# Go-On-ARM SIMD优化入门案例

## 1 Golang开发环境准备

环境准备

工具介绍

perf

Benchstat

## 2参与社区方法

包含签署贡献协议，下载源码、运行别人CL、自己提交等完整过程

参考一下资料写

<https://stackoverflow.com/questions/25716920/what-does-cl-mean-in-a-commit-message-what-does-it-stand-for>

<https://golang.org/doc/contribute.html>

## 3 使用SIMD优化

场景介绍

SIMD技术全称Single Instruction Multiple Data，即单指令多数据流，通过单条指令并行操作一组数据替换原来的多条指令或循环操作，实现性能提升。ARM64支持的SIMD指令数约400个左右，包含数据加载和存储、数据处理、压缩、加解密等。ARM64包含32个SIMD向量寄存器用于SIMD操作，可以批量加载一组数据到向量寄存器中，使用SIMD指令对向量寄存器中的数据运算后，批量存到内存。SIMD技术常用于多媒体、数学库、加解密算法库等包含循环处理数组元素的场景，通过SIMD指令和向量寄存器的帮助减少其中数据加载和存储、数学运算、逻辑运算、移位等常用操作所需的指令条数。

3.1分析

我们通过Golang官方的一个SIMD优化案例来进行介绍，该CL地址为：

<https://go-review.googlesource.com/c/go/+/71110>

优化前的代码使用Golang汇编编写，实现在src/runtime/asm\_arm64.s中，如下所示：

// memequal(p, q unsafe.Pointer, size uintptr) bool  
TEXT memequal(SB),NOSPLIT,$-8-25  
 MOVD a+0(FP), R1  
 MOVD b+8(FP), R2  
 MOVD size+16(FP), R3  
 ADD R1, R3, R6  
 MOVD $1, R0  
 MOVB R0, ret+24(FP)  
 CMP R1, R2  
 BEQ done  
loop:  
 CMP R1, R6  
 BEQ done  
 MOVBU.P 1(R1), R4  
 MOVBU.P 1(R2), R5  
 CMP R4, R5  
 BEQ loop  
  
 MOVB $0, ret+24(FP)  
done:  
 **RET**

该函数的定义在src/runtime下面的stubs.go文件中，如下所示：

func memequal(a, b unsafe.Pointer, size uintptr) bool

参数a, b的类型实际是两个int指针，表示两个数组，参数size表示数组的大小，简单来说该函数就是按顺序挨个比较两个数组中的元素是否相等。

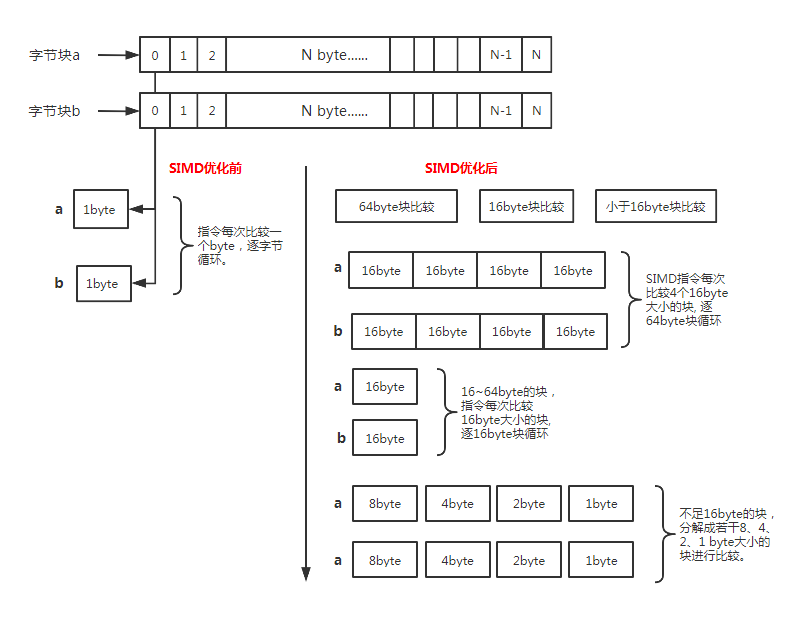
优化前代码逻辑简析见下图：



根据上图代码解析可以看到，该函数是循环操作两个数组中的元素，当前的处理方式每次循环只能取1 byte进行比较，即每byte数据要耗费两个读取操作和两个比较操作，性能不高。

3.2优化

**SIMD优化原理图**



**byte块比较函数的SIMD指令优化详解**

优化后的代码使用Golang汇编编写，实现在src/runtime/asm\_arm64.s中，跟SIMD指令相关的代码如下：(添加了关键指令的注释，红色部分）

// input:  
// R0: pointer a  
// R1: data len  
// R2: pointer b  
// R8: address to put result  
**TEXT** memeqbody<>(SB),NOSPLIT,$0  
 CMP $1, R1  
 // handle 1-byte special case for better performance  
 BEQ one  
 CMP $16, R1  
 // handle specially if length < 16  
 BLO tail  
 BIC $0x3f, R1, R3 //位清除指令，清除R1的后6位存放到R3  
 CBZ R3, chunk16 //跳转指令，R3为0，跳转到chunk16  
 // work with 64-byte chunks R6为64byte块尾部指针  
 ADD R3, R0, R6 // end of chunks   
chunk64\_loop: //循环处理64的byte块  
 VLD1.P (R0), [V0.D2, V1.D2, V2.D2, V3.D2]

//SIMD加载指令RO数据块加载到V0-V3寄存器,偏移64byte  
 VLD1.P (R2), [V4.D2, V5.D2, V6.D2, V7.D2]

//SIMD加载指令R2数据块加载到V0-V3寄存器,偏移64byte  
 VCMEQ V0.D2, V4.D2, V8.D2  
 VCMEQ V1.D2, V5.D2, V9.D2  
 VCMEQ V2.D2, V6.D2, V10.D2  
 VCMEQ V3.D2, V7.D2, V11.D2

//SIMD比较指令，比较寄存器V0和V1的每-位，若相等则在V8向量中标志1，否则否则表示0  
 VAND V8.B16, V9.B16, V8.B16  
 VAND V8.B16, V10.B16, V8.B16  
 VAND V8.B16, V11.B16, V8.B16

//SIMD位与操作，比较结果向量的每一位做与操作，保存在V8寄存器  
 CMP R0, R6 //比较指令，比较RO和R6的值，修改寄存器标志位  
 VMOV V8.D[0], R4 //SIMD转移指令，保存V8向量0-63位到R4寄存器  
 VMOV V8.D[1], R5 //SIMD转移指令，保存V8向量64-127位到R5寄存器  
 CBZ R4, not\_equal // 跳转指令，R4中0bit位表示不相等，跳转not\_equal  
 CBZ R5, not\_equal   
 BNE chunk64\_loop //标志位不等于0，对应上面RO!=R6则跳转chunk64\_loop  
 AND $0x3f, R1, R1 //仅保存R1的后6位  
 CBZ R1, equal //R1为0,跳转equal,否则继续往下执行  
chunk16:  
 // work with 16-byte chunks  
 BIC $0xf, R1, R3 //位清除指令，清除R1的后4位存到R3  
 CBZ R3, tail //R3为0，表示R1块小于16byte，跳转到tail块  
 ADD R3, R0, R6 // end of chunks R6为16byte块尾部指针

chunk16\_loop: //循环处理16byte

VLD1.P (R0), [V0.D2]

//SIMD加载指令RO数据块加载到V0寄存器,偏移16byte

VLD1.P (R2), [V1.D2]

//SIMD加载指令R2数据块加载到V1寄存器,偏移16byte

VCMEQ V0.D2, V1.D2, V2.D2

CMP R0, R6 //比较指令，比较RO和R6的值，修改寄存器标志位

VMOV V2.D[0], R4

VMOV V2.D[1], R5

CBZ R4, not\_equal //判断是否有不等，如有0位，跳not-equal

CBZ R5, not\_equal

BNE chunk16\_loop //循环继续

AND $0xf, R1, R1

CBZ R1, equal //若无剩余块（小于16byte），则跳转equal

tail: //处理小于16byte块的尾部

// special compare of tail with length < 16

TBZ $3, R1, lt\_8

//跳转指令，若R1[3]==0,也就是R1小于8,跳转到lt\_8

MOVD.P 8(R0), R4

MOVD.P 8(R2), R5

CMP R4, R5

BNE not\_equal //比较R4和R5，不等则跳转not-equal

lt\_8://处理小于8的byte块

TBZ $2, R1, lt\_4

MOVWU.P 4(R0), R4

MOVWU.P 4(R2), R5

CMP R4, R5

BNE not\_equal

lt\_4://处理小于4的byte块

TBZ $1, R1, lt\_2

MOVHU.P 2(R0), R4

MOVHU.P 2(R2), R5

CMP R4, R5

BNE not\_equal

lt\_2://处理小于2的byte块

TBZ $0, R1, equal

one://处理一个byte的块

MOVBU (R0), R4

MOVBU (R2), R5

CMP R4, R5

BNE not\_equal

equal:

MOVD $1, R0

MOVB R0, (R8)

RET

not\_equal:

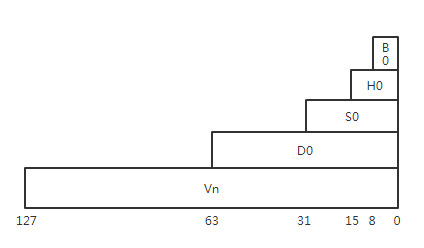
MOVB ZR, (R8)

RET

byte块比较函数优化使用了VLD1,VCMEQ,VADD,VMOV等SIMD指令，VLD1.P是SIMD加载指令，.p表示先加载再偏移，VCMEQ用于比较SIMD寄存器存储的内容，VADD用于SIMD寄存器间的与操作，VMOV 用于SIMD寄存器之间复制转移。指令的用法在代码注释中写明。

ARM64 SIMD 寄存器用Vn表示，其中n取值范围[0,31]，SIMD寄存器有128位。

SIMD寄存器支持字节B、半字H、字S、双字D四种数据格式用于数据加载存储等处理，下图表示了Vn寄存器上D,S,H,B的所占的位关系。ARM64中SIMD寄存器的一般表示为Vn{.2D,.4S,.8H,.16B}。byte块比较函数优化中，VLD1，VCMEQ指令使用Vn.D2形式表示SIMD Vn寄存器，VADD指令使用Vn.B16表示Vn寄存器。



3.3验证

要求图文结合，图片为主，可不借助其他材料上手完成整个操作