A Brief Introduction to GCC and LLVM

莫泽威

4/22/2021

中山大學
SUN YAT-SEN UNIVERSITY

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

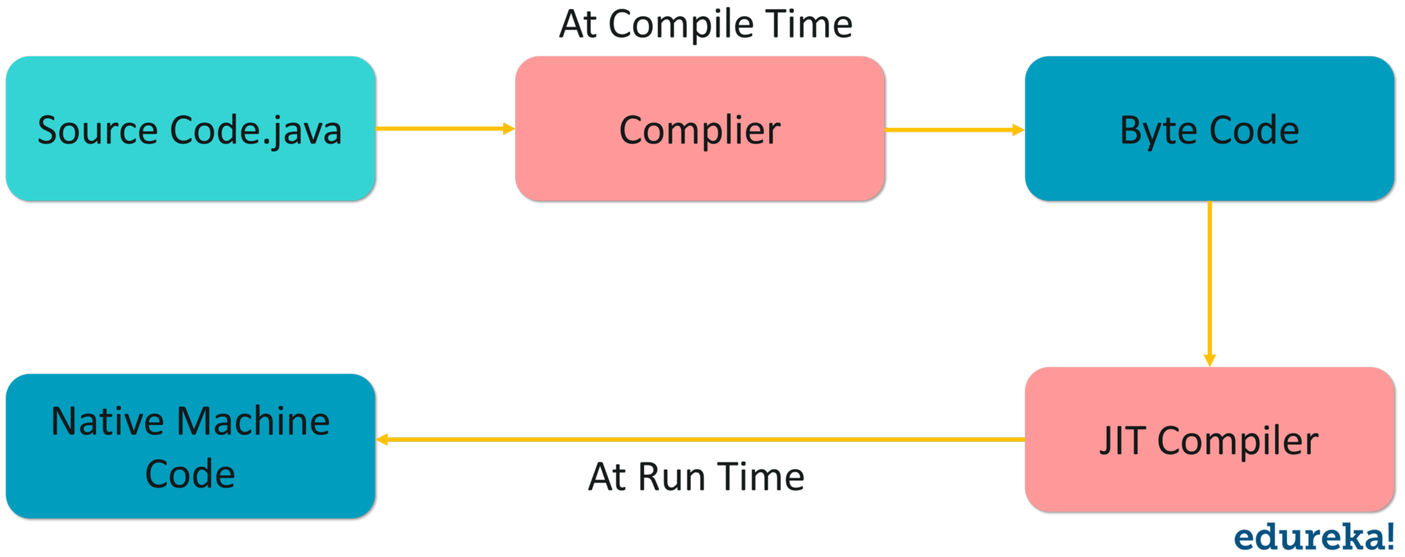
Agenda

- Category of Compiler
- What is LLVM
- History of GCC and LLVM
- Differences Between GCC and LLVM
- Optimizations of Compiler
  - Official Optimizations in GCC/LLVM
  - Custom Optimizations in GCC/LLVM
- GCC/LLVM, Which to Use
- Question?

Category of Compiler

1. Target Platform (native and cross)

2. Compilation Time
   - AOT : Ahead-of-Time
   - JIT : Just-in-Time
3. Output Code Type
   - High-level Language
   - Bytecode (Python/Java)
   - Assembly/Machine Code



What type do GCC and LLVM belong to?

What is LLVM?

Low Level Virtual Machine(低级虚拟机)

It is a compiler framework and use uniform language (IR) to present multiple languages

Also it can generate instructions for multiple platforms

It’s bigger than you think, it includes frontend(Clang), assembler and so on…

History of GCC* and LLVM*

GCC (GNU C Compiler → GNU Compiler Collection)

In 1987 March, Stallman and MIT released GCC under GPL License.

Many developers were free to develop their own fork of GCC but this is inefficient and hard to get accepted by GCC project.

EGCS merged multiple forks into one project and released it as official GCC.

GCC was implemented by C/C++ in version 4.8.

[1]https://en.wikipedia.org/wiki/GNU_Compiler_Collection#History
[2]https://en.wikipedia.org/wiki/LLVM#History

History of GCC and LLVM (cont.)

LLVM

LLVM project began from 2000 by University of Illinois by Chris Lattner and it's formal published paper was on 2004 CGO.

Apple hired Chris Lattner to develop LLVM as formal development tool in 2005.

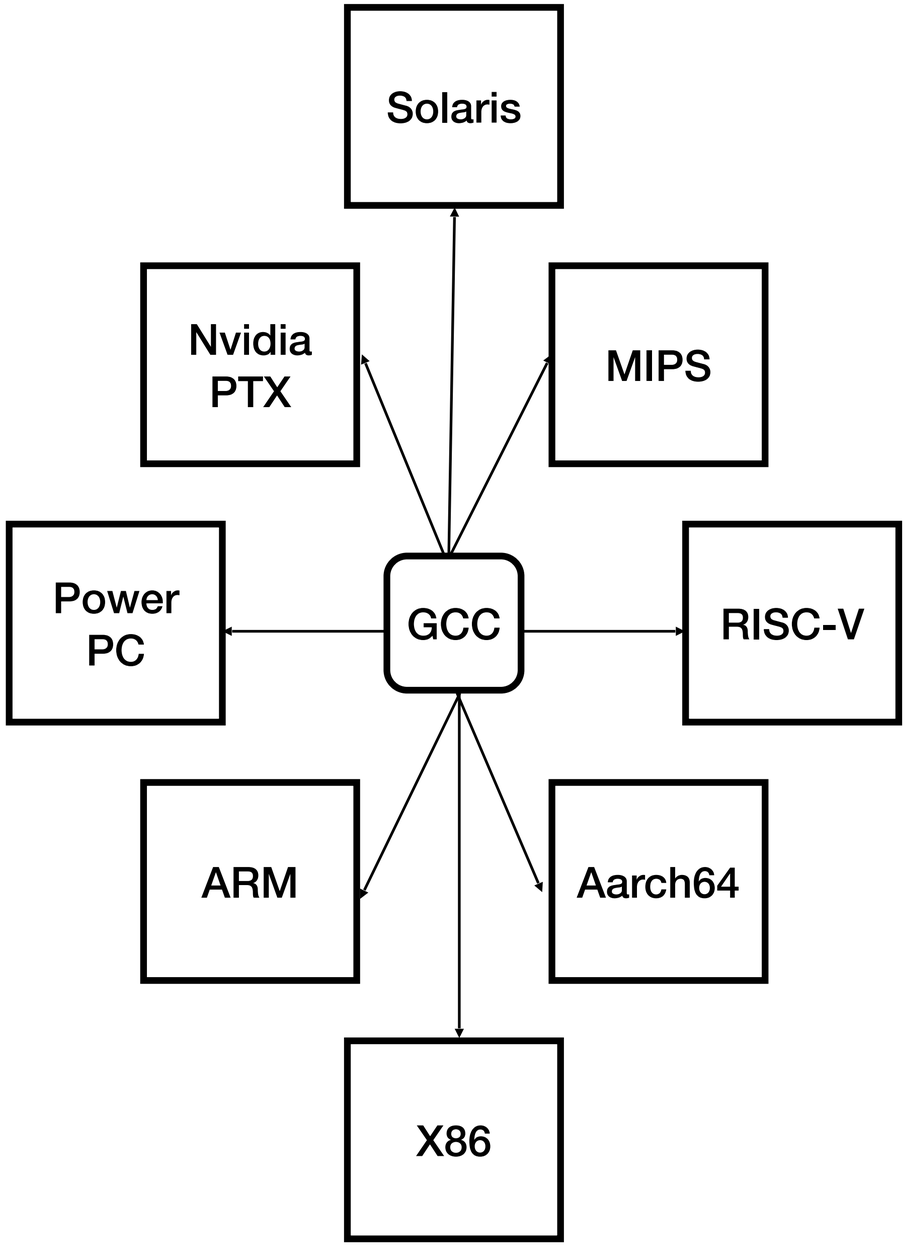
Within 15 years, LLVM's community grew fast and LLVM became more powerful and more widely used.

Since LLVM 9.0, it was under Apache License 2.0.
(What differences are between AL2.0 and GPLv3?)

GCC vs. LLVM

GCC's Advantages

1. Support more platforms
2. Perform better performance optimization
3. More widely used by older system



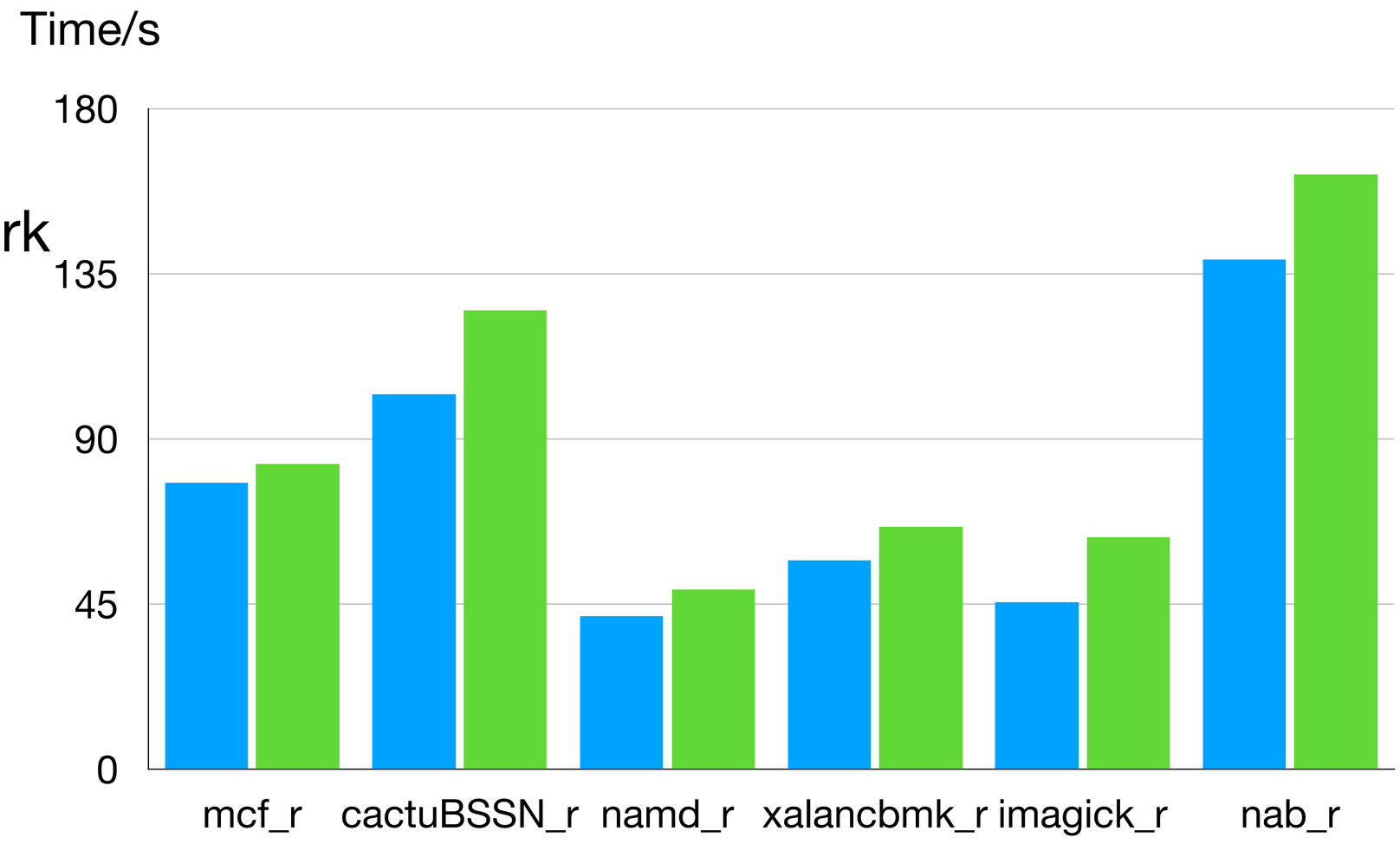
GCC vs. LLVM (cont.)

Optimized Performance Comparison of GCC and LLVM

Tested on CPU2017* benchmark

Both with -O3 option

Blue is GCC, Green is LLVM



* https://www.spec.org/cpu2017/

GCC vs. LLVM (cont.)

LLVM's Advantages

1. Code of LLVM has better reusability（可重用性） and readability（可读性）, is more convenient to do custom modify
2. Most tools can be used as API by other programs
3. More helpful and complete bug information and debug advice
4. Save original information about every token and doesn't easily modify them
5. Able to output ASTs, IRs to disk file in a readable way
6. Occupy less memory when running

Secret of Compiler Optimizations

What does -O3 mean?

Compilation Options:

-O0, -O1, -O2, -Os, …

It means a lot of complicated optimizations on code, binary file and compilation time

Including:

1. Vectorization （向量化）
2. Register Rename （寄存器重命名）
3. Instruction Generate and Scheduling （指令生成与调度）
4. Inline Function （内联函数）
5. Platform-specific Compiler （平台特定编译器）

Example: Matrix-Matrix Product

```
for (int i = 1; i <= N; i++)
{
   for (int j = 1; j <= M; j++)
   {
      C[ i ][ j ] = 0;
      for (int p = 1; p <= P; p++)
      {
         C[ i ] [ j ] += A[ i ][ p ] * B[ p ][ j ];
      }
   }
}
```

Matrix times Matrix: **by inner products**

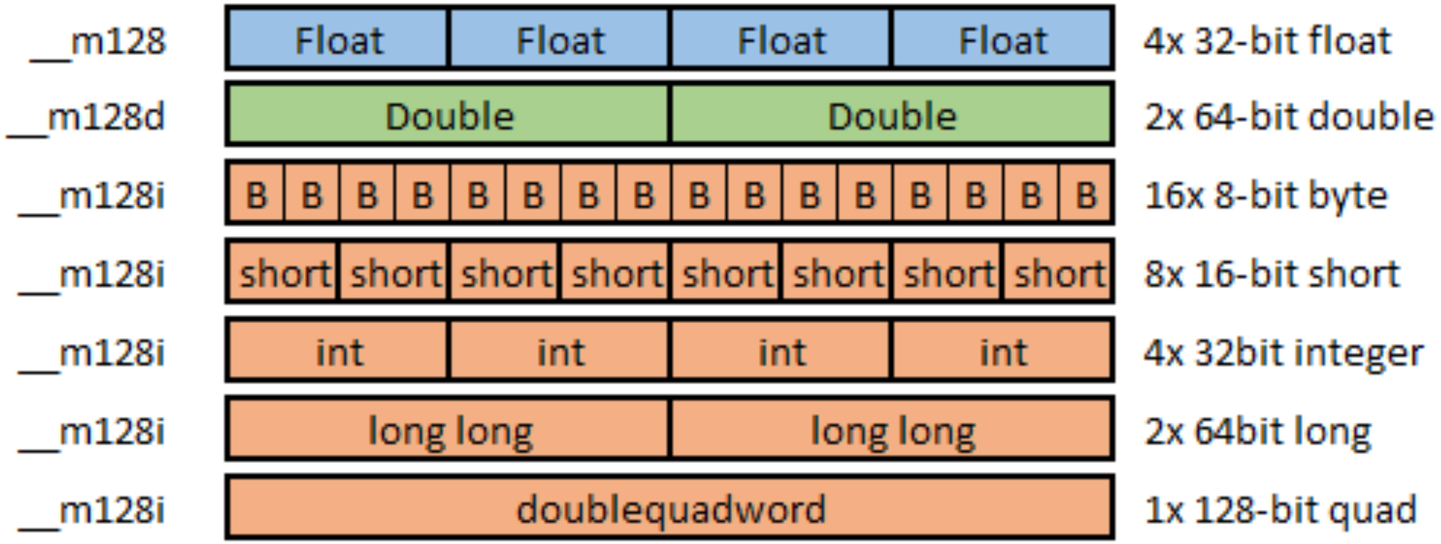
$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1P} \\ A_{21} & A_{22} & \cdots & A_{2P} \\ \vdots & \vdots & & \vdots \\ A_{i1} & A_{i2} & \cdots & A_{iP} \\ \vdots & \vdots & & \vdots \\ A_{N1} & A_{N2} & \cdots & A_{NP} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1j} & \cdots & B_{1M} \\ B_{21} & B_{22} & \cdots & B_{2j} & \cdots & B_{2M} \\ \vdots & \vdots & & \vdots & & \vdots \\ B_{P1} & B_{P2} & \cdots & B_{Pj} & \cdots & B_{PM} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1M} \\ C_{21} & C_{22} & \cdots & C_{2M} \\ \vdots & \vdots & C_{ij} & \vdots \\ C_{N1} & C_{N2} & \cdots & C_{NM} \end{pmatrix}$$

- C_{ij} is the inner product of the i^{th} row with the j^{th} column

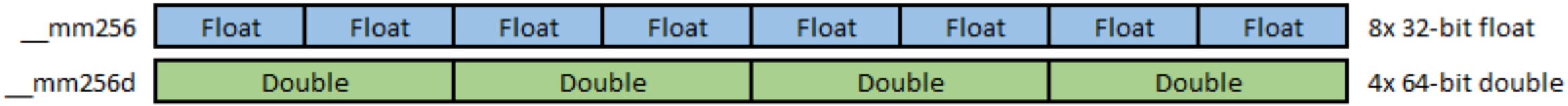
One inner-inner loop takes 2*P cycles

Vectorization Optimization

SSE Data Types (16 XMM Registers)



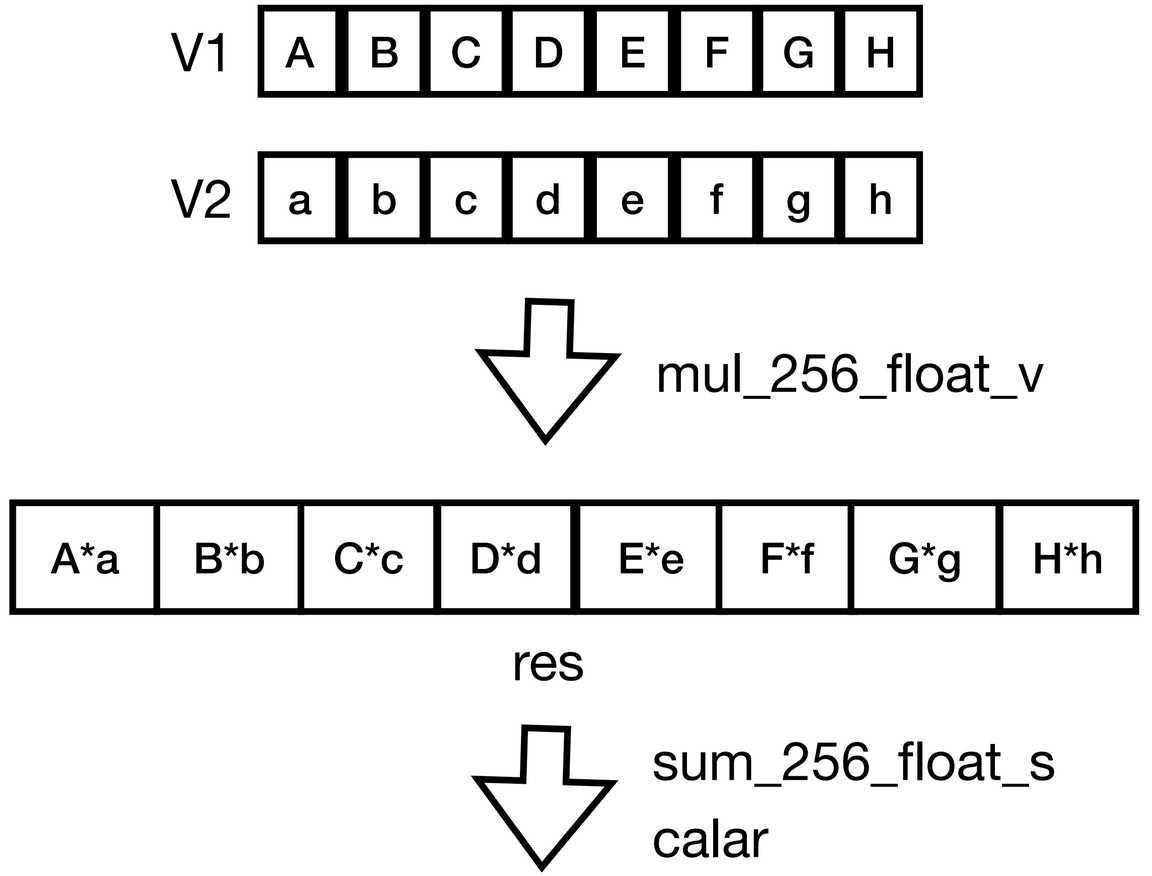
AVX Data Types (16 YMM Registers)



__mm256i *256-bit Integer registers. It behaves similarly to __m128i.Out of scope in AVX, useful on AVX2*

Vectorization Optimization (cont.)

```
for(int i = 1; i <= N; i ++)
{
   for(int j = 1; j <= M; j ++)
   {
      C[ i ][ j ] = 0;
      _mm256 res = initzero_256_float_v( );
      for(int p = 1; p <= P; p+=8)
      {
         _mm256 vec1 = load_256_float_v(A,i,p);
         _mm256 vec2 = load_256_float_col_v(B, j, p);
         res += mul_256_float_v(vec1,vec2);
      }
      C[ i ][ j ] = sum_256_float_scalar(res);
   }
}
```



A*a + B*b + C*c + D*d + E*e + F*f + G*g + H*h

One inner-inner loop takes P/4 cycles

Code Generation

Is i++ slower than ++i?

```
int main()
{
        int i = 0;
        i++;
        ++i;
}
```

Does i++ use one more register?

```
main:
.LFB1522:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, -4(%rbp)
        addl    $1, -4(%rbp)
        addl    $1, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Compiler generates the same instruction

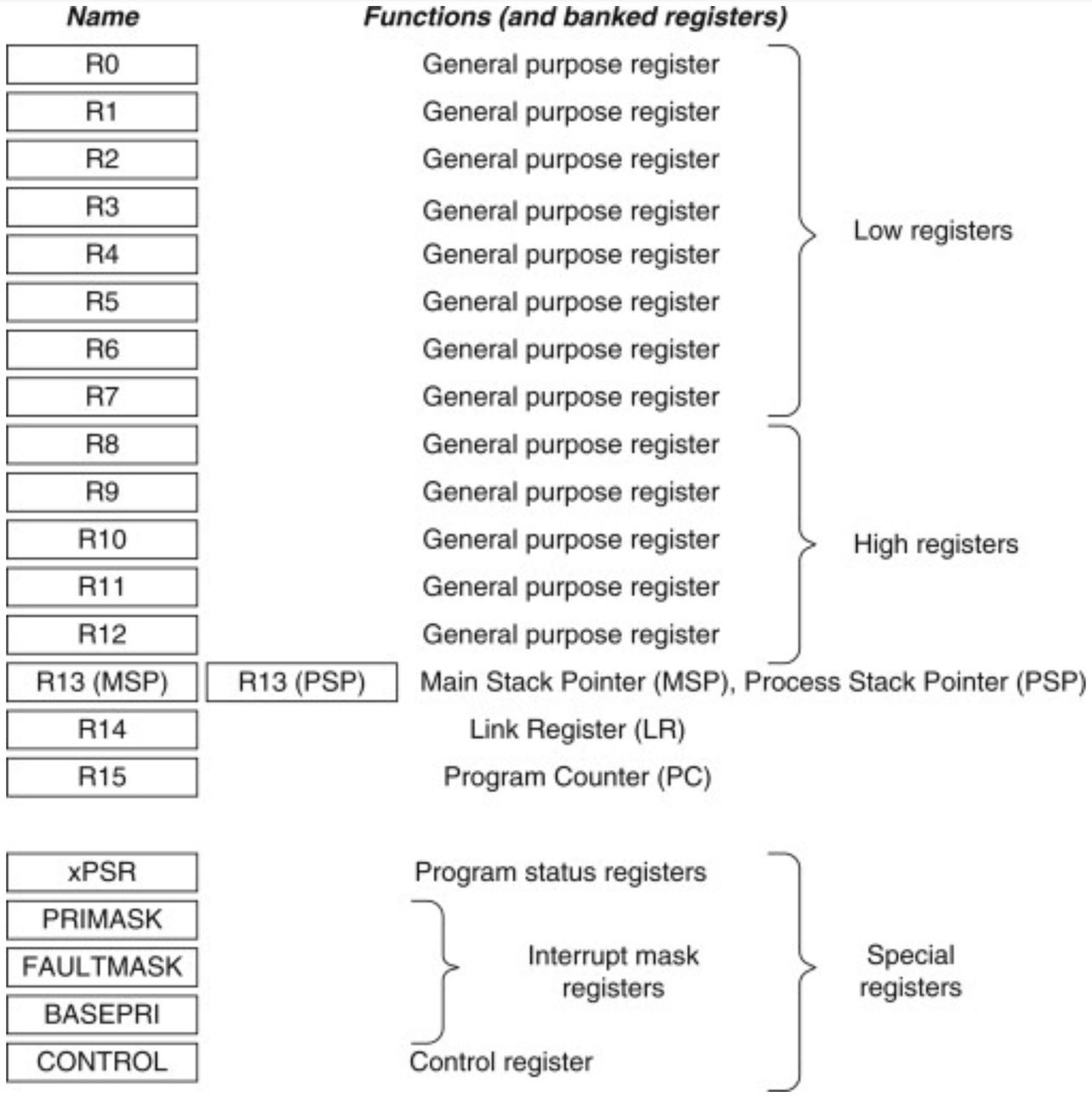
Register Rename

On x86, we have 16 registers in total

How to efficiently use all these registers?

On an out-of-order (乱序执行) CPU

```
r1 := m[1024]
r1 := r1 + 2
m[1032] := r1
r1 := m[2048]
r1 := r1 + 4
m[2056] := r1
```

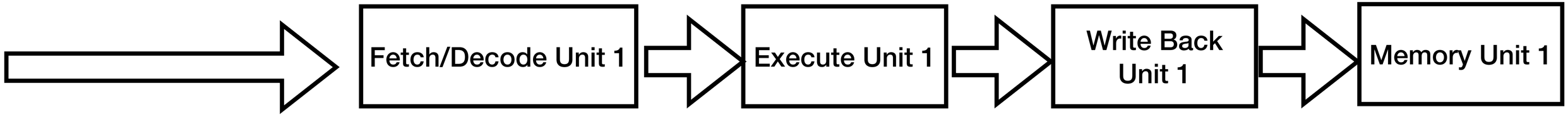


Register Rename (cont.)

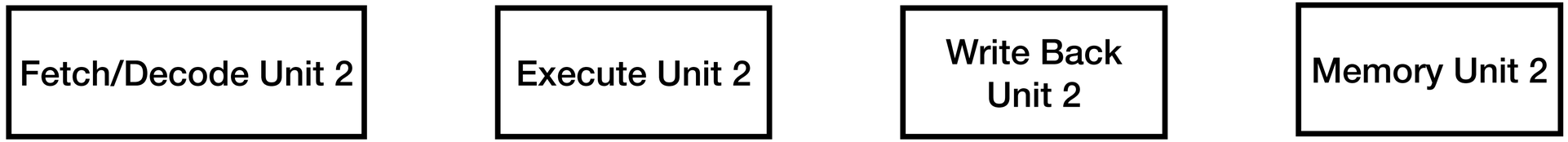
Data dependency（数据依赖性） makes un-efficient use of hardware unit

Need roughly 10 cycles to finish all these instructions

1. R1 = m[1024]
2. R1 = R1 + 2
3. m[1032] = R1



4. R1 = m[2048]
5. R1 = R1 + 4
6. m[2056] = R1



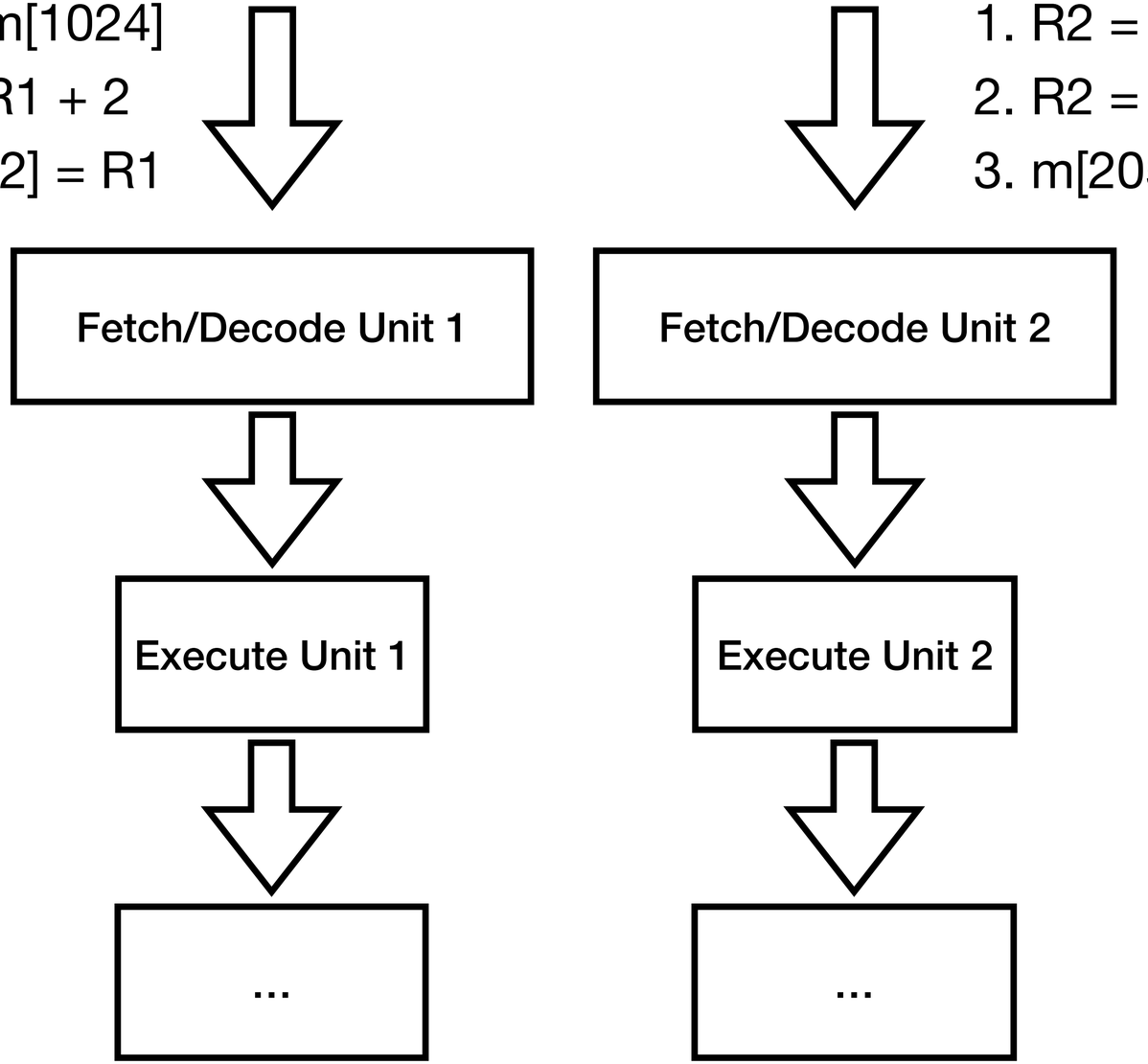
Register Rename (cont.)

Use another register to replace R1, in order to get rid of data dependency

```
r1 := m[1024]
r1 := r1 + 2
m[1032] := r1
r2 := m[2048]
r2 := r2 + 4
m[2056] := r2
```

1. R1 = m[1024]
2. R1 = R1 + 2
3. m[1032] = R1

1. R2 = m[2048]
2. R2 = R2 + 4
3. m[2056] = R2

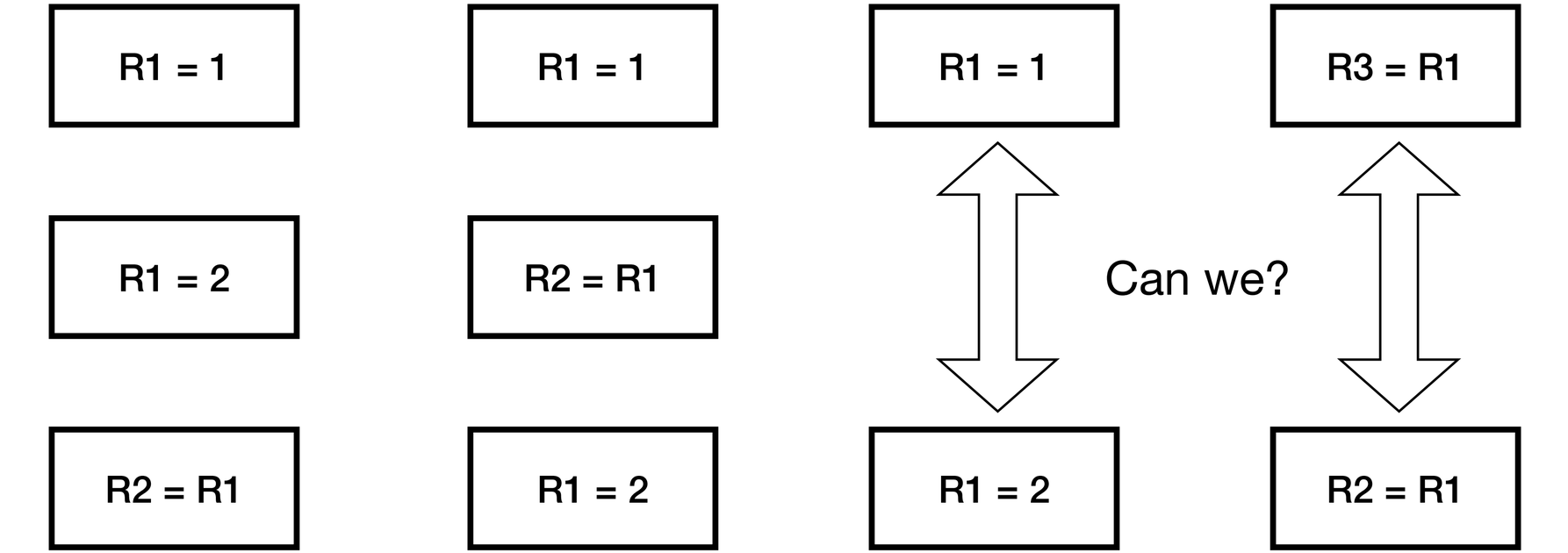


Need roughly 7 cycles to finish these instructions

Code Scheduling

Data Dependency Matters.

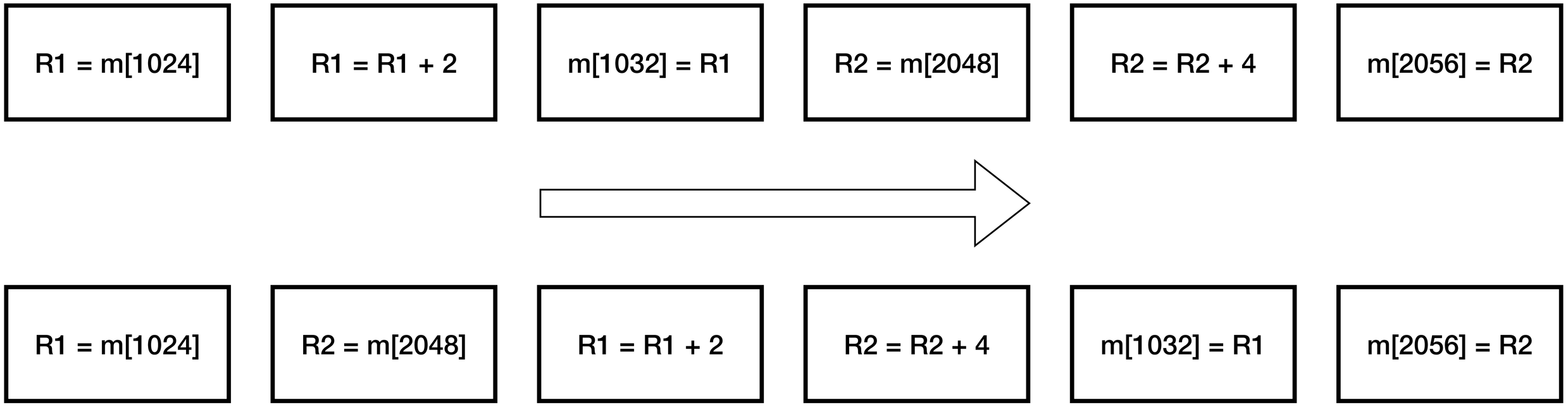
1. RAW (Read after Write)
2. WAR (Write after Read)
3. WAW (Write after Write)
4. RAR (Read after Read)



What about in multi-thread program?

Code Scheduling (cont.)

Data Dependency Matters.



Hide the second memory access latency with the first one

Inline Function

Inline expansion（内联展开）

Save the function-calling overhead

Compiler decides which function to be inlined or not

But it may increase the code-size (How does code-size affect performance?)

```
int func(int y)
{
    return pred(y) + pred(0) + pred(y+1);
}
```

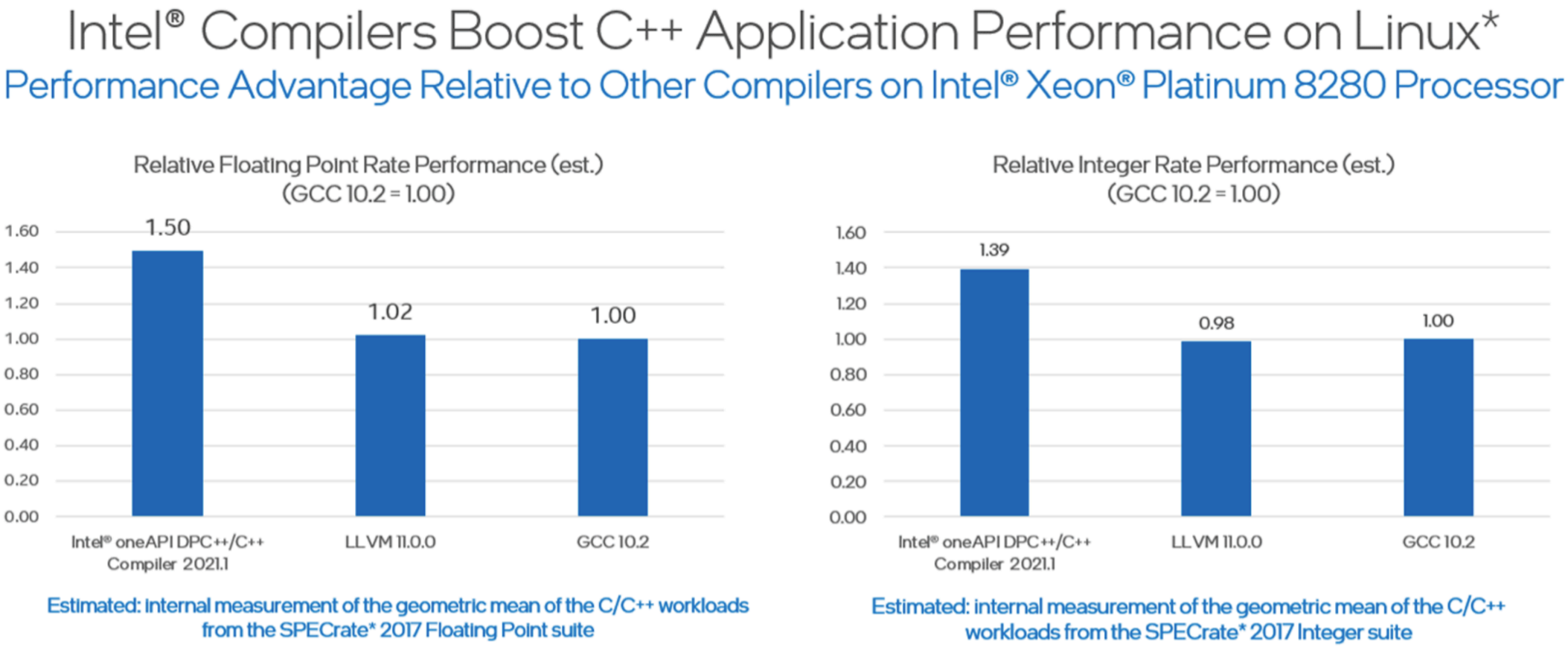
After inlining

```
int func(int y)
{
    int tmp;
    if (y   == 0) tmp  = 0; else tmp  = y - 1;        /* (1) */
    if (0   == 0) tmp += 0; else tmp += 0 - 1;        /* (2) */
    if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1; /* (3) */
    return tmp;
}
```

Platform-Specific Compiler

Platform's developers knows their platform better.

Platform-specific compiler can generate better code on the target platform, like Intel C++ Compiler.



中山大學 SUN YAT-SEN UNIVERSITY 国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Other platforms: ARM, MIPS, GPU, RISC-V

How to Customize(定制化) the Compiler?

GCC’s code is hard to reuse and modify, we only can make plugin to complete our goal.

LLVM provides multiple ways for us to modify/optimize:

1. If we need to write a new IR-pass to optimize IR using our new approach, just need to complete a separate pass under pass manager.*
2. Add new function in original optimization pass or in-time information collection pass.*
3. Check out the stack safety* while running a program
4. Build Link-Time Optimization* on LLVM
5. Implement our own JIT compiler*

[1]https://llvm.org/docs/WritingAnLLVMNewPMPass.html
[2]https://llvm.org/docs/Passes.html
[3]https://llvm.org/docs/StackSafetyAnalysis.html
[4]https://llvm.org/docs/GoldPlugin.html
[5]https://llvm.org/docs/MCJITDesignAndImplementation.html

GCC/LLVM, Which to Use?

GCC:

If we need easiest and highest performance improvement

If we need to run project on pretty old Linux systems

LLVM:

If we need to make use of middle-product（中间产物） of compiler

If we need to write specific IR optimization

If we need to optimize binary file / bitcode to improve performance

If we need to implement a compiler for one new language

Summary

Biggest Difference between GCC and LLVM:

1. Performance
2. Code Reusability and Readability
3. Platform Supported
4. Custom Modify

General Compiler Optimization:

1. Vectorization
2. Register Renaming
3. Code Generating and Scheduling
4. Inline Function
5. Platform-specific Optimization

Question Time

Question ?

Grazie.

Thank you!