-11 9-10 12-15 13-14, 0xcccc)
    V←Sub-stage(V, 0-1 2-3 4-5 6-7 8-9 10-11 12-13 14-15, 0xaaaa)
    V←Sub-stage(V, 0-7 1-6 2-5 3-4 8-15 9-14 10-13 11-12, 0xf0f0)
    V←Sub-stage(V, 0-2 1-3 4-6 5-7 8-10 9-11 12-14 13-15, 0xcccc)
    V←Sub-stage(V, 0-1 2-3 4-5 6-7 8-9 10-11 12-13 14-15, 0xaaaa)
    V←Sub-stage(V, 0-15 1-14 2-13 3-12 4-11 5-10 6-9 7-8, 0xff00)
    V←Sub-stage(V, 0-4 1-5 2-6 3-7 8-12 9-13 10-14 11-15, 0xf0f0)
    V←Sub-stage(V, 0-2 1-3 4-6 5-7 8-10 9-11 12-14 13-15, 0xcccc)
    V←Sub-stage(V, 0-1 2-3 4-5 6-7 8-9 10-11 12-13 14-15, 0xaaaa)
    return V

Function Sub-stage(V, Swap method, Blend method)
    Vperm ← permutation of V by the Swap method
    Vmi ← vec_min(V, Vperm)
    Vmx ← vec_max(V, Vperm)
    // Pick the right result from Vres based on the bits of
       Blend method.
    Vres ← blend Vmi and Vmx by the Blend method
    return Vres
```

Based on the sorting of one vector, we can hard-code the sorting network of multiple vectors. Taking two vectors as an example, first, sort the two vectors by Algorithm 2, respectively. Two sorted vectors form a bitonic sequence, and BitonicCore in Algorithm 1 can be used to sort the vectors. Second, similar to Algorithm 2, we hard-code every sub-stage in the BitonicCore of the sorting network. The hard-coding of sorting network of two vectors is shown in Algorithm 3.

Algorithm 3 Bitonic sort for two vector(32 elements)

```
Input: V1, V2: two vectors to sort
Output: V1, V2: two vectors sorted
Function vector_sort_32(V1, V2)
    V1←vector_sort_16(V1)
    V2←vector_sort_16(V2)
    merge_2V_sorted(V1, V2)
    return V1, V2

Function aftermerge(V)
    V←Sub-stage(V, 0-8 1-9 2-10 3-11 4-12 5-13 6-14 7-15, 0xff00)
    V←Sub-stage(V, 0-4 1-5 2-6 3-7 8-12 9-13 10-14 11-15, 0xf0f0)
    V←Sub-stage(V, 0-2 1-3 4-6 5-7 8-10 9-11 12-14 13-15, 0xcccc)
    V←Sub-stage(V, 0-1 2-3 4-5 6-7 8-9 10-11 12-13 14-15, 0xaaaa)
    return V

Function merge_2V_sorted(V1, V2)
    Vtmp←permute V2 by the Swap method(0-15 1-14 2-13 3-12 4-11
     5-10 6-9 7-8)
    V1←vec_min(V1, Vtmp) // Choose smaller elements from
     two vectors.
    V2←vec_max(V1, Vtmp) // Choose larger elements from two
     vectors.
    // 1st sub-stage completed.
    V1←aftermerge(V1)
    V2←aftermerge(V2)
    // all sub-stages completed.
    return V1, V2
```

To use vector instructions and improve parallelism, we hard-code the sorting network. However, there is an issue of efficiency in this procedure. In a sub-stage, we use the *vec_min* and *vec_max* instructions. *vec_min* instruction makes 16 comparisons, but only eight are valid, same for *vec_max*, as shown in Fig. 5. In other words, only 50% of comparisons are valid. Meanwhile, the single-core DSP sorting consists of micro-kernel and vectorized merge, and the vectorized merge presented in Sect. 4.1.2 has all comparisons valid. Therefore, the performance does not keep growing as the size of the hard-coded sorting network increases. We eventually hard-code a sorting network of 24 vectors in our implement and use it as the micro-kernel of our algorithm, as shown in Fig. 4.

4.1.2 Merge in DSP core

In this paper, we develop a multi-level merge algorithm, as shown in Fig. 4. The vectorized merge and Merge Path, a parallel merging algorithm proposed by Green et al. (Green et al. 2012), correspond to single-core DSP sorting and multi-core DSP sorting. In this section, We present the vectorized merge.

After implementing the micro-kernel (sorting network of 24 vectors), we need an efficient algorithm to merge each of the 24 vectors. Two vectors can be compared using the *vec_min* and *vec_max* instructions. The merge operation can be accelerated by using the two instructions. In Algorithm 4, we presents a vectorized merge algorithm that is based on the article (Chhugani et al. 2008). Suppose the two sub-arrays to be merged are [left, middle] and [middle+1,right] of the whole array, which called left-array and right-array, respectively, and suppose that the elements of two arrays is both a multiple of the size of 16 (vector size). The procedure of the vectorized merge is as follows:

- Load V1 and V2 vectors from the left-array and right-array, respectively. Use *merge_2V_sorted* which is defined in Algorithm 3 to merge V1 and V2, and store V1 in the result array.
- Load a vector to V1 from the array whose first unloaded element is smaller. Use *merge_2V_sorted* to merge V1 and V2, and store V1 in the result array. Repeat until all the elements of an array have been loaded.

- If the elements in the left-array or right-array are not loaded, load 16 elements into V1. Use *merge_2V_sorted* to merge V1 and V2, and store V1 in the result array. Repeat until all the elements of the array have been loaded.
- Store V2 in the result array.

Due to the space limitations of AM, there is a limit to the number of elements that vectorized merge can handle. When the size of the sorted array exceeds the capacity of the AM, we can chunk the array. Assume that the chunk size is C, which is limited by many factors as follows: (1) due to the implementation of the algorithm, the following equation needs to be met: $C = 384 \times 2^k (k = 0, 1, 2, 3...)$. (2) Since we use an auxiliary array to store the merged subarray, the following equation needs to be met: $2 \times C \leq 760$ KB. Therefore, the chunk size is 24,576, which means the algorithm can merge two arrays of 12,288 into one array of 24576.

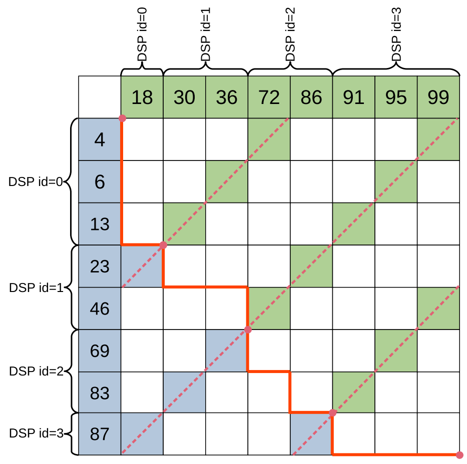


Fig. 9 Merge Path matrix showing intersection lines and points

Algorithm 4 Vectorized merge algorithm

```
Input: left_array&right_array: two arrays to be merged
Output: array: the array sorted
Function vectorized_merge(left_array, right_array)
    V1 ← load a vector from left_array
    V2 ← load a vector from right_array
    merge_2V_sorted(V1, V2)
    store V1 in array // add V1 to the final result.
    while both left_array and right_array have vectors to load do
        // Compare the first of the remaining elements of
         left_array and right_array.
        left1st ← the first element of unloaded left_array
        right1st ← the first element of unloaded right_array
        if left1st<right1st then
            V1 ← load a vector from left_array
        else
            V1 ← load a vector from right_array
        merge_2V_sorted(V1, V2)
        store V1 in array // add V1 to the final result.

    while only left_array have vectors to load do
        V1 ← load a vector from left_array
        merge_2V_sorted(V1, V2)
        store V1 in array // add V1 to the final result.

    while only right_array have vectors to load do
        V1 ← load a vector from right_array
        merge_2V_sorted(V1, V2)
        store V1 in array // add V1 to the final result.
    store V2 in array // add V2 to the final result, and merge
     complete.
    return array
```

4.2 Multi-core DSP sorting

In this section, we propose multi-core DSP sorting, which is carried out on the basis of single-core DSP sorting. The sorted arrays obtained from single-core DSP sorting are merged by the Merge Path algorithm, which is the second level of multi-level merge, as shown in Fig. 4. There are two reasons for proposing multi-core DSP sorting:

Space limitation. The data must be stored in the private memory of the DSP core in order to implement vectorization. Each DSP core has its own AM with a size of 760 KB. The amount of elements a single DSP core can sort is limited by the size of the AM.

Parallelism. We expect all DSP cores to work in parallel, and each DSP core is responsible for a part of the work. When n elements are sorted using p DSP cores, the elements are divided into p arrays of size n/p. However, utilizing all the DSP cores to merge these arrays is a challenge. For example, when using the normal merge algorithm, $p/2$ DSP cores deal with p arrays to get $p/2$ sorted arrays, and $p/4$ DSP cores deal with $p/2$ arrays to get $p/4$ sorted arrays, and so on. Normal merge algorithm cannot utilize the all DSP cores. To further sort these arrays, an efficient parallel merge algorithm is needed. We present the Merge Path algorithm as an alternative solution to address this issue.

4.2.1 Merge Path

Merge Path is an efficient parallel merge algorithm with many advantages. One of the key advantages of the algorithm is that it is in general load-balanced, which means that all DSP cores can be assigned the same amount of work. This is achieved by dividing the input sequences into chunks of equal size and assigning each chunk to a different DSP core. Additionally, Merge Path is a lock-free algorithm, which means that it does not use atomic instructions or locks to synchronize access to shared resources (Green et al. 2012). The time complexity of the algorithm for n elements and p processors is given by $O(n/p + log(n))$. For $p < N/log(n)$, this algorithm is considered to be optimal. The procedure of Merge Path is shown in Algorithm 5.

Algorithm 5 Merge Path

```
Input: A, B: two array to be merged
Output: Res: the array sorted
1  Function $MergePath(A, B)$
2      $A_{diag}[DSPCores] \leftarrow A_{length}$
3      $B_{diag}[DSPCores] \leftarrow B_{length}$
4      for each DSP core in parallel do
5          $index \leftarrow DSPid \times (A_{length} + B_{length})/DSPCores$
6          $a_{top} \leftarrow min(index, A_{length})$
7          $b_{top} \leftarrow max(index - A_{length}, 0)$
8          $a_{bottom} \leftarrow b_{top}$
9          // binary search for diagonal intersections
10         while true do
11             $offset = (a_{top} - a_{bottom})/2$
12             $a_i \leftarrow a_{top} - offset$
13             $b_i \leftarrow b_{top} + offset$
14             if $A[a_i] > B[b_i - 1]$ then
15                 if $A[a_i - 1] \leq B[b_i]$ then
16                     $A_{diag}[DSPid] \leftarrow a_i$
17                     $B_{diag}[DSPid] \leftarrow b_i$
18                     break
19                 else
20                     $a_{top} \leftarrow a_i - 1$
21                     $b_{top} \leftarrow b_i + 1$
22             else
23                 $a_{bottom} \leftarrow x + 1$
24     All DSP cores are synchronized here.
25     for each DSP core in parallel do
26         // Merge A and B arrays in parallel.
27         $a_{stop} \leftarrow A_{diag}[DSPid]$
28         $b_{stop} \leftarrow B_{diag}[DSPid]$
29         merge $A[a_i, a_{stop})$ and $B[b_i, b_{stop})$ results into Res
30         return Res
```

Suppose there are two sorted arrays A and B, and place them in a grid as shown in Fig. 9. A and B are placed on the left and top, respectively. For the convenience of the description, A and B are set to the same length, and the algorithm can be used in practice with different lengths of A and B.

The merging procedure begins at the top-left corner, where individual elements of two arrays undergo pairwise comparison. If the element from array B is smaller than that from array A, the paths move to the right by one position. Otherwise, the paths move down by one position. Until the path reaches the bottom-right corner, the procedure ends. The path formed in this procedure is called the merge path. The cross diagonals in Fig. 9 are used to assign the same amount of work to the DSP cores. The top-left and bottom-right corners are considered two cross diagonals of length zero. The cross diagonals divide the merge path into segments of equal length. In the Merge Path algorithm, each DSP core is responsible for one of the segments. Now notice the intersection of each cross diagonal with the merge path, which determines the start and end of the array where each DSP core needs to merge.

For better time complexity, a binary search can be used to figure out the intersection between the cross diagonal and merge path. The diagonal will pass through several blocks, and each block corresponds to A_i and B_i from A and B,

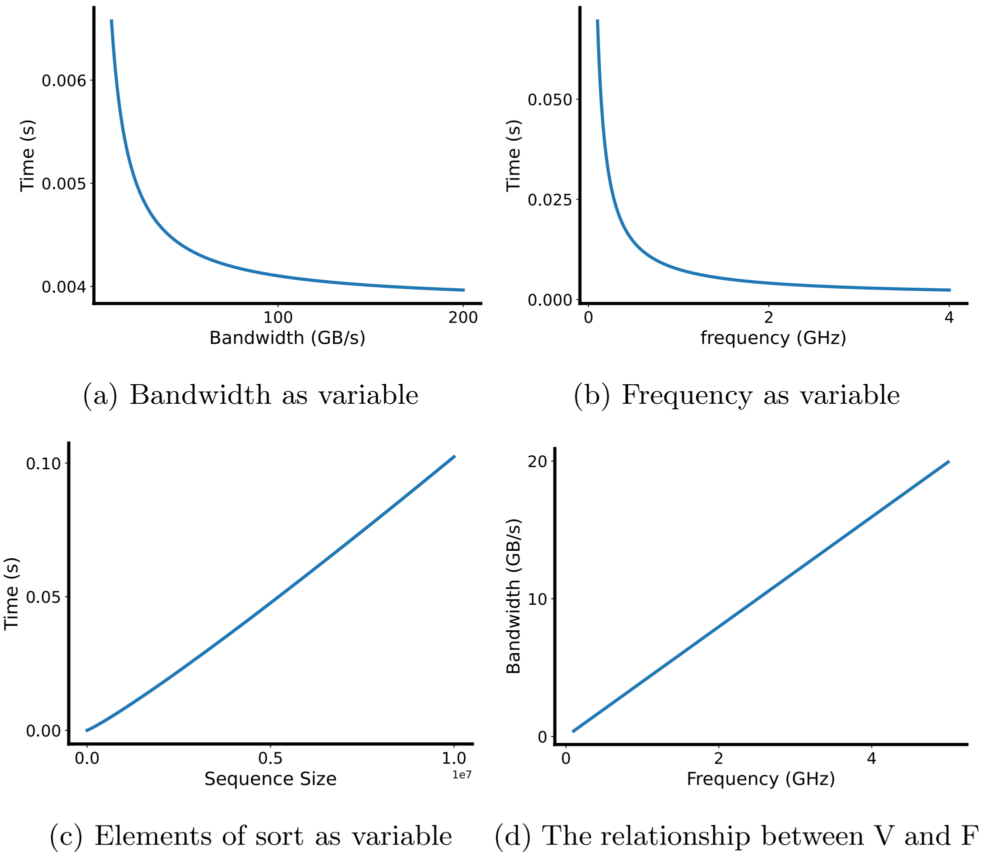


Fig. 10 The effect of the variation of each parameter in the performance model on the results (Default B = 42.62 GB/s, F = 1.8 GHz, N = 589,824)

Fig. 11 The theoretical performance and the composition of time consumption of performance model. As the time complexity of our sort algorithm is $O(n \times log_2(n))$, the performance degrades as the number of input elements increases

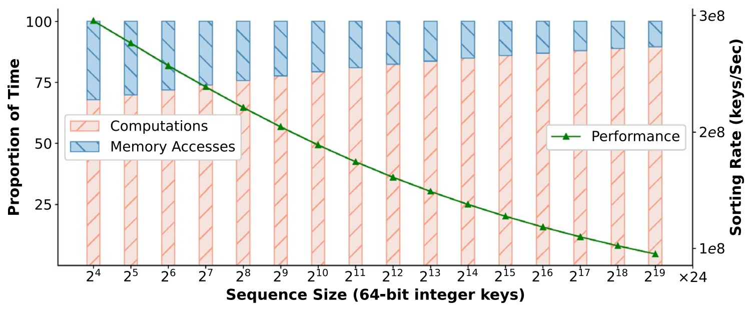


Table 2 Experimental setup for performance testing

Name	Version
Operating system	Ubuntu 19.04
Host compiler	GCC 8.3.0
Device compiler	M3CC 1.0
Programming framework	Hthreads 9.8
CPU	16 ARMv8 Cores
DSP	8 GPDSP cores $\times$ 4
DDR	32 GB $\times$ 4

respectively. For blocks to the left of the intersection, they satisfy A_i not less than B_i. For blocks to the right, they satisfy A_i less than B_i. Therefore, binary search can be used to find the intersection, and the time complexity is $O(log(N))$.

5 Performance model

In this section, we propose a performance model of our sorting algorithm that provides insight into performance optimization. Based on the theoretical memory accesses and computations, we can obtain the performance model of our sorting algorithm. Assume that each DSP core sorts B elements, which means $n = B \times p$. In single-core DSP sorting, one element requires one load and one store operations. In multi-core DSP sorting, the merge algorithm requires $n \times log(p)$ load and store operations. The total number of memory accesses is as follows:

$$Memory\ Accesses = 2 \times B \times p + 2 \times N \times log(p)$$

For the scalability of the performance model, vectorization is not considered when calculating the theoretical computations. $T(n)$ denotes the number of comparisons for the bitonic sorting network of n elements, and $T(n) = T(n/2) \times 2 + log(n) \times n/2$. In single-core DSP sorting, each DSP core sorts B elements, which requires $T(B)$ comparisons. In multi-core DSP sorting, merge algorithms require $n \times log(p)$ comparisons. The total computations is as follows:

$$Computations = T(B) \times p + log(p) \times n$$

Assuming a processor with a frequency of F Hz and a DDR bandwidth of $8 \times V$ bytes per second. The theoretical execution time for sorting n numbers using p DSP cores is as follows:

$$Total\ time = \frac{2 \times B + 2 \times n \times log(p)}{V} + \frac{T(B) + n \times log(p)}{F}$$

After determining F and V of a multi-core processor and n of the array sorted, the theoretical fastest execution time of our sorting algorithm can be obtained by the performance model. By controlling two of the parameters F, V, and n to be constant, we can obtain the trend of time consumption when changing the third parameter, which is shown in Fig. 10a–c respectively, and Fig. 10d shows the relationship between V and F when the memory access time is equal to the computation time. Figure 11 further refines the details of Fig. 10c. Similarly, the frequency is set to 1.8 GHz and the bandwidth is set to 46.62 GB/s. By changing the number of sorted elements, the time proportion of two parts (*Computations & Memory Access*) and the theoretical performance are shown in Fig. 11.

The performance model can reveal the fact that thSORT is compute-bound. In Fig. 10, the bandwidth corresponding to the frequency of 1.8 GHz is about 10 GB/s, which is less than the actual bandwidth of 46.62 GB/s. In Fig. 11, computations take up the majority of the running time. Besides, the model can also guide the architectural design of a processor. In order to design an architecture for sorting, the allocation of hardware resources, including computing hardware and memory hardware, can be changed by analyzing the relationship between V and F.

6 Experimental results

6.1 Experimental setup

We assess our method on FT-M7032 with 16 ARMv8 CPU cores and one cluster. A comprehensive overview of the experimental setup and configuration employed in this study is shown in Table 2.

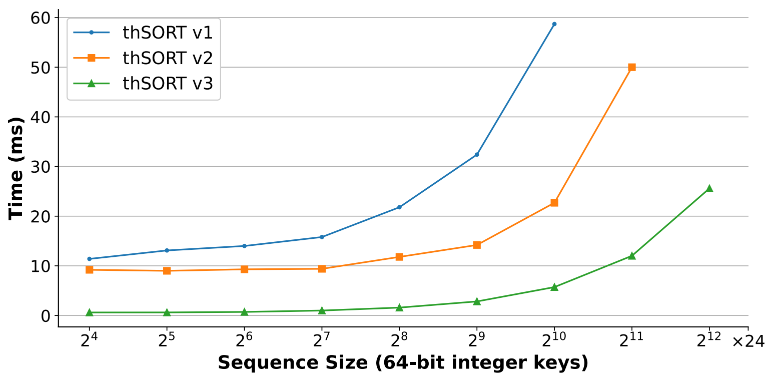


Fig. 12 The performance of one DSP with memory hierarchy optimization

For single-core DSP sorting, we optimize the memory access of thSORT and compare the performance before and after optimization. For multi-core DSP sorting, we compare it against the *block_indirect_sort* of Boost C++ Libraries 1.67.0 and the *std::sort* of GUN libc++.so.6. *Boost::block_indirect_sort* is highly regarded for its robust performance and exceptional handling of large-scale datasets. *Std::sort*, being an integral part of the C++ standard library, is considered a well-vetted and reliable sorting algorithm. By comparing these two sorting algorithms to thSORT, we aim to comprehensively evaluate the performance of our algorithm implementation. In all performance tests, the arrays to sort are populated with randomly generated values.

6.2 Single-core DSP sorting

The single-core DSP sorting consists of micro-kernel and vectorized merge. Micro-kernel is designed to sort 24× VEC_SIZE elements (384 elements). When sorting more than 384 elements, vectorized merge continues to conduct the sorting on the results of the micro-kernel.

Analyzing our algorithm implementation, we find that most time data is stored in DDR, which means that the memory hierarchy of FT-M7032 is not fully utilized. Therefore, we conduct a series of optimizations for this by utilizing AM and GSM. We optimized the memory access of the thSORT and obtained three versions. thSORT v1 is the base version, thSORT v2 is optimized on the basis of thSORT v1 using AM, and thSORT v3 is optimized on the basis of thSORT v2 using GSM. In addition, DMA is used to optimize data transfer in v2 and v3. The performance improvement of single-core DSP sorting resulting from the optimization of the memory access is shown in Fig. 12.

The micro-kernel of single-core DSP sorting sorts 384 elements. Based on the size of 384, the size of our test is multiplied by 2 per step. The optimized version showed an

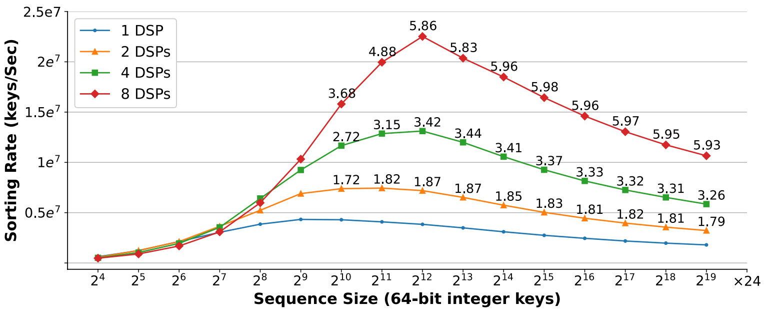


Fig. 13 Sorting between DSPs. Execution time divided by sort number, and the speedup of the 2, 4, and 8 DSPs sorting against the 1 DSPs sorting is shown above the lines

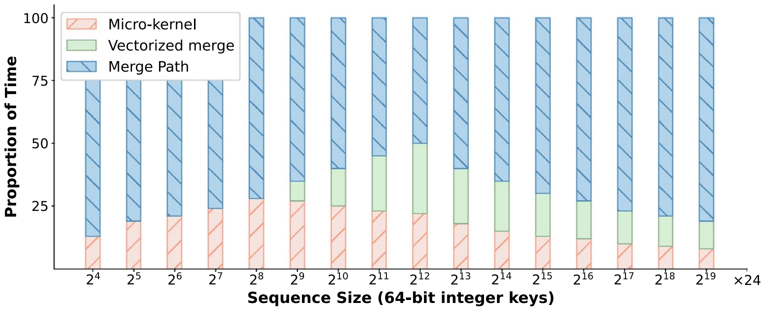


Fig. 14 The composition of time consumption of thSORT (bitonic sort micro-kernel, vectorized merge and Merge Path)

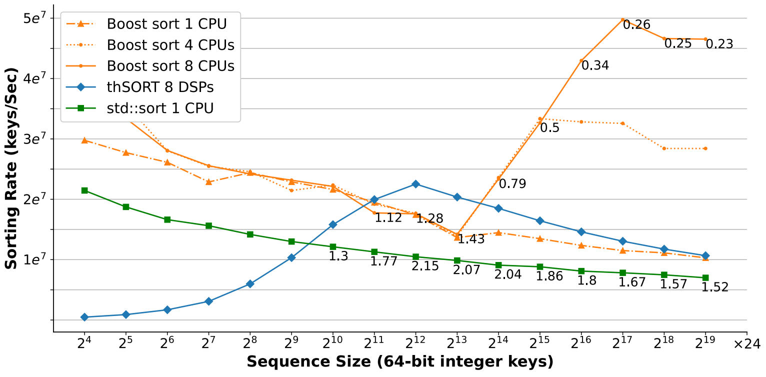


Fig. 15 Comparison of CPUs and DSPs. The speedup of the thSORT against the Boost sort of eight CPUs is shown above the line for Boost sort, against the *std::sort* of one CPU is shown above the line for *std::sort*

advantage at the beginning, and this advantage continued to increase as the scale increased. Through the memory optimization, the performance of thSORT has been significantly improved. This also shows that high bandwidth is very important for the performance improvement of sorting, especially when sorting large-scale arrays. Unless otherwise specified, the subsequent experiments are conducted using the thSORT v3 version.

6.3 Multi-core DSP sorting

6.3.1 Optimization results

Figure 13 shows the performance to sort arrays up to a size of 1.2×10^7 (24×2^{19}) elements using 1, 2, 4, and 8 DSPs. As the number of DSP grows, the speedup grows as well, which demonstrates the good scalability of thSORT. In addition, we observe that when sorting elements exceeds 24×2^{12}, the performance starts to drop while the speedup increases. This number holds significance as it equals the maximum number of elements that AM can accommodate. Hence, we conjecture that the performance degradation is associated with memory access, and we combine the performance model to further analyze the reasons for this phenomenon from the perspective of time consumption.

Figure 14 shows the composition of time consumption of the vectorized merge, micro-kernel, and Merge Path across the entire sort. After the sorting elements exceed 24×2^{12}, the proportion of Merge Path increases as performance decreases. Therefore, the phenomenon of performance degradation is related to Merge Path. FT-M7032 consists of two parts: single-core DSP sorting and multi-core DSP sorting. The former part, including the micro-kernel and vectorized merge, occurs inside the DSP core, which means almost no data exchange between the DSP core and DDR. On the contrary, there is frequently memory access to DDR when conducting Merge Path. In Sect. 6.2, we optimized the memory access of the thSORT by utilizing AM and GSM. However, their space is limited, while AM has 768KB and GSM has 6MB. When the scale rises, this will inevitably result in an increase in the amount of data exchanged between AM, GSM, and DDR. Therefore, Merge Path is the reason for the performance degradation.

We can roughly regard micro-kernel and vectorized merge as the calculation part of the algorithm, and Merge Path as the memory access part of the algorithm. Comparing Figs. 11 and 14, we find that memory access occupies the majority of time in our implementation, which means our implementation is bandwidth-bound. This is inconsistent with the compute-bound conclusion in the performance model, which means that our implementation requires more memory access optimization as guided by the performance model. Therefore, the performance of our implementation is much poorer than the theoretical performance shown in Fig. 11. We achieve only 13.97% of the theoretical performance at a scale suitable for thSORT.

6.3.2 Comparison with other sorting

Figure 15 shows the performance for different numbers of cores of a parallel CPU sort implementation from the Boost C++ Libraries (*block_indirect_sort*), and the performance of *std::sort* against thSORT using eight DSPs. thSORT is faster when sorting 24×2^{12} elements. At a scale suitable, thSORT of eight DSPs is 1.43× faster than *block_indirect_sort* of eight CPUs, and is 2.15× faster than *std::sort*. The *block_indirect_sort* exhibits lower performance at small scales, but as the sorting scale exceeds 24×2^{13} elements, its performance experiences a rapid improvement, eventually stabilizing due to the associated overhead in block creation, merging, and limited cache utilization. From a hardware perspective, the single-core performance of the DSP core is inferior to that of the CPU. This is primarily attributed to architectural differences, such as the CPU's proficiency in utilizing advanced features like out-of-order execution and a sophisticated cache hierarchy. When analyzing performance outcomes, it is imperative to take these differences into account. Analyze the performance at different scales: When the scale is small, the features of FT-M7032, such as high-bandwidth on-chip memory, cannot be fully utilized. Furthermore, the overhead of startup and synchronization costs becomes more pronounced at smaller scales. When the scale is large, there are memory access problems mentioned in Sect. 6.2.

7 Conclusion

The FT-M7032 is a CPU-DSPs heterogeneous processor designed for HPC. This paper describes thSORT, an efficient parallel bitonic sorting algorithm on FT-M7032. thSORT consists of two parts: single-core DSP sorting and multi-core DSP sorting, both aiming to tap the features of FT-M7032. The single-core DSP sorting leverages the VPU in DSP for vectorization and optimizes memory access through AM. The multi-core DSP sorting reduces access latency by using GSM and improves the parallelism of sorting by implementing the Merge Path algorithm. Besides, we propose a performance model for performance improvement and architecture design. Through the performance model, we analyze the theoretical performance of thSORT and compare it with our implementation to guide our optimization efforts.

The experimental results show an enhancement in performance. By utilizing memory hierarchy, we have optimized memory access and greatly improved the performance of thSORT. We compare the sort implementations from the Boost C++ Libraries and std::sort with thSORT, our implementation is 1.43× faster than Boost sort, and is 2.15× faster than std::sort.

Acknowledgements We deeply appreciate the invaluable comments from all the reviewers. This research was supported by the National Natural Science Foundation of China under Grant Nos. 61972415 and 61972408.

Data availability The data that support the findings of this study is available upon request from the authors.

Declaration

Conflict of interest The authors declare no competing interests.

References

Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, Spring Joint Computer Conference, AFIPS ’68 (Spring), pp. 307–314. Association for Computing Machinery, New York (1968)

Chen, R., Prasanna, V.K.: Computer generation of high throughput and memory efficient sorting designs on FPGA. IEEE Trans. Parallel Distrib. Syst. **28**(11), 3100–3113 (2017)

Chhugani, J., Nguyen, A.D., Lee, V.W., Macy, W., Hagog, M., Chen, Y.-K., Baransi, A., Kumar, S., Dubey, P.: Efficient implementation of sorting on multi-core simd cpu architecture. Proc. VLDB Endow. **2**, 1313–1324 (2008)

Fang, J., Zhang, P., Huang, C., Tang, T., Lu, K., Wang, R., Wang, Z.: Programming bare-metal accelerators with heterogeneous threading models: a case study of matrix-3000. Front. Inf. Technol. Electron. Eng. **24**(4), 509–520 (2023)

Graefe, G.: Implementing sorting in database systems. ACM Comput. Surv. **38**(3), 10-es (2006)

Green, O., McColl, R., Bader, D.A.: Gpu merge path: a gpu merging algorithm. In: Proceedings of the 26th ACM International Conference on Supercomputing (ICS), pp. 331–340. Association for Computing Machinery, New York (2012)

Gueron, S., Krasnov, V.: Fast quicksort implementation using avx instructions. Comput. J. **59**(1), 83–90 (2016)

Guo, C., Chen, H., Li, C., Wu, T.: A memory access reduced sort on multi-core gpu. In: 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 586–593. Exeter, UK (2018)

Hou, K., Wang, H., Feng, W.-C.: A framework for the automatic vectorization of parallel sort on x86-based processors. IEEE Trans. Parallel Distrib. Syst. **29**(5), 958–972 (2018)

Jun, S.-W., Xu, S., Arvind.: Terabyte sort on fpga-accelerated flash storage. In: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 17–24. Napa, CA (2017)

Nassimi, S.: Bitonic sort on a mesh-connected parallel computer. IEEE Trans. Comput. **28**(1), 2–7 (1979)

Peters, H., Schulz-Hildebrandt, O., Luttenberger, N.: Fast in-place, comparison-based sorting with cuda: a study with bitonic sort. Concurr. Comput. Pract. Exp. **23**(7), 681–693 (2011)

Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore gpus. In: 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–10. Rome, Italy (2009)

Stehle, E., Jacobsen, H.-A.: A memory bandwidth-efficient hybrid radix sort on gpus. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD ’17, pp. 417–432. Association for Computing Machinery, New York (2017)

Yin, Z., Zhang, T., Müller, A., Liu, H., Wei, Y., Schmidt, B., Liu, W.: Efficient parallel sort on avx-512-based multi-core and many-core architectures. In: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 168–176. Zhangjiajie, China (2019)

Yin, S., Wang, Q., Hao, R., Zhou, T., Mei, S., Liu, J.: Optimizing irregular-shaped matrix–matrix multiplication on multi-core dsps. In: 2022 IEEE International Conference on Cluster Computing (CLUSTER), pp. 451–461. Heidelberg, Germany (2022)