Contents

1 About This Document.............................................................................................................1

2 Multi-core and NUMA............................................................................................................ 4
  2.1 Single-core CPU, Hyper-threading, and SMP................................................................. 4
  2.2 NUMA Structure.......................................................................................................... 5
  2.3 Affinity and Core Binding........................................................................................... 6

3 Pipeline......................................................................................................................................8
  3.1 Overview....................................................................................................................... 8
  3.2 Three-Level Pipeline................................................................................................... 9
  3.3 Kunpeng Pipelines..................................................................................................... 10
  3.4 Pipeline Orchestration Acceleration Practices.......................................................... 14
  3.5 Summary.................................................................................................................... 15

4 Cache and Prefetch...............................................................................................................16
  4.1 CPU Cache................................................................................................................ 16
  4.2 Cacheline................................................................................................................... 20
  4.3 Prefetch..................................................................................................................... 24

5 SIMD Programming...............................................................................................................27
  5.1 SIMD Instruction Set................................................................................................ 27
  5.2 Development of the x86 SIMD Instruction Set......................................................... 28
  5.3 Development of the SIMD Instruction Set in the ARM Architecture....................... 28
  5.4 NEON Assembly Programming................................................................................ 29
  5.4.1 Introduction........................................................................................................ 29
  5.4.2 NEON Instructions............................................................................................. 30
  5.4.3 Inline Assembly Implementation......................................................................... 30
  5.5 NEON Intrinsics Programming................................................................................ 31
    5.5.1 Introduction...................................................................................................... 31
    5.5.2 Data Types....................................................................................................... 32
    5.5.3 NEON Intrinsics Interfaces............................................................................. 32
    5.5.4 Intrinsics Implementation............................................................................... 32

6 Compiler.................................................................................................................................. 35
  6.1 Introduction................................................................................................................ 35
  6.2 Principles..................................................................................................................... 35
6.3 Kunpeng Compiler................................................................................................................................................................ 36
6.4 Compiler Optimization Methods..................................................................................................................................... 36

7 Kunpeng BoostKit Library................................................................................................................................................. 46
7.1 Overview.................................................................................................................................................................................. 46
7.2 Introduction to the Kunpeng BoostKit Library............................................................................................................46
7.3 Application Scenarios.......................................................................................................................................................... 49
7.4 Kunpeng Math Library........................................................................................................................................................ 49
7.4.1 Introduction to KML......................................................................................................................................................... 49
7.4.2 KML Application Cases.................................................................................................................................................... 49
7.4.2.1 Hotspot Analysis.............................................................................................................................................................50
7.4.2.2 KML Installation and Usage....................................................................................................................................... 50
7.4.2.3 Verification....................................................................................................................................................................... 51

A Change History....................................................................................................................................................................... 52
This document is intended for Kunpeng native application developers. Based on the features of the Kunpeng 920 processor, this document describes how to optimize code performance during Kunpeng native application development from multiple dimensions.

First, the Kunpeng processor is briefly introduced. The Kunpeng processor is an ARM-based enterprise-level processor. In terms of the general-purpose computing processor, Huawei released the first ARM-based 64-bit Kunpeng 912 processor in 2014, and the Kunpeng 916 processor released in 2016 is the industry's first ARM-based processor that supports multi-socket interconnection. The third-generation Kunpeng 920 processor released in January 2019 is the industry's first ARM-based processor at data center level that uses the 7 nm process.

Figure 1-1 shows the evolution roadmap of the Huawei Kunpeng processor family.
This document describes the Kunpeng programming optimization methods in terms of hardware and software. Figure 1-2 shows the programming optimization framework.

**Figure 1-2 Programming optimization framework**

At the hardware level, the optimization and tuning methods are described based on the features of the Kunpeng processor.

The Kunpeng 920 processor uses the non-uniform memory access (NUMA) architecture, which solves the restriction of the symmetrical multiprocessing (SMP) technology on the number of CPU cores. Therefore, multi-core is one of major advantages of the Kunpeng processor. However, when programs run concurrently, cross-NUMA memory access hinders program performance. This document describes some programming methods for improving NUMA affinity of programs in **2 Multi-core and NUMA**.

The SoC of the Kunpeng 920 processor uses the TaiShan V110 core developed by Huawei HiSilicon. For a single processor core, the instruction time parallel technology, that is, instruction pipeline, plays a leading role in performance improvement. The Kunpeng 920 processor supports multi-level pipelines. **3 Pipeline** describes how to optimize pipeline orchestration to improve the throughput and efficiency of the pipeline and give full play to the processor...
performance. In **4 Cache and Prefetch**, based on the L3 cache of the Kunpeng 920 processor, this document introduces some program data structure arrangement and access methods, which can significantly improve the cache hit ratio.

At the software level, instruction sets, compilers, and acceleration libraries are introduced.

The Kunpeng 920 processor is fully compatible with the Armv8-A instruction set. For details about how to use the NEON instructions, see **5 SIMD Programming**. After the code development is complete, the compiler needs to translate the code into executable files that adapt to different platforms. This document describes how to use the features of the compiler to optimize code in **6 Compiler**.

In addition to the preceding common methods, Huawei releases the Kunpeng BoostKit, a full-stack optimization solution for Kunpeng hardware, base software, and application software. It provides high-performance open source components, basic acceleration software packages, and application acceleration software packages to enable optimal application performance. In **7 Kunpeng BoostKit Library**, this document describes how to use the Kunpeng basic acceleration libraries.
# 2 Multi-core and NUMA

## 2.1 Single-core CPU, Hyper-threading, and SMP

In the early stage of computer science, most computers were equipped with a chip on the mainboard, which is called a microprocessor or a single-core CPU. These processors communicate with other components on the mainboard through a connector or socket. It is very inefficient for processors to communicate with each other by using a system bus. A performance bottleneck often occurs, and the computing capability of a CPU cannot be maximized.

To deal with it, the hyper-threading technology is developed. The hyper-threading technology copies CPU units (such as registers or the L1 cache) to the same processor, so that two execution threads can share data. This mode accelerates multi-process execution and provides higher overall performance than conventional single-core mode (hyper-threading disabled).

With the rapid development of informatization and intelligence in modern society, more and more devices are connected to the Internet, Internet of Things (IoT), and Internet of Vehicles (IoV), resulting in huge computing requirements. The multi-core architecture is an inevitable trend. The symmetric multi-processing (SMP) technology uses the symmetric multi-processor structure. Each processor is equal and has the same permissions to use the memory. Any program, process, or thread can be allocated to any processor. With the support of the OS, perfect load balancing can be achieved, which greatly improves the performance and throughput of the entire system. However, because multiple cores use a same bus to access the memory, as the core number increases, memory access conflicts increase rapidly, and the bus becomes a bottleneck, which restricts scalability and performance of the system.
2.2 NUMA Structure

The SMP centralized shared memory limits the memory access frequency of the processor. Therefore, the processor may be frequently in data access starvation. To better solve this problem, the NUMA architecture is developed.

In the NUMA architecture, multiple cores are bound into a node, and each node could be considered as a symmetric multiprocessor. Nodes of a CPU communicate with each other through the on-chip network, and different CPUs communicate with each other through Hydra interfaces with high bandwidth and low latency. In the NUMA architecture, the entire memory space is physically distributed, and a set of all these dual in-line memory modules (DIMMs) is the global memory of the entire system. The memory access time of each core depends on the location of the memory relative to the processor. The access to the local memory (on the local node) is faster. The Linux kernel supports the NUMA architecture since version 2.5. The current OSs also provide various tools and interfaces to help optimize and configure the nearest memory access. The Kunpeng processor supports the NUMA architecture. Therefore, a computer system implemented by using the Kunpeng processor can achieve good performance and resolve bus bottlenecks in the SMP architecture through performance tuning, providing stronger multi-core scalability and better and more flexible computing capability.
2.3 Affinity and Core Binding

Affinity is the tendency of a process to run on a specified CPU as long as possible without being scheduled to other processors. On a multi-core server, each CPU has its own cache that stores the information used by processes. A process may be scheduled by the OS to other CPUs. As a result, the CPU cache hit ratio is low. After the CPU is bound, the process keeps running on the specified CPU, and the OS does not schedule the process to other CPUs. In this way, the CPU cache hit ratio is greatly improved, and thus the performance is improved.

1. Run the shell command to bind the running task to a NUMA node and CPU.
   The numactl command is a manual optimization command provided by Linux. It can be used to specify a process to run on a NUMA node or a specific CPU core.
   a. Bind a NUMA node: `numactl --cpubind=0 --membind=0 java SIMDTest_Compare_Max2`
   b. Bind a CPU core: `numactl -C 0-19 --membind=0 java SIMDTest_Compare_Max2`
   c. Run the top command to check whether CPU core binding is successful. The output also shows the process to which the core is bound.

2. Specify a CPU core by calling the system APIs in the program code.
   The sched_getaffinity interface in the glibc library is used to obtain the current CPU affinity of the application. The sched_setaffinity interface can be used to bind the application to one or more CPU cores.

   The API syntax is as follows:
   ```c
   #include <sched.h>
   int sched_setaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask);
   int sched_getaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask);
   void CPU_CLR(int cpu, cpu_set_t *set);
   int CPU_ISSET(int cpu, cpu_set_t *set);
   void CPU_SET(int cpu, cpu_set_t *set);
   void CPU_ZERO(cpu_set_t *set);
   ```

   The following is an example of core binding:
   ```c
   #include <sched.h>
   cpu_set_t cpu_mask;
   memset((VOS_VOID *)(&cpu_mask), 0, sizeof(cpu_mask));
   cpu_mask.__bits[0] = 1 << 0;
   (VOS_VOID)sched_setaffinity(0, sizeof(cpu_mask), &cpu_mask);
   ```

3. Bind application software to cores.
   The Kunpeng 920 processor provides two super CPU clusters (SCCLs). Each SCCL contains six to eight CPU clusters, and each CPU cluster contains four cores. When binding CPUs to a KVM, you are advised to use CPUs across multiple CPU clusters to improve the KVM performance. This method can reduce bandwidth bottlenecks between the L3 cache and memory caused by core contention in the same CPU cluster.

   a. Query the NUMA node information and topology in the Linux system.
      ```bash
      numactl -H
      ```
b. Edit the VM XML configuration file in the Linux system and bind vCPUs to cores in as many CPU clusters as possible. The following is an example:

```xml
<domain type="KVM">
  ...
  <vcpu placement='static' cpuset='0,1,4,5,8,9,12,13'>8</vcpu>
  <cputune>
    <vcpupin vcpu='0' cpuset='0'/>
    <vcpupin vcpu='1' cpuset='1'/>
    <vcpupin vcpu='2' cpuset='4'/>
    <vcpupin vcpu='3' cpuset='5'/>
    <vcpupin vcpu='4' cpuset='8'/>
    <vcpupin vcpu='5' cpuset='9'/>
    <vcpupin vcpu='6' cpuset='12'/>
    <vcpupin vcpu='7' cpuset='13'/>
  </cputune>
  ...
</domain>
```

**NOTE**

(0, 1), (4, 5), (8, 9), and (12, 13) are in different CPU clusters.
3 Pipeline

3.1 Overview

The system performance varies according to application scenarios. In computing-intensive scenarios, performance optimization aims to enable CPUs to process more data per unit time. In scenarios with a large number of database accesses, performance optimization aims to improve I/O efficiency, reduce I/O waits, and allow more data to be read and written per unit time. In scenarios with heavy network traffic, performance optimization aims to improve the network throughput and reduce the delay, that is, increase the quantity of data packets sent or received per unit time. Although the objectives in different scenarios are different, they are all to improve a certain capability of the system per unit time. For the CPU, performance optimization is reflected in the instruction execution efficiency of the CPU. The objective of performance optimization is to increase the number of instructions executed by the CPU per unit time, that is, instructions per cycle (IPC).

When an instruction is executed in a CPU, the instruction passes through many CPU components, and each component performs its own function and is functionally independent of each other. However, a downstream component generally depends on the execution result of an upstream component. Such flow is similar to a production line. Therefore, the entire process of instruction execution is called an instruction pipeline. The number of levels (lengths) of CPU pipelines in different architectures varies greatly, ranging from several levels to dozens of levels. A pipeline with more levels indicates a complex CPU structure, more powerful functions, and high power consumption. Conversely, a pipeline with fewer levels indicates a simple CPU structure and low power consumption. The following table lists some typical ARM pipeline levels.
## Table 3-1 Pipeline levels

<table>
<thead>
<tr>
<th>Model</th>
<th>Instruction Set</th>
<th>Pipeline Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM 7</td>
<td>Armv4</td>
<td>3</td>
</tr>
<tr>
<td>ARM 9</td>
<td>Armv5</td>
<td>5</td>
</tr>
<tr>
<td>ARM 11</td>
<td>Armv6</td>
<td>8</td>
</tr>
<tr>
<td>Cortex-A8</td>
<td>Armv7-A</td>
<td>13</td>
</tr>
<tr>
<td>Kunpeng 920/Cortex-A55</td>
<td>Armv8</td>
<td>8</td>
</tr>
</tbody>
</table>

### 3.2 Three-Level Pipeline

A three-level instruction pipeline includes three parts: instruction fetching, decoding, and execution. Instruction fetching and decoding can be completed in one cycle. However, the execution unit completes a large amount of work, including read/write operations on operand-related registers and memory, ALU operations, and data transmission between related components. As a result, the execution unit may need to occupy multiple clock cycles, which blocks execution of other instructions on the pipeline and becomes a performance bottleneck of the system.

![Figure 3-1 Ideal three-level pipeline](image)

From T₁, one instruction is executed in each cycle, that is, IPC=1. This is the most efficient instruction execution mode. However, not all instructions are single-cycle...
instructions. For example, the LDR instruction for accessing the memory is a non-single-cycle instruction. This instruction interrupts the execution of the pipeline, as shown in Figure 3-2.

**Figure 3-2 Non-single-cycle instruction pipeline**

The LDR instruction is used to load data from the memory to a register. The execution of this instruction occupies three cycles from T3 to T5. At T3, the signal control cable is occupied to calculate the address of the memory. However, the decoding process also needs the cable. Therefore, at T3, the decoding operation of the first SUB cannot be performed, and the instruction fetch operation of the second SUB is not affected. At T4, the LDR instruction is accessing the memory, and the signal control cable is released. Therefore, the decoding operation of the first SUB is performed, and the decoding operation of the second SUB cannot be performed in this cycle. Because the von Neumann architecture is used, data and instructions share the same memory, and instructions cannot be read when the memory is accessed. Therefore, the instruction fetch operation of the MOV instruction is interrupted. At T5, data is fetched from the memory and stored in the register of the CPU. This operation occupies the execution unit. The first SUB instruction is interrupted, and the second SUB instruction enters the decoder for decoding. The instruction fetch operation of the MOV instruction is also performed.

Only one instruction is executed in the three cycles from T3 to T5. In general, six instructions are executed in eight cycles from T1 to T8, that is, IPC=0.75. According to the preceding instruction execution process, the multi-cycle LDR instruction occupies three clock cycles in the execution phase. As a result, the execution of subsequent instructions is blocked and the pipeline efficiency is hindered.

### 3.3 Kunpeng Pipelines

Execution of the three-level pipeline can be easily interrupted, resulting in low instruction execution efficiency. A subsequently developed five-level instruction
pipeline technology is considered as a classic processor configuration, and has been widely used in a plurality of RISC processors. On a basis of the three-level pipeline (instruction fetching, decoding, and execution), two levels is added. The execution phase is further divided into execution, memory access, and write-back, which resolves the memory access instruction delay in the instruction execution phase of a three-level pipeline. However, new problems such as register interlocking may occur, which causes pipeline interruption. The Kunpeng 920 processor uses an eight-level pipeline structure. It first fetches instructions, and then decodes instructions, renames registers, and schedules the instructions. Once the scheduling is complete, instructions are sent to one of the eight execution pipes in an unordered manner. Each execution pipe can receive and complete an instruction in each cycle. Finally, the memory access and write-back operations are performed.

**Figure 3-3** Kunpeng pipeline structure

Kunpeng execution pipes support multiple operation instructions, as shown in **Table 3-2**.

<table>
<thead>
<tr>
<th><strong>Execution Pipe (Mnemonic)</strong></th>
<th><strong>Function</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU1 (ALU)</td>
<td>Integer operation</td>
</tr>
<tr>
<td>ALU2/3/BRU1/2 (ALU/BRU)</td>
<td>Integer operation, branch jump</td>
</tr>
<tr>
<td>Multi-cycle (MDU)</td>
<td>Integer shift, multiplication, division, and CRC operations</td>
</tr>
<tr>
<td>LoadStore 0/1 (LS)</td>
<td>Access operation</td>
</tr>
<tr>
<td>FP/ASIMD 1 (FSU1)</td>
<td>ASIMD ALU, ASIMD misc, ASIMD integer multiply, FP convert, FP misc, FP add, FP multiply, FP divide, crypto uops, Hivector</td>
</tr>
<tr>
<td>FP/ASIMD 2 (FSU2)</td>
<td>ASIMD ALU, ASIMD misc, FP convert, FP misc, FP add, FP multiply, FP sqrt, ASIMD shift uops, Hivector</td>
</tr>
</tbody>
</table>
No matter how perfect the pipeline design is, an interrupt is inevitable, which affects the execution efficiency of the pipeline. The following describes several common pipeline interrupts and solutions.

**Branch Jump**

Some statements that interrupt program execution in sequence are often used in code. For example, statements such as if, switch, for, while, and return interrupt code execution in sequence. After compilation, the compiler changes the statements to jump instructions such as B, BL, BX, and BLX. These jump instructions block the execution of the pipeline, as shown in Figure 3-4.

**Figure 3-4 Pipeline interruption caused by branch jump**

When the BL instruction is executed, the following two instructions ADD and SUB have entered the pipeline but cannot be executed. After the BL instruction is executed, the PC pointer is adjusted based on the calculated target address, and then the instruction is fetched from the address and enters the pipeline for execution. The ADD and SUB instructions that previously enter the pipeline are cleared, and the pipeline that is originally in the decoding and instruction fetch state is blocked.

Current processors use the branch prediction technology to reduce the impact of jump instructions on the pipeline. It includes static prediction and dynamic
prediction. Static prediction is processed at compile time. As for dynamic prediction, in the instruction fetch phase of code execution, the most possible location of the next instruction to be fetched is pre-determined within a range. By doing so, instructions are fetched following the code execution sequence, instead of following the storage sequence of the program instructions in the memory.

Program developers can use the static prediction method to improve the accuracy of branch prediction. For example, the Linux kernel source code shows that the likely() and unlikely() macros are defined in the Linux kernel.

```c
# ifndef likely
# define likely(x) (__builtin_expect(!!(x), 1))
# endif

# ifndef unlikely
# define unlikely(x) (__builtin_expect(!!(x), 0))
# endif
```

__builtin_expect is a preprocessing command provided by GCC. __builtin_expect((x), 1) indicates that the value of x is likely to be true, and __builtin_expect((x), 0) indicates that the value of x is likely to be false. That is, likely() tends to execute the statement after if, and unlikely() tends to execute the statement after else. In this way, the compiler closely follows code with a higher probability after the branch jump statement to increase the instruction cache hit ratio and improve pipeline execution efficiency. The following code is an example:

```c
if (likely(sem->count > 0))
    sem->count--;
else
    __down(sem);
```

However, many code branches are closely related to service running scenarios and cannot be manually determined. To solve this problem, the common practice in the industry is as follows: During source code compilation, the compiler automatically adds variables to code branches and runs the compiled program in the real or simulated environment for a long time. In this process, the statistics on the number of operations that each code branch is entered and the proportion of each branch is collected and fed back to the compiler. Then the compiler rearranges the code and places pieces of code with a high execution probability after branch jump statements to reduce pipeline blocking, thereby improving pipeline efficiency and performance.

**Processor Interrupts**

During system running, an interrupt may be generated at any time and is irrelevant to the current command. When an interrupt is generated, the processor does not interrupt the instruction that is being executed. Instead, the processor responds to the interrupt after executing the instruction, as shown in Figure 3-5 (taking the three-level pipeline as an example).
If an interrupt is generated when the ADD instruction is being executed, the processor automatically jumps to the exception interrupt vector table 0x18 (B target address) starting from address 0, saves the interrupt site, and then jumps to the interrupt handler (SUB) to process the interrupt. In this process, two jumps occur, and an interrupt is returned, which wastes a lot of CPU cycles and greatly reduces the pipeline efficiency.

Interrupts are unpredictable. Therefore, the compiler cannot rearrange the code to optimize the pipeline, but interrupts can be manually intervened. For example, in some scenarios, you can modify kernel parameters to reduce the number of interrupts by combining interrupts. Alternatively, bind interrupts to a core and use a specific CPU core to process the interrupts. The core where the service process is located can be more focused on processing service data without being interrupted.

3.4 Pipeline Orchestration Acceleration Practices

The following uses several examples to describe pipeline orchestration.

**Eliminating Data Dependency**

ADD x1, x2, x3
SUB x4, x1, x5

In the preceding code, the calculation of the SUB instruction depends on the result of the x1 register in the ADD instruction. Therefore, insert an empty instruction before the SUB instruction to wait for the SUB instruction to write the calculation result to the register file. After `-mtune=tsv110` is added, the compiler adds two irrelevant instructions to eliminate data risks:

ADD x1, x2, x3
Instruction 1
Eliminating Structure Dependency

ADD x1, x2, x3
SUB x4, x5, x6
LDR x6, [x7, #12]
SDR x6, [x7, #12]

In the preceding code, the ADD and SUB instructions need to use ALU resources at the same time, and the LDR and STR instructions need to use MEM resources at the same time. Structure risks occur when the hardware cannot support the execution of multiple instructions at the same time. After `-mtune=tsv110` is added, the compiler adds two irrelevant instructions to eliminate structure dependency.

ADD x1, x2, x3
Instruction 1
SUB x4, x5, x6
Instruction 2
LDR x6, [x7, #12]
Instruction 3
SDR x6, [x7, #12]

3.5 Summary

When writing code, consider not only the optimization of assembly instructions, but also the impact of pipeline orchestration on performance. During pipeline orchestration, minimize interlocking. CPUs can work well in most cases, and the compiler has been optimized for out-of-order processors for nearly two decades. When instructions and data are executed in sequence, CPUs can yield the best performance.

So, use simple code. Simple code helps the optimization engine of the compiler identify and optimize the code. Try not to use jump instructions. When you have to, try to jump to the same direction each time. Complex designs, such as dynamic jump tables, do have powerful functions, but neither the processor nor the compiler can perform good prediction processing. Therefore, complex code is likely to cause pipeline pauses and guess errors. As a result, the performance of the processor is greatly affected.

Second, use simple data structures. Keeping data sequential, contiguous, and continuous prevents data pauses. Using the correct data structure and data distribution can greatly improve performance. As long as the code and data structures are as simple as possible, you can leave the rest of the work to the compiler’s optimization engine.
4 Cache and Prefetch

4.1 CPU Cache

The CPU reads data from the memory, stores the data in the register, and then performs computing. With development of computers, the access speed difference between the memory and register becomes larger, and a program has high overheads when reading data from the memory, affecting overall performance of the computer. To solve this problem, the CPU cache is added between memory and registers. Because the access speed of reading data from the CPU cache by the register is faster than that of directly reading data from the memory, adding the CPU cache can effectively improve the overall performance of the computer. Currently, the CPU cache has three layers: L1, L2, and L3. 4.1 CPU Cache shows the storage hierarchy.

Figure 4-1 CPU cache

<table>
<thead>
<tr>
<th>CPU Core</th>
<th>CPU Core</th>
<th>CPU Core</th>
<th>CPU Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1i</td>
<td>L1d</td>
<td>L1i</td>
<td>L1d</td>
</tr>
<tr>
<td>L1i</td>
<td>L1d</td>
<td>L1i</td>
<td>L1d</td>
</tr>
<tr>
<td>L2</td>
<td>L2</td>
<td>L2</td>
<td>L2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>L3</td>
</tr>
</tbody>
</table>

L1 is the CPU cache layer closest to the register. It has the fastest access speed among L1 to L3, but its capacity is relatively small. In addition, different from L2 and L3, L1 is divided into an instruction cache and a data cache. The instruction cache is dedicated to storing instructions, and the data cache is dedicated to storing data. The instruction cache and the data cache are not mixed. This design is based on the fact that L1 is closest to the register and has the smallest capacity. If no division is made, instructions may be read to L1 and then be replaced by data to be operated. After the instructions are replaced, the instructions need to be obtained from L2, which is not desirable.
The access speed of L2 is slower than that of L1, but the L2 capacity is larger. L2 is not divided into the two caches. Note that each CPU core has its own L1 cache and L2 cache.

L3 has the largest capacity but also the slowest access speed among the three layers. Different from L1 and L2, L3 is shared by multiple CPU cores.

When the CPU needs data to perform an operation, the CPU first searches for the data from L1. If the data cannot be found, the CPU reads the data from L2 and L3 in sequence. If no data is read, the data is read from the memory. This case is called a cache miss.

Kunpeng and x86 servers use the three-layer cache. However, the cache size and access speed of each layer are different. For program development, if data can be accessed in L1, the program performance can be improved. Sound array access sequence, structure member alignment, and structure layout optimization are effective methods to improve program performance based on the CPU cache principle.

### Array Access Sequence

Arrays are continuous data and are stored continuously in the memory. If arrays are accessed based on the storage sequence of the arrays in the memory, the cache hit ratio can be effectively improved, thereby improving performance. The following uses two-dimensional arrays as an example. The arrays are traversed. When the two-dimensional array B is added to the array A by column, the column elements are inconsecutive in the cache. Due to the size limitation of L1, after an element is obtained and calculated at L1, the element in the next column cannot be obtained at L1. In the next calculation, data needs to be read from L2, L3, or even the memory. The performance is relatively poor. After the array access sequence is adjusted by adding row elements of two-dimensional arrays A and B, data can be continuously read from the cache. It takes 235,722 μs after rearrangement, reduced from the previous 544,939 μs.

Read by column:

```c
for (int i = 0; i < ARRAYLEN; i++) {
    for (int j = 0; j < ARRAYLEN; j++) {
        arrayC[i][j] = arrayA[i][j] + arrayB[j][i];
    }
}
```

Read by row:

```c
for (int i = 0; i < ARRAYLEN; i++) {
    for (int j = 0; j < ARRAYLEN; j++) {
        array[i][j] = arrayA[i][j] + arrayB[i][j];
    }
}
```

### Structure Member Alignment

The principle for aligning data members of a structure is as follows: The first member is placed at the position where the offset is 0, and the start position for storing each subsequent member must be an integer multiple of the size of the first member. See the following example. The total size of the structure, that is, the result of `sizeof`, must be an integer multiple of the maximum member in the structure. If the size is insufficient, supplement it.
struct Test1{
    char a;
    double b;
    int c;
    short d;
} Test1;

According to the principle, the storage position in the memory for Test1 is shown in Figure 4-2. The memory positions occupied by the structure are 0 to 23. However, the memory positions of 1 to 7, 22, and 23 are padded.

**Figure 4-2** Structure memory distribution (before adjustment)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th></th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>8</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>17</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>18</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>19</td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>20</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>21</td>
<td>13</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>22</td>
<td>14</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>23</td>
<td>15</td>
<td>23</td>
</tr>
</tbody>
</table>

There is another structure Test2. Test1 and Test2 have related functions, but the sequence of Test2 member variables is adjusted.

struct Test2{
    double b;
    int c;
    short d;
    char a;
} Test2;

According to the structure member alignment principle, the storage position in the memory for Test2 is shown in Figure 4-3. The memory positions occupied by the structure are 0 to 15, and only the position 15 is padded.
According to the preceding example, Test2 saves one third of the memory space compared with Test1 for implementing the same functions during programming. This greatly improves the memory utilization and data compactness, facilitates L1 cache hit, and improves the performance.

**Structure Layout Design**

For the structure layout, various programming layouts bring different effects. The following describes two data organization modes: structure array and array structure. The structure array is defined as follows:

Structure array:

```c
struct Array{
    int x;
    int y;
};
```

```c
struct Array stArray[N];
```

In this case, each group of `x` and `y` is stored continuously. The storage format in the memory is as follows:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>y0</td>
<td>x1</td>
<td>y1</td>
</tr>
<tr>
<td>x2</td>
<td>y2</td>
<td>x3</td>
<td>y3</td>
</tr>
</tbody>
</table>

The array structure is defined as follows:

Array structure:

```c
struct Array{
    int x[N];
    int y[N];
};
```

```c
struct Array stArray;
```
In this case, \( x \) and \( y \) are stored separately. The storage format in the memory is as follows:

<table>
<thead>
<tr>
<th></th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>y0</th>
<th>y1</th>
<th>y2</th>
<th>y3</th>
</tr>
</thead>
</table>

When operations mainly involve \( x \) in the service scenario, compared with the structure array, the data loaded to the memory and cache from \( x \) in the array structure is continuous, which can improve the cacheline validity and cache hit ratio, thereby improving the performance. Therefore, you are advised to design an appropriate structure layout based on the actual scenario during programming.

### 4.2 Cacheline

When the CPU reads data, it does not read data byte by byte. Instead, it reads data by cacheline. The CPU identifies whether the data in the cache is valid data in the unit of cacheline instead of memory bit width. This mechanism may cause false sharing, which reduces the CPU cache hit ratio. The common cause of false sharing is that frequently accessed data is not aligned based on the cacheline size.

The cache space is divided into different cachelines, as shown in Figure 4-4. Although `readHighFreq` is not modified and is in the cache, it is read from the memory when false sharing occurs.

**Figure 4-4 Cache space**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>128B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128B</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>readHighFreq</code></td>
<td><code>writeHighFreq</code></td>
<td><code>writeHighFreq</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For example, two variables are defined in the following code. The two variables are in the same cacheline, and the cache reads the variables at the same time.

```c
int readHighFreq, writeHighFreq
```

`readHighFreq` is a variable with a high read frequency, and `writeHighFreq` is a variable with a high write frequency. After `writeHighFreq` is modified in a CPU core, data of a cacheline length corresponding to the cache is identified as invalid, that is, `readHighFreq` is identified as invalid data by the CPU core. Although `readHighFreq` is not modified, when the CPU accesses `readHighFreq`, the data is still imported from the memory. As a result, false sharing occurs and the performance deteriorates.

The cacheline size of the Kunpeng 920 processor is different from that of the x86 processor. As a result, the performance of an optimized program on x86 may be
low when the program runs on Kunpeng 920. In this case, you need to modify the memory alignment size of the service code. The cacheline size of the x86 L3 is 64 bytes, and the cacheline size of Kunpeng 920 is 128 bytes.

### Cacheline Alignment Programming

Data that is frequently read and written needs to be aligned based on the cacheline size. There are two modification methods: dynamic memory application and padding.

The alignment method for dynamically applying for memory is as follows:

```c
int posix_memalign(void **memptr, size_t alignment, size_t size)
```

When the `posix_memalign` function is successfully called, the dynamic memory of `size` bytes is returned, and the start address of the memory is a multiple of `alignment`.

Local variables can be padded as follows:

```c
int writeHighFreq;
char pad[CACHE_LINE_SIZE - sizeof(int)];
```

In the code, `CACHE_LINE_SIZE` indicates the size of the cacheline on the server. The `pad` variable is used to fill the remaining space of the `writeHighFreq` variable. The sum of the two values is the size of the cacheline.

The MySQL source code has made much 64-byte cacheline alignment for the x86 platform. The L3 cacheline of the Kunpeng 920 processor is 128 bytes. Therefore, the alignment mode for the MySQL source code needs to be changed to 128 bytes. The TPM is improved by 3% to 4% after the cacheline of the MySQL data structure is changed to 128 bytes.

- `brt_search_latches` btr_search_sys
- `ReadView::m_view_list`
- `trx_sys_t::rw_trx_list`
- `trx_sys_t::mysql_trx_list`
- `trx_sys_t::rsegs`
- `srv_conc_t::n_active`
- `srv_conc_t::n_active`
- `lock_sys_t::mutex`
- `lock_sys_t::wait_mutex`

### Cacheline Alignment Implementation

Code before cacheline alignment:

```c
#define TIME_S 99999999
#define NUM_THREADS 4

struct foo {
    int x;
    int y;
};

static struct foo f;
```
static struct foo testf;

/* The two following functions are running concurrently: */

void *inc_b1(void)
{
    cpu_set_t mask;
    cpu_set_t get;
    CPU_ZERO(&mask);
    CPU_SET(0, &mask);
    if(sched_setaffinity(0, sizeof(mask), &mask) == -1){
        printf("warning: could not set CPU affinity, continuing...
");
    }
    CPU_ZERO(&get);
    if(sched_getaffinity(0, sizeof(get), &get) == -1){
        printf("warning: could not get thread affinity, continuing...
");
    }
    if(CPU_ISSET(0, &get)){
        printf("inc_b1 is running in %d\n", get);
    }
    for (int i = 0; i < TIME_S; ++i)
        ++testf.y;
}

int *sum_a1(void)
{
    cpu_set_t mask;
    cpu_set_t get;
    CPU_ZERO(&mask);
    CPU_SET(1, &mask);
    if(sched_setaffinity(0, sizeof(mask), &mask) == -1){
        printf("warning: could not set CPU affinity, continuing...
");
    }
    CPU_ZERO(&get);
    if(sched_getaffinity(0, sizeof(get), &get) == -1){
        printf("warning: could not get thread affinity, continuing...
");
    }
    if(CPU_ISSET(1, &get)){
        printf("sum_a1 is running in %d\n", get);
    }
    int s = 0;
    for (int i = 0; i < TIME_S; ++i)
        s += testf.x;
    return s;
}

int *sum_a(void)
{
    cpu_set_t mask;
    cpu_set_t get;
    CPU_ZERO(&mask);
    CPU_SET(2, &mask);
    if(sched_setaffinity(0, sizeof(mask), &mask) == -1){
        printf("warning: could not set CPU affinity, continuing...
");
    }
    CPU_ZERO(&get);
    if(sched_getaffinity(0, sizeof(get), &get) == -1){
        printf("warning: could not get thread affinity, continuing...
");
    }
    if(CPU_ISSET(2, &get)){
        printf("sum_a is running in %d\n", get);
    }
    int s = 0;
}
for (int i = 0; i < TIME_S; ++i)
    s += f.x;
return s;
}

void *inc_b(void)
{
    cpu_set_t mask;
    cpu_set_t get;
    CPU_ZERO(&mask);
    CPU_SET(3, &mask);
    if(sched_setaffinity(0, sizeof(mask), &mask) == -1){
        printf("warning: could not set CPU affinity, continuing..\n");
    }
    CPU_ZERO(&get);
    if(sched_getaffinity(0, sizeof(get), &get) == -1){
        printf("warning: could not get thread affinity, continuing.. \n");
    }
    if(CPU_ISSET(3, &get)){
        printf("inc_b is running in %d\n", get);
    }
    for (int i = 0; i < TIME_S; ++i)
        ++f.y;
}

int main(){
    int ret = 0;
    pthread_t tids[NUM_THREADS];
    printf("start the threads\n");
    ret = pthread_create(&tids[0], NULL, sum_a, NULL);
    if(ret != 0){
        printf("pthread_create error: error_code = %d\n", ret);
    }
    ret = pthread_create(&tids[1], NULL, inc_b, NULL);
    if(ret != 0){
        printf("pthread_create error: error_code = %d\n", ret);
    }
    ret = pthread_create(&tids[2], NULL, sum_a1, NULL);
    if(ret != 0){
        printf("pthread_create error: error_code = %d\n", ret);
    }
    ret = pthread_create(&tids[3], NULL, inc_b1, NULL);
    if(ret != 0){
        printf("pthread_create error: error_code = %d\n", ret);
    }
    pthread_join(tids[0], NULL);
    return 0;
}

The execution time is 2.955s.

New code based on the cacheline alignment principle:

The foo structure is adjusted. The x and y member variables are separated in two different cachelines to avoid false sharing. The execution time is shortened to 2.248s.

struct foo {
    int x;
    char padx[124];
    int y;
    char pady[124];
};
4.3 Prefetch

For data stored in the memory, the CPU needs to first fetch data from the memory to L3, then from L3 to L2 and L2 to L1. Finally, the data in L1 is fetched to the register. Then, the CPU can process the data. If the data to be processed next time by the CPU is in L1, the performance of the program is improved. Prefetch can be classified into hardware prefetch and software prefetch. Hardware fetches a possible memory access unit into the cache in advance according to historical information of memory access, so that the cache does not fail when data is needed. Hardware prefetch is universal. Software prefetch means that a programmer prefetches a specific location by writing a prefetch instruction in service code. Software prefetch is specific.

This document describes only the programming based on the Kunpeng platform. Therefore, only software prefetch is described.

Software Prefetch

Software prefetch is implemented by using a prefetch instruction, and prefetch instructions provided by architectures are different. On the Kunpeng platform, the format of the prefetch instruction is as follows:

```
PRFM prfop, [Xn|SP{, #pimm}]
```

- **prfop** consists of **type**, **target**, and **policy**.
  - The options of **type** are as follows:
    - **PLD**: data preload
    - **PLI**: instruction prefetch
    - **PST**: data pre-storage
  - The options of **target** are as follows:
    - **L1**
    - **L2**
    - **L3**
  - Operations are performed on the specified cache layer.
  - The options of **policy** are as follows:
    - **KEEP**: Data is stored for a certain period of time after being prefetched. This policy applies to scenarios where data is used for multiple times.
    - **STRM**: streaming or non-temporary prefetch. This policy applies to scenarios where data is used only once and will be eliminated after being used.
    - **Xn|SP**: a 64-bit general register or stack pointer, which is usually the prefetch start address.
    - **pimm**: offset in bytes. It indicates the length of the bytes to be prefetched. The value is an integer multiple of 8 and ranges from 0 to 32760. The default value is 0. The prefetch data length can be set based on service requirements. You are advised to prefetch data of various lengths to obtain the optimal value.
In terms of instruction composition, the core part of the prefetch instruction is prfop, which determines the prefetch type, prefetch cache level, and usage of prefetched data. This section describes PLD data prefetch. Other modes are similar. The core instructions of data prefetch and their functions are as follows:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLDL1KEEP</td>
<td>Data is prefetched to the L1 cache in keep mode and stored for a certain period of time after being used.</td>
</tr>
<tr>
<td>PLDL2KEEP</td>
<td>Data is prefetched to the L2 cache in keep mode and stored for a certain period of time after being used.</td>
</tr>
<tr>
<td>PLDL3KEEP</td>
<td>Data is prefetched to the L3 cache in keep mode and stored for a certain period of time after being used.</td>
</tr>
<tr>
<td>PLDL1STRM</td>
<td>Data is prefetched to the L1 cache in strm mode and eliminated from the cache after being used.</td>
</tr>
<tr>
<td>PLDL2STRM</td>
<td>Data is prefetched to the L2 cache in strm mode and eliminated from the cache after being used.</td>
</tr>
<tr>
<td>PLDL3STRM</td>
<td>Data is prefetched to the L3 cache in strm mode and eliminated from the cache after being used.</td>
</tr>
</tbody>
</table>

The GCC compiler has built-in functions for prefetch. The format is as follows:

```c
__builtin_prefetch (const void *addr, int rw, int locality)
```

In this function:
- `addr` indicates the memory address of the data.
- `rw` is optional and can be set to 0 or 1. The value 0 indicates the read operation, and the value 1 indicates the write operation.
- `locality` is an optional parameter, indicating the duration for which data is stored in the cache, that is, the validity period. It can be set to 0 to 3. 0 indicates that the accessed data will be eliminated from the cache and cannot be accessed later. 3 indicates that the accessed data will be accessed again. 1 and 2 indicate short and medium duration respectively. The default value is 3.

Not using software prefetch:

```c
int add_vector(int *dst, int *src1, int *src2, int size)
{
    for (int index =0 ; index < size; index += 4) {
        ... // do something
    }
}
```

Using software prefetch:

```c
```
static inline void prefetch(const void* data)
{
    __asm__ __volatile__ (
        "prfm PLDL1STRM, [%[data]] \n\t"
        :: [data] "r" (data));
}

int add_vector(int *dst, int *src1, int *src2, int size)
{
    for (int index =0 ; index < size; index += 4) {
        prefetch(src1 + 256);
        prefetch(src2 + 256);
        ... // do something
    }
}
5 SIMD Programming

5.1 SIMD Instruction Set

A single-instruction, multiple-data (SIMD) instruction set is a set of instructions for processing multiple data streams by using a single instruction. A single-instruction, single-data (SISD) instruction set is a set of instructions that can only process a single data stream by using a single instruction.

Conceptually, SIMD outperforms SISD in data-intensive computing scenarios. Take the addition instruction as an example, as shown in Figure 5-1. The CPU executes one instruction: A1 + B1 = C1, and then executes the next instruction: A2 + B2 = C2. The subsequent calculations are completed in this sequence. The four addition operations need to be performed serially. When SIMD is used, the CPU needs to execute only one instruction to complete four addition calculation operations, and the four addition calculation operations are executed in parallel. In such a scenario, SIMD instruction performance is better than conventional SISD instruction performance.
The parallel execution is implemented because some dedicated vector registers are added during physical design of the CPU. The length of these registers is usually greater than that of the general-purpose register. For example, the length of the vector register of the Kunpeng processor is 128 bits, and the maximum length of the general-purpose register is 64 bits. Therefore, multiple pieces of data can be put into these registers at the same time. Note that the data types of the multiple pieces of data must be the same.

The vector registers vary based on architectures. Therefore, SIMD instruction sets supported by different architectures vary. When developing and using an SIMD instruction set on the Kunpeng platform, check whether the SIMD instruction set is supported.

### 5.2 Development of the x86 SIMD Instruction Set

Intel launched the first SIMD instruction set MMX in 1997. MMX uses 64-bit vector registers MM0 to MM7. However, these vector registers are not dedicated. They are parts of 80-bit floating-point registers ST0 to ST7.

In 1999, Intel launched the Streaming SIMD Extensions (SSE) instruction set. SSE uses independent vector registers instead of floating-point registers. In addition, the length of vector registers is increased to 128 bits. AMD developed SSE in 2001. Since then, SIMD instructions in the x86 architecture have been growing, and SIMD instruction sets such as SSE2, SSE3, SSE4, AVX, and AVX-512 have been developed.

### 5.3 Development of the SIMD Instruction Set in the ARM Architecture

The earliest SIMD instruction set supported by ARM is Armv6. The vector register on which the SIMD instruction set depends is the general register of ARM. It
supports 8-bit or 16-bit integers to implement parallel computing of four 8-bit integers or two 16-bit integers. In the Armv7-A architecture, Arm further develops its own SIMD instruction set and names it NEON. This instruction set has 32 64-bit NEON vector registers and supports single-precision floating-point operations. In the Armv8-A architecture, the NEON instruction set is further developed, and the length of each of the 32 vector registers is increased to 128 bits. The NEON instruction set supports 8-bit, 16-bit, 32-bit, and 64-bit integers as well as single-precision and double-precision floating-point operations.

Although the NEON vector registers are 128 bits long, they can be used as 32-bit Sn registers or 64-bit Dn registers. Figure 5-2 shows the usage mode.

**Figure 5-2** NEON vector registers

![NEON vector registers](image)

## 5.4 NEON Assembly Programming

### 5.4.1 Introduction

NEON assembly programming generally has two implementation manners: assembly file and inline assembly. Because inline assembly is relatively simple, most developers use this manner for NEON assembly programming.

### 5.4.2 NEON Instructions

The general NEON instruction format is as follows:

```
{<prefix>}<op>{<suffix>} Vd.<T>, Vn.<T>, Vm.<T>
```

- `{}` indicates that the parameter is optional.
- `<prefix>` is generally used to identify the data type processed by the NEON instruction. For example, `S` indicates a signed integer, `U` indicates an unsigned integer, `F` indicates a floating point number, and `P` indicates a Boolean data type.
- `<op>` indicates the operation type of the NEON instruction. For example, `ADD` indicates an addition operation.
- `<suffix>` indicates an operation behavior of the NEON instruction. For example, `P` indicates that a vector is operated by pair, `V` indicates that an operation is performed across all data channels, and `2` indicates that an operation is performed on the high-order bits of data in a wide or narrow instruction.
- `<T>` indicates the width of data processed by the NEON instruction. For example, `B` indicates an 8-bit data width, `H` indicates a 16-bit data width, `S` indicates a 32-bit data width, and `D` indicates a 64-bit data width.

The following describes two NEON instructions.
- **ADDHN2**: Two 128-bit vectors are added to generate a 64-bit vector, which is stored as the upper 64 bits of the NEON register.

- **SADDL2**: Two upper 64-bit vectors of the NEON register are added to generate a 128-bit vector.

NEON instructions can be classified into normal instructions, wide instructions, narrow instructions, saturating instructions, and long instructions based on operand types.

- Normal instruction: generates a vector of the same size and type (usually the same) as the operand vector.

- Long instruction: performs an operation on a double-word vector operand to produce a quad-word vector. The generated elements are twice the width of the operand elements and of the same type. Its flag is L, for example, VMOVL.

- Wide instruction: performs an operation on a double-word vector operand and a quad-word vector operand to generate a quad-word vector. Its flag is W, for example, VADDW.

- Narrow instruction: performs an operation on a quad-word vector operand to generate a double-word vector. The result is generally half the width of the original. Its flag is N, for example, VMOVN.

- Saturating instruction: automatically limits the data width to the range when the actual width exceeds the range specified by the data type. Its flag is Q, for example, VQSHRUN.

### 5.4.3 Inline Assembly Implementation

C language implementation:

```c
void AddFloatC(float* dst, float* src1, float* src2, int count) {
    for (int i = 0; i < count; i++) {
        dst[i] = src1[i] + src2[i];
    }
}

int main() {
    float dst[ARRAY_NUMS] = {0.0};
    float src1[ARRAY_NUMS];
    float src2[ARRAY_NUMS];

    struct timeval start;
    struct timeval end;
    double dt;

    InitArray(dst, src1, src2);
    gettimeofday(&start, NULL);
    AddFloatNeonAsm(dst, src1, src2, ARRAY_NUMS);
    gettimeofday(&end, NULL);
    dt=(end.tv_sec-start.tv_sec) * 1000+(end.tv_usec-start.tv_usec) / 1000.0;
    cout<<"Time used of Normal NEON ASM code: "<<dt<<"ms"<<endl;
    return 0;
}
```

Inline assembly implementation:
void AddFloatNeonAsm(float* dst, float* src1, float* src2, int count)
{
    __asm__ volatile(
        "1:                               
        "ld1 {v0.4s}, [%[src1]], #16 \n"
        "ld1 {v1.4s}, [%[src2]], #16 \n"
        "fadd v0.4s, v0.4s, v1.4s \n"
        "subs %[count], %[count], #4 \n"
        "st1 {v0.4s}, [%[dst]], #16 \n"
        "bgt 1b                       
        : [dst] "+r" (dst)
        : [src1] "r" (src1), [src2] "r" (src2), [count] "r" (count)
        : "memory", "v0", "v1"
    );
}

int main()
{
    float dst[ARRAY_NUMS] = {0.0};
    float src1[ARRAY_NUMS];
    float src2[ARRAY_NUMS];

    struct timeval start;
    struct timeval end;
    double dt;
    InitArray(dst, src1, src2);
    gettimeofday(&start,NULL);
    AddFloatC(dst, src1, src2, ARRAY_NUMS);
    gettimeofday(&end,NULL);
    dt=(end.tv_sec-start.tv_sec) * 1000 + (end.tv_usec-start.tv_usec) / 1000.0;
    cout<<"Time used of Normal C code: "<<dt<<"ms"<<endl;
    return 0;
}

NOTE

The bgt jump instruction reads a corresponding NZCV flag in the NZCV system register to check whether the jump condition is met. Because the subs instruction is different from a common sub instruction, when subtraction is performed, operations are performed on corresponding condition flags.

b in bgt 1b: b is added after 1 to prevent bgt from processing 1 as an immediate. It notifies the compiler that 1 indicates a label instead of an immediate.

The execution of the C language code takes 3.062 ms, and the execution of the NEON inline assembly code takes 0.331 ms.

5.5 NEON Intrinsics Programming

5.5.1 Introduction

NEON intrinsics refer to built-in C/C++-like functions and data types supported by the compiler. These functions and data types are expanded into NEON instructions and NEON registers in the compile stage. It can be regarded as encapsulating NEON instructions into C/C++ functions.
5.5.2 Data Types

The main data types of NEON intrinsics are int8x8_t, int8x16_t, int16x4_t, int16x8_t, int32x2_t, int32x4_t, int64x1_t, float32x2_t, and float32x4_t.

The format of each data type is fixed. The first word indicates the data type, the first digit indicates the width of the data type, and the second digit indicates the amount of data stored in a vector register. For example, in int8x16_t, int8 indicates an 8-bit signed number, and 16 indicates 16 pieces of such data.

5.5.3 NEON Intrinsics Interfaces

Similar to the data type definition, the format of each NEON intrinsics interface is fixed. The following is an example of the NEON intrinsic addition function:

```c
int16x8_t vqaddq_s16 (int16x8_t, int16x8_t);
```

- The first letter v indicates a vector instruction, that is, a NEON instruction.
- The second letter q indicates a saturating instruction, that is, the subsequent addition result is automatically saturated.
- The third field add indicates an addition instruction.
- The fourth field q indicates the width of the operation register. When the field is q, the operation register QWORD is 128 bits. When the field is not specified, the operation register DWORD is 64 bits.
- The fifth field s16 indicates that the basic unit of the operation is a signed 16-bit integer. The value ranges from –32768 to 32767.

For details about the NEON intrinsics interface list, visit the official Arm website at https://developer.arm.com/architectures/instruction-sets/intrinsics/.

5.5.4 Intrinsics Implementation

To call NEON intrinsics in the code, add the header file arm_neon.h. Take array addition as an example.

C language implementation:

```c
using namespace std;

void add(int* out, int* input1, int* input2, int count)
{
    for(int i = 0; i < count; i += 1)
    {
        out[i] = input1[i] + input2[i];
    }
}

int main()
{
    int count;
    count = 10000 * 4;
    int a[count];
    int b[count];
    int c[count];

clock_t start, finish;
double duration;
```
for(int i = 0; i < count; i += 1)
{
    a[i] = rand();
}

for(int i = 0; i < count; i += 1)
{
    b[i] = rand();
}

start = clock();
for(int i = 0; i < count; i += 1)
{
    add(c, a, b, count);
}
finish = clock();
duration = (double)(finish - start) / CLOCKS_PER_SEC;
printf("%f seconds\n", duration);
return 0;

Return:
1.910000 seconds

NEON intrinsics implementation:

```cpp
using namespace std;

void add_neon(int* out, int* input1, int* input2, int count)
{
    int32x4_t input1_neon, input2_neon, out_neon;
    for(int i = 0; i < count; i += 4)
    {
        input1_neon = vld1q_s32(input1);
        input1 += 4;
        input2_neon = vld1q_s32(input2);
        input2 += 4;
        out_neon = vaddq_s32(input1_neon, input2_neon);
        vst1q_s32(out, out_neon);
        out += 4;
    }
}
```

int main()
{
    int count;
    count = 10000 * 4;
    int a[count];
    int b[count];
    int c[count];

clock_t start, finish;
double duration;

for(int i = 0; i < count; i += 1)
{
    a[i] = rand();
}

for(int i = 0; i < count; i += 1)
```c
{  
    b[i] = rand();
}

start = clock();
for(int i = 0; i < count; i += 1)
{
    add_neon(c, a, b, count);
}
finish = clock();
duration = (double)(finish - start) / CLOCKS_PER_SEC;
printf("%f seconds for neon\n", duration);
return 0;
```

Return:

```
0.360000 seconds for neon
```

It shows that the performance is greatly improved when NEON intrinsics are used.
6 Compiler

6.1 Introduction

A compiler is a software system that converts a program written in an advanced language into an equivalent object code or machine language program that can be executed on a computer. Advanced languages are easy to read and write, and machine language programs can be directly run. The main workflow of a modern compiler includes preprocessing, compiling and linking.

Figure 6-1 Compilation process

Currently, the main open source compilers include GNU Compiler Collection (GCC) and Low Level Virtual Machine (LLVM). GCC is a programming language compiler developed by GNU. It is a standard compiler for most Unix-like operating systems. LLVM is a framework for building compilers. It is an open source project compiled using C++. Because the architecture is decoupled and easy to integrate, more and more commercial compilers are using LLVM.

6.2 Principles

In the traditional three-stage compiler design, the frontend parses source code, checks source code errors, and establishes an abstract syntax tree to generate an intermediate file (IR). The optimizer reorganizes and optimizes the logic of the IR. The backend generates machine code based on the operating environment and optimizes links. libc is used to link the static library and call the SO library during running.
GCC is a single executable program compiler. There is no clear boundary between the frontend, IR, and backend. GCC is strongly coupled and cannot be developed independently. During the compilation process, much information cannot be reused by other programs. Inheriting the traditional three-stage design, LLVM normalizes the input and output interfaces and data of the optimizer. That is, the frontends of different languages parse the data and generate IRs with the same syntax rules. After optimization, common code is output to different backends for generating object code. The object code running platforms are limited, the backends are relatively fixed, and the input format of the frontend is fixed. Therefore, LLVM has convenient integration capabilities in terms of developing a compiler for a new language. In addition, multiple frontends are used as development examples, which promotes the prosperity of the LLVM framework. Compared with GCC, LLVM has a faster compilation speed, better performance of the target program, and more friendly prompts on compilation errors.

6.3 Kunpeng Compiler

The Kunpeng Compiler is developed and optimized based on open source software. It is a high-performance compiler for the Kunpeng platform.

The BiSheng Compiler is a high-performance compiler developed based on the open source LLVM and optimized for the Kunpeng platform. It supports the Fortran language. In addition to general functions and optimization of LLVM, the BiSheng Compiler has the middle and back-end key technologies optimized and the AutoTuner feature integrated to support automatic tuning of the compiler.

GCC for openEuler is a compilation toolchain (including compilers, assemblers, and linkers) developed based on the open source GCC. It is released in the openEuler open source community, and provides free binary packages in the Kunpeng community. It supports multiple processor architectures including ARM and x86.

6.4 Compiler Optimization Methods

Compiler optimization options can complete code optimization in the compile stage. The following lists common optimization options and their principles, which can greatly improve program running performance in some scenarios.

Instruction Sets and Pipelines

During C/C++ code compilation, the compiler translates the source code into an instruction sequence that can be identified by the CPU and writes the instruction sequence into the binary file of the executable program. The CPU usually executes an instruction in a pipeline to improve performance. Therefore, the instruction execution sequence greatly affects pipeline efficiency. Generally, factors such as the quantity of hardware resources for instruction computing, execution periods of different instructions, and data dependency between instructions need to be considered in an instruction pipeline. You can obtain better instruction sequence
orchestration by notifying the compiler of the target platform (CPU) instruction set and pipeline.

GCC 9.1.0 supports the Armv8 instruction set and TaiShan v110 pipeline that are compatible with the Kunpeng processor.

Usage:
Add compilation options to CFLAGS and CPPFLAGS on the GCC for openEuler, BiSheng Compiler, and GCC (version 9.1.0 or later).

```
-mtune=tsv110 -march=armv8-a
```

### Optimization Levels

The compiler uses `-O` to control program optimization levels, as shown in **Table 6-1**.

**Table 6-1 Optimization levels**

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-O</code></td>
<td>The default value is <code>-O0</code>, which does not add any optimization options. It enables the fastest compilation speed, and the program is debugging-friendly.</td>
</tr>
<tr>
<td><code>-O1</code></td>
<td>Add common optimization options.</td>
</tr>
<tr>
<td><code>-O2</code></td>
<td>Add more optimization options compared with <code>-O1</code>.</td>
</tr>
<tr>
<td><code>-O3</code></td>
<td>This is the highest level of optimization. It takes a long time to compile and generate programs that can be executed faster.</td>
</tr>
<tr>
<td><code>-Ofast</code></td>
<td>Add the same optimization options as <code>-O3</code> and some non-standard optimization options.</td>
</tr>
<tr>
<td><code>-Os</code></td>
<td>Add the same optimization options as <code>-O2</code> and some options to reduce program code.</td>
</tr>
<tr>
<td><code>-Og</code></td>
<td>Add debugging information.</td>
</tr>
</tbody>
</table>

GCC uses `gcc -Q --help=optimizers -O2` to view the optimization options selected for each optimization level.
The compiler has many optimization options. The following are some examples of scenarios where the compiler can be optimized.

- **Function inlining:**
  
  You can use the compilation option `-finline-functions` to enable function inlining. By default, function inlining is enabled at the `-O2`, `-O3`, and `-Os` optimization levels. Function inlining reduces the function calling and return overheads and improves the instruction cache hit ratio. It also saves code space and provides opportunities for other optimization measures.

  **Before:**
  ```c
  int funA (int a) {
    return a * a;
  }

  int funB (int b) {
    return funA(b) + 1;
  }
  ```

  **After:**
  ```c
  int funB (int b) {
    return b * b + 1;
  }
  ```

- **Constant propagation and folding:**
  
  Compilation options `-fdevirtualize`, `-fipa-cp`, `-fipa-cp-clone`, `-fipa-bit-cp`, `-fipa-vrp`, `-ftree-bit-ccp`, `-ftree-ccp`, `-ftree-dominator-opts`, and `-ftree-vrp` are used to control the enabling of this function. By default, these options are enabled in `-O2`, `-O3`, and `-Os` levels. In the compilation phase, variables that can be directly calculated or the calculation results of multiple variables can be replaced with constants, which reduces calculation time during running.

  **Before:**
  ```c
  ```

  ```c
  ```
int funA (int a) {
    return a * a;
}

int funB () {
    int a = funA(2);
    int b = a + 1;
    return b;
}

After:
int funB () {
    return 5;
}

- Eliminating public expressions:
  This function is controlled by compilation options `-fgcse`, `-fgcse-lm`, `-fgcse-sm`, `-fgcse-las`, and `-fgcse-after-reload`, and is enabled by default in `-O2`, `-O3`, and `-Os` levels. The calculation process is optimized during compilation to reduce the calculation amount during running.

Before:
int a, b, c;
b = (a + 1) * (a + 1);
c = (a + 1) / 2;

After:
int a, b, c, tmp;
tmp = a + 1;
b = tmp * tmp;
c = tmp / 2;

- Loop unrolling:
  You can use `-funroll-loops` and `-funroll-all-loops` to enable this function. In scenarios where the number of loops is small, the compiler copies loop body code for multiple times to eliminate the loop control overheads, facilitate data expectation, and improve the cache hit ratio.

Before:
int a[4];
for (int i = 0; i < 4; ++i) {
    a[i] = i;
}

After:
int a[4];
a[0] = 0;
a[1] = 1;
a[2] = 2;
a[3] = 3;

- Hoisting invariant loop code:
  This function is controlled by the `-fmove-loop-invariants` compilation option. It is enabled by default when the optimization level is higher than `-O1`. After the optimization, the repeated calculation of the loop body is reduced.

Before:
int a[100];
void funA(int b) {
    for (int i = 0; i < 100; ++i) {
        a[i] = b * b + i;
    }
}
After:

```c
int a[100];
void funA(int b) {
    int tmp = b * b;
    for (int i = 0; i < 100; ++i) {
        a[i] = tmp + i;
    }
}
```

- **Variable induction:**

  Use the `-fivopts` compilation option to enable this function. It is enabled by default. The value of a variable in a loop is increased (or decreased) by a fixed value in each loop iteration. Therefore, multiplication can be replaced by addition, thereby accelerating the calculation.

Before:

```c
int a[100];
for (int i = 0; i < 100; ++i) {
    a[i] = i * 9 + 3;
}
```

After:

```c
int a[100];
int tmp = 3
for (int i = 0; i < 100; ++i) {
    a[i] = tmp;
    tmp += 9;
}
```

- **After querying the virtual table, using function pointers in the virtual table to call virtual functions:**

  If the compiler can determine which virtual function is to be called, it can directly call the function, which reduces function call overheads. Use the `-fdcvirtualize` compilation option to enable this function. By default, this function is enabled in `-O2`, `-O3`, and `-Os` levels. The sample code is as follows:

  ```c
  class C0 {
  public:
      virtual void funA () {}
  };

  class C1 : public C0 {
  public:
      virtual void funA () {}
  };

  int main () {
      C1 c1;
      C0* c = &c1;
      c->funA();
      return 0;
  }
  ```

Before the optimization, the virtual table is used for function calling:
After the optimization, the function is directly called:

```cpp
[int main() { struct C0 * c; struct C1 * c1; int d[0.4128];
  int (* vtbl_ptr_type) () * _1;
  int (* _vtbl_ptr_type) () * _3;
  int _2;

  _2 = C0 vptr-C0;
  _2 = C0 vptr-C0;

  _2 = C0 vptr-C0;

  _2 = C0 vptr-C0;
  cl C0; cl C0;
  cl C0; cl C0;

  return 0;
}
```

- Vectorization:
  During compilation, code is vectorized and the NEON attribute is automatically used. NEON is a technology based on SIMD and can perform operations on multiple data at the same time based on a single instruction. When GCC uses `-O3`, the `-f-tree-vectorize` option is automatically enabled. You need to add the `-f-tree-vectorize` option under `-O1` and `-O2` to perform vectorization. In `-O0` mode, vectorization cannot be performed even if `-f-tree-vectorize` is added.

Reference code:

```cpp
int a[64 * 4];
int b[64 * 4];
int c[64 * 4];

int main ()
{
  for (int i = 0; i < 64 * 4; i++) {
    c[i] = a[i] + b[i];
  }
  return 0;
}
```
PGO

Profile-guided optimization (PGO) is to make optimization decisions by collecting program runtime information (profile). PGO needs to be compiled and run twice. In the first compilation process, the compiler inserts some functions or instructions into the program for obtaining program runtime features. Then the compiler runs the program and saves feature information as a profile. In the second compilation process, the program feature profile saved during first program running is read. Then the compiler provides guidance for various optimization technologies to make optimization decisions according to these features, to generate a target program for a performance test.

PGO supports two feedback-based optimization technologies. For the first one, the compiler instrumentation, running, and profile-guided compilation process is used. For the other, the compiler instrumentation is not required, and the perf tool is used to run programs, collect information, and conduct profile-guided compilation. The following describes the PGO methods of the compiler instrumentation. For details about the PGO methods of the perf tool, visit official websites:


Using GCC

**Step 1** Add the `-fprofile-generate` compilation option to enable the instrumentation application to generate profile information.

gcc -O2 -fprofile-generate vec.cpp

**Step 2** Run the program to generate the profile information, that is, the gcda file.

./a.out

**Step 3** Add the `-fprofile-use` compilation option and use the profile information to recompile the program.
Using Clang

**Step 1** Add the `-fprofile-instr-generate` compilation option to enable the instrumentation application to generate profile information.

```
clang -O2 -fprofile-instr-generate vec.cpp
```

**Step 2** Run the program to generate the profile information. The default file name is `default.profraw`.

```
./a.out
```

**Step 3** Use the llvm-profdata tool to convert the `default.profraw` file to a profile file that can be identified by Clang.

```
llvm-profdata merge -output=code.profdata default.profraw
```

**Step 4** Add the `fprofile-instr-use` compilation option, specify profile information, and recompile the program.

```
clang -O2 -fprofile-instr-use=code.profdata vec.cpp
```

--- End

PGO items:

- **Register allocation:** Non-profile-guided compilation generally uses a static heuristic register allocation algorithm to keep the variable value or calculation result in the register. PGO uses a priority-driven register allocation method. Priorities are determined based on the execution frequency of basic blocks to ensure that frequently used variables are preferentially allocated to registers.

- **Cold and hot partitioning:** When PGO is not used, the compiler statically performs cold and hot partitioning based on the program structure, which is not accurate enough. The profile information is used to accurately record the call frequency of basic blocks, so that hot and cold block partitioning can be more accurate. Then basic blocks are optimized, including loop unrolling and function inlining. The profile information is also used to rearrange basic blocks. Cold blocks are placed in a remote zone, and hot blocks are gathered, which helps improve utilization of the instruction cache.
Function rearrangement: The function definition sequence in the source code determines the function sequence in code segments, and the function sequence in code segments determines the sequence of functions loaded to the memory. As a result, cold and hot functions are mixed. The compiler obtains the function call relationship based on the profile information and rearranges the function sequence in code segments based on the call stack sequence. It strips cold functions to the end of a code segment, and arranges the hot functions based on the function call stack, thereby reducing the jump instruction overheads and improving the cache hit ratio.

Branch rearrangement: If conditional jump statements such as if/else and switch/case fail to be predicted, cache miss occurs. PGO uses instrumentation to collect the probability of each branch and adjust the branch calling sequence to reduce cache miss occurrences.

Others: function inlining, constant propagation/folding, and loop unrolling mentioned in Optimization Levels.

LTO

Link time optimization (LTO) is program optimization during linking. Multiple intermediate files are combined to form a global calling diagram to optimize the entire program. Link time optimization is the analysis of the entire program and is cross-module.

Add the -flto compilation option to enable LTO. LTO is performed after compilation. Therefore, the problem that multiple .o files are unaware of each other can be solved, and the entire program is optimized globally (see Optimization Levels). For example, global function inlining optimization is more comprehensive than inlining optimization of a single .o file. Besides, whether useless code exists can be determined to reduce the code volume (whether code is called cannot be determined without enabling LTO due to multiple .o files exist).

It should be noted that LTO improves the program performance, but brings the problem that the compile time is long and the memory usage is high during compilation. To reduce the compile time caused by enabling LTO, LLVM proposes the ThinLTO technology. That is, add -flto=thin in the LLVM compiler.

AutoTuner

AutoTuner is an automated iteration process that optimizes a given program by manipulating compilation options for optimal performance. It works with the BiSheng Compiler and the AutoTuner CLI tool.

The AutoTuner optimization process consists of two phases: initial compilation and tuning.
In the initial compilation phase, the `-fautotune-generate` compilation option is added to the BiSheng Compiler. During the compilation, the BiSheng Compiler generates some YAML files that contain all adjustable structures, showing which structures in the target program can be tuned.

In the tuning phase, the AutoTuner first reads the generated YAML files of the adjustable structures to generate the corresponding search space. Then, the AutoTuner tries the values of a group of parameters based on the specified search algorithm to generate a compilation configuration file in YAML format, and then compiles the binary file of the target program. Finally, AutoTuner runs the compiled file in user-defined mode and returns performance information. After certain iterations, AutoTuner finds the final optimal configuration, generates the optimal compilation configuration file, and stores the file in YAML format.

Currently, AutoTuner can be used in two modes with two CLI tools respectively: llvm-autotune and auto-tuner. For details, see AutoTuner Feature Guide (BiSheng Compiler).
7 Kunpeng BoostKit Library

7.1 Overview

Based on the computer architecture, an acceleration library improves the computing efficiency of base software libraries by changing data structures and algorithms of software code and making full use of the high performance of a dedicated chip. Under constraints such as costs, improving the hardware computing capability cannot quickly solve problems. An acceleration library must be used to achieve the optimal hardware performance and provide better services to improve the computing speed.

Functions of an acceleration library:

- For a device with a single-core CPU, if tasks are blocked, the CPU will be in the waiting state, and the computing power of the CPU will be wasted. The multi-thread technology divides time slices to make full use of the CPU capability. This is a software technology.
- For a device with a multi-core CPU, multiple threads need to be scheduled by software to make full use of multiple cores.
- For a device with multiple CPUs, software needs to be used to coordinate CPUs and memory to effectively use resources.

7.2 Introduction to the Kunpeng BoostKit Library

The Kunpeng BoostKit Library is used to optimize the performance of base software libraries and build the performance competitiveness of common software libraries on the Kunpeng platform. Up to now, 18 hardware and software acceleration libraries in total have been developed. It is an acceleration technology based on the Kunpeng platform instruction optimization and integrated software and hardware.
The Kunpeng BoostKit Library is optimized based on ARM instructions and
developed based on the Kunpeng Accelerator Engine (KAE). It covers the system
library, math library, compression, encryption & decryption, media, storage, and
network libraries, and provides high-performance acceleration capabilities for
application scenarios such as big data encryption and decryption, distributed
storage and compression, and video transcoding.

### Kunpeng BoostKit Library

<table>
<thead>
<tr>
<th>No.</th>
<th>Category</th>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>System library</td>
<td>Glibc, HyperScan, and AVX2Neon</td>
<td>Kunpeng instructions are used to optimize the performance of the common basic libraries of the system based on the Kunpeng microarchitecture features, and map the instruction functions of the x86 platform to the common modules of the Kunpeng platform.</td>
</tr>
<tr>
<td>2</td>
<td>Compression library</td>
<td>Gzip, ZSTD, Snappy, and KAEzip</td>
<td>The Kunpeng hardware acceleration module and Kunpeng instructions are used to improve the performance of mainstream open source compression libraries.</td>
</tr>
<tr>
<td>No.</td>
<td>Category</td>
<td>Library</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>---------------------------</td>
<td>----------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3</td>
<td>Encryption and decryption</td>
<td>KAE encryption and decryption</td>
<td>The Kunpeng hardware acceleration module and Kunpeng instructions are used to optimize the performance of the OpenSSL library. Automatic collaboration between hardware acceleration and instruction acceleration is supported. You can use the encryption and decryption acceleration library without modifying the application logic.</td>
</tr>
<tr>
<td>4</td>
<td>Media library</td>
<td>HMPP, HW265, X265, and X264</td>
<td>The high-performance media primitive library and video encoding/decoding library are provided based on Kunpeng acceleration instructions.</td>
</tr>
<tr>
<td>5</td>
<td>Math library</td>
<td>KML_FFT, KML_BLAS, and KML_SPBLAS</td>
<td>Basic high-performance math libraries are provided based on the Kunpeng microarchitecture and Kunpeng acceleration instructions.</td>
</tr>
<tr>
<td>6</td>
<td>Storage library</td>
<td>Smart Prefetch, SPDK, and ISA-L</td>
<td>High-speed cache drives and the efficient prefetch algorithm are used to improve the storage I/O performance, thereby improving the overall system performance in storage I/O-intensive scenarios.</td>
</tr>
<tr>
<td>7</td>
<td>Network library</td>
<td>XPF and DPDK</td>
<td>TCP/IP and OVS-based network acceleration technologies are provided based on the Kunpeng architecture.</td>
</tr>
</tbody>
</table>
7.3 Application Scenarios

The Kunpeng BoostKit Library is actually more underlying, somewhere between hardware resources and solutions. It can be used in various solutions, such as big data, distributed storage, and database, to support industries such as government, carrier, and finance. It can adapt to and be embedded in mainstream OSs, such as openEuler and CentOS, or be used as a function library to support solutions. By making full use of hardware capabilities, it improves solution performance and cost-effectiveness.

7.4 Kunpeng Math Library

7.4.1 Introduction to KML

Kunpeng Math Library (KML) is an acceleration library based on Huawei Kunpeng processors for high-performance mathematical computing. It provides mathematical functions optimized based on the Kunpeng platform to meet service requirements. It consists of five sub-libraries: a basic linear algebra library (KML_BLAS), a sparse linear algebra library (KML_SPBLAS), a vector operation library (KML_VML), a basic mathematics library (KML_MATH), and a fast Fourier transform library (KML_FFT).

In addition to ensuring mathematical operation precision, KML_FFT performs in-depth optimization on the FFT by means of vectorization and algorithm improvement, so that the performance of FFT functions is greatly improved.

7.4.2 KML Application Cases
### 7.4.2.1 Hotspot Analysis

Tools such as perf are used to collect hotspot data during code running. If a hotspot function is an existing function in KML, KML can accelerate it.

For example, when the HPC software CP2K is running, use perf to collect hotspot data. It is found that the dgemm function accounts for 79.91%.

![Hotspot Analysis Chart](image)

According to the hotspot analysis result, the dgemm function is the biggest bottleneck. Mathematical functions in KML optimized based on the Kunpeng platform can be used for acceleration.

### 7.4.2.2 KML Installation and Usage

**Step 1** Download KML.

Obtain the software digital certificate and software installation package from Huawei enterprise website or carrier website.

**Step 2** Install KML.

1. Obtain the KML software package and decompress it to obtain the binary RPM package.
2. Install KML.
   ```
   rpm -ivh boostkit-kml-xxxxx64.rpm
   ```
3. Verify KML.
   Run the `source` command or log in to the terminal again for the environment variable to take effect.
   ```
   source /etc/profile
   ```

**Step 3** Use KML.

The dgemm function is a KML_BLAS level 3 function. For details about the definition, see Function Description.

In the arch configuration file of CP2K, add the KML math library to the LIBS variable.

```bash
LIBS += $(MATHLIB_PATH)/libkblas_armv8ss_v1.2.0.a
```

----End
7.4.2.3 Verification

Run CP2K again. Table 7-1 shows the test result.

**Table 7-1 Test result**

<table>
<thead>
<tr>
<th>Test Round</th>
<th>Running Duration (s)</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline test</td>
<td>1601.49</td>
<td>The running duration is used as the performance reference. The shorter the duration, the better the performance.</td>
</tr>
<tr>
<td>Optimization test</td>
<td>433.58</td>
<td>The running duration is used as the performance reference. The shorter the duration, the better the performance.</td>
</tr>
</tbody>
</table>

Performance improvement: \((1601.49 - 433.58) / 1601.49 = 72.93\%\)

After KML is used, the running duration is reduced by 72.93\%.
## Change History

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2021-12-29</td>
<td>This is the first official release.</td>
</tr>
</tbody>
</table>