# A4实验报告

## Write-up 1:找出 gcc/g++和 clang/clang++的对应可执行文件分别在哪并且解释有何不同？为什么要使用符号链接?

我们首先通过which指令找到了可执行文件所在的目录

```
harry@harry-virtual-machine:~/CppPerformanceBenchmarks$ which gcc g++
/usr/bin/gcc
/usr/bin/g++
harry@harry-virtual-machine:~/CppPerformanceBenchmarks$ which clang clang++
/usr/bin/clang
/usr/bin/clang++
```

使用符号链接后，作为用户，我们只需要知道调用的通用名称，而不需要关心具体的版本。同时在版本升级时，只需要将更新的符号链接指向目标，而不必修改每个依赖于编译器路径的配置。

## Write-up 2:请说明一下你选择了哪个测试例，以及你选择这个测试例的原因。

***我选择了matrix_multiply，原因有以下三点***

- **广泛应用**：`matrix_multiply` 是一个基础算法，广泛用于科学计算、机器学习和图像处理等领域，测试其性能具有实际意义。
- **计算密集型**：矩阵乘法涉及大量计算，能够很好地体现不同编译器及优化选项的性能差异。

- **优化潜力**：矩阵乘法的性能容易受到编译器优化的影响，特别是对循环展开、缓存利用和 SIMD 指令的优化。

# Write-up 3:你对 makefile 及所选测试例对应的源文件做了哪些修改，为什么?

首先我在开头添加了如下三行

```
CC ?= gcc
CXX ?= g++
OPTLEVEL ?= -O3
```

这三行是为了我们在执行make的时候可以正确地输入我们的六组参数

接下来我注释掉了 `BINARIES` 和 `report` 中除了matrix_multiply 的所有部分，因为我们只需要编译和测试 matrix_multiply

***对于这个Makefile的主要部分和功能的解释如下***

1.Macros（宏定义部分）

```
INCLUDE = -I.#指定编译器查找头文件的路径。-I.表示将当前目录作为头文件的搜索路径。

CC ?= gcc
CXX ?= g++#指定 C 和 C++ 的编译器。

OPTLEVEL ?= -O3#指定优化等级

CFLAGS = $(INCLUDE) $(OPTLEVEL)
CPPFLAGS = -std=c++14 $(INCLUDE) $(OPTLEVEL)
#分别用于 C 和 C++ 文件的编译选项:
#$(INCLUDE): 包含头文件路径。
#$(OPTLEVEL): 设置优化级别。

CLIBS = -lm#编译链接时所需的库选项，这里包括 -lm，用于链接数学库。
CPPLIBS = -lm
```

```
DEPENDENCYFLAG = -M#用于生成依赖项的标志，-M 告诉编译器输出目标的依赖关系。
```

总结来看，这部分设定了编译时的各项参数配置

## 2.Targets（目标部分）

```
BINARIES = matrix_multiply
```

这部分用于定义需要生成的可执行文件目标

## 3.Build Rules（构建规则部分）

```
all : $(BINARIES)#指定 make 默认生成的目标，这里BINARIES在上一部分已经给定

SUFFIXES:
.SUFFIXES: .c .cpp#显式声明文件的后缀类型规则，清空原有规则后重新定义 .c 和
.cpp

.PHONY : clean dependencies#声明 clean 和 dependencies 是伪目标，不依赖具
体文件

clean :
    rm -f *.o $(BINARIES)#删除所有生成的目标文件和中间文件

#使用 -M 生成所有源文件的依赖信息
SOURCES = $(wildcard *.c)  $(wildcard *.cpp)
dependencies :   $(SOURCES)
  $(CXX) $(DEPENDENCYFLAG) $(CPPFLAGS) $^
```

## 4.Special Case Compiles（特殊编译规则）

```
#编译时使用 $(CC) 和 $(CFLAGS) 进行构建
exceptions : exceptions.c
  $(CC) $(CFLAGS) -o $@ exceptions.c $(CLIBS)

exceptions_cpp : exceptions.c
  $(CXX) $(CPPFLAGS) -D TEST_WITH_EXCEPTIONS=1 -o $@ exceptions.c
$(CPPLIBS)
```

5.Run the Benchmarks and Generate a Report（运行基准测试并生成报告）

```
#定义了report目标，用于运行测试并生成报告
REPORT_FILE = report.txt

report:  $(BINARIES)
  echo "##STARTING Version 1.0" > $(REPORT_FILE)#写入报告开头信息
  date >> $(REPORT_FILE)#记录开始时间
  echo "##Compiler: $(CC) $(CXX)" >> $(REPORT_FILE)#记录编译器和编译选项
  echo "##CFlags: $(CFLAGS)" >> $(REPORT_FILE)
  echo "##CPPFlags: $(CPPFLAGS)" >> $(REPORT_FILE)
  ./matrix_multiply >> $(REPORT_FILE)  #运行目标程序 matrix_multiply 并将
输出附加到报告中
  date >> $(REPORT_FILE)#记录结束时间
  echo "##END Version 1.0" >> $(REPORT_FILE)
```

至于对应的源文件，我根据其提示修改了iterations。提示如下

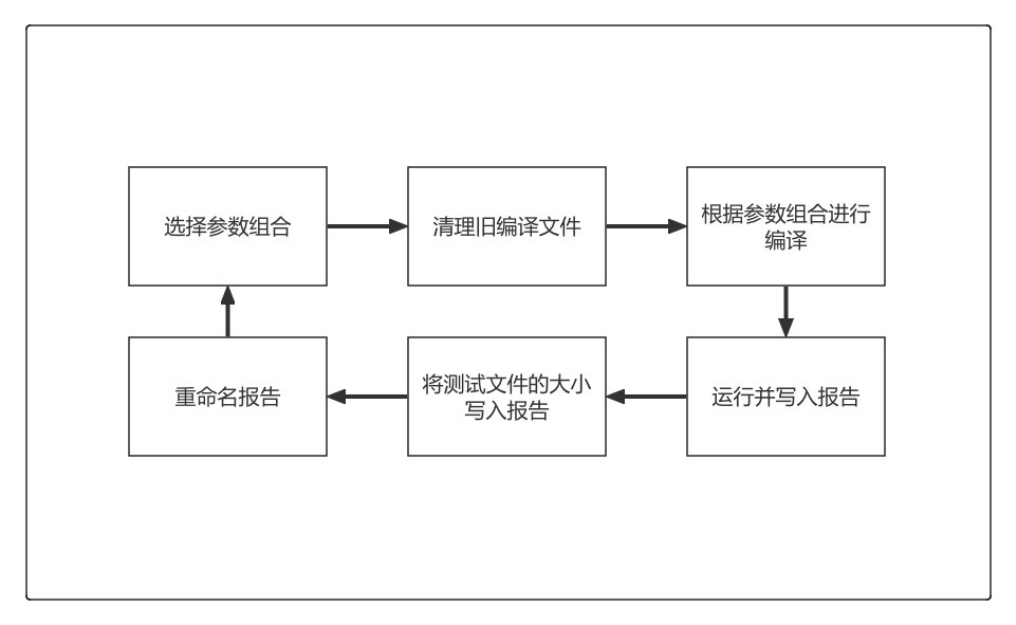
```
// this constant may need to be adjusted to give reasonable minimum
times
// For best results, times should be about 1.0 seconds for the minimum
test run
```

因此我调整了iteration，使得最短的测试时间为1s左右。

# Write-up 4：自动化脚本实现对测试组合进行测试并获得结果

为了完成自动化地测试，我通过编写sh脚本来实现。

首先构建脚本运行的流程




该流程循环进行直到遍历完六种参数配置

根据流程可以编写如下sh脚本，我将最终的报告重命名并移动到了output目录下

```
#!/bin/bash

# 创建 output 文件夹（如果不存在的话）
mkdir -p output

# 遍历不同的编译器和优化级别
for compiler in gcc clang
do
  # 根据编译器选择正确的 CXX
  if [ "$compiler" == "gcc" ]; then
```

```
    CXX="g++"
  elif [ "$compiler" == "clang" ]; then
    CXX="clang++"
  fi

  for optlevel in O0 O1 O2
  do
    # 每次测试前清理旧的编译文件
    echo "Cleaning previous builds..."
    make clean

    # 编译并生成报告
    echo "Running test with $compiler and -$optlevel optimization..."
    make report CC=$compiler CXX=$CXX OPTLEVEL=-$optlevel

    # 获取编译后的测试例文件的大小
    test_binary="matrix_multiply"
    if [ -f $test_binary ]; then
      file_size=$(stat --format=%s $test_binary)
      echo "File size of $test_binary: $file_size bytes"
    else
      file_size="N/A"
    fi

    # 如果报告文件存在，重命名并移到 output 文件夹
    if [ -f report.txt ]; then
      mv report.txt output/report.txt.$compiler-$optlevel
      echo "Report saved as output/report.txt.$compiler-$optlevel"
      # 将文件大小信息添加到报告中
      echo "File size of $test_binary: $file_size bytes" >>
output/report.txt.$compiler-$optlevel
    else
      echo "Error: report.txt not found. Test might have failed."
    fi
  done
done
```

```
echo "All tests completed."
```

## Write-up 5：请针对选定的测试例及收集到的 6 个测试数据结果文件（以及相应测试例编译后的代码大小），进行数据分析，总结分析洞见。

*测试例大小分析*

六种测试例的代码大小为

| 参数配置 | 测试例大小（bytes） |
| --- | --- |
| gcc-O0 | 227,856 |
| gcc-O1 | 167,008 |
| gcc-O2 | 155,856 |
| clang-O0 | 197,168 |
| clang-O1 | 219,912 |
| clang-O2 | 342,792 |

***我们可以发现***

- GCC随着优化级别提高，代码体积逐渐减小
- Clang则相反，优化级别越高代码体积越大，特别是O2级别下增长显著
- 在O0级别下，Clang生成的代码比GCC小；但在O1和O2级别下，Clang生成的代码显著大于GCC

我认为可能的原因有

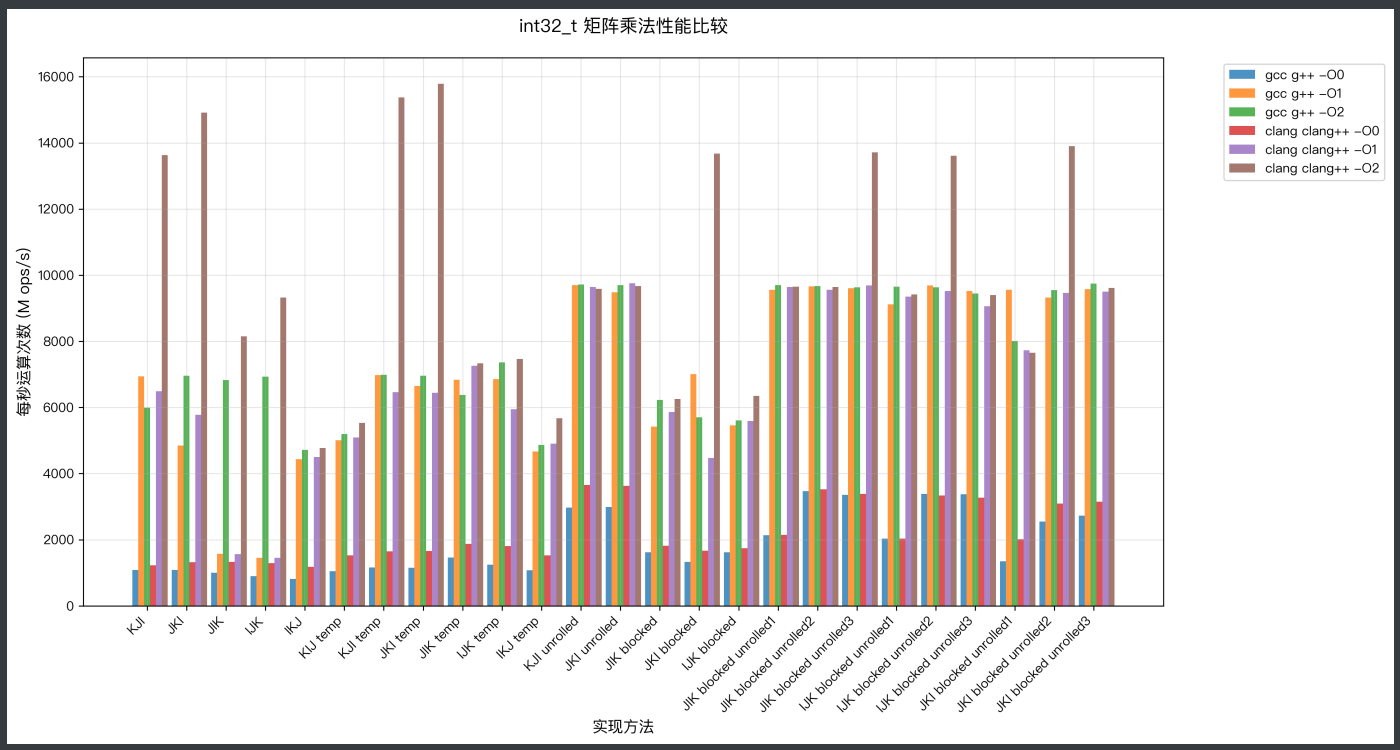
- GCC在高优化级别下倾向于合并重复代码，消除死代码
- Clang在高优化级别下可能进行了更激进的循环展开和向量化，导致代码膨胀

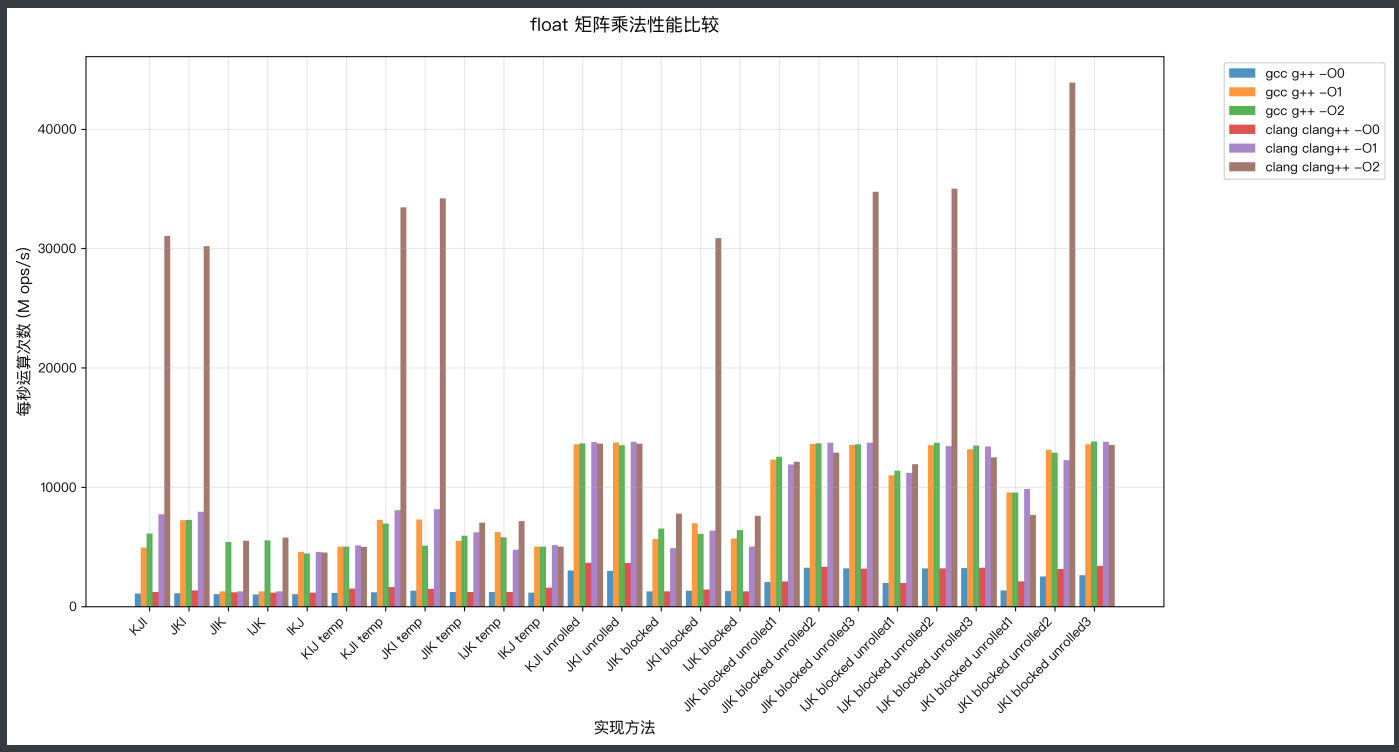
## *性能分析*

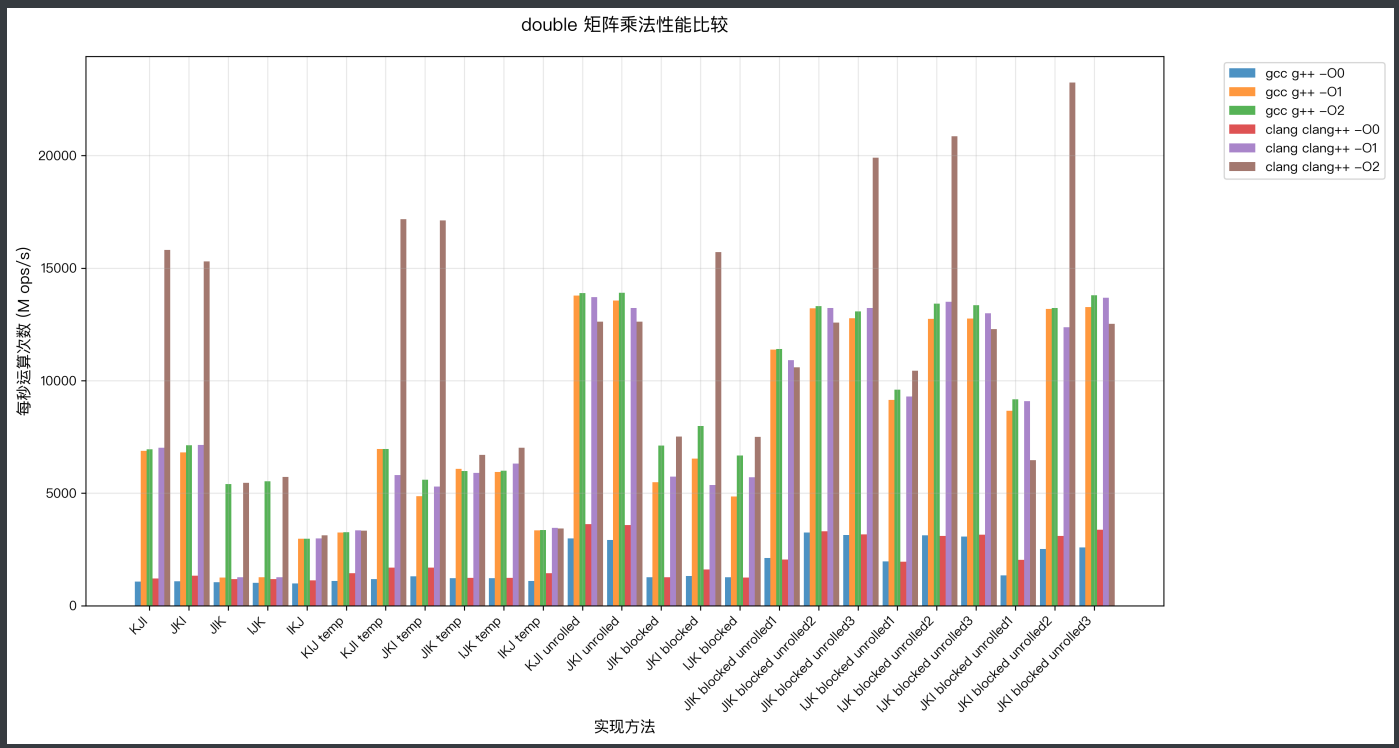
### *我们首先将报告的结果进行可视化*

由于单个实现方法的运行耗时和每秒的的操作数是反应同一性能的不同指标，因此这里我选择以每秒的操作数为主

我们将以三张不通的数据类型的 `operations per second` 结果来绘制三张表






整体上来看，gcc/g++同一测试中，O2级别的优化基本能取得最佳的性能。

clang/clang++和gcc/g++的情况类似，但是出现O2优化等级性能差于O1的情况略多。不过其在多个任务下有着远超其他优化等级和编译器的性能，Clang-O2在float类型上，最高性能可达43,904 MOPS（JKI blocked unrolled2）。

综合来看，相比O0，O2优化可带来5-10倍的性能提升。

*接下来我们进行更细致的分析*

**数据类型影响：**

- float类型在高优化级别下性能最好
- double类型总体性能略低于float
- int32_t在基础实现方式下性能较好，但优化提升空间较小

**实现方法比较：**

- 基础实现（KIJ, KJI等）在无优化时差异不大
- blocked + unrolled组合在所有情况下都表现较好
- JKI和KJI在优化后性能提升明显

由此我们可以总结出**编译器特点：**

GCC优化更保守，性能提升平稳

Clang在O2级别的优化更激进，性能波动较大但峰值更高

## 深入分析 1： 为什么float类型在高优化级别下性能最好

对于float类型为什么在高优化级别下性能最好我猜测是CPU对于浮点数的计算有特别的优化，通过查阅资料，我得知SIMD指令对于浮点数的操作有对应的优化。我们首先来做个简单的测试，我们编写代码来测试编译器在是否开启SIMD优化的情况下，运行的结果有什么差异。

```
#include <iostream>
#include <chrono>
#include <vector>

// 避免编译器过度优化
```

```cpp
volatile int dummy = 0;

template<typename T>
void simple_compute(const std::vector<T>& a, const std::vector<T>& b,
std::vector<T>& c, int n) {
    for(int i = 0; i < n; i++) {
        // 简单的计算: c[i] = a[i] * b[i] + a[i]
        // 使用volatile防止编译器因为c[i]没有使用过而将循环完全优化掉
        c[i] = a[i] * b[i] + a[i];
        dummy += (c[i] > T(0)) ? 1 : 0;
    }
}

template<typename T>
void test_type() {
    const int N = 10000000;
    std::vector<T> a(N, T(1.5));
    std::vector<T> b(N, T(2.0));
    std::vector<T> c(N);

    auto start = std::chrono::high_resolution_clock::now();


    for(int i = 0; i < 100; i++) {
        simple_compute(a, b, c, N);
    }

    auto end = std::chrono::high_resolution_clock::now();
    auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(end - start);

    std::cout << "Type size: " << sizeof(T) << " bytes\n";
    std::cout << "Time taken: " << duration.count() << " ms\n";
    std::cout << "Dummy value: " << dummy << "\n\n";
}
```

```cpp
int main() {
    std::cout << "Testing int32_t:\n";
    test_type<int32_t>();

    std::cout << "Testing float:\n";
    test_type<float>();

    std::cout << "Testing double:\n";
    test_type<double>();

    return 0;
}
```

我们采用如下指令编译并运行

```bash
# 使用GCC，不开启SIMD
g++ -O2 verify_simd.cpp -o verify_gcc_no_simd -mno-sse -mno-avx

# 使用GCC，开启SIMD
g++ -O2 verify_simd.cpp -o verify_gcc_simd -march=native

# 使用Clang，不开启SIMD
clang++ -O2 verify_simd.cpp -o verify_clang_no_simd -mno-sse -mno-avx

# 使用Clang，开启SIMD
clang++ -O2 verify_simd.cpp -o verify_clang_simd -march=native

# 运行所有版本
./verify_gcc_no_simd
./verify_gcc_simd
./verify_clang_no_simd
./verify_clang_simd
```

最终我们得到如下结果，由于在double类型的计算中，dummy为int_32，发生了溢出，但是这不影响我们的测试需求

| Test Type | Data Type | Type Size (bytes) | Time Taken (ms) | Dummy Value |
|---|---|---|---|---|
| gcc_no_simd | int32_t | 4 | 1581 | 1000000000 |
| gcc_no_simd | float | 4 | 1539 | 2000000000 |
| gcc_no_simd | double | 8 | 1724 | -1294967296 |
| gcc_simd | int32_t | 4 | 1562 | 1000000000 |
| gcc_simd | float | 4 | 1458 | 2000000000 |
| gcc_simd | double | 8 | 1421 | -1294967296 |
| clang_no_simd | int32_t | 4 | 1624 | 1000000000 |
| clang_no_simd | float | 4 | 1547 | 2000000000 |
| clang_no_simd | double | 8 | 1518 | -1294967296 |
| clang_simd | int32_t | 4 | 1579 | 1000000000 |
| clang_simd | float | 4 | 1392 | 2000000000 |
| clang_simd | double | 8 | 1408 | -1294967296 |

我们可以发现，时间消耗最少的确实是clang/clang++在开启SIMD优化的float类型。我们可以查看一下 `verify_clang_simd` 的汇编代码，得到如下结果。

***int_32***

```
# 数据加载使用256位YMM寄存器，一次处理8个int32
vbroadcastss .LCPI1_0(%rip), %ymm0    # 广播到ymm0的8个位置
vmovups %ymm0, 40000000(%r14,%rax)    # 向量存储

# 计算部分
movl 40000000(%r14,%rax), %ecx        # 标量加载
imull %ecx, %edx                      # 标量乘法
addl %ecx, %edx                       # 标量加法
```

***float***

```
# 数据加载使用256位YMM寄存器，一次处理8个float
vbroadcastss .LCPI2_0(%rip), %ymm0    # 广播到ymm0的8个位置
vmovups %ymm0, 40000000(%r14,%rax)    # 向量存储

# 关键计算部分使用SIMD指令
vmovss 40000000(%r14,%rax), %xmm1     # 加载单个float
vfmadd231ss 40000000(%r15,%rax), %xmm1, %xmm1  # 融合乘加指令(FMA)
# vfmadd231ss执行: xmm1 = (xmm1 * mem) + xmm1
```

***double***

```
# 使用256位YMM寄存器，一次处理4个double
vbroadcastsd .LCPI3_0(%rip), %ymm0    # 广播到ymm0的4个位置
vmovupd %ymm0, 80000000(%r14,%rax)    # 向量存储

# 计算部分
vmovsd 80000000(%r14,%rax), %xmm1     # 加载单个double
vfmadd231sd 80000000(%r15,%rax), %xmm1, %xmm1  # 融合乘加指令
```

总结来看

float和int_32相比具有指令优势：

- 使用了高效的FMA指令

和double相比具有数据密度：

- 一个YMM寄存器可以处理8个float
- 内存带宽利用效率高
- 缓存命中率好

## 深入分析 2： 为什么同一任务下，clang/clang++会出现O2优化等级性能远超其他参数配置的情况

为了分析这个问题，我选择了两个测试例来进行分析，首先是float的matmul_JKI_blocked_unrolled2，用于分析为何会出现clang/clang++会出现O2优化等级性能远超其他参数配置的情况，其次是float的matmul_JKI_blocked_unrolled3，用于分析为什么clang/clang++会出现O2优化等级的性能并没有出现质的超越，甚至不如clang/clang++O1优化等级的性能的情况。

***第一个问题，为何会出现clang/clang++会出现O2优化等级性能远超其他参数配置的情况***

首先由于原测试文件过于繁琐，我们将需要的函数截取并独立出为文件，去除原本的测试框架进行编译，注意在完成代码的时候要防止编译器出现优化掉死代码的情况（gcc/g++在不做预防的时候出现了这个情况）。随后我们编译三个组合，gcc/g++ O2、`clang/clang++ 01`、`clang/clang++ 02`，我们找到我们关注的计算部分汇编代码。

```
# gcc O2中的实现
.L6:
    movss   (%r11,%rax,4), %xmm0
    movss   (%rdx,%rax,4), %xmm1
    mulss   %xmm3, %xmm0        # 标量乘法
    mulss   %xmm2, %xmm1
    addss   %xmm1, %xmm0        # 标量加法
```

```
# clang O1中的实现
.LBB1_11:
    movss   (%r11,%r13,4), %xmm4        # 标量加载
    mulss   %xmm1, %xmm4                # 标量乘法
    movss   (%r8,%r13,4), %xmm5
    mulss   %xmm0, %xmm5
    addss   %xmm4, %xmm5                # 标量加法
    movss   (%rcx,%r13,4), %xmm4
    mulss   %xmm2, %xmm4
    addss   %xmm5, %xmm4
    movss   (%r9,%r13,4), %xmm5
    mulss   %xmm3, %xmm5
```

```
    addss   %xmm4, %xmm5
    addss   (%r14,%r13,4), %xmm5
    movss   %xmm5, (%r14,%r13,4)
```

```
# clang O2中的实现
.LBB1_16:
    movups  (%rsi,%rax,4), %xmm4
    mulps   %xmm9, %xmm4        # 向量化乘法
    movups  (%rdx,%rax,4), %xmm5
    mulps   %xmm8, %xmm5
    addps   %xmm4, %xmm5        # 向量化加法
    movups  (%r10,%rax,4), %xmm4
    mulps   %xmm6, %xmm4
    addps   %xmm5, %xmm4
    movups  (%rdi,%rax,4), %xmm5
    mulps   %xmm7, %xmm5
    addps   %xmm4, %xmm5
```

到这里我们其实已经可以知道了第一个问题的原因，clang充分利用了SIMD指令的能力。

***第二个问题，为什么clang/clang++会出现O2优化等级的性能并没有出现质的超越，甚至不如clang/clang++O1优化等级的性能的情况***

同上，我们替换一下测试代码的函数为 `matmul_JKI_blocked_unrolled3`，我们这次编译clang/clang++O1和clang/clang++O2两个配置的版本，并得到汇编代码。首先单从行数来看差距已经非常明显，O2版本的有***1579***行，O1仅有***631***行，这里我们就可以猜测到O2进行了极度激进的优化导致代码的复杂度提高。我们查看关键的计算部分的汇编代码。

```
#O1版本的最内层的循环
.LBB1_11:
    # 4x4循环展开，每次处理4个元素
    movss   -12(%rbp,%rax,4), %xmm4     # 加载单个float
    mulss   %xmm1, %xmm4                # 标量乘法
    movss   -12(%r8,%rax,4), %xmm5
    mulss   %xmm0, %xmm5
    addss   %xmm4, %xmm5                # 标量加法
    # ... (重复4次)
    addq    $4, %rax                    # 递增索引
    cmpq    %r9, %rcx                   # 循环条件检查
    jb      .LBB1_11                    # 循环跳转
```

```
# O2版本的最内层循环
.LBB1_29:
    # 使用SIMD指令进行向量化
    movups  (%r14,%rdi,4), %xmm4        # 加载4个float
    mulps   %xmm9, %xmm4                # 向量乘法(4个同时)
    movups  (%rsi,%rdi,4), %xmm5
    mulps   %xmm8, %xmm5
    addps   %xmm4, %xmm5                # 向量加法(4个同时)
    # ... (更多向量操作)

    # 复杂的数据重排
    shufps  $36, %xmm7, %xmm12          # 向量元素重排
    movaps  %xmm1, %xmm7
    unpcklps %xmm11, %xmm7              # 向量解包
```

如此对比，我们就可以得知，O2优化进行了太多复杂的向量操作，尤其是数据重排造成额外延迟和缓存的缺失，同时更多的操作也会产生更多的时间开销，我们可以用valgrind工具来进行测试。

首先是clang/clang++ O1编译 `matmul_JKI_blocked_unrolled3` 的结果

```
harry@harry-virtual-machine:~/CppPerformanceBenchmarks$ valgrind --tool=cachegrind ./test_matmul
==37928== Cachegrind, a cache and branch-prediction profiler
==37928== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==37928== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==37928== Command: ./test_matmul
==37928==
--37928-- warning: L3 cache found, using its data for the LL simulation.
### unhandled dwarf2 abbrev form code 0x25
### unhandled dwarf2 abbrev form code 0x25
### unhandled dwarf2 abbrev form code 0x25
### unhandled dwarf2 abbrev form code 0x23
Result checksum: 2.03179e+09
==37928==
==37928== I   refs:      9,117,711,637
==37928== I1  misses:            2,337
==37928== LLi misses:            2,245
==37928== I1  miss rate:          0.00%
==37928== LLi miss rate:          0.00%
==37928==
==37928== D   refs:      3,036,961,104  (2,524,759,264 rd   + 512,201,840 wr)
==37928== D1  misses:       60,616,342  (   60,238,924 rd   +     377,418 wr)
==37928== LLd misses:          384,616  (        7,967 rd   +     376,649 wr)
==37928== D1  miss rate:           2.0% (          2.4%     +         0.1%  )
==37928== LLd miss rate:           0.0% (          0.0%     +         0.1%  )
==37928==
==37928== LL refs:          60,618,679  (   60,241,261 rd   +     377,418 wr)
==37928== LL misses:           386,861  (       10,212 rd   +     376,649 wr)
==37928== LL miss rate:            0.0% (          0.0% _    +         0.1%  )
```

接下来是clang/clang++ O2编译 `matmul_JKI_blocked_unrolled3` 的结果

```
harry@harry-virtual-machine:~/CppPerformanceBenchmarks$ valgrind --tool=cachegrind ./test_matmul
==36139== Cachegrind, a cache and branch-prediction profiler
==36139== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==36139== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==36139== Command: ./test_matmul
==36139==
--36139-- warning: L3 cache found, using its data for the LL simulation.
### unhandled dwarf2 abbrev form code 0x25
### unhandled dwarf2 abbrev form code 0x25
### unhandled dwarf2 abbrev form code 0x25
### unhandled dwarf2 abbrev form code 0x23
Result checksum: 2.03179e+09
==36139==
==36139== I   refs:      6,542,222,728
==36139== I1  misses:            2,396
==36139== LLi misses:            2,303
==36139== I1  miss rate:          0.00%
==36139== LLi miss rate:          0.00%
==36139==
==36139== D   refs:      1,811,366,546  (1,394,361,173 rd   + 417,005,373 wr)
==36139== D1  misses:       62,391,771  (   62,014,339 rd   +     377,432 wr)
==36139== LLd misses:          384,684  (        8,035 rd   +     376,649 wr)
==36139== D1  miss rate:           3.4% (          4.4%     +         0.1%  )
==36139== LLd miss rate:           0.0% (          0.0%     +         0.1%  )
==36139==
==36139== LL refs:          62,394,167  (   62,016,735 rd   +     377,432 wr)
==36139== LL misses:           386,987  (       10,338 rd   +     376,649 wr)
==36139== LL miss rate:            0.0% (          0.0%     +         0.1%  )
```

这里我为了对比，添加了一个clang/clang++ O2编译 `matmul_JKI_blocked_unrolled2` 的结果

```
harry@harry-virtual-machine:~/CppPerformanceBenchmarks$ valgrind --tool=cachegrind ./test_matmul
==36383== Cachegrind, a cache and branch-prediction profiler
==36383== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==36383== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==36383== Command: ./test_matmul
==36383==
--36383-- warning: L3 cache found, using its data for the LL simulation.
### unhandled dwarf2 abbrev form code 0x25
### unhandled dwarf2 abbrev form code 0x25
### unhandled dwarf2 abbrev form code 0x25
### unhandled dwarf2 abbrev form code 0x23
Result checksum: 2.03179e+09
==36383==
==36383== I   refs:      2,732,216,955
==36383== I1  misses:            2,360
==36383== LLi misses:            2,267
==36383== I1  miss rate:          0.00%
==36383== LLi miss rate:          0.00%
==36383==
==36383== D   refs:      1,121,764,125  (906,760,402 rd   + 215,003,723 wr)
==36383== D1  misses:       61,170,983  ( 60,793,560 rd   +     377,423 wr)
==36383== LLd misses:          384,646  (      7,997 rd   +     376,649 wr)
==36383== D1  miss rate:           5.5% (        6.7%     +         0.2%  )
==36383== LLd miss rate:           0.0% (        0.0%     +         0.2%  )
==36383==
==36383== LL refs:          61,173,343  ( 60,795,920 rd   +     377,423 wr)
==36383== LL misses:           386,913  (     10,264 rd   +     376,649 wr)
==36383== LL miss rate:            0.0% (        0.0%     +         0.2%  )
```

还有gcc/g++ O2编译 `matmul_JKI_blocked_unrolled2` 的结果

```
harry@harry-virtual-machine:~/CppPerformanceBenchmarks$ valgrind --tool=cachegrind ./test_matmul
==36507== Cachegrind, a cache and branch-prediction profiler
==36507== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==36507== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==36507== Command: ./test_matmul
==36507==
--36507-- warning: L3 cache found, using its data for the LL simulation.
Result checksum: 2.03179e+09
==36507==
==36507== I   refs:      8,106,300,619
==36507== I1  misses:            2,325
==36507== LLi misses:            2,232
==36507== I1  miss rate:          0.00%
==36507== LLi miss rate:          0.00%
==36507==
==36507== D   refs:      3,037,555,785  (2,525,354,906 rd   + 512,200,879 wr)
==36507== D1  misses:       60,425,942  (   60,048,526 rd   +     377,416 wr)
==36507== LLd misses:          384,602  (        7,953 rd   +     376,649 wr)
==36507== D1  miss rate:           2.0% (          2.4%     +         0.1%  )
==36507== LLd miss rate:           0.0% (          0.0%     +         0.1%  )
==36507==
==36507== LL refs:          60,428,267  (   60,050,851 rd   +     377,416 wr)
==36507== LL misses:           386,834  (       10,185 rd   +     376,649 wr)
==36507== LL miss rate:            0.0% (          0.0%     +         0.1%  )
```

我们绘制表格

| 指标 | clang++ O1 unrolled3 | clang++O2 unrolled3 | g++ O2unrolled2 | clang++O2 unrolled2 |
| --- | --- | --- | --- | --- |
| 指令引用(Irefs) | 9.12B | 6.54B | 8.11B | 2.73B |
| 数据引用(Drefs) | 3.04B | 1.81B | 3.04B | 1.12B |
| - 读操作 | 2.52B | 1.39B | 2.53B | 0.91B |
| - 写操作 | 0.51B | 0.42B | 0.51B | 0.22B |
| D1缓存未命中率 | 2.0% | 3.4% | 2.0% | 5.5% |
| D1未命中次数 | 60.62M | 62.39M | 60.43M | 61.17M |
| LL未命中次数 | 386.9K | 387.0K | 386.8K | 386.9K |

根据内容我们可以发现，unrolled3中，clang++O2的缓存未命中次数和未命中率均高于clang++O1。且在unrolled2中，虽然clang++O2的缓存命中率高于g++O2，但是其指令引用远低于其他三种情况，并且clang++的数据引用都低于g++。

## 总结

至此我们其实可以总的来看两个深入分析的现象。

由于clang++O2优化会使用SIMD指令进行向量化的操作，使得其数据的引用次数降低。这时会有两种情况，第一种为clang++O2的优化非常好，使得指令的引用大幅度降低，和数据引用结合，产生了飞跃般的性能提升；第二种情况为clang++O2的优化过于激进，导致指令引用很高，性能提升不大，甚至有时候会出现性能不如O1。