鲲鹏920代码调优指南

HUAWEI

鲲鹏920体系结构特点及性能优化点

速度快

容量大

	鲲鹏920规格	性能优化点	实施手段
Cpu 寄存器	指令集：ARMv8-A, ARMv 8.2 64C/64T,48C/48T 2.6GHz 31个通用寄存器 32个128bit的向量寄存器	指令级并行(去除指令间的依赖关系) 向量化并行(NEON) 循环并行化(串行循环转化为并行循环) 多线程编程等	**编程语言：**c/c++、汇编 实施手段例： **1.编译器自动优化：** (1)编译选项： O2,O3,循环展开,自动向量化，流水线 (2)去掉全局变量便于编译器优化 (3)受限的指针运用 **2.算法和数据结构** (1)数据组织(如：缓存利用) (2)算法实现的策略(如：查表) **3.函数** (1)传参设计 (2)内联小函数 **4.循环** 展开、累积、拆分、合并 **5.语句** 避免生成不需要的指令，让语句生成更高效的指令 (FMA,Move指令排布) **6.指令** 使用高吞吐量的指令 **7.多核并发** 多线程编程 **辅助：性能分析工具**
L1Cache	I Cache Size :64KB D Cache Size:64KB CacheLine:64B	指令对齐 预取 Cache line 对齐	
L2Cache	Cache Size:512KB Cache Line:64B	预取 Cache line 对齐 Cache分片	
L3Cache	Cache Size:1MB/Core Cache Line:128B	预取 Cache line 对齐 Cache分片 线程级存储 避免false-sharing	
内存	内存通道：8x2933MT/s	Numa编程 内存重用	

算法和数据结构优化指南 之 对齐cacheline

优化示例：1. 假设是行列数相同(M)的float方阵相乘。2. 为说明简单起见，假设M为items的倍数

优化前代码

```
#define M 1600
for(int i=0; i < M;i++) {
   for(int j=0;j < M;j++) {
      dest[i][j] = 0;
      for(int k=0; k < M;k++) {
         dest[i][j] += src1[i][k] * src2[k][j];
      }
   }
}
```

优化前性能

```
 4,444,887,982      cache-misses

  79.208280470 seconds time elapsed

  79.085865000 seconds user
   0.015972000 seconds sys
```

优化后代码

```
#define M 1600
#define CacheLineSize 64
int items = CacheLineSize / sizeof(float);
for(int i=0;i<M;i++) {
   for(int j=0;j<M;j += items ) {
      for(int jj=0;jj< items; jj++) {
         dest[i][j+jj] = 0;
      }
      for(int k=0;k<M;k++) {
         for(int jj=0;jj<items; jj++) {
            dest[i][j+jj] += src1[i][k] * src2[k][j+jj];
         }
      }
   }
}
```

优化后性能

Cache miss降低。运行时间降低一半

```
   298,703,303      cache-misses

  40.429041360 seconds time elapsed

  40.366073000 seconds user
   0.007988000 seconds sys
```

核心方法：

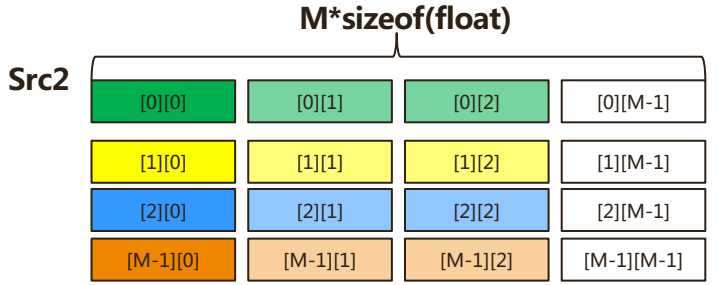
处理数据量对齐鲲鹏920 cache line(L1 L2 64B)，提升cache命中率

优化说明：

[优化前]

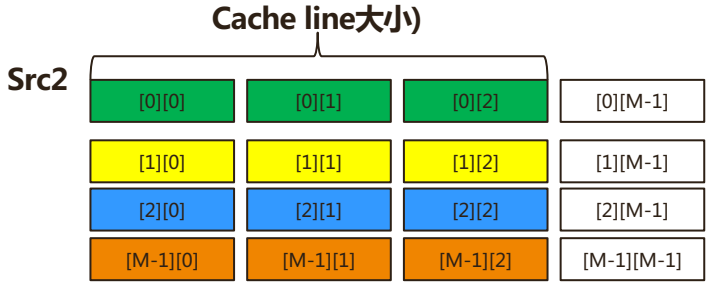
如图所示，同色系代表cacheline预期进来的数据，带有透明色的代表被浪费的数据。

在M很大的情况下，如果M*sizeof(float)大于cache line，则每次访问src2[k][j]都不能命中缓存。



[优化后]

Cache友好，不浪费cache line预取进来的数据



HUAWEI

算法和数据结构优化指南 之 Cache分片

优化示例：1. 假设是行列数相同(M)的float方阵相乘。2. 为说明简单起见，假设指定T能为M整除的倍数

优化前代码片段

```
#define M 1600
for(int i=0; i < M;i++) {
    for(int j=0;j < M;j++) {
        dest[i][j] = 0;
        for(int k=0; k < M;k++) {
            dest[i][j] += src1[i][k] * src2[k][j];
        }
    }
}
```

优化前性能

```
Performance counter stats for './block_gemm':

   4,444,226,429      cache-misses

    79.091859780 seconds time elapsed

    78.976563000 seconds user
     0.011981000 seconds sys
```

优化后代码片段

```
int T = 64;//
int N = M/T;//25
int k = 0;
int i = 0;
int j = 0;
for(int kt = 0; kt < N; ++ kt){
    for(int it=0; it<N; ++it){
        for(int jt=0; jt<N; ++jt){
            int ktt = kt*T;
            int itt = it*T;
            int jtt = jt*T;
            for(k = ktt; k<ktt+T; ++k){
                for(i = itt; i<itt+T; ++i){
                    float r = src1[i][k];
                    for(j=jtt; j<jtt+T; ++j){
                        dest[i][j] += r *src2[k][j];
                    }
                }
```

优化后性能

Cache miss降低。运行时间降低一半以上

```
Performance counter stats for './block_gemm 64':

      13,388,096      cache-misses

    26.934739940 seconds time elapsed

    26.893215000 seconds user
     0.011983000 seconds sys
```

核心方法：

当计算量大的时候，进行分片，将计算量限定在 cache容量以内，降低cache miss以提升计算性能

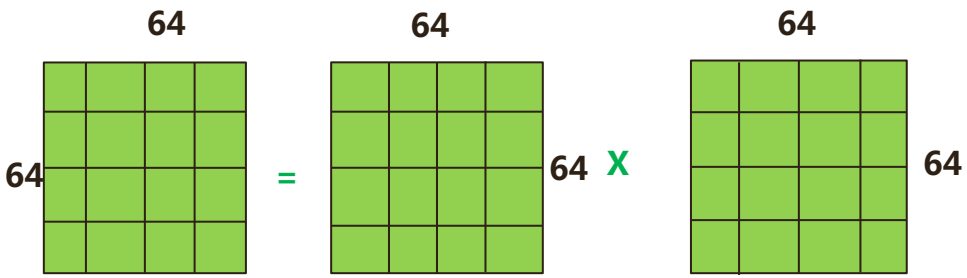
优化说明

矩阵大小1600*1600*sizeof(float)*3 Byte = 29.2M 超过了Cache的容量

将分块调整为

64*64*3*sizeof(float) = 48K

在L1CacheSize 64KB以内



HUAWEI

算法和数据结构优化指南之查表

核心方法：

(1)如果某些运算频繁的进行，可以提前保存所有的可能的结果。运算时通过查表获取结果即可。查表法可获得性能提升

(2)可结合查表法和线性插值法达到精度和性能的平衡。

优化前代码

```
for(int j = 0; j<100000000; j++) {
        float val = expf(j%5);
}
```

优化前性能

```
0.979564990 seconds time elapsed

0.978498000 seconds user
0.000000000 seconds sys
```

优化后代码

```
float a[5]={2.71828,7.389,20.085,54.5981,148.413};
for(int j = 0; j<100000000; j++) {
        float val = a[j%5];
}
```

优化后性能

运行时间降低2/3

```
0.334317780 seconds time elapsed

0.334055000 seconds user
0.000000000 seconds sys
```

VML的vexp、VSL的vRang中均实践了查表法。

函数优化指南之传参

优化前代码

```
struct WordModel {
    int x[100];
    float y[20];
    float z[10];
};
float getY0ByVal(WordModel bs)
{
    return bs.y[0];
}

int main(int argc, char **argv){
    WordModel val;
    ......
    for(int j = 0; j<100000000; j++) {
            float y0 = getY0ByVal(val);
    }
return 0;
}
```

优化前性能

```
2.019290750 seconds time elapsed

2.017644000 seconds user
0.000000000 seconds sys
```

优化后代码

```
struct WordModel {
    int x[100];
    float y[20];
    float z[10];
};
float getY0ByPtr(WordModel* bs)
{
    return bs->y[0];
}

int main(int argc, char **argv){
    WordModel val;
    ......
    for(int j = 0; j<100000000; j++) {
            float y0 = getY0ByPtr(&val);
    }
return 0;
}
```

优化后性能

```
0.208923610 seconds time elapsed

0.208749000 seconds user
0.000000000 seconds sys
```

核心方法：

函数参数优先通过寄存器传递，大小超量通过栈传递。

如果参数是大结构体或者类时，会有调用时的复制开销和返回时的销毁开销。

改成指针传递则只需要传递一个指针即可。开销较小。

此为通用的优化方法

函数优化指南 之 内联小函数

示例

```
//求0-9的平方
inline int inlineFunc(int num)
{
  if(num>9||num<0){
     return -1;
  }
  return num*num;
}

int main(int argc,char* argv[])
{
    int a=8;
    int res=inlineFunc(a);
    cout<<"res:"<<res<<endl;
}
```

优点：

(1)消除函数调用的开销。

(2)如果函数调用在循环体内，则编译器很难进行向量化。此时循环体内调用的函数如果采用内联形式的话，则编译器可识别优化

缺点：

如果内联后的函数较长，会增加寄存器压力

如果函数内有分支，内联后会对指令流水线产生不利影响

循环优化指南之循环展开

原理知识：

- 循环展开（Loop unroll）牺牲函数的尺寸，降低循环带来的开销，加快程序执行速度的方法；
- 针对Kunpeng 920处理器具有多个功能单元（FSU，ALU，LD/ST）的特点，循环展开也有利于充分使用功能单元进行指令级并行，优化指令流水线的调度。

使用方法：

- 针对循环之间不存在循环依赖或访问冲突的场景，可以使用SIMD改写、手工循环展开或者在编译选项中添加-funroll-loops；
- 针对循环之间存在依赖或访问冲突的场景，可以考虑手工循环展开的方式，减少循环开销。

注意：

- 小循环且内部没有判断逻辑，收益较高；
- 大循环展开有可能会引起通用寄存器的溢出，降低性能（寄存器重命名/外溢至内存）；
- 内部有判断逻辑可能会增加分支预测的开销，需要具体情况具体分析。

```
#include <stdlib.h>
#include <math.h>
#include <string.h>

void vpow(int length, float* src)
{
    for (int i = 0; i < length; i++) {
        powf(src[i], src[i]);
    }
}

int main(int argc, char **argv)
{
    float *src = (float *)malloc(1000000 * sizeof(float*));
    for (int i = 0; i < 1000000; i++) {
        src[i] = rand() * 100;
    }
    vpow(1000000, src);
}
[100383570@localhost test_opti]$ perf stat ./vexp

 Performance counter stats for './vexp':

           56.84 msec task-clock:u              #    0.988 CPUs utilized
               0      context-switches:u        #    0.000 K/sec
               0      cpu-migrations:u          #    0.000 K/sec
             473      page-faults:u             #    0.008 M/sec
     135,525,102      cycles:u                  #    2.384 GHz
     173,681,547      instructions:u            #    1.28  insn per cycle
   <not supported>    branches:u
         727,443      branch-misses:u

       0.057522960 seconds time elapsed

       0.049852000 seconds user
       0.007677000 seconds sys
 Performance counter stats for './vexp_unroll':

           42.64 msec task-clock:u              #    0.991 CPUs utilized
               0      context-switches:u        #    0.000 K/sec
               0      cpu-migrations:u          #    0.000 K/sec
              57      page-faults:u             #    0.001 M/sec
     107,324,851      cycles:u                  #    2.517 GHz
     170,681,504      instructions:u            #    1.59  insn per cycle
   <not supported>    branches:u
         724,012      branch-misses:u

       0.043041660 seconds time elapsed

       0.042992000 seconds user
       0.000000000 seconds sys
```

未做循环展开
汇编规模50行
IPC ~1.28

循环展开
汇编规模190行
IPC ~1.59

上述例子为使用编译选项-funroll-loops进行循环展开；另外可手动对循环体进行展开，或者使用SIMD改写循环内容，均能在合适场景将循环性能进行提升

循环优化指南 之 SIMD优化

原理知识：

- SIMD（Single Instruction Multi Data）单指令多数据流，支持将数据打包在一个大型寄存器中进行处理的一组指令集，在性能上相比标量操作吞吐量更大，适合数据密集型的操作；

使用方法：

- 编译器自动向量化优化，加入-ftree-vectorize，或编译优化设置为-O3，可添加-ftree-vectorizer-verbose参数查看编译器优化；
- 使用Neon Instrinsics改写循环体，优化处理过程；

```
#include <stdlib.h>
#include <math.h>
#include <string.h>

void vadd(int length, float* src)
{
    for (int i = 0; i < length; i++) {
        src[i] += src[i] * src[i];
    }
}

int main(int argc, char **argv)
{
    float *src = (float *)malloc(100000000 * sizeof(float
    for (int i = 0; i < 100000000; i++) {
        src[i] = rand() * 100;
    }
    vadd(100000000, src);
}
```

```
Performance counter stats for './vadd':

      2,299.48 msec task-clock:u              #    0.99
             0      context-switches:u        #    0.00
             0      cpu-migrations:u          #    0.00
           389      page-faults:u             #    0.16
 5,761,850,685      cycles:u                  #    2.50
 7,690,540,540      instructions:u            #    1.33
<not supported>     branches:u
     3,234,224      branch-misses:u

   2.301245250 seconds time elapsed

   2.244850000 seconds user
   0.047914000 seconds sys
```

```
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <arm_neon.h>

static inline void vadd(int length, float* src)
{
    float32x4_t vsrc;
    float32x4_t vtemp;
    for (int i = 0; i < length / 4; i++) {
        vsrc = vld1q_f32(src + i * 4);
        vtemp = vmlaq_f32(vsrc, vsrc, vsrc);
        vst1q_f32(src + i * 4, vtemp);
    }
}

int main(int argc, char **argv)
{
    float *src = (float *)malloc(100000000 * sizeof(float*));
    for (int i = 0; i < 100000000; i++) {
        src[i] = rand() * 100;
    }
    vadd(100000000, src);
}
```

```
Performance counter stats for './vadd_inline':

      2,130.46 msec task-clock:u              #    1.000 CPUs utilized
             0      context-switches:u        #    0.000 K/sec
             0      cpu-migrations:u          #    0.000 K/sec
           510      page-faults:u             #    0.239 K/sec
 5,366,448,557      cycles:u                  #    2.519 GHz
 7,690,540,553      instructions:u            #    1.43  insn per cycle
<not supported>     branches:u
     3,232,127      branch-misses:u

   2.131350040 seconds time elapsed

   2.048101000 seconds user
   0.080029000 seconds sys
```

循环优化指南之合并小循环

原理知识：

- 对于小循环体，可以考虑将函数上下流中的多个循环合并至一个循环内执行，减少对循环变量的操作；
- 合并小循环后也有利于增加Kunpeng处理器乱序执行的机会，提升指令级的并行能力。

注意：

- 循环的大小不绝对以代码规模而定，需要视循环的计算复杂性而言 ；
- 小循环内若均存在分支判断，则需要视情况而定，过多的分支可能会导致合并后性能劣化。

```
for (int i = 0; i < 100000000; i++) {
    src[i] = rand();
}
// Transform to [lBound, rBound)
for (int i = 0; i < 100000000; i++) {
    src[i] = lBound + (rBound - lBound) * src[i];
}
```

```
10.0 20.0
Two loops:
2281 ms
Merge loops:
2186 ms
```

```
Two loops:
2213 ms

 Performance counter stats for './trans':

        2,216.96 msec task-clock:u              #    1.000 CPUs utilized
               0      context-switches:u        #    0.000 K/sec
               0      cpu-migrations:u          #    0.000 K/sec
           1,057      page-faults:u             #    0.477 K/sec
   5,399,335,821      cycles:u                  #    2.435 GHz
   7,642,129,804      instructions:u            #    1.42  insn per cycle
  <not supported>     branches:u
       3,244,608      branch-misses:u

     2.217507250 seconds time elapsed

     2.098436000 seconds user
     0.115771000 seconds sys
```

```
Merge loops:
2013 ms

 Performance counter stats for './trans 1':

        2,017.17 msec task-clock:u              #    0.999 CPUs utilized
               0      context-switches:u        #    0.000 K/sec
               0      cpu-migrations:u          #    0.000 K/sec
             704      page-faults:u             #    0.349 K/sec
   5,097,076,200      cycles:u                  #    2.527 GHz
   7,642,129,806      instructions:u            #    1.50  insn per cycle
  <not supported>     branches:u
       3,245,118      branch-misses:u

     2.018326200 seconds time elapsed

     1.950886000 seconds user
     0.063984000 seconds sys
```

以生成[lBound, rBound)的均匀分布随机数为例，首先生成[0, 1)均匀分布随机数，然后通过分布转换将数据映射到对应的空间。

操作相对简单，可做循环合并，增加指令的并行能力和乱序执行的可能，从微观数据来看IPC利用率更高。

循环优化指南之拆分大循环

原理知识：

- 针对较为复杂或计算密集的循环，可以将大循环拆分至多个小循环内执行，可以提升寄存器的利用效率；
- 拆分大循环与合并小循环的使用是相对概念，难以绝对而言。

以生成标准高斯分布随机数为例，首先生成两个[0, 1)均匀分布随机数，然后通过BoxMuller分布转换方法将数据映射到对应的空间；其中BoxMuller方法需要使用Ln、Sqrt和Sin操作，属于计算密集操作。

在实际案例中，将循环拆分后性能得到大幅度提升（业务场景性能提升200%），减轻同一个循环的计算密度，提升寄存器的利用效率，同时也在一定程度上减少page-faults。

```
for (int i = 0; i < 100000000; i++) {
    src[i] = sqrt(-2 * log(rand())) * sin(2 * 3.1415926 * rand());
}
Merge loops:
21106 ms

 Performance counter stats for './trans 1':

       21,109.45 msec task-clock:u              #    1.000 CPUs utilized
               0      context-switches:u        #    0.000 K/sec
               0      cpu-migrations:u          #    0.000 K/sec
           1,449      page-faults:u             #    0.069 K/sec
  54,398,614,358      cycles:u                  #    2.577 GHz
  87,046,315,419      instructions:u            #    1.60  insn per cycle
  <not supported>     branches:u
     205,055,486      branch-misses:u

    21.112294430 seconds time elapsed

    20.939576000 seconds user
     0.131779000 seconds sys
```

```
for (int i = 0; i < 100000000; i++) {
    src[i] = sin(2 * 3.1415926 * rand());
}
for (int i = 0; i < 100000000; i++) {
    src[i] *=  sqrt(-2 * log(rand()));
}
Two loops:
20450 ms

 Performance counter stats for './trans':

       20,451.83 msec task-clock:u              #    1.000 CPUs utilized
               0      context-switches:u        #    0.000 K/sec
               0      cpu-migrations:u          #    0.000 K/sec
             447      page-faults:u             #    0.022 K/sec
  52,854,526,474      cycles:u                  #    2.584 GHz
  86,596,010,231      instructions:u            #    1.64  insn per cycle
  <not supported>     branches:u
     194,837,063      branch-misses:u

    20.454634500 seconds time elapsed

    20.367134000 seconds user
     0.035933000 seconds sys
```

语句优化指南 之 减少内存读写和分配

原理知识：

- 使用临时变量等手段，减少对某一块固定内存空间的访问，可以提升编译器的优化能力；
- 针对一些场景，可以重复利用内存空间，减少分配内存的次数来进行性能提升。

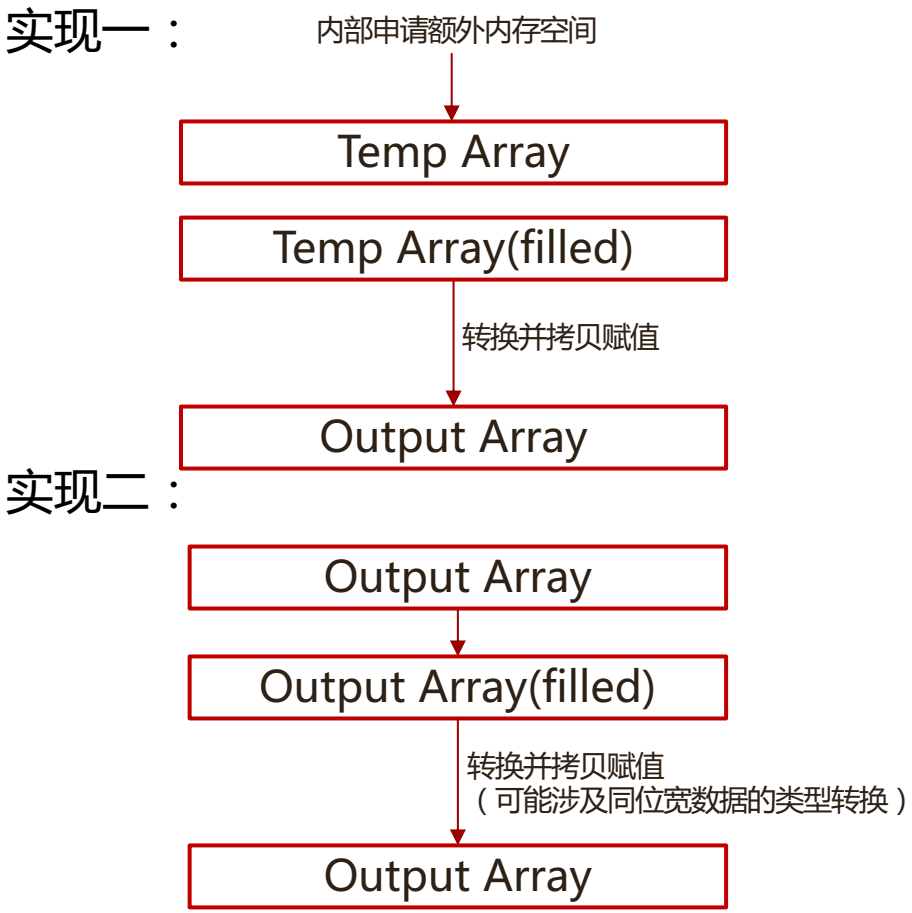
在某案例中，生成最终输出数据的流程包含前端和后端转换，在编码实现时实际上有两种实现模式：

一种是内部分配内存A用于存放算法核心生成的数据内容，转换赋值至目标内存空间B内；

另一种是直接用目标数组空间B作为中间存储，转换赋值在内存空间B内进行。

前者内存模式为copy，且需要内部分配空间，在输出长度较大时带来额外开销；
后者内存模式为override，无需分配额外空间，经实测override的带宽也比copy的带宽更高。

端到端性能提升10%。



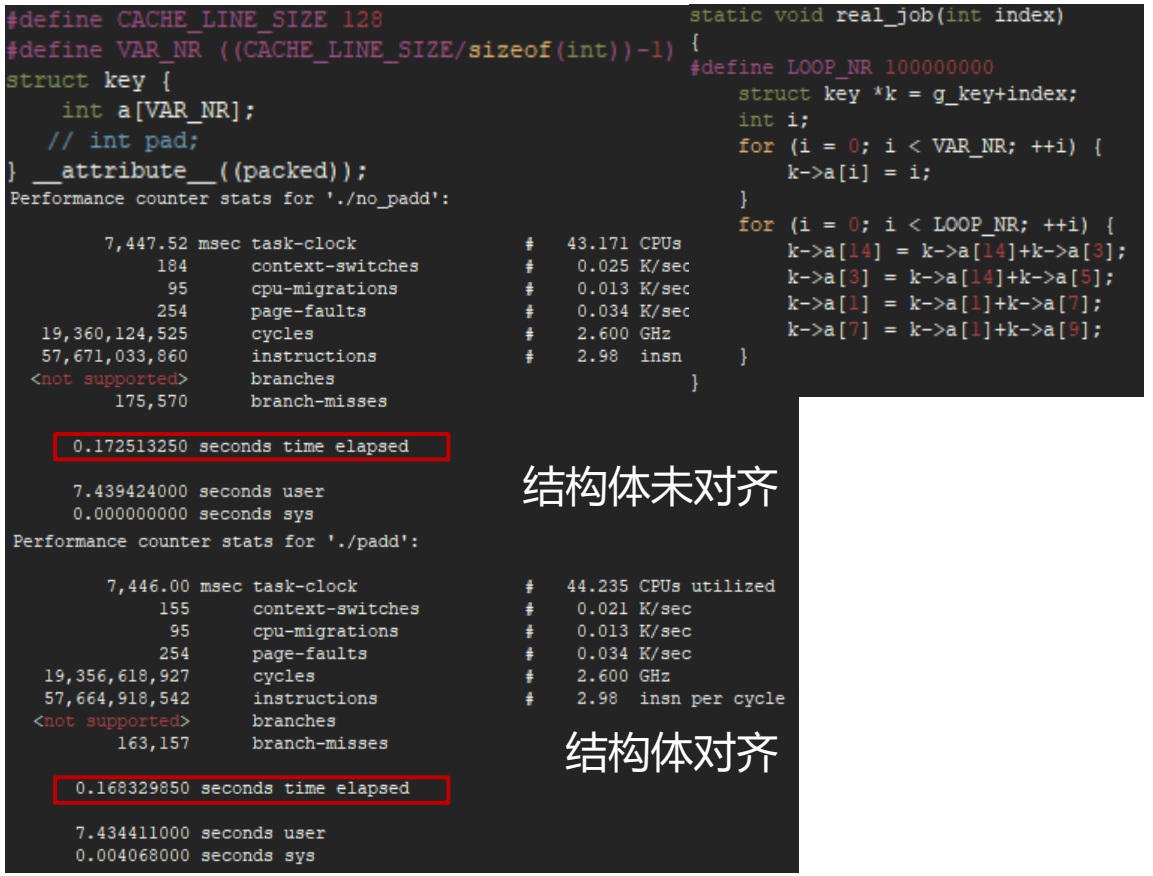
语句优化指南 之 结构体对齐

原理知识：

- 尽量保持结构体的定义保持字节对齐，有利于在编译器优化后运行时性能最佳，避免内存不对齐。

优化方法：

- 较小结构体进行机器字对齐，较大结构体进行Cache Line对齐（L1/L2 64Bytes，L3 128Bytes）；
- 只读字段与读写字段隔离对齐，使得其集中在Cache的不同Cache Line中，降低只读字段的Cache Line频繁失效，也减少读写字段污染的Cache Line条数；
- 尽量大数据类型在前，小数据类型在后，保持数据紧密；
- 总字节数尽量是2的幂次；
- 每个域的起始地址为其大小的整数倍；
- 可以使用编译器提供的按字节对齐的编译原句__attribute__((aligned(x))), __attribute__(packed())优化。



如上图所示，结构体构造为CacheLine大小减去一个整形数据的大小，进行数据更新的操作。CPU在进行数据获取和更新操作时，由于结构体与Cache Line Size的匹配关系，直接影响了cache是否会产生失效。

语句优化指南 之 分支优化

原理知识：

- 编译器会对分支进行预判，正确的编码方式可以提高分支预测的准确性和流水线效率；

优化方法：

- 避免将判断放在循环内；
- 拆分循环，将循环内的多条件分支运行拆分成多个循环，避免判断（例如奇偶拆分）；
- 合并条件，多个条件合并成一个分支预测；
- 使用三目条件运算符，编译器将优化会条件赋值语句，可以提升性能；
- 使用查表的方式减少分支；
- 调整判断顺序或使用unlikely()/likely()语句显性标识分支可能性，提升编译器对分支预测的准确性。

```
Performance counter stats for './bn 1000':

        1,671.60 msec task-clock:u              #    1.000 CPUs utilized
               0      context-switches:u        #    0.000 K/sec
               0      cpu-migrations:u          #    0.000 K/sec
             509      page-faults:u             #    0.304 K/sec
   4,331,418,158      cycles:u                  #    2.591 GHz
   9,000,475,297      instructions:u            #    2.08  insn per cycle
  <not supported>     branches:u
           7,066      branch-misses:u

     1.672383670 seconds time elapsed

     1.669342000 seconds user
     0.000000000 seconds sys
Performance counter stats for './b0 1000':

        1,655.60 msec task-clock:u              #    0.999 CPUs utilized
               0      context-switches:u        #    0.000 K/sec
               0      cpu-migrations:u          #    0.000 K/sec
             508      page-faults:u             #    0.307 K/sec
   4,285,812,428      cycles:u                  #    2.589 GHz
   9,000,475,312      instructions:u            #    2.10  insn per cycle
  <not supported>     branches:u
           6,878      branch-misses:u

     1.657070110 seconds time elapsed

     1.653240000 seconds user
     0.000000000 seconds sys
Performance counter stats for './b1 1000':

        3,435.19 msec task-clock:u              #    1.000 CPUs utilized
               0      context-switches:u        #    0.000 K/sec
               0      cpu-migrations:u          #    0.000 K/sec
             987      page-faults:u             #    0.287 K/sec
   8,883,871,595      cycles:u                  #    2.586 GHz
   9,000,475,206      instructions:u            #    1.01  insn per cycle
  <not supported>     branches:u
           7,214      branch-misses:u

     3.436001530 seconds time elapsed

     3.420168000 seconds user
     0.007971000 seconds sys
```

无分支优化

正确分支优化

错误分支优化

针对同一程序代码，添加unlikely()/likely()语句实际运行结果

多线程并发优化指南 之 避免不必要共享写

- **原理**
  - 多核写入共享变量会导致所在cache line在这些核的私有cache之间重复无效化，降低cache命中率
  - 伪共享：写入位于相同cache line的不同变量，和写入共享变量效果相同
- **鲲鹏特点**
  - 鲲鹏920的L1和L2 cache line大小为64字节、L3 cache line为128字节
- **优化方案**
  - OpenMP代码中使用reduction子句替代直接写入共享变量（循环过程中写入线程私有变量）
  - 线程私有的变量按照cache line大小对齐（线程栈上的变量除外）
  - 使用线程私有变量（如GCC支持__thread，C11支持_Thread_local关键字）
- **优化例1（见右上）**
  - 增加reduction子句，当n=1G时运行时间缩短到2/3
- **优化例2（见右下）**
  - 对于线程私有的变量增加按照cache对齐，100线程下运行时间缩短到1/3

```
    int domainError = 0;
#pragma omp parallel for reduction(|:domainError)
    for (long i = 0; i < n; i++) {
        if (a[i] <= 0) {
            domainError = 1;
        }
        y[i] = logf(a[i]);
    }
    return domainError;
```

```
struct ThreadInfo {
    unsigned long tid;
    float *input;
    unsigned long sum;
    unsigned long length;
} __attribute__ ((aligned (128)));

struct ThreadInfo tinfo[N];

void *proc(void *arg)
{
    struct ThreadInfo *info = (struct ThreadInfo *)arg;
    info->sum = 0;
    for (unsigned long i = 0; i < info->length; i++) {
        info->sum += info->input[i];
    }
}
```

调优工具（perf）

- perf可以在运行程序时收集性能计数器（events），分析程序性能、发现热点或性能瓶颈
  - 列出perf支持的events：perf list
  - 运行app并收集events信息，运行结束后输出结果：perf stat -c events ./app
    - 例如右上例在运行/bin/ls时收集了cycles（CPU时间）和cache-misses信息
  - 运行app并收集信息后存入文件：perf record -c events ./app，然后可用perf report查看
    - 例如右例是使用perf report后可见的界面，给出每个函数的cycles事件分布
    - 根据此比例发现程序热点，决定首先优化的函数
    - 进入具体函数后可以看到每条指令的cycles分布，可以看到str指令为热点
      - 注：编译时需要加入调试信息才能显示C源代码（GCC使用-g）

```
[251 1.48 opt]$ perf stat -e cycles,cache-misses /bin/ls /
bin   dev  home  lib64       media  opt   root  sbin  sys  usr
boot  etc  lib   lost+found  mnt    proc  run   srv   tmp  var

Performance counter stats for '/bin/ls /':

          909159      cycles:u

            5112      cache-misses:u


      0.001482320 seconds time elapsed

      0.001525000 seconds user
      0.000000000 seconds sys
```

```
Samples: 972  of event 'cycles:u', Event count (approx.
Overhead  Command  Shared Object      Symbol
  82.02%  ins      ins                [.] getx
  17.85%  ins      ins                [.] main
   0.10%  ins      libc-2.28.so       [.] _IO_file_xsput
   0.02%  ins      ld-2.28.so         [.] _dl_lookup_sym
   0.00%  ins      ld-2.28.so         [.] _dl_start_fina
   0.00%  ins      ld-2.28.so         [.] _dl_start
```

```
                  return __builtin_aarch64_ld1v4sf ((const
14.00  80:  ldr    q1, [x0], #16
            vmulq_f32():
              return __a * __b;
 1.03         fmul   v1.4s, v1.4s, v1.4s
            main():
                for (int i = 0; i < N; i += 4) {
10.65         cmp    x0, x2
            vst1q_f32():

            __extension__ extern __inline void
            __attribute__ ((__always_inline__, __gnu_i
            vst1q_f32 (float32_t *a, float32x4_t b)
            {
              __builtin_aarch64_st1v4sf ((__builtin_aa
72.05         str    q1, [x1], #16
            main():
 0.68         b.ne   80
```

调优工具优化举例：perf

- 优化前perf发现访存指令所用CPU cycles较多，考虑优化访存
- 优化方法：交换内外层循环，提高数组B访问局部性
- 优化后性能明显提升且perf统计发现数组B加载指令不再是热点

优化前

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < P; j++) {
        C[i][j] = 0;
        for (int k = 0; k < M; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
                mov     x1, x6
25.20   40:     ldr     s1, [x0]
30.10           add     x0, x0, x2
38.85           ldr     s2, [x1], #4
 5.00           cmp     x0, x3
                fmadd   s0, s2, s1, s0
 0.60         ↑ b.ne    40
 0.06           str     s0, [x5, x4]
                add     x4, x4, #0x4
```

加载数组B

优化后

```
for (int i = 0; i < N; i++) {
    for (int k = 0; k < M; k++) {
        for (int j = 0; j < P; j++) {
            if (k == 0) {
                C[i][j] = 0;
            }
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
                mov     x2, #0x0
        28:     ldr     s2, [x5, x2, lsl #2]
                mov     x0, #0x0
 0.60   30: ↓   cbz     x2, 78
 0.06           ldr     s1, [x1, x0]
21.15   38:     ldr     s0, [x3, x0]
                fmadd   s0, s2, s0, s1
78.19           str     s0, [x1, x0]
                add     x0, x0, #0x4
                cmp     x0, x4
```

加载数组B