

实验准备

由于我本次实验使用的是MacBookAir(M2)，因此我需要对**MakeFile**做一点修改

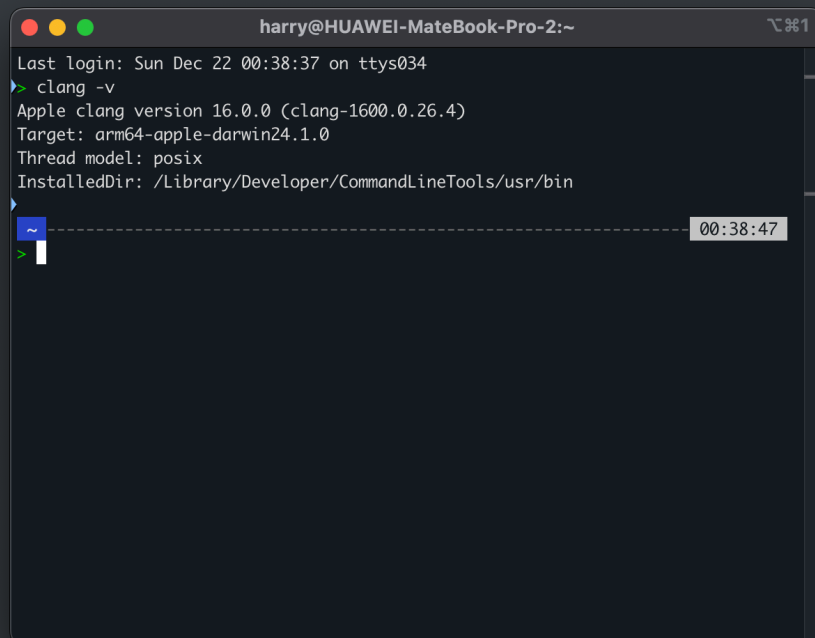
```
TARGET := x86_64-apple-darwin

CFLAGS := -Wall -g -std=gnu99 -target $(TARGET)
```

这里在开头添加了一个新的参数**TARGET**用于指定目标架构和平台，然后我们修改一下**CFLAGS**，添加上我们的参数。

由于现在Mac上的Clang可以直接进行交叉编译，我们至此便完成了实验准备。

本次实验我采用的Clang版本为16.0.0（clang-1600.0.26.4）



```
harry@HUAWEI-MateBook-Pro-2:~
Last login: Sun Dec 22 00:38:37 on ttys034
> clang -v
Apple clang version 16.0.0 (clang-1600.0.26.4)
Target: arm64-apple-darwin24.1.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
~----- 00:38:47
>
```

由于我得到的汇编文件保留了大量调试信息，实验报告中我只会展示关键的部分，完整文件将放置于**code**文件夹

Write-up1

我们首先根据要求生成制定的汇编代码，并重命名为`wr-example1.s`

任务1：绘制流程图

简单分析我们可以发现其执行流程如下

BB0_0: 函数入口点，进行初始检查

- 第一个判断 `cmpq %rdi, %rax`: 检查数组 a 的结束位置是否小于等于数组 a 的起始位置

BB0_1: 继续进行第二重检查

- 第二个判断 `cmpq %rsi, %rax`: 继续检查数组 a 的结束位置是否小于等于数组 b 的起始位置

BB0_2 和 BB0_3: 向量化的循环实现

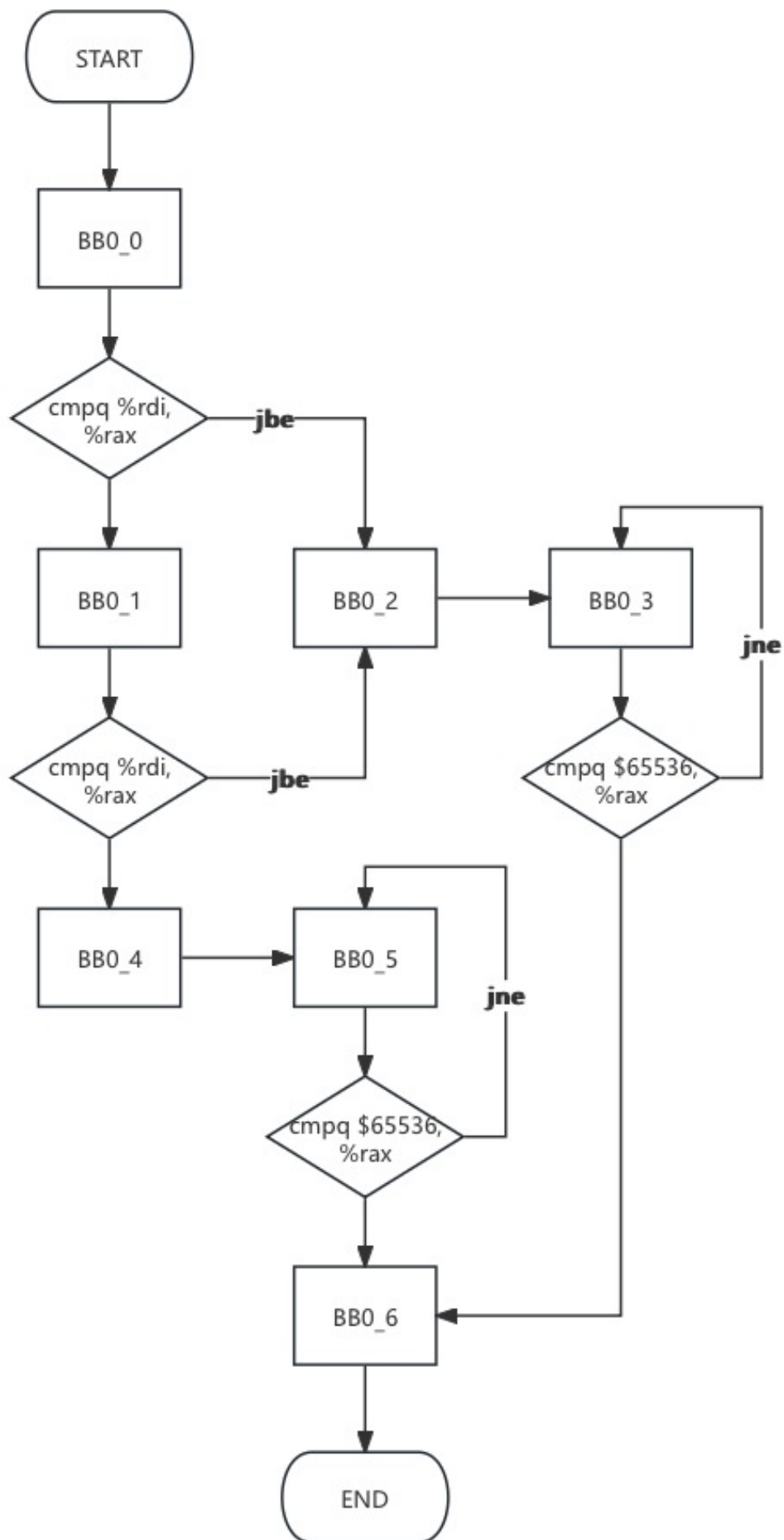
- BB0_2 是向量化循环的入口，进行循环的初始化
- BB0_3 为 向量化加法指令执行部分

BB0_4 和 BB0_5: 非向量化的循环实现

- BB0_4 是非向量化循环的入口，进行循环的初始化
- BB_05 为 非向量化加法的执行部分

BB0_6: 函数退出点

接下来我们来画出这段汇编代码的控制流程图



任务2:比较汇编代码

通过阅读给定的图中的汇编和我得到的汇编，我发现二者的主要区别为循环的初始化。

我们可以看到，图中的汇编代码在内存是否有重叠的初始化都将%rax初始化为-65536

```
# %bb.4:                                     # %for.body.preheader
    #DEBUG_VALUE: test:b <- %rsi
    #DEBUG_VALUE: test:a <- %rdi
    .loc      1 0 3 is_stmt 0                  # example1.c:0:3
    movq      $-65536, %rax                    # imm = 0xFFFF0000

    .p2align  4, 0x90
```

```
.LBB0_2:                                     # %vector.body.preheader
    #DEBUG_VALUE: test:b <- %rsi
    #DEBUG_VALUE: test:a <- %rdi
    .loc      1 0 3                          # example1.c:0:3
    movq      $-65536, %rax                    # imm = 0xFFFF0000
    .p2align  4, 0x90
```

而我得到的则将%eax初始化为0

```
Ltmp2:
## %bb.4:
    ##DEBUG_VALUE: test:i <- 0
    ##DEBUG_VALUE: test:b <- $rsi
    ##DEBUG_VALUE: test:a <- $rdi
    .loc  0 0 3 is_stmt 0                    ## example1.c:0:3
    xorl  %eax, %eax

LBB0_2:
    ##DEBUG_VALUE: test:i <- 0
    ##DEBUG_VALUE: test:b <- $rsi
```

```
##DEBUG_VALUE: test:a <- $rdi
.loc 0 0 3                                     ## example1.c:0:3
xorl %eax, %eax
```

这也导致了二者的循环控制的逻辑是不同的

图中的汇编代码向量化后的代码，从-65536开始向上计数到0，然后利用加法操作后的零标志位直接判断是否要继续进行循环

```
120 .Ltmp4:
121     .loc 1 12 26 is_stmt 1                # example1.c:12:26
122
123     addq $64, %rax
124     jne .LBB0_3
```

而我得到的汇编代码则是不断比较%eax和%rax以控制循环

```
Ltmp12:
    .loc 0 12 26 is_stmt 1                ## example1.c:12:26
    addq $64, %rax
    cmpq $65536, %rax                     ## imm = 0x10000
    jne LBB0_3
```

同时，图片中给出的版本未显式指定对齐；而我得到的版本使用 `.p2align 4, 0x90` 进行代码对齐。并且我得到的版本保留了更多调试信息。

对于二者的优缺点，对比如下

实验手册图中给出的汇编代码

优点：

- 使用零初始化寄存器启动循环相较于我得到的那个版本的方法个更加简单
- 可以允许根据程序的上下文进行动态初始化

缺点：

- 没有为循环变量设置特定值，可能会导致数组越界访问、内存访问违规
- 循环中操作的数字较大，运算较复杂，可能导致运算慢、寻址慢

我得到的汇编代码

优点：

- 循环中操作的数字较小，运算简单，可能运算快、寻址快
- 为循环变量设置了特定值

缺点：

- 多使用了一条比较指令，指令数略多
- 初始化值是固定的，不容易更改

Write-up2

我根据要求修改代码并得到新的汇编文件，重命名为**wr2-example1.s**

```
.section __TEXT,__text,regular,pure_instructions
.build_version macos, 15, 0 sdk_version 15, 1
.globl _test                                ## -- Begin function test
.p2align 4, 0x90
_test:                                     ## @test
Lfunc_begin0:
.file 0
"/Users/harry/work/\350\275\257\344\273\266\347\263\273\347\273\237\34
4\274\230\345\214\226/\345\256\236\351\252\214\351\241\271\347\233\256
A5/recitation3" "example1.c" md5 0x094e920f5f12746d7cc7ddf73cde262d
.loc 0 9 0                                ## example1.c:9:0
.cfi_startproc
## %bb.0:
##DEBUG_VALUE: test:a <- $rdi
##DEBUG_VALUE: test:b <- $rsi
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
xorl %eax, %eax
Ltmp0:
##DEBUG_VALUE: test:i <- 0
.p2align 4, 0x90
LBB0_1:                                ## =>This Inner Loop Header:
Depth=1
##DEBUG_VALUE: test:i <- 0
##DEBUG_VALUE: test:b <- $rsi
##DEBUG_VALUE: test:a <- $rdi
.loc 0 13 13 prologue_end              ## example1.c:13:13
movdqu (%rsi,%rax), %xmm0
```

```

movdqu 16(%rsi,%rax), %xmm1
.loc 0 13 10 is_stmt 0                                ## example1.c:13:10
movdqu (%rdi,%rax), %xmm2
paddb %xmm0, %xmm2
movdqu 16(%rdi,%rax), %xmm0
paddb %xmm1, %xmm0
movdqu 32(%rdi,%rax), %xmm1
movdqu 48(%rdi,%rax), %xmm3
movdqu %xmm2, (%rdi,%rax)
movdqu %xmm0, 16(%rdi,%rax)
.loc 0 13 13                                ## example1.c:13:13
movdqu 32(%rsi,%rax), %xmm0
.loc 0 13 10                                ## example1.c:13:10
paddb %xmm1, %xmm0
.loc 0 13 13                                ## example1.c:13:13
movdqu 48(%rsi,%rax), %xmm1
.loc 0 13 10                                ## example1.c:13:10
paddb %xmm3, %xmm1
movdqu %xmm0, 32(%rdi,%rax)
movdqu %xmm1, 48(%rdi,%rax)
Ltmp1:
.loc 0 12 26 is_stmt 1                            ## example1.c:12:26
addq $64, %rax
cmpq $65536, %rax                                ## imm = 0x10000
jne LBB0_1
Ltmp2:
## %bb.2:
##DEBUG_VALUE: test:i <- 0
##DEBUG_VALUE: test:b <- $rsi
##DEBUG_VALUE: test:a <- $rdi
.loc 0 15 1 epilogue_begin                        ## example1.c:15:1
popq %rbp
retq
Ltmp3:
Lfunc_end0:
.cfi_endproc

```



```
## -- End function

.file 1
"/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/_types" "_uint8_t.h" md5 0x8b64ccf8c67b8c006b07b8daf1b49be5
.file 2
"/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/_types" "_uint64_t.h" md5 0x77fc5e91653260959605f129691cf9b1
.section __DWARF,__debug_abbrev,regular,debug
```

我们可以发现，这段汇编代码中并没有判断数组a和数组b是否发生重叠，而是直接进行向量化的循环。这是因为指向 restrict 变量的指针不能用于同时访问该变量的其他部分，这样就使得无论数组a和数组b是否有重叠，都不会对我们的向量化产生影响。

Write-up3

我们根据要求继续修改代码，得到汇编文件并重命名为`wr3-example1.s`

和上一问的主要区别在于`LBB0_1`

```
LBB0_1:                                ## =>This Inner Loop Header:
Depth=1
    ##DEBUG_VALUE: test:i <- 0
    ##DEBUG_VALUE: test:b <- $rsi
    ##DEBUG_VALUE: test:a <- $rdi
    .loc 0 16 10 prologue_end          ## example1.c:16:10
    movdqa (%rdi,%rax), %xmm0
    movdqa 16(%rdi,%rax), %xmm1
    movdqa 32(%rdi,%rax), %xmm2
    movdqa 48(%rdi,%rax), %xmm3
    paddb (%rsi,%rax), %xmm0
    paddb 16(%rsi,%rax), %xmm1
    movdqa %xmm0, (%rdi,%rax)
    movdqa %xmm1, 16(%rdi,%rax)
    paddb 32(%rsi,%rax), %xmm2
    paddb 48(%rsi,%rax), %xmm3
    movdqa %xmm2, 32(%rdi,%rax)
    movdqa %xmm3, 48(%rdi,%rax)
```

我们做出的修改为添加如下两行

```
a = __builtin_assume_aligned(a, 16);
b = __builtin_assume_aligned(b, 16);
```

这两行代码告知编译器，**指针a**和**指针b**按照 **16** 字节对齐

我们可以发现write-up2中由于未进行16字节对齐，使用的是 `movdqu` ,而这一问中进行了16字节对齐后，使用的是 `movdqa` ,这是用于执行对齐的128位XMM寄存器版本。

`movdqa` 指令，即***Move Aligned Double Quadword***，要求16字节对齐的内存访问，用于将128位（16字节）的数据加载到对应的XMM寄存器中。

`movdqu` 指令，即***Move Unaligned Double Quadword***，则支持非对齐的内存访问，用于将128位（16字节）的数据加载到对应的XMM寄存器中。

Write-up4

我们根据要求得到汇编代码，重命名保存为 **wr4-example1.s**

这段汇编代码和上一问中的区别主要为 **LBB0_1**

```
LBB0_1:                                     ## =>This Inner Loop Header:
Depth=1
    ##DEBUG_VALUE: test:i <- 0
    ##DEBUG_VALUE: test:b <- $rsi
    ##DEBUG_VALUE: test:a <- $rdi
    .loc 0 16 10 prologue_end               ## example1.c:16:10
    vmovdqu (%rdi,%rax), %ymm0
    vmovdqu 32(%rdi,%rax), %ymm1
    vmovdqu 64(%rdi,%rax), %ymm2
    vmovdqu 96(%rdi,%rax), %ymm3
    vpaddb  (%rsi,%rax), %ymm0, %ymm0
    vpaddb  32(%rsi,%rax), %ymm1, %ymm1
    vpaddb  64(%rsi,%rax), %ymm2, %ymm2
    vpaddb  96(%rsi,%rax), %ymm3, %ymm3
    vmovdqu %ymm0, (%rdi,%rax)
    vmovdqu %ymm1, 32(%rdi,%rax)
    vmovdqu %ymm2, 64(%rdi,%rax)
    vmovdqu %ymm3, 96(%rdi,%rax)
```

可以发现这段汇编代码是以 32 位为单位进行向量化的

AVX2 是 Intel 开发的 SIMD 指令集扩展，我们在启用后，这里将整数以 256 bit 为单位进行打包然后向量化

为了使其对齐将原本的对齐方式16改为32即可，我们再次make得到汇编代码并保存为 **wr4-2-example1.s**

可以看到 vmovdqu 指令变为了 vmovdqa 指令

LBB0_1:

=>This Inner Loop Header:

Depth=1

##DEBUG_VALUE: test:i <- 0

##DEBUG_VALUE: test:b <- \$rsi

##DEBUG_VALUE: test:a <- \$rdi

.loc 0 16 10 prologue_end

example1.c:16:10

vmovdqa (%rdi,%rax), %ymm0

vmovdqa 32(%rdi,%rax), %ymm1

vmovdqa 64(%rdi,%rax), %ymm2

vmovdqa 96(%rdi,%rax), %ymm3

vpaddb (%rsi,%rax), %ymm0, %ymm0

vpaddb 32(%rsi,%rax), %ymm1, %ymm1

vpaddb 64(%rsi,%rax), %ymm2, %ymm2

vpaddb 96(%rsi,%rax), %ymm3, %ymm3

vmovdqa %ymm0, (%rdi,%rax)

vmovdqa %ymm1, 32(%rdi,%rax)

vmovdqa %ymm2, 64(%rdi,%rax)

vmovdqa %ymm3, 96(%rdi,%rax)

Write-up5

我们根据要求生成 example2.s, 把 example2.s 重命名成 **example2.s.ORG**

```
LBB0_1:                                ## =>This Inner Loop Header:
Depth=1
    ##DEBUG_VALUE: test:i <- 0
    ##DEBUG_VALUE: test:y <- $rsi
    ##DEBUG_VALUE: test:x <- $rdi
    ##DEBUG_VALUE: test:b <- $rsi
    ##DEBUG_VALUE: test:a <- $rdi
    .loc 0 17 9                          ## example2.c:17:9
    movdqa (%rsi,%rax), %xmm0
    movdqa (%rdi,%rax), %xmm1
    .loc 0 17 14 is_stmt 0                ## example2.c:17:14
    pminub %xmm0, %xmm1
    pcmpeqb %xmm0, %xmm1
    movd %xmm1, %ecx
    notb %cl
```

根据要求修改代码, 再次生成汇编代码, 保存为**wr5-example2.s**

```
LBB0_1:                                ## =>This Inner Loop Header:
Depth=1
    ##DEBUG_VALUE: test:i <- 0
    ##DEBUG_VALUE: test:y <- $rsi
    ##DEBUG_VALUE: test:x <- $rdi
    ##DEBUG_VALUE: test:b <- $rsi
    ##DEBUG_VALUE: test:a <- $rdi
    .loc 0 17 13 prologue_end            ## example2.c:17:13
    movdqa (%rsi,%rax), %xmm0
    movdqa 16(%rsi,%rax), %xmm1
    .loc 0 17 12 is_stmt 0                ## example2.c:17:12
    pmaxub (%rdi,%rax), %xmm0
    pmaxub 16(%rdi,%rax), %xmm1
```

```

.loc 0 17 10                                ## example2.c:17:10
movdqa %xmm0, (%rdi,%rax)
movdqa %xmm1, 16(%rdi,%rax)
.loc 0 17 13                                ## example2.c:17:13
movdqa 32(%rsi,%rax), %xmm0
movdqa 48(%rsi,%rax), %xmm1
.loc 0 17 12                                ## example2.c:17:12
pmaxub 32(%rdi,%rax), %xmm0
pmaxub 48(%rdi,%rax), %xmm1
.loc 0 17 10                                ## example2.c:17:10
movdqa %xmm0, 32(%rdi,%rax)
movdqa %xmm1, 48(%rdi,%rax)

```

我们可以很明显地发现，**example2.s.ORG**中并没有进行向量化操作，而**wr5-example2.s**进行了以16位为单位的向量化操作

这是因为if-else语句在编译的时候编译器需要维护跳转表，而三元运算符表达式不需要维护，所以更容易进行向量化。

(代码的主要区别是**LBB0_1**，因此两个文件只展示对应的部分)

Write-up6:

根据要求生成汇编代码保存为`wr6-example3.s`

exampl3.c的源代码为

```
// Copyright (c) 2015 MIT License by 6.172 Staff

#include <stdint.h>
#include <stdlib.h>
#include <math.h>

#define SIZE (1L << 16)

void test(uint8_t * restrict a, uint8_t * restrict b) {
    uint64_t i;

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

为什么没有出现向量化指令？

这段代码中，数组 `b` 的访问模式是 `b[i+1]`，这种错位访问模式会导致向量化变得复杂。编译器需要保证内存访问的对齐，而 `b[i+1]` 的访问模式会破坏内存对齐

编译器对这个循环进行向量化优化是否可以生成更加高性能的代码？为什么？

不一定

代码已经是连续的内存访问模式，这种访问模式对缓存友好，可以很好地利用硬件预取机制，CPU的缓存行已经在硬件层面优化了连续内存访问。而向量化可能需要额外的数据对齐和打包/解包操作，这些额外操作的开销可能抵消向量化带来的收益。

对于这种简单的内存拷贝操作，标量代码可能已经足够高效

Write-up7

根据要求生成 example4.s, 把 example4.s 重命名成 **example4.s.ORG**

```
Ltmp0:
    ##DEBUG_VALUE: test:x <- $rdi
    xorpd %xmm0, %xmm0
    xorl  %eax, %eax
Ltmp1:
    ##DEBUG_VALUE: test:i <- 0
    ##DEBUG_VALUE: test:y <- 0.000000e+00
    .p2align 4, 0x90
LBB0_1:                                ## =>This Inner Loop Header:
Depth=1
    ##DEBUG_VALUE: test:x <- $rdi
    ##DEBUG_VALUE: test:a <- $rdi
    ##DEBUG_VALUE: test:y <- $xmm0
    ##DEBUG_VALUE: test:i <- $rax
    .loc 0 18 7 prologue_end           ## example4.c:18:7
    addsd (%rdi,%rax,8), %xmm0
Ltmp2:
    ##DEBUG_VALUE: test:i <- [DW_OP_constu 1, DW_OP_or,
DW_OP_stack_value] $rax
    ##DEBUG_VALUE: test:y <- $xmm0
    addsd 8(%rdi,%rax,8), %xmm0
Ltmp3:
    ##DEBUG_VALUE: test:i <- [DW_OP_constu 2, DW_OP_or,
DW_OP_stack_value] $rax
    ##DEBUG_VALUE: test:y <- $xmm0
    addsd 16(%rdi,%rax,8), %xmm0
Ltmp4:
    ##DEBUG_VALUE: test:i <- [DW_OP_constu 3, DW_OP_or,
DW_OP_stack_value] $rax
    ##DEBUG_VALUE: test:y <- $xmm0
    addsd 24(%rdi,%rax,8), %xmm0
```

```

Ltmp5:
    ##DEBUG_VALUE: test:i <- [DW_OP_constu 4, DW_OP_or,
DW_OP_stack_value] $rax
    ##DEBUG_VALUE: test:y <- $xmm0
    addsd 32(%rdi,%rax,8), %xmm0
Ltmp6:
    ##DEBUG_VALUE: test:i <- [DW_OP_constu 5, DW_OP_or,
DW_OP_stack_value] $rax
    ##DEBUG_VALUE: test:y <- $xmm0
    addsd 40(%rdi,%rax,8), %xmm0
Ltmp7:
    ##DEBUG_VALUE: test:i <- [DW_OP_constu 6, DW_OP_or,
DW_OP_stack_value] $rax
    ##DEBUG_VALUE: test:y <- $xmm0
    addsd 48(%rdi,%rax,8), %xmm0
Ltmp8:
    ##DEBUG_VALUE: test:i <- [DW_OP_constu 7, DW_OP_or,
DW_OP_stack_value] $rax
    ##DEBUG_VALUE: test:y <- $xmm0
    addsd 56(%rdi,%rax,8), %xmm0
Ltmp9:
    ##DEBUG_VALUE: test:y <- $xmm0
    .loc 0 17 26                                ## example4.c:17:26
    addq $8, %rax
Ltmp10:
    ##DEBUG_VALUE: test:i <- $rax
    .loc 0 17 17 is_stmt 0                        ## example4.c:17:17
    cmpq $65536, %rax

```

修改 Makefile, 修改

```
CFLAGS := -Wall -g -std=gnu99 -target $(TARGET)
```

改成

```
CFLAGS := -Wall -g -std=gnu99 -ffast-math -target $(TARGET)
```

接下来生成汇编代码并保存为**wr7-example4.s**

```
Ltmp0:
    ##DEBUG_VALUE: test:x <- $rdi
    xorpd %xmm0, %xmm0
    xorl  %eax, %eax
Ltmp1:
    ##DEBUG_VALUE: test:i <- 0
    ##DEBUG_VALUE: test:y <- 0.000000e+00
    xorpd %xmm1, %xmm1
Ltmp2:
    .p2align 4, 0x90
LBB0_1:                                ## =>This Inner Loop Header:
Depth=1
    ##DEBUG_VALUE: test:y <- 0.000000e+00
    ##DEBUG_VALUE: test:i <- 0
    ##DEBUG_VALUE: test:x <- $rdi
    ##DEBUG_VALUE: test:a <- $rdi
    .loc 0 18 7 prologue_end          ## example4.c:18:7
    addpd (%rdi,%rax,8), %xmm0
    addpd 16(%rdi,%rax,8), %xmm1
    addpd 32(%rdi,%rax,8), %xmm0
    addpd 48(%rdi,%rax,8), %xmm1
    addpd 64(%rdi,%rax,8), %xmm0
    addpd 80(%rdi,%rax,8), %xmm1
    addpd 96(%rdi,%rax,8), %xmm0
    addpd 112(%rdi,%rax,8), %xmm1
Ltmp3:
    .loc 0 17 26                      ## example4.c:17:26
    addq  $16, %rax
    cmpq  $65536, %rax                ## imm = 0x10000
    jne  LBB0_1
Ltmp4:
```

```

## %bb.2:
##DEBUG_VALUE: test:y <- 0.000000e+00
##DEBUG_VALUE: test:i <- 0
##DEBUG_VALUE: test:x <- $rdi
##DEBUG_VALUE: test:a <- $rdi
.loc 0 17 3 is_stmt 0                ## example4.c:17:3
addpd %xmm0, %xmm1
movapd %xmm1, %xmm0
unpckhpd %xmm1, %xmm0                ## xmm0 = xmm0[1],xmm1[1]
addsd %xmm1, %xmm0

```

我们可以发现，在修改前，编译器并没有采用向量化操作，修改后采用了向量化操作。

主要的问题是源代码中这个操作

```

for (i = 0; i < SIZE; i++) {
    y += x[i]; // 累加操作
}

```

-ffast-math是GCC编译器的一个优化选项，它实际上是多个浮点优化标志的组合这里的情况与**-funsafe-math-optimizations**这一标志有关

-funsafe-math-optimizations

- 允许可能改变结果的优化
- 假设参数和结果不是NaN（非数字）
- 假设参数和结果不是±Inf（无穷大）

在不启用-ffast-math这个选项的时候，C/C++的浮点数运算必须严格遵循IEEE 754标准，这意味着编译器必须保持运算的顺序，即浮点数运算不满足结合律： $(a + b) + c$ 可能不等于 $a + (b + c)$ 。

而开启-ffast-math这个选项后，我们将允许编译器重新排序浮点运算以提高性能。

Write-up8

首先我们安装Rosetta 2

```
> softwareupdate --install-rosetta
I have read and agree to the terms of the software license agreement. A list of Apple SLAs may be found here: https://www.apple.com/legal/sla/
Type A and press return to agree: A
2024-12-22 01:59:44.218 softwareupdate[48159:8405004] Package Authoring Error: 072-30124: Package reference com.apple.pkg.RosettaUpdateAuto is missing installKBytes attribute
Install of Rosetta 2 finished successfully
```

安装后我们就可以直接执行编译出的x86_64可执行文件了

安装成功后，我们需要修改一下MakeFile，类似于上一个MakeFile，作出如下修改

```
TARGET := x86_64-apple-darwin#添加这一行
```

```
CFLAGS := -Wall -g -std=gnu99 -target $(TARGET) -arch x86_64#修改这一行
```

```
LDFLAGS := -arch x86_64#修改这一行
```

这样我们就指定编译出的 x86_64 版本的程序

我们编写一个python脚本来自动化测试并绘图

```
import subprocess
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import time
import os
from datetime import datetime

def run_command(command):
    """运行命令并返回输出"""
    process = subprocess.Popen(command, shell=True,
                                stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    output, error = process.communicate()
```

```

        return output.decode(), error.decode()

def get_system_info():
    """获取系统信息"""
    info = {}

    uname_output, _ = run_command("uname -a")
    info['OS'] = uname_output.strip()

    cpu_output, _ = run_command("sysctl -n machdep.cpu.brand_string")
    info['CPU'] = cpu_output.strip()

    clang_output, _ = run_command("clang --version")
    info['Compiler'] = clang_output.split('\n')[0]

    return info

def run_experiment(config_name, make_command):
    """运行单个实验配置"""
    times = []

    run_command("make clean")

    print(f"\n编译 {config_name}...")
    run_command(make_command)

    print(f"运行 {config_name} 测试...")
    for i in range(100):
        print(f"运行 #{i+1}/100")
        output, _ = run_command("./loop")
        time_str = output.split("time:")[1].split("sec")[0].strip()
        times.append(float(time_str))

    return times

def create_plots(results, output_dir):

```

```
"""创建性能对比图表"""
os.makedirs(output_dir, exist_ok=True)

plt.rcParams['font.sans-serif'] = ['Arial Unicode MS']
plt.rcParams['axes.unicode_minus'] = False

df = pd.DataFrame(results)

plt.figure(figsize=(10, 6))
df.mean().plot(kind='bar')
plt.title('平均执行时间对比')
plt.ylabel('平均执行时间(秒)')
plt.xticks(rotation=45)
plt.tight_layout()
plt.savefig(f'{output_dir}/performance_barplot.png')
plt.close()

def save_results(results, system_info, output_dir):
    """保存实验结果"""

    os.makedirs(output_dir, exist_ok=True)

    df = pd.DataFrame(results)
    df.to_csv(f'{output_dir}/raw_results.csv')

    stats = pd.DataFrame({
        'mean': df.mean(),
        'std': df.std(),
        'min': df.min(),
        'max': df.max()
    })

    baseline_mean = stats.loc['基础版本', 'mean']
    stats['speedup'] = baseline_mean / stats['mean']
    stats['improvement_percentage'] = (stats['speedup'] - 1) * 100
```



```
with open(f'{output_dir}/results_summary.txt', 'w') as f:
    f.write("系统信息:\n")
    for key, value in system_info.items():
        f.write(f"{key}: {value}\n")
    f.write("\n")

    f.write("统计结果:\n")
    f.write(stats.to_string())
    f.write("\n\n")

    f.write("性能提升分析:\n")
    for config in stats.index:
        if config != '基础版本':
            improvement = stats.loc[config,
'Improvement_Percentage']
            f.write(f"{config} 相比基础版本提升了
{improvement:.2f}%\n")

def main():
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    output_dir = f"experiment_results_{timestamp}"

    system_info = get_system_info()

    experiments = {
        '基础版本': 'make',
        '向量化': 'make VECTORIZE=1',
        'AVX2': 'make VECTORIZE=1 AVX2=1'
    }

    results = {}
    for config_name, make_command in experiments.items():
        results[config_name] = run_experiment(config_name,
make_command)

    create_plots(results, output_dir)
```

```
save_results(results, system_info, output_dir)

print(f"\n实验完成! 结果保存在 {output_dir} 目录中")

if __name__ == "__main__":
    main()
```

这里我们执行脚本后，得到如下结果

系统信息

- 操作系统: Darwin HUAWEI-MateBook-Pro-2.local 24.1.0
 - Kernel: Darwin Kernel Version 24.1.0 (Thu Oct 10 21:02:45 PDT 2024)
 - 架构: root:xnu-11215.41.3~2/RELEASE_ARM64_T8112 arm64
- CPU: Apple M2
- 编译器: Apple clang version 16.0.0 (clang-1600.0.26.4)

统计结果

版本	mean	std	min	max	speedup	improvement_percentage
基础版本	0.052364	0.079150	0.043658	0.835937	1.000000	0.000000
向量化	0.030741	0.041762	0.025484	0.439185	1.703428	70.342802
AVX2	0.030679	0.040886	0.026126	0.435445	1.706846	70.684620

性能提升分析

- 向量化版本相比基础版本提升了 **70.34%**
- AVX2 版本相比基础版本提升了 **70.68%**

这和我猜测的不太一样，因为我认为AVX2 提供了对 256 位寄存器的支持，相比于之前的 SSE指令集，可以一次性处理更多的数据，理论来说应该执行的时间短于普通的向量化，但是结果却是AVX2的性能更差

我查找了一下苹果官方对于Apple Silicon的文档，对于Rosetta找到了如下内容

What Can't Be Translated?

Rosetta can translate most Intel-based apps, including apps that contain just-in-time (JIT) compilers. However, Rosetta doesn't translate the following executables:

- Kernel extensions
- Virtual Machine apps that virtualize x86_64 computer platforms

Rosetta translates all x86_64 instructions, but it doesn't support the execution of some newer instruction sets and processor features, such as AVX, AVX2, and AVX512 vector instructions. If you include these newer instructions in your code, execute them only after verifying that they are available. For example, to determine if AVX512 vector instructions are available, use the [sysctlbyname](#) function to check the `hw.optional.avx512f` attribute.

但是我这里成功运行了，我猜测，虽然 Rosetta 2 最初不支持 AVX、AVX2 等指令集，但随着 macOS 的更新，苹果可能对 Rosetta 进行了改进，增加了对AVX2指令的支持，或者通过软件模拟的方式处理AVX2指令，但是即使可以执行，也无法利用 AVX2 指令的优化优势，从而导致出现如上的结果。

于是我又在x86_64平台进行了一次测试，结果如下

系统信息

- **操作系统:** Linux harry-virtual-machine 6.8.0-49-generic
 - **架构:** x86_64 GNU/Linux
- **版本:** #49~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Wed Nov 6 17:42:15 UTC 2
- **CPU:** 13th Gen Intel(R) Core(TM) i5-13600KF
- **编译器:** Ubuntu clang version 14.0.0-1ubuntu1.1

同时我的CPU支持AVX2指令集

```

BogoMIPS: 6988.80
Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush m
mx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon rep
_good nopl xtopology tsc_reliable nonstop_tsc cpuid tsc_known_freq pni pclmulqdq
ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsav
e avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch pti ssbd ibrs ibpb stibp f
sgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt cl
wb sha_ni xsaveopt xsavec xgetbv1 xsaves arat umip gfni vaes vpclmulqdq rdpid fsr
m md_clear flush_l1d arch_capabilities

```

统计结果

优化版本	mean	std	min	max	speedup	improvement_percentage
基础版本	0.018615	0.002562	0.016485	0.026773	1.000000	0.000000
向量化	0.008169	0.000426	0.007323	0.009669	2.278777	127.877741
AVX2	0.006883	0.001057	0.005419	0.008940	2.704560	170.456037

性能提升分析

向量化版本相比基础版本提升了 127.88%

AVX2 版本相比基础版本提升了 170.46%

可以看到，得益于AVX 指令可以进行 256 位宽的操作，并行度更高，在使用AVX2后我们程序执行的效率更高，运行速度更快。