In [426]:

```
# This cell runs automatically.  Don't edit it.
from CSE142L.notebook import *
from notebook import *
from cfiddle import *
import os
setup_lab()
```

Double Click to edit and enter your
1. Name
2. Student ID
3. @ucsd.edu email address

Lab 2: The Compiler

Rules of Optimization:

- Rule 1: Don't do it.
- Rule 2 (for experts only): Don't do it yet.

-- M.A. Jackson

The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

-- Donald Knuth in The Art of Computer Programming, 1968.

This lab will give you a much clearer understanding of the role that the compiler plays in translating your source code into executable binaries. Our focus is on the optimizations that compilers perform and the critical role that they play in efficiently implementing modern compiled programming languages (e.g., C, C++, Rust, Go, and Java). We will use C++, but many of the same lessons apply to other languages.

This lab includes a programming assignment. You should start thinking about it early, so read that section now (or at least soon).

Check gradescope schedule for due date(s).

1 How To Read the Lab For the Reading Quiz

This is a long lab. For the reading quiz you should focus these sections.

- Section 9-14: "New Tools" through "Practical Rules for Using Compiler..."
  - You can skip Section 12 ("C++ Revisited")
- Section 16: "The Programming Assignment" (just the introduction and section 17.1)

(It's possible the section numbers might not match the names. When in doubt, use the section name)

Guidance

- Some of the sections are collapsed it the PDF so that there is a heading and no content. This is intentional. You don't need to read the hidden text.
- For the quiz, you can just read the introductory text at the top of each section or subsection.
- Many of the code cells and their output print very poorly. If you can't read it easily, it won't be on the quiz.

2 FAQ and Updates [...]

3 Using in the Correct Environment on DataHub [...]

4 Browser Compatibility [...]

5 About Labs In This Class [...]

6 Logging In To the Course Tools [...]

7 Grading

Your grade for this lab will be based on the following components.

Part	value
Reading quiz	3%
Jupyter Notebook	55%
Programming Assignment	40%

Part	value
Post-lab survey.	2%

- No late work or extensions will be allowed.
- We will grade 5 of the "completeness" problems. They are worth 3 points each. We will grade all of the "correctness" questions.
- You'll follow the directions at the end of the lab to submit the lab write up and the programming assignment through gradescope.
- Please check gradescope for exact due dates.

8 A Note About The Examples

There are a bunch of short functions in the examples in the lab. Their purpose is to illustrate the impact of different compiler optimizations in a clear way, and that is what I designed them for (often by trial and error). As a result, none of them do anything *useful*. Please don't expend effort trying to understand what the functions are doing or what they are for, there's nothing to find :-).

If you look closely, you'll also see that I compile some of the functions using complex sets of compiler flags. I do this to highlight specific optimizations, but it's not usually helpful or necessary in the real world. GCC and other compilers provide a [huge number of flags (https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html)](https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html) that control how the compiler optimizes. Using these flags is *almost never necessary* in the real world. When you need to ship your code, just compile it with `-O3`.

9 New Tools

There are few new tools we will be using in this lab.

9.1 Control Flow Graphs

Control flow graphs (CFGs) are a common way to visually inspect the structure of *one function* in a program.

A CFG is a collection of nodes and edges. Each node is a *basic block* -- a sequence of instructions that always execute together. This means that a basic block ends with either 1) a branch, 2) a jump, or 3) the target of a branch or jump. The edges in the CFG are possible *control transfers* between basic blocks. So, the CFG shows all possible paths of control flow through the function.

CFiddle can generate CFGs for most functions. Take a look at this code then run the cell to see it's CFG:

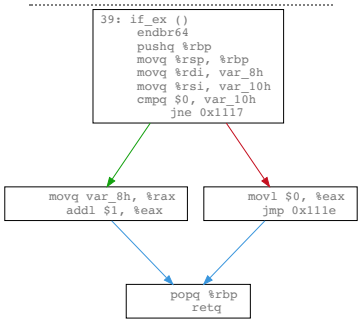
In [427]:

```
if_ex = build(code(r"""
#include<cstdint>
#include<cstdlib>

extern "C"
int if_ex(uint64_t array, unsigned long int size) {
    if (size == 0) {
        return 0;
    }
    return array + 1;
}
""", file_name="if_ex.cpp"), arg_map(DEBUG_FLAGS="-g0 -O0"))

compare([if_ex[0].source("if_ex"),
         if_ex[0].cfg("if_ex", remove_assembly=False),
        if_ex[0].asm("if_ex")])

```

```
  0%|          | 0/1 [00:00<?, ?it/s]
```

```
ERROR: Cannot determine entrypoint, using 0x00001040
```

```
int if_ex(uint64_t ar
ray, unsigned long in
t size) {
    if (size == 0) {
        return 0;
    }
    return array + 1;
}
```




```
if_ex:
.LFB15:
        .cfi_startpro
c
        endbr64
        pushq   %rbp
        .cfi_def_cfa_
offset 16
        .cfi_offset
6, -16
        movq    %rsp,
%rbp
        .cfi_def_cfa_
register 6
        movq    %rdi,
-8(%rbp)
        movq    %rsi,
-16(%rbp)
        cmpq    $0, -
16(%rbp)
        jne     .L2
        movl    $0, %
eax
        jmp     .L3
.L2:
        movq    -8(%r
bp), %rax
        addl    $1, %
eax
.L3:
        popq    %rbp
        .cfi_def_cfa
7, 8
        ret
        .cfi_endproc
```

This error is not an error: You'll get this error every time you render a CFG:

```
Cannot determine entrypoint, using 0x00001040.
Warning: run r2 with -e bin.cache=true to fix relocations in disassembly
```

You can ignore it.

There are also sometimes stray boxes (like above). You can ignore those too.

The CFG shows that there are two paths through the function depending on the value of the `if` condition. So there are two paths along which *control* can *flow*. The green and red lines correspond to the taken and not-taken outcomes of the branch at the end of the top block. We'll discuss them in more detail below.

Here's a more complex piece of code (below the question). Study it, answer the question, and then run the code cell below.

Note: Remember that you can paste images into a text cell while it's in edit mode.

Question 1 (Completeness)

What do you think the CFG for the code below will look like (describe it briefly or use ASCII art or paste in screen capture)?

```
ASCII art or text drawing/description of the CFG her
e.
```

How many paths through the code are there?

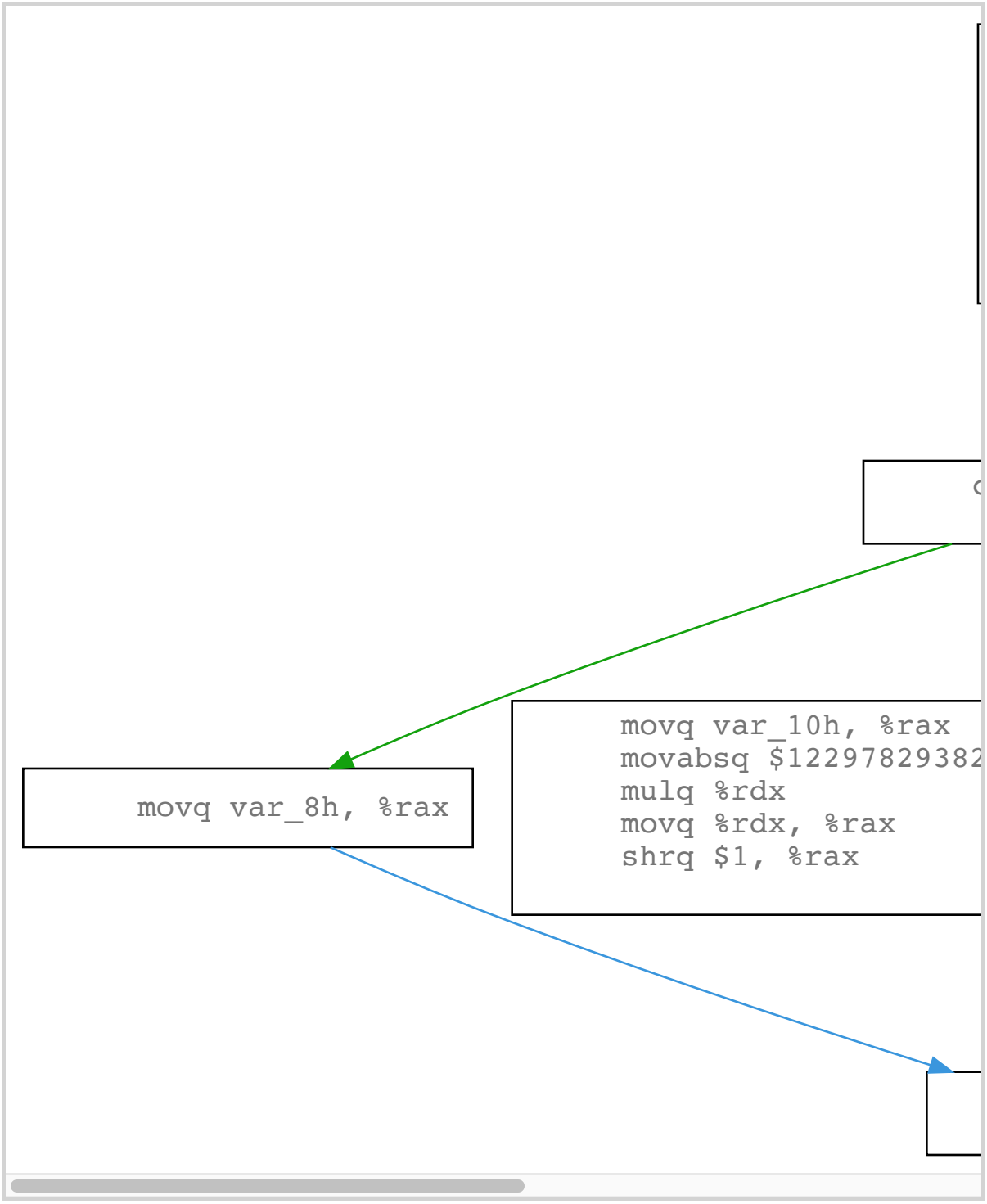
Or paste an image out here (outside the triple backticks).

In [428]:

```
t = build(code(r"""
#include<cstdint>
#include<cstdlib>

extern "C"
uint64_t if_else_if_else(uint64_t array, unsigned long int size)
    if (size/2) {
        return NULL;
    } else if (size/3) {
        return size/3;
    } else {
        return array;
    }
}
"""))
display(t[0].cfg("if_else_if_else"))
```

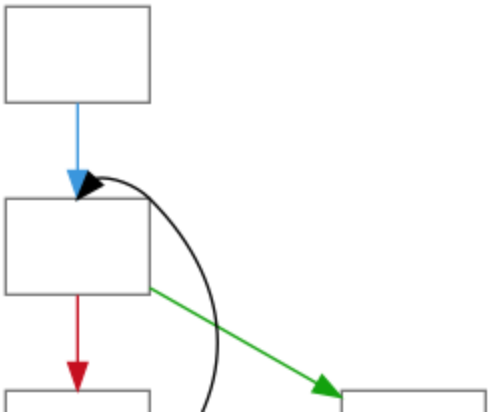
```
100%                                                1/1 [00:00<00:00, 41.34it/s]
```

```
ERROR: Cannot determine entrypoint, using 0x00001040
```



Question 2 (Correctness - 2pts)

In the code below, modify `foo()` so that its CFG looks like this (you can edit the code as many times as you'd like; Extra empty boxes -- or boxes with "None" in them -- are fine.):



In [302]:

```
1
2  build(code(r"""
3  extern "C"
4  int foo(int a) {
5      return 0;
6  }
7  """))[0].cfg("foo", remove_assembly=False)
```

Finally, take a look at this code and the resulting CFG, and then answer the question below. As you think about the question, look at the code and figure out what decision the program will need to make. Each of these decisions maps to node with two out-going edges. If you do that starting at the top of the loop, the role of each basic block should become clear.

In [303]:

```
1  loop_if = build(code(r"""
2  #include<cstdint>
3  #include<cstdlib>
4
5  extern "C"
6  uint64_t loop_if(uint64_t array, unsigned long int size) {
7      uint64_t t= 0;
8      int k = 0;
9      for(uint i = 0; i < size; i++) {
10         if (i-size != 0) {
11             k = 4;   //  L1
12         } else if (i+size != 0) { // L2
13             k = 5;
14         }
15     }
16     return t + k; // L3
17
18 }
19
20 """))
21
22 compare([loop_if[0].source("loop_if"), loop_if[0].cfg("loop_if",
```

Question 3 (Correctness - 3pts)

Fill out the table below to show which nodes in the CFG correspond to the labeled line in the source code:

Line	Node
L1	
L2	
L3	

9.2 Call Graphs

CFGs show the control flow *within a single function* , but they cannot tell us much about the flow of control through an entire program. To understand how functions call one another, we will use *call graphs*.

In a call graph, the nodes are functions, and an edge exists from one function to another, if the first function calls the second.

We will build call graphs by running the program and keeping track of which function calls occur. We'll use the `gprof` profiler to collect the data, which means that building a call graph is a three stop process:

1. Compile the program with `gprof` enabled.
2. Run the program on a representative input.
3. Generate the call graph for that execution of the program.

Here's an example:

Question 4 (Completeness)

What do you think the call graph for the code below will look like (describe it briefly or use ASCII art or paste in screen capture)?

```
ASCII art or text drawing/description of the CFG her
e.
```

Or paste an image out here (outside the triple backticks).

In [165]:

```
call_graph(code(r"""
extern "C" {
int one(int a);
int two(int a);
int three(int a);

int main(int argc, char * argv[]) {
    for(int i = 0; i < 100; i++) one(i);
    return one(4);
}

int one(int a) {
    if (a & 1)
        return two(a);
    else
        return three(a);
}

int two(int a) {
    return three(a);
}

int three(int a) {
    return a;
}
}
"""), root="one")
```

The call graph has a bunch of information in it regarding how many times each function was called and how often a function was called from one location rather than another, but for now, we'll just focus on which function called which.

Question 5 (Completeness)

Modify the code above to add a loop to the call graph (but be sure the function still terminates). Describe what you did below.

Call graphs can reveal the internal workings of libraries. For instance, take a look at the function below and answer this question:

Question 6 (Completeness)

How deep do you think the call graph is (i.e., how long is the longest chain of one function calling another, calling another, etc.)? How many different functions do you think are invoked? How many function calls are made in the course of running the program? (It's OK if you have no concrete way to answer this. But think about the code you've written and what reasonable values might be.)

1. How deep is the graph?:
2. How many different functions are called?:
3. How many function calls are made?:

In [166]:

```
call_graph(code(r"""
#include<algorithm>

extern "C" int one(int a);

int main(int argc, char * argv[]) {
    return one(4);
}
#define ARRAY_SIZE (16*1024)
extern "C"  int one(int a) {
    int * array = new int[ARRAY_SIZE];
    std::sort(array, &array[ARRAY_SIZE]);
    return array[a];
}

"""), root="one", opt="-O0")
```

Wow, what a mess!

You can double click on the image to zoom in and pan around. You'll see lots of functions called many, many times, and each of those function calls requires a few extra instructions (e.g., to call the function and return from it.)

This seems shockingly inefficient. Someone should really clean that up!

This kind of complex, deep call graph is very common in modern object-oriented programming languages.

Question 7 (Optional)

Modify the code to use other parts of the standard template library (STL). For instance, you could create and initialize a `std::vector` or an `std::list`.

What did you find?

9.3 Looking At Assembly

CFGs and call graphs are good tools for looking at the high-level structure and behavior of programs, but we also need to understand what's going on at a finer level: Inside the basic blocks. To do that, we'll have to look at the assembly language that the compiler generates.

The lectures in 142 provided an overview of the x86 assembly and our main goal in both courses is that you will be able to look at some x86 assembly and (with the help of google) get *some idea* of what is going on.

Use the Slides, Luke! (and google!): This lab doesn't contain everything you need to know about x86 assembly. Please look back over the slides from 142 and relevant parts of the textbook for more details, especially about instruction suffixes and the register file. Google is a good resource as well: Searching for "x86 *inst name* " will get you information about any instruction.

9.3.1 The Basics

We'll start simple by revisiting the `if_ex()` function we saw earlier (run the cell):

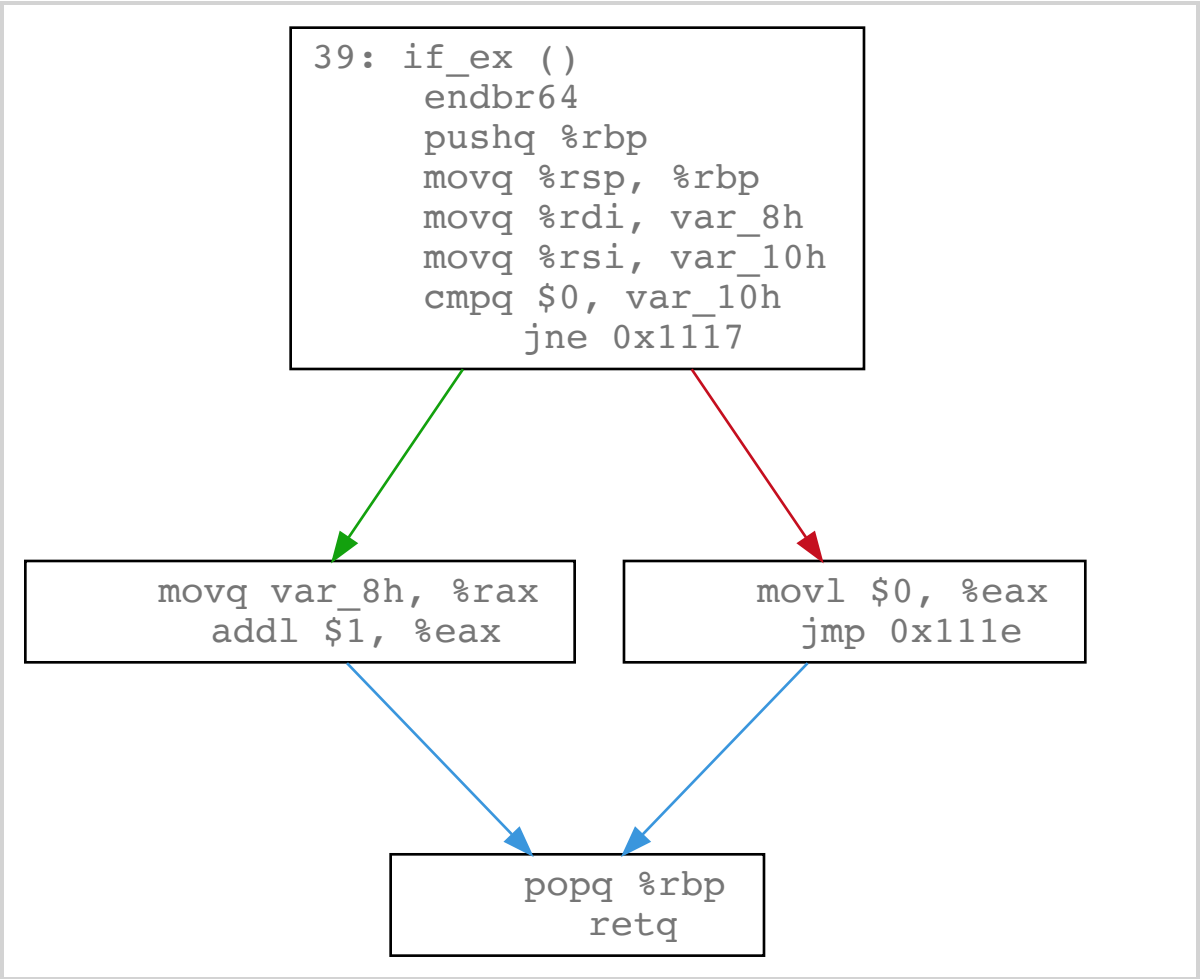
In [429]:

```
1 compare([if_ex[0].source("if_ex"), if_ex[0].asm("if_ex")])
2 display(if_ex[0].cfg("if_ex"))
```

```
int if_ex(uint64_t array, unsign
ed long int size) {
    if (size == 0) {
        return 0;
    }
    return array + 1;
}
```

```
if_ex:
.LFB15:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movq    %rdi, -8(%rbp)
        movq    %rsi, -16(%rbp)
        cmpq    $0, -16(%rbp)
        jne     .L2
        movl    $0, %eax
        jmp     .L3
.L2:
        movq    -8(%rbp), %rax
        addl    $1, %eax
.L3:
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

```
ERROR: Cannot determine entrypoint, using 0x00001040
```



Take some time to study the source code (top left) and assembly output (top right) and how it corresponds to the CFG annotated with the assembly (bottom).

A few things to note:

1. Comments in assembly start with `;`
2. Things like `.L3:` that end in `:` are labels in the assembly and can be used as the "targets" of branches. Some of them (e.g., `if_ex:`) mark the beginnings of functions.
3. Other things like `.cfi_endproc` are assembly directives. They provide metadata about the instruction, functions, and symbols. They don't affect execution. We will mostly ignore them.
4. In x86 the first 6 function arguments are passed in %rdi, %rsi, %rdx, %rcx, %r8, and %r9.

Finally, note how the assembly in the CFG is different than the assembly from the compiler.

The assembly on the right is the actual x86 assembly that the compiler generated. The code in the CFG is equivalent, but it's not x86 assembly. We'll call it *pseudo-assembly*. It was generated by a *disassembler* from the compiled binary. There's a one-to-one correspondence between the assembly and the pseudo-assembly but there are some important differences:

1. The targets for branch and jump instructions are raw addresses in the pseudo-assembly rather than labels.
2. The pseudo-assembly provides some information about argument types in comments (e.g., `; var int64_t var_10h @ rbp-0x10`) and then uses `var_10h` in the

pseudoassembly.

What's going on? Why does the CFG contain pseudo-assembly instead of the actual assembly?

Internally, CFiddle use Redare2 (https://rada.re/n/) to generate the CFGs. It's a really powerful tool for reverse engineering software, but it operates only binaries rather than assembly code.

Question 8 (Completeness)

How does the variable `size` appear in the assembly and the unoptimized pseudo-assembly? That is, how can we tell that an instruction is operating on the value stored in `size` ?

1. What does `size` look like in x86 assembly:
2. What does `size` look like in pseudo-assembly:
3. That is, how can we tell that an instruction is operating on the value store in `size` ?

You should also be able to spot the boundaries between the basic blocks in the assembly and see how the CFG makes them explicit. You can see that the instruction at the start of each basic block is either

1. A branch target (i.e., a label that a branch might jump to), or
2. The instruction right after a branch.

The last instruction in each basic block is either:

1. The instruction before a branch target (i.e., label), or
2. A branch, jump, or return instruction.

If the last instruction in the block is a branch, then there will be two lines leaving the block -- one green, one red. The green line is the path that control will take if the branch is *taken*. The red line is the branch's *not taken* path. Notice how the red (not-taken) path leads to the block that contains `movl $0, %eax` which sets the return value for the function to 0. This corresponds to the if condition `size == 0` being `true`. The compiler is free to use any branch it wants to implement an if condition. In this case, it used `jne` (jump on not equal), but it could have used `je` (jump on equal) and reversed the position of the two basic blocks in the assembly.

For instance, here's a comparison between the assembly above and the same code compiled with more optimizations turned on. The more optimized code (on the right) uses `je` instead of `jne` so the `movl $0, %eax` is on the green path instead of the red path.

In [168]:
```
1 if_ex = build("./if_ex.cpp", build_parameters=arg_map(OPTIMIZE=["
2 compare([x.cfg("if_ex") for x in  if_ex], [f"OPTIMIZE = {x.get_bu
3
```

Question 9 (Correctness - 2pts)

In the assembly below, add lines that say `" ; basic block boundary "` between each of the basic blocks in the assembly.

```
bb:
.LFB0:
    .cfi_startproc
    endbr64
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    $0, -8(%rbp)
    movl    $0, -4(%rbp)
    jmp    .L2
.L5:
    cmpl    $0, -24(%rbp)
    jle    .L3
    movl    -4(%rbp), %eax
    addl    %eax, -8(%rbp)
    jmp    .L4
.L3:
    movl    -4(%rbp), %eax
    subl    %eax, -8(%rbp)
.L4:
```

9.3.2 A More Complex Example [...]

Here's the assembly for `loop_if()`. Run the cell. Be sure to scroll down.

(The `-g0` option tells gcc to supress debugging information in the assembly. It makes it easier to read, but doesn't change any of the instructions.)

9.3.3 Counting Instructions With the CFG [...]

In the next section you will study how compiler optimizations change which instructions execute. Let's see what we can learn about which instructions execute how often by looking at the assembly.

10 Understanding C++ Assembly Output

C++ is a big, complex mess of a language that includes a bunch of powerful tools that make it possible to write fast code without too much pain. However, all that power translates into a lot of complexity that shows up in the assembly code generated for C++ programs.

In order to read C++ assembly output, you need to understand a few details about one aspect of this implementation process: linking.

10.1 What Is Linking?

Linking is the final step in compiling a program. Non-trivial programs are spread across multiple source files that are compiled one-at-a-time into *object files* (`.o`) that contain binary instructions and static data (e.g., string constants from your code). Each function and global variable in the object file has a name called a *symbol*. We say that the object file *defines* the symbols it contains. For instance, if `foo.cpp` contains the source code for a function `bar()` then, `foo.o` will define the symbol `bar` .

The code in the object files will also *reference* symbols defined in other object files. For instance, if another file, `baz.cpp` , calls `bar` , then `baz.o` will have reference to `bar` . Prior to linking, that reference is *undefined*.

The linker takes all the `.o` files and copies their contents into a single executable file. As it copies them it *resolves* the undefined references. In this example, the linker resolves the undefined reference in `bar.o` by replacing the reference with pointer to the code for `bar()` in `foo.o` .

One important thing about linkers is that they are language-agnostic -- the linker will happily link object file generated from C++, C, Go, or Rust as long as the symbols match.

There's a lot more to linking (https://www.amazon.com/Linkers-Kaufmann-Software-Engineering-Programming/dp/1558604960), but this is enough to see what's problematic about C++.

10.2 C++ Name Mangling

The linker restricts what strings can serve as valid symbols: Symbols must start with a letter (or `_`) and only contain letters, numbers, and `_` .

For C, this poses no problems. If you declare a function `bar` in file `foo.c` :

```
int bar(int a) {
    return 1;
}
```

The compiler will generate exactly one symbol with the name `bar` . Then you can call it from another file `baz.c` :

```
main() {
     bar(4);
}
```

and the linker will know what function you mean (i.e., the function named `bar` from `foo.c`).

However, C++ allows function overloading, so we might have this in `foo.cpp` :

```
int bar(int a) {
}

float bar(float a) {
}
```

This will generate two functions, so they need two symbols. But what symbols should the compiler choose? The compiler needs a systematic way of naming functions *that includes their type information*. This will ensure that when we have `baz.cpp` with

```
main() {
    bar(4);
    bar(4.0);
}
```

The linker will know that we mean to call two different functions.

Things get more complex with templates, since we could have:

```
int bar(const std::map<std::string, std::vector<int>> & a) {
}
```

That's a lot of information to pack into one symbol!

In [126]:

```
build(code(r"""
#include<map>
#include<vector>
#include<string>
int foo(int a) {
    return 0;
}

float foo(float a) {
    return 0;
}

int foo(const std::map<std::string, std::vector<int>> & a) {
    return 0;
}
"""))[0].asm(demangle=False, show=(0,100))
```

Question 10 (Completeness)

What's the mangled name for each of these functions?

function	mangled name
`int foo(int a)`	
`float foo(float a)`	

function	`int foo(const std::map<std::string, std::vector<int>> & a)`
mangled name	

As you can see, mangled names make assembly pretty hard to read. To make matters worse, mangled names show up in other places as well (e.g., the output of profiling tools).

Fortunately, C++ compilers usually come with a utility to de-mangle names. For `g++` it's called `c++filt` and it takes in text, looks for mangled names and demangles them. For instance:

In [127]:
```
!echo _Z3foof | c++filt
!echo _Z3fooRKSt3mapINSt7__cxx1112basic_stringIcSt11char_traitsIc
```

You'll notice that the full name for `int foo(const std::map<std::string, std::vector<int>> & a)` is very long. This is because it includes full type names (including the C++ namespace) and all the default template parameters.

To see how it works on assembly, change `demangle=False` in the code above to `demangle=True` and re-run it. The resulting assembly is no longer valid code assembly (since the symbol names are invalid), but it's much easier to read.

From now on in the examples, the assembly code in the examples will be demangled, but you will probably run into some mangled names occasionally. Just remember to use `c++filt` to clean them up.

10.3 C vs C++ Linkage

The way that the compiler generate symbols for a function is called the function's *linkage*. We've seen two kinds: C linkage which just uses the function name and C++ linkage which uses mangled names.

You might have noticed that most of the code examples have `extern "C"` before some functions. This is a way of telling the compiler that it should use C linkage for those functions (i.e., just use the function names). You can use it for one function:

```
extern "C" int foo()
```

or a group of functions:

```
extern "C" {
    int foo(){}
    int bar(){}
}
```

This is useful if you want to call the function from a language other than C++ (e.g., C). We will

11 Optimization

Now, we have all the tools we need to study how compiler optimizations affect program performance. In the exercises below, we'll look at some of the most important optimizations that compilers perform and why and how they work.

We have several goals:

1. To provide some intuition about what the compiler can and cannot do, so you can predict when it will need your help and when you should trust it to "do the right thing".
2. To see how and why optimization is so important for languages like C++.
3. To gain further insight into how the way a computation is implemented affects its performance (via the performance equation).

11.1 Register assignment

The first and simplest optimization is *register assignment*. Register assignment takes local variables and intermediate values and stores them in registers rather than on the stack. This saves `mov` instructions and memory accesses. You can see it in action below:

In [430]:

```
1
2  foo = build(code(r"""
3  extern "C"
4  int foo(int a, int b){
5      return a * b;
6  }
7  """), arg_map(OPTIMIZE=["-O0 -g0", "-O1 -g0"]))
8  display(foo[0].source())
9  compare([x.asm("foo") for x in foo], [html_parameters(v.get_build
```

100% 2/2 [00:00<00:00, 55.41it/s]

```
extern "C"
int foo(int a, int b){
    return a * b;
}

// Cfiddle-signature=10e87d07873cb1e07d1e687af5b74abd
```

OPTIMIZE = -O0 -g0

```
foo:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    -4(%rbp), %eax
        imull   -8(%rbp), %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

OPTIMIZE = -O1 -g0

```
foo:
.LFB0:
        .cfi_startproc
        endbr64
        movl    %edi, %eax
        imull   %esi, %eax
        ret
        .cfi_endproc
```

The code on the left is unoptimized. The code on the right is optimized.

Wow! The optimized code is much shorter!

A few things to notice about the code and to remember about x86 assembly:

First, the "base pointer" is in `%rbp`. This is the base of the stack frame for this function call. Local variables typically live on the stack and are accessed relative to the base pointer.

Second, in x86, the first two function arguments are passed in `%edi` and `%esi`. Return values are stored in `%eax`.

Third, in unoptimized code, `a` and `b` are on the stack at locations `-4(%rbp)` and `-8(%rbp)`, respectively. In fact, the compiler goes through the trouble of storing `%edi` and `%esi` into `-4(%rbp)` and `-8(%rbp)`.

Then it, *immediately* uses `movl` to load `a` back into the `%eax` before the `imull` instruction loads `b` from the stack, multiplies it by `%eax`, storing the result in `%eax`.

The optimized code avoids all that nonsense with the stack. It just copies `%edi` into `%eax`, and multiplies it by `%esi`, once again leaving the result in `%eax`.

Pro Tip: Identifying unoptimized assembly Register assignment is one of the most basic optimizations that compilers perform and the difference between the optimized and unoptimized versions is starkly obvious. This means that you can usually just tell by looking at assembly code whether it was compiled with optimizations enabled: if it has

Question 11 (Correctness - 2pts)

Assuming constant `CPI` and `CT` (i.e., they are the same for the optimized and unoptimized code) how much speedup did register assignment provide for `foo()` (show your work)?

11.1.1 The `register` Keyword

You can provide some guidance to the compiler about what to put in registers with the `register` keyword:

In [129]:
```
register = build(code(r"""
extern "C"
int foo(register int a, register int b){
    int c = a *b;
    return c * b;
}
"""), arg_map(OPTIMIZE=["-O0 -g0", "-Og -g0", "-O1 -g0"]))
display(register[0].source())
compare([x.asm("foo") for x in register], [html_parameters(v.get_
```

Note how, with no optimizations, the compiler uses loads and stores to access `c` but not `a` or `b` (which were both declared `register`). But with optimizations, all three are in registers.

You don't see `register` very much in code because compilers put things in register automatically. So you shouldn't ever use it...except maybe in programming assignments...

11.1.2 Register to Replace Memory

The compiler will also combine multiple memory updates into one by storing the intermediate values in a register. For instance:

In [130]:
```
enregister = build(code(r"""
#include"cfiddle.hpp"
extern "C"
void foo(register uint64_t * sum, uint k) {
    for(uint i =0 ; i < k; i++) {
        *sum += i;
    }
}
"""), arg_map(OPTIMIZE=["-O0 -g0", "-O1 -g0"]))
compare([x.asm("foo") for x in enregister], [html_parameters(v.ge
```

In the unoptimized code (on the left), `s` lives at the address in `%rax`. In the basic block starting at `.L3`, it loads and stores `s`, adds to it, and then stores it back. It does this *on every iteration*. With optimizations (on the right), `s` lives at the address in `%rdi`. The code loads it once (into `%rdx`) with `movq (%rdi), %rdx` at the beginning, updates the register in the loop body, and stores `%rdx` back into `%rdi` at the end.

11.2 Common sub-expression elimination

A *common sub-expression* is a piece of repeated computation in a program. Since calculating the same thing twice is a waste of time, the compiler will eliminate the second instance and reuse the result of the first. Here's an example:

In [131]:
```

optimization_example(code(r"""
extern "C" int foo(register int a, register int b) {
    register int c = a * b;
    return a * b + c;
}
"""), "foo")
```

Again, the unoptimized code does some inefficient things. I encourage you to trace through the assignments/ `movl`s (`a` is in `%edi` and `b` is in `%esi`), but the key thing is that it performs two `imull` instructions that compute the same result.

The optimized code, just computes the product once and stores it in `%edi`. Then it uses `leal` to add `%rdi` to itself and store the result in `%eax`.

Pro Tip: `lea` in action Recall that `leal` computes the effective address of its first argument and stores that address in its second argument. In this case, it uses the `(r1,r2)` addressing mode which adds `r1` and `r2` together to compute the

Question 12 (Completeness)

Play around with the code above and see how complex of a sub-expression you can get the compiler to eliminate.

It's useful to know what common sub-expressions the compiler can eliminate because it lets you write more natural code. Consider these two (equivalent) code snippets:

```
if (k < array[len - 1] ) {
   k = array[len - 1];
}
```

and

```
int t = len - 1;
if (k < array[t] ) {
   k = array[t];
}
```

In the second, the programmer has effectively performed common sub-expression elimination explicitly leading to longer and (I would argue) less readable code.

A programmer without the benefit of CSE142L might think the longer code is faster, but the savvy alumnus of this class will know they can rely on the compiler to eliminate the extra work automatically.

11.3 Loop invariant code motion

Loop invariant code motion identifies computations in the body of a loop that don't change from one iteration to the next. The compiler can *hoist* that code out of the loop, saving instructions. For example:

In [132]:
```
optimization_example(code(r"""
extern "C" int foo(register int a, register int b){
    register int c = 0;
    for(register int i = 0; i < a; i++) {
        c += b*a;
    }
    return c;
}

"""), function="foo", cfg=True)
```

Quite a bit changes when we turn on optimizations, but the key thing to notice is that the unoptimized code has an `imull` in the loop body while the optimized code does not. In the optimized code, the `imull` has been moved into a new basic block called a *loop header*.

11.4 Strength reduction

In *strength reduction* the compiler converts a "stronger" (i.e., more general and/or slower) operation into a "weaker" (i.e. less general and/or faster) operation. The most common example is converting multiplication and division by powers of two into left and right shifts.

For example:

In [133]:
```

optimization_example(code(r"""
extern "C" int foo(register unsigned int a, register unsigned int
    return a *8;
}

"""), function="foo")
```

We don't even need to look at the optimized code to find strength reduction. Strength reduction is such a common optimization that the compiler does it even when we tell it not to optimize. Note that there is no `mull` instruction, but there is a shift arithmetic left long (`sall`) instruction with a constant `$3` that multiplies `%eax` by 8.

The optimized code does one better and folds the whole function in one `leal` . The `n(,%r,k)` addressing mode multiplies register `%r` times `k` and adds it to `n` . `k` must be power of two, which means that the processor can use a left shift to implement it.

Changing multiplies and divides in to shifts is not the only kind of strength reduction that is possible. In the code below, change the `a*8` to the expressions given in the question below and see what the compiler does:

In [134]:

```
optimization_example(code("""
extern "C" int foo(register unsigned int a, register unsigned int
    return a *8;
}
"""), function="foo")
```

Question 13 (Completeness)

Try replacing `a*8` with each of the following. Describe what the compiler does:

	What the compiler did
`a*3`	
`a*5`	
`a*11`	
`a/b`	
`a/3`	

Optional: Find an expression for which the compiler has to do something significantly different.

11.5 Constant propagation

Constant propagation allows the compiler to identify the value of constant expressions at compile time and use those constant values to simplify computations. This effectively executes part of the program *at compile time* and embeds the result in the assembly.

For example:

In [135]:
```
optimization_example(code("""

extern "C" int foo(register int a){
    register int c = 4;
    register int d = 4;
    return a + c + d;
}

"""), function="foo")

```

Again, the compiler is doing multiple things at once, but the constant propagation is visible: In the unoptimized code, it moves `$4` into both `%r12d` and `%ebx` and then adds both those register to `%eax` on the next two lines. In the optimized code it's folded both `4`s into the `8` in the `leal` instruction. Here, `leal` is using the `n(%r)` addressing mode which adds a constant `n` to `%r`. In this case, that is enough to implement the entire function.

So what happened to variables `c` and `d`? They are gone!

The compiler can make bigger things disappear:

In [136]:
```

optimization_example(code("""
extern "C" int foo(register int a, register int b){
    register int i, s = 0;
    for(i = 0; i < 10; i++) {
        s+= i;
    }
    return s;
}
"""), function="foo")
```

Since the compiler can evaluate the whole loop at compile time, it does. Bye, bye loop!

Question 14 (Optional)

Play around with the code to test the limits of constant propagation. Can you write code that could be evaluated at compile time but the compiler can't do it? What code constructs/patterns does the compiler have trouble with when it comes to constant propagation?

11.6 Loop Unrolling

The example above also demonstrates *loop unrolling*. In loop unrolling, the compiler "unrolls" a loop so that the loop body contains the computation for multiple iterations of the loop. For instance:

In [138]:
```
optimization_example(code(r"""

extern "C" int foo(register unsigned int b, int *array){
    register unsigned int i, s = 0;
    for(i = 0; i < b; i++) {
        s+= array[i];
    }
    return s;
}
""", file_name="foo.cpp"), function="foo", OPTIMIZE= ["-O0", "-Og

```

Whoa! What happened?

Let's take it step by step.

Question 15 (Completeness)

Which role does each register play (recall that the `r` or `e` prefixes don't distinguish between registers, so `%rax` and `%eax` are the same register/value)? Where is `i` ?o

register	where	role
%di	all	
%si	beginning of n2	
%si	end of n2	
%ax	from n4 on.	
%dx	from n2 on.	

Where is `i` (this is a little tricky and has to do with `%dx`)?

Show Solution

Now that we know which values are where, we can make sense of all those basic blocks and branches.

Let's start at the bottom with `n3`

```
.L3:
    addl    (%rdx), %eax
    addl    4(%rdx), %eax
    addl    8(%rdx), %eax
    addl    12(%rdx), %eax
    addl    16(%rdx), %eax
    addl    20(%rdx), %eax
    addl    24(%rdx), %eax
    addl    28(%rdx), %eax
    addq    $32, %rdx
    cmpq    %rsi, %rdx
    jne    .L3
```

The first `8` `addl` instructions are 8 copies of the loop body. The `addq` is 8 advances `%rdx` across the 8 values we just summed. The `cmpq` checks whether `%rdx` now points to the end of `array` (which we computed in `n2` and stored in `%rsi`.

In C-like pseudocode, this would look like:

```
top:
    ax += *rdx;
    ax += *(rdx + 4);
    ax += *(rdx + 8);
    ax += *(rdx + 12);
    ax += *(rdx + 16);
    ax += *(rdx + 20);
    ax += *(rdx + 24);
    ax += *(rdx + 28);
    rdx += 32;
    if (rdx != rsi)
        goto top;
```

Question 16 (Completeness)

What are blocks `n4` - `n17'` doing? How do we know that, eventually, `rdx == rsi` will be true and the loop in `n3` will terminate? What role does `%ecx` play?

Show Solution

Question 17 (Correctness - 5pts)

If `b` is 16 and CPI and CT remains constant (i.e., change in speedup is due just to change in `IC`), how much speedup would you expect from unrolling the loop (Show your work)? You only need to fill in the entries that affect the result.

Unoptimized

Basic block	static instructions	execution count	dynamic count
n0			
n1			
n2			
n3			

Optimized

Basic block	static instructions	execution count	dynamic count
n0			
n1			
n2			
n3			
n4			
n5			
n6			
n7			
n8			
n9			
n10			
n11			
n12			
n13			
n14			
n15			
n16			
n17			

Speedup:

Let's see if we can make the compiler's job easier.

Question 18 (Completeness)

Replace `i < b` with `i < 8*b` . What further simplication could the compiler make? What did the compiler actually do?

While, in principle, a compiler could unroll any loop, it will refuse to unroll some loops because they are too complicated:

In [139]:

```
optimization_example(code(r"""

extern "C" int foo(register unsigned int a, register unsigned int
    register unsigned int i, s = 0;

    i = 8*b;     // LOOP B
    while(i > 0) {
        i -= b;
        s += i;
        if (i == a)
            continue;
        if (i == b)
            break;
        if (i % 4) {
            s++;
        }
    }

    return s;
}

"""), function="foo",OPTIMIZE=["-O0", "-O4 -funroll-loops"], cfg=
```

Question 19 (Completeness)

Simplify the loops above so that the compiler will unroll it. You can change the computation that the loop performs, but try to alter the loop as little as possible. What characteristics or parts of the loop are

Question 20 (Optional)

Did the compiler use the same approach to unrolling this loop? If not, what's different? Compute the instruction count for the optimized and unoptimized code. Is this approach as effective as what you saw earlier? Why or why not?

11.7 Dead Code Elimination

Compilers will also remove code that will never run. This code is called *dead code*. It might seem unlikely that such code would exist, but shows up pretty often due to some common programming practices or as a byproduct of other optimizations.

For instance, the first `if` condition is always false, so the compiler eliminates the branch and the `return 0`. This is another example of an optimization that happens even at `-O0`.

In [11]:

```
dead = build(code(r"""
extern "C" int foo(register unsigned int b, register unsigned c){
    if (b != b + b - b)
        return 0;
    else if (b == 1)
        return c;
    else
        return 2*c;
}
""", file_name="foo.cpp"), arg_map(OPTIMIZE=["-O0"]))

compare([dead[0].source("foo"), dead[0].cfg("foo")])

```

11.8 Combining Single-function Optimizations

Each of these optimizations is interesting in isolation, but they are more powerful together.

Consider this code. I've used a macro `DIV` to make it clearer where division occurs. This code uses division in several different ways. Study it, and, assuming `size = 30`, calculate how many divides the program will execute?

Run the cell and let's see what the compiler does:

In [93]:

```
optimization_example(code(r"""
#include<cstdint>
#include<cstdlib>
extern "C" uint32_t div_loop(uint64_t * array,  unsigned long int
#define X 3
#define Y 8
#define DIV(a,b) (a / b)

    for(uint32_t i = 0; i < DIV(size, 3); i++) {
        array[DIV(i, 2) +  DIV(Y, X)] = DIV(size, 3);
    }
    return array[0];
}


"""), cfg=True, function="div_loop", OPTIMIZE=["-O0", "-O1 -fno-i

```

To start, how many divides (i.e., instructions with `div` in their names) are there? -- zero!

Something strange is going on. How is the compiler dividing (hint: it's the weirdest looking thing in this code) Even if you don't know how *exactly* it's dividing, can you tell *how many times* its dividing?

Question 21 (Completeness)

For each `DIV()` in the code above, list the optimizations that the compiler applied to the code and the combined effect they had. (hint: the optimizations we've discussed are sufficient)

	optimizations	effect
DIV(size,3)		
DIV(8/3)		
DIV(i,2)		

Show Solution

11.9 Function Inlining

So far, all the optimizations have only affected a single function and none of them will have any impact on the call graph of our program. This means they cannot hope to fix those monstrous call graphs that we got from invoking relatively simple STL functions. Function *inlining* will change all that.

11.9.1 Function Call Overhead

Before we get to inlining, let's talk a little about functions. When you a write a function, the code you write turns into the "body" of the function. However, the processor has to do some work to *make* the function call and each function includes some overhead instructions in addition to instructions for code the function contains. For example, consider this code:

In [12]:
```
prologue = build(code(r"""

extern "C"
long int sum(long int a, long int b) {
    return a + b;
}

int main(){
    return sum(1,2);
}
"""), arg_map(DEBUG_FLAGS="-g0", OPTIMIZE="-O4"))
compare([prologue[0].source(), prologue[0].asm()])

```

The body of `sum` is very simple: It should just be a single add instruction, but instead it has to return as well and the `endbr64` instruction (https://stackoverflow.com/a/56910435/3949036) which is a recently-added security feature. The *call site* in `main` takes 5 instructions: Adjust the stack, store two arguments into registers, call function, and re-adjust the stack pointer. The `ret` is part of the overhead for calling `main`, not `sum`.

In this case, the *function call overhead* is eight instructions: 1 `x subq`, 2 `x movl`, 1 `x call`, 1`x endbr64`, 1 `x ret`, and 1`x addq`.

Recall the earlier example with `std::sort` and how many function calls were involved. Each of them incurred this kind of overhead. What a waste!

The Application Binary Interface (ABI)

There are several standardized protocols for how arguments are passed to functions and even how names are mangled. These protocols are called "application binary interfaces" or ABIs. It's important that the caller (the function that calls) and the callee (the function that is called) agree on the ABI. The ABI dictates which arguments go in which register and in what order, the number of bits in an `int` vs a

`long int`, how things like pass-by-value vs. pass-by-references are implemented, and how C++ virtual function tables (which implement virtual functions) are laid out. Generally speaking, if two object files (i.e., `.o` files) were compiled with the same ABI, functions in one object file can call functions in another.

For the most part, you can think of there being one ABI per operating system, but that's not completely accurate. Linux has (at least) two: one for the kernel and one for user programs. Microsoft has one. Intel has defined a standard as well. The [wikipedia page (https://en.wikipedia.org/wiki/](https://en.wikipedia.org/wiki/) has a little more detail.

If you're curious, use the code cell to see how the compiler passes, `struct`s, pointers to `struct`, and C++ references to `struct`s. What's surprising about how it implements those three different language constructs?

Question 22 (Optional)

What happens to function call overhead if you add more arguments (something interesting happens past 8)? What if pass a struct? How does the complexity of *the caller* affect function call overhead?

11.9.2 Removing Function Call Overheads

One way to remove the function call overhead is to copy the body of the function (i.e., the useful part) to the caller. Then, we don't need to pass arguments, make the `call`, or do the `ret`. The compiler can do this automatically by *inlining* the function.

For instance, the compiler can inline `foo` into `loop`:

In [25]:

```
overhead = build(code(r"""
#include<iostream>
#include <unistd.h>
#include"fastrand.h"

int k = 2;
extern "C"
int inline __attribute__ ((used)) foo( register int a, register i
    if (a + k)
        return b * k * a;
    else
        return 2 * k * a;
}

extern "C"
int loop(int bound){
    register int i;
    register int s = 0;
    for(i = 0; i < bound; i++) {
        s += foo(i, i+1);
    }
    return s;
}


"""), arg_map(OPTIMIZE=["-O0 -g0", "-O1 -g0"]))

compare([overhead[0].source("foo"), overhead[0].cfg("foo", remove
compare([overhead[0].source("loop"), overhead[0].cfg("loop", remo
compare([overhead[1].cfg("loop", remove_assembly=True)], ["loop()
```

You can see the inlining clearly in the control flow graph. And you we examine the assembly, you'll notice that all those overhead instructions are gone:

In [23]:
```
overhead[1].cfg("loop")
```

But this is just the beginning of inlining's power, because it also vastly increases the opportunities to apply other optimizations. Consider `foo()` in the example above. Without inlining, the compiler can only apply optimizations that will work for *all* values of `a` and `b`. However, once `foo()` is inlined, the compiler can optimize *that copy* of `foo()` for the values of `a` and `b` at that call site. Then it is free to apply all the other optimizations we've discussed already.

For instance:

In [28]:
```
byebye = build(code(r"""
extern "C" inline int the_loop(register int a) {
    register int i;
    register int sum = 0;
    for(i = 0; i < a; i++) {
        sum += i;
    }
    return sum;
}

extern "C"
int caller(int k) {
    int sum = 0;
    for(int i = 0; i < k; i++) {
        sum += the_loop(20);
    }
    return sum;
}

"""), arg_map(OPTIMIZE=["-O0 -g0 -fkeep-inline-functions", "-O2 -

compare([byebye[0].source(), byebye[0].cfg("the_loop")], ["the_lo
compare([byebye[0].source("caller"), byebye[0].cfg("caller")], ["
compare([byebye[1].cfg("caller")], ["caller() with inlining"])


```

Bye bye, function call! Bye Bye, loops!

Question 23 (Completeness)

Which optimizations did the compiler apply to come up with inlined, optimized version of `caller()`? For each optimization explain what it accomplished.

Show Solution

12 C++ Revisited

C++ is an amazing language and it places a large burden on the compiler which has to implement all its interesting features and make it go fast. Below, let's look at how the optimizations you've explored can fix the messy call graph we saw earlier. Then, we'll investigate the impact of virtual functions.

12.1 Optimizations in C++

We now have all the tools we need to see how a compiler can handle the messiness of C++ and its standard library. Here's the sort example from earlier with and without optimizations:

In [31]:

```
no_inlining = call_graph(code(r"""
#include<algorithm>
#include"cfiddle.hpp"


extern "C" uint64_t stl_sort(uint64_t * array, uint64_t size);

extern "C" void sort_harness(uint64_t size, uint64_t seed) {

    uint64_t *array = new uint64_t[size];
    for(uint64_t i = 0; i < size; i++) {
        array[i] = fast_rand(&seed);
    }
    stl_sort(array, size);
}

int main() {
    sort_harness(1000, 1);
    return 0;
}

extern "C" uint64_t stl_sort(uint64_t * array, uint64_t size) {
    start_measurement();
    std::sort(array, &array[size]);
    end_measurement();
    return array[size-1];
}

""", file_name="./stl_sort.cpp"), root="stl_sort", quiet_on_succe

inlining = call_graph("./stl_sort.cpp",opt="-O1", root="stl_sort"

compare([no_inlining, inlining], ["Without inlining", "With inlin

sort = build("./stl_sort.cpp", build_parameters=arg_map(OPTIMIZE=

compare([sort[0].cfg("stl_sort"), sort[1].cfg("stl_sort")], ["Wit


```

What a difference some optimization can make! A few things to note about the optimized code:

1. The call to `std::sort()` is gone. It's been inlined into `stl_sort()`, which now calls several other functions that `std::sort()` calls.
2. The CFG for `stl_sort()` is more complex because it contains parts of `std::sort`.
3. The optimized call graph above is *much* shallower, and there are vastly few function calls (you'll have to double click and zoom in to see that). The call graph is probably missing some calls due to some limitations of `gprof` (there are a few other function calls in `one()`), but the situation is clearly much better.

In [34]:
```
sort = build("./stl_sort.cpp",
             build_parameters=arg_map(OPTIMIZE=["-O0", "-Og", "-O
sort_run = run(sort, function="sort_harness", arguments=arg_map(s
```

In [35]:
```
sort_df = PE_calc(sort_run.as_df())
display(sort_df)
plotPEBar(df=sort_df,
          what=[("OPTIMIZE", "IC"),
                ("OPTIMIZE", "CPI"),
                ("OPTIMIZE", "CT"),
                ("OPTIMIZE", "ET")])
```

Question 24 (Correctness - 3pts)

Based on the data above compute the speedup of `-O3` over `-O0` for `IC`, `CPI`, and `ET`.

	speedup
IC	
CPI	
ET	

That is why you should compile your C++ code with optimizations turned on.

12.2 C++ Virtual Functions

C++ has a lot of fancy object-oriented features, and one of the most powerful is virtual functions (sometimes also called "virtual methods"). However, an often-cited downside of virtual functions is that they are more expensive than normal functions. A google search (https://www.google.com/search?q=C%2B%2B+virtual+function+call+overhead) for "C++ virtual function call overhead" produces an astonishing number of hits. Let's see for ourselves!

Virtual function refresher: Virtual Functions in C++ allow child classes to override member functions of parent classes. Then, when a member function is called on a pointer to an instance of the class (e.g., `p->foo()`), C++ determines *at run time* which version of the function to call. This means the compiler doesn't know what function is

being called.

In the code below, we have a class with a single, virtual function that we'll call in two different ways.

In `static_call()` we allocate an instance of `A` as a local variable. This lets the compiler know, for certain, that `a` is actually of type `a` and not a subclass of `A` that has overridden `foo()`. As a result, when we invoke `a.foo()`, it is not a virtual call. It's a "static" call.

In `virtual_call()`, we create an instance of `A` using `new` and store *a pointer to it* in `a` which is of type `A*`. Now, when we invoke `a->foo()`, all the compiler knows is that `a` points to an instance of `A` or a *subclass of* `A` that might have overridden `foo()`. In this case, it has to make a "virtual" or "dynamic" call to `foo()`. (It's worth noting that it seems like the compiler could infer that `a` points to an instance of `A` instead of an instance of a subclass. However, our compiler seems to not be that smart.)

In [92]:

```
virt = build(code(r"""
#include<cstdint>
#include"cfiddle.hpp"

class A {
public:
    virtual void bar() {}
    virtual int foo(int x) {
        int s = 0;
        for(int i = 0; i < 10; i++) {
            s += x;
        }
        return s;
    }
};


extern "C" int static_call(uint64_t  size) {
    A a;
    register int sum = 0;

    start_measurement();
    for(register uint64_t i = 0; i < size ; i++)
        sum += a.foo(4);
    end_measurement();

    return sum;
}

extern "C" int virtual_call(uint64_t  size) {
    register A * a = new A();
    register int sum = 0;

    start_measurement();
    for(register uint64_t i = 0; i < size ; i++)
        sum += a->foo(4);
    end_measurement();

    return sum;
}

""", file_name="virt.cpp"), build_parameters=arg_map(OPTIMIZE=["-


```

In [96]:

```
display(heading("Static call (no inlining)"))
display(virt[0].cfg("static_call", number_nodes=True))
display(heading("virtual call (no inlining)"))
display(virt[0].cfg("virtual_call", number_nodes=True))
```

On the top, is `static_call()`. In block `n2` you can see the call to `A::foo(int)` in the form of `callq 0x10d0`.

`virtual_call()` is on the bottom. The structure is similar the same, and still ends up in `n2`. Instead of invoking the function via a fixed address, it's calling the function whose address is in `%rax`: `callq *8(%rax)`.

The instructions before `callq *8(%rax)` are looking up the address of `a`'s virtual method `foo` in `a`'s *virtual table* or *vtable*.

Here's what's going on in `n3` of `virtual_call()`:

1. `a` is in `%rbp`
2. `movq` loads the first word of `b` into `%rax`. According to the C++ ABI, the first word of an object with virtual methods is the address of the object's vtable, so `%rax` now has the base of the vtable.
3. `movl` set's the function's second argument for `foo` to `4`.
4. `movq` set's the function's first argument to the address of `a`. This is the implicit `this` parameter that every method call receives.
5. `callq *8(%rax)` adds 8 to the base address of the vtable, loads that value as a function pointer, and calls it. The 8 is the offset of `foo` in `a`'s virtual table (the function `bar()` is the first slot at offset 0).

The invocation of `a.foo()` on the is simpler: `a` is in `%rsi`. The code still passes `4` and `this`, but it doesn't have to load the vtable, it just calls `A::foo` directly.

In this code, the difference between the virtual and non-virtual invocation is pretty small: Just a few instructions per loop iteration.

But let's see what happens when we turn on more optimizations:

In [140]:
```
display(heading("Static call (Lots of optimizations)"))
display(virt[1].cfg("static_call", number_nodes=True))
display(heading("virtual call (Lots of optimizations)"))
display(virt[1].cfg("virtual_call", number_nodes=True))
```

For `static_call()`, the compiler could apply many of the optimizations we've studied: It inlines `a.foo()`, unrolls the loop and evaluates it at compile time, and then multiplies it times `size`. It's all wrapped up in `leal` and `shll` in block `n1`.

For `virtual_call()`, the compiler...sure does something complicated. I haven't traced through what exactly it is (If you figure it out, let me know). However, it's clear what it did not do: It did not get rid of the virtual function call. It's there in `n4`.

Why can't the compiler inline `a->foo()`? Or at least just call it once and multiply the result by `size`? Because it doesn't know what function it will invoke. The version that actually runs could return random numbers or never return at all, so the compiler *must* execute it just as the code calls it.

Let's see what performance looks like.

In [142]:

```
virtual_data = run(virt, function=["static_call", "virtual_call"]
                   arguments=arg_map(size=100000000)).as_df()
virtual_data["experiment"] = virtual_data["function"] + "-" + vir
virtual_df = PE_calc(virtual_data)
display(virtual_df)
plotPEBar(df=virtual_df, what=[("experiment", "IC")])

```

Question 25 (Correctness - 2pts)

How much speedup do optimizations provide in for the static calls? For the dynamic?

Speedup in static calls:

Speedup in dynamic calls:

This is the main cost of virtual functions: It's not that calling virtual functions is expensive, it's that using virtual functions vastly reduces the effectiveness of compiler optimizations.

And this is why the `stl` containers don't use virtual functions -- they are all template-based instead. Templates are processed at compile time, so the compiler always knows what's getting called and it can apply inlining. The massive reduction in call graph complexity we saw for `std::sort` would not have been possible if the `std::sort` used virtual functions to implement a generic sorting algorithm.

Question 26 (Optional)

What did the compiler do to achieve speedup for the virtual call case?

13 Compilers are Easily Confused

So far, we have seen the compiler do some pretty remarkable things. The transformations it performed on `std::sort()` are pretty impressive, but we have also seen how some program constructs (like virtual functions) can limit what the compiler can do.

There is a second, bigger problem that limits how effectively compiler can optimize: Memory.

13.1 Aliases

Consider this code and it's assembly:

In [143]:

```
alias = build(code(r"""
#include"cfiddle.hpp"

extern "C" void values(int * c, int a, int b) {
    *c += a + b;
    *c += a + b;
    *c += a + b;
    *c += a + b;
    *c += a + b;
}

extern "C" void pointers(int *c, int * a, int * b) {

    *c += *a + *b;
    *c += *a + *b;
    *c += *a + *b;
    *c += *a + *b;
    *c += *a + *b;
}"""
), build_parameters=arg_map(OPTIMIZE=["-O4"], DEBUG_FLAGS="-Og"))
compare([alias[0].asm("values"), alias[0].asm("pointers")], ["val
```

Despite the code looking almost the same, the assembly output is quite different.

Question 27 (Completeness)

Add and remove copies of the expression to each function. What happens to the relative lengths of the resulting assembly code? Why? What can and can't the compiler assume in each function? What optimization can it apply?

Show Solution

A situation where two pointers refer to the same values is called an "alias", and "alias analysis" is compiler's attempt to determine whether or not two variables might alias with one another. Unfortunately, alias analysis is, in general, very difficult. In the case of the code above, it is not possible: It really is the case that `c` could be equal to `a` or `b`, so the compiler has to take that into account.

However, the *programmer* might know that the alias does not exist. Aliases have a big enough effect on optimizations that the latest standards for C and most dialects of C++ provide a keyword to tell the compiler that aliases don't exist. In C, it's `restrict`. For the dialect of C++ that `g++` implements it's `__restrict__`. Here it is in action:

In [144]:
```
noalias = build(code(r"""
#include"cfiddle.hpp"

extern "C" void pointers(int * __restrict__ c, int *__restrict__

    *c += *a + *b;
    *c += *a + *b;
    *c += *a + *b;
    *c += *a + *b;
    *c += *a + *b;
}"""
), build_parameters=arg_map(OPTIMIZE=["-O2"], DEBUG_FLAGS="-Og"))
compare([noalias[0].asm("pointers")])
```

And now, `*a + *b` *is* a common subexpression and the compiler can optimize things.

13.2 Function Calls and Memory

The `restrict` keyword can only do so much. Check out this code:

In [145]:

```
side_effect = build(code(r"""
#include"cfiddle.hpp"

extern "C" void something() {
}

extern "C" void pointers(int * __restrict__ c, int *__restrict__
    *c += *a + *b;
    something();
    *c += *a + *b;
    something();
    *c += *a + *b;
    something();
    *c += *a + *b;
    something();
    *c += *a + *b;
}

""",
file_name="side_effect.cpp"), build_parameters=arg_map(OPTIMIZE=[
compare([side_effect[0].asm("pointers")], [html_parameters(side_e
```

Question 28 (Completeness)

What's preventing common subexpression elimination here? What is an optimization that would alleviate the problem?

Show Solution

14 Practical Rules For Using Compiler Optimizations

The single most important lesson to learn from this lab is that you should compile your code with optimizations turned on. It is the easiest 2-10x boost in performance you can get.

Fortunately, it's pretty simple to do that. Somewhat overwhelmingly, gcc provides around 300 flags (https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html) that control optimization and a bunch of tunable parameters as well, but in practice you don't need to worry about them.

There are just a handful that are typically useful. Here's what the gcc docs have to say about them:

- `-O0` : Perform no optimizations. You should never use this unless you're just playing around.
- `-O1` : "the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time."
- `-O2` : "GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff." "Space" in this context means the number of static instructions generated.
- `-O3` : "Optimize yet more"
- `-Og` : "Optimize debugging experience. -Og should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience."

A final option that can be useful is `-march=` . This allows you to set the specific version of the processor your are compiling for. By default, gcc generates assembly that'll run on any 64-bit x86 machine (the first of these appeared in 2002). There are many options here (you can see them all with `gcc -Q --help=target`), but most useful is `-march=native` . This setting compiles your code for the machine you are running on. It might not run on an older machine.

So, if you want to run code on the local machine. A good default set of flags is `-O3 -march=native` . In this class, we compile locally an run stuff in the cloud. For the cloud machines, the right setting is `-march=skylake` , so that's a good default for this class.

The benefits of `-march=native` are highly variable.

Among these, `-Og` is a relatively new flag that "optimizes the debugging experience". What does that mean? The optimizations we described above (especially function inlining, but others as well) can cause strange behavior when you debug. For instance, consider the inlined version of `side_effect()` in the previous section. If you set a break point in the `something()` function that `pointers()` calls in the example above, your code would never stop because that function is never called. Likewise, we've seen loops and variables disappear. This can make debugging really difficult and frustrating. On the other hand, compiling with `-O0` will make the code much, much slower (just look at the graphs above).

So `-Og` strikes a balance: It optimizes but avoids these problems in debugging.

Here's what that balance looks like:

In [148]:
```
plotPEBar(df=sort_df,
          what=[("OPTIMIZE", "IC"),
                ("OPTIMIZE", "CPI"),
                ("OPTIMIZE", "CT"),
                ("OPTIMIZE", "ET")])
```

There's not a huge difference between `-O3` and `-Og` ... You might conclude that `-O3` is not worth it, but I'd want to see data for a range of different, more realistic programs. Our examples here are tiny.

So, in practice you should:

- Use `-Og` when debugging and developing.
- Use `-O3` when deploying (and maybe `-march` if you can be sure of what hardware you'll be running on).

The other 298 options have their uses, but unless you are interested in squeezing out the very last drop of performance (and doing the experiments to check that the optimizations help), they are not worth the effort and are pretty hard to use productively. It's usually a lot of trial and error.

That said, by the time you've finished this class, you'll have a pretty deep understanding of CPU performance and how to look "under the hood" at what the compiler is doing. So, you'll be in a good position to read about those other options and design meaningful experiments that let you measure their impact.

15 How the Compiler Implements (and Optimizes) Common C++ Constructs [...]

16 Programming Assignment: Can You Beat the Compiler?

> Here's how to approach this lab.
>
> 1. Read through the whole thing and run the example code in the code cells.
> 2. Do your work in the "Running the Code" section. It has the key commands you'll need to evaluate and analyze your results.
> 3. When you are all done, go to the Final Measurement section and follow those instructions.

In this programming assignment you'll be taking the place of the compiler to optimize several invocations of a simple implementation of matrix multiply.

The code uses a very simple matrix library. Here's the header;

In [108]:
```
1 render_code("matmul.hpp")
```

And the rest of the implementation:

In [109]:
```
1 build("matmul.cpp")[0].source(show=("START_IMPL","END_IMPL"))
```

The code you'll be modifying is below. Please read through it and the comments carefully.

In [15]:

```
matmul_solution = build(code(r"""

#include"cfiddle.hpp"
#include"matmul.hpp"


// Compute the matrix product: c = a * b
// This is a reference implementation.  You don't need to modify
extern "C" void matrix_product(Matrix * c, Matrix * a, Matrix * b
    for(uint i = 0; i < a->rows; i++) {
        for(uint j = 0; j < b->columns; j++) {
            matrix_write(c, j, i, 0);
            for(uint k = 0; k < a->columns; k++) {
                matrix_write(c, j, i,
                        matrix_read(c, j, i) +
                           (matrix_read(a, k, i) *
                            matrix_read(b, j, k)));
            }
        }
    }
}

// These are the functions you'll be modifying.   Right now, they
// implementation above, but when you are done, they should be sp
// are called by the go() function in the next cell.
extern "C" void matrix_product_1(Matrix * c, Matrix * a, Matrix *
    for(uint i = 0; i < a->rows; i++) {
        for(uint j = 0; j < b->columns; j++) {
            matrix_write(c, j, i, 0);
            for(uint k = 0; k < a->columns; k++) {
                matrix_write(c, j, i,
                        matrix_read(c, j, i) +
                           (matrix_read(a, k, i) *
                            matrix_read(b, j, k)));
            }
        }
    }
}

extern "C" void matrix_product_2(Matrix * c, Matrix * a, Matrix *
    for(uint i = 0; i < a->rows; i++) {
        for(uint j = 0; j < b->columns; j++) {
            matrix_write(c, j, i, 0);
            for(uint k = 0; k < a->columns; k++) {
                matrix_write(c, j, i,
                        matrix_read(c, j, i) +
                           (matrix_read(a, k, i) *
                            matrix_read(b, j, k)));
            }
        }
    }
}

extern "C" void matrix_product_3(Matrix * c, Matrix * a, Matrix *
    for(uint i = 0; i < a->rows; i++) {
```

```
        for(uint j = 0; j < b->columns; j++) {
            matrix_write(c, j, i, 0);
            for(uint k = 0; k < a->columns; k++) {
                matrix_write(c, j, i,
                             matrix_read(c, j, i) +
                                (matrix_read(a, k, i) *
                                 matrix_read(b, j, k)));
            }
        }
    }
}

extern "C" void matrix_product_4(Matrix * c, Matrix * a, Matrix *
    for(uint i = 0; i < a->rows; i++) {
        for(uint j = 0; j < b->columns; j++) {
            matrix_write(c, j, i, 0);
            for(uint k = 0; k < a->columns; k++) {
                matrix_write(c, j, i,
                             matrix_read(c, j, i) +
                                (matrix_read(a, k, i) *
                                 matrix_read(b, j, k)));
            }
        }
    }
}

extern "C" void matrix_product_5(Matrix * c, Matrix * a, Matrix *
    for(uint i = 0; i < a->rows; i++) {
        for(uint j = 0; j < b->columns; j++) {
            matrix_write(c, j, i, 0);
            for(uint k = 0; k < a->columns; k++) {
                matrix_write(c, j, i,
                             matrix_read(c, j, i) +
                                (matrix_read(a, k, i) *
                                 matrix_read(b, j, k)));
            }
        }
    }
}

""", file_name="matmul_solution.cpp"), build_parameters=arg_map(M

```

SourceCodeModified Errors If you get this error when you run the cell above

```
SourceCodeModified: The contents of matmul_solution.cpp have changed since cfiddle wrote them last.  Aborting to prevent loss of work.
```

That means that you've modified the contents of `matmul_solution.cpp` outside of the Jupyter notebook, and cfiddle is refusing to overwrite your work. This will happen, for instance, if you're using using VSCode to write your solution.

You have several options:

1. Run the `build("matmul_solution.cpp")` cell below to compile your modified code.
2. Copy the contents of your modified `matmul_solution.cpp` into the cell above.

3. Delete your modified `matmul_solution.cpp` and re-run the cell above to rewrite the file.

The five `matrix_product_*()` functions are what we will actually be testing. Right now they are all identical.

Then there's the test harness that calls them:

In [431]:

```
1 build("matmul.cpp")[0].source(show=("START_TESTS", "END_TESTS"))
```

```
100%                                                    1/1 [00:00<00:00, 42.49it/s]
```

Out[431]:

```
//START_TESTS
extern "C" void test_matrix_product_1(uint rows, uint size1, uint col
umns, uint size2, uint log_size, uint64_t seed) {
        ////////////////////////////////////

        Matrix *a1 = matrix_new(columns, size1);
        Matrix *b1 = matrix_new(size1, rows);
        Matrix *c1 = matrix_new(columns, rows);
        matrix_random(a1, &seed);
        matrix_random(b1, &seed);

        start_measurement();
        matrix_product_1(c1, a1, b1);
        end_measurement();

        matrix_delete(a1);
        matrix_delete(b1);
        matrix_delete(c1);
}


extern "C" void test_matrix_product_2(uint rows, uint size1, uint col
umns, uint size2, uint log_size, uint64_t seed) {
        ////////////////////////////////////

        Matrix *a2 = matrix_new(size2, size2);
        Matrix *b2 = matrix_new(size2, size2);
        Matrix *c2 = matrix_new(size2, size2);
        matrix_random(a2, &seed);
        matrix_random(b2, &seed);

        start_measurement();

        matrix_product_2(c2, a2, b2);
        end_measurement();

        matrix_delete(a2);
        matrix_delete(b2);
        matrix_delete(c2);
}

extern "C" void test_matrix_product_3(uint rows, uint size1, uint col
umns, uint size2, uint log_size, uint64_t seed) {
        ////////////////////////////////////

        Matrix *a3 = matrix_new(1 << log_size, 1 << log_size);
        Matrix *b3 = matrix_new(1 << log_size, 1 << log_size);
        Matrix *c3 = matrix_new(1 << log_size, 1 << log_size);
        matrix_random(a3, &seed);
        matrix_random(b3, &seed);

        start_measurement();

        matrix_product_3(c3, a3, b3);
```

```
        end_measurement();

        matrix_delete(a3);
        matrix_delete(b3);
        matrix_delete(c3);
}

extern "C" void test_matrix_product_4(uint rows, uint size1, uint col
umns, uint size2, uint log_size, uint64_t seed) {
        ////////////////////////////////////

#define SIZE 512
        Matrix *a4 = matrix_new(SIZE, SIZE);
        Matrix *b4 = matrix_new(SIZE, SIZE);
        Matrix *c4 = matrix_new(SIZE, SIZE);
        matrix_random(a4, &seed);
        matrix_random(b4, &seed);

        start_measurement();

        matrix_product_4(c4, a4, b4);

        end_measurement();
        matrix_delete(a4);
        matrix_delete(b4);
        matrix_delete(c4);
}

extern "C" void test_matrix_product_5(uint rows, uint size1, uint col
umns, uint size2, uint log_size, uint64_t seed) {
        ////////////////////////////////////

        Matrix *a5 = matrix_new(SIZE, SIZE);
        Matrix *b5 = matrix_new(SIZE, SIZE);
        Matrix *c5 = matrix_new(SIZE, SIZE);
        seed = 1;
        matrix_random(a5, &seed);
        matrix_random(b5, &seed);

        start_measurement();

        matrix_product_5(c5, a5, b5);

        end_measurement();

        ////////////////////////////////////
        matrix_delete(a5);
        matrix_delete(b5);
        matrix_delete(c5);

}
//END_TESTS
```

Each function (`test_matrix_product_1()` to `test_matrix_product_5()`) calls one of the implementations in `matmul_solution.cpp` . The difference between the tests lies in how their arguments are set: `matrix_product_1()` is called with arbitrary-sized matrices, while the others constrain the shape of the matrices they pass to the matrix product implementation. The last one also limits what `seed` will be.

For instance, `test_matrix_product_2()` calls `matrix_product_2()` only with square matrices. For `matrix_product_3()` , `test_matrix_product_3()` only calls it with square where the dimensions are a power of two.

Your task is to create optimized implementations of `matrix_product_1()` through `matrix_product_5()` by manually applying a sequence of compiler optimizations.

All the `test_matrix_product_*()` take the same arguments, even though they ignore some of them. This just makes it easier to call with Cfiddle.

The test suite in `run_tests.cpp` contains tests for each of the `matrix_product_*()` functions, but they only test for matrices that could be passed to those functions using the code in the corresponding `test_matrix_product_*()` (e.g., the tests for `matrix_product_2()` only include square matrices).

This means your implementation of the `matrix_product_*()` functions need not be correct for all matrices. They only need to generate the correct answer when called as they are called in `test_matrix_product_*()` . This is, in essence, what the compiler does when it inlines a function -- once it makes a copy it can tune that copy to the particular call site.

You can make the following assumptions:

- No dimension of any matrix is larger than 2048
- The calls in the `test_matrix_product_*()` function are the only calls to the corresponding `matrix_product_*()` functions.

And you are subject to the following constraints:

- Your code will be compiled with `-O0` .
- You can only use "vanilla" C++, so you can't use
  - inline assembly language.
  - The `__attribute__` keyword.
- Multithreading is not allowed.
- You cannot use the `inline` keyword (but you can inline by hand).
- You can, however, use `__restrict__` and `register` .
- You are free to leverage the C/C++ preprocessor.
- You are also free to inspect the assembly that the compiler produces to get ideas.
- Code generation (i.e., writing code that write code) is allowed.
- You can only modify the code in `matmul_solution.cpp` (not `matmul.cpp` or `matmul.hpp`).

**What is `fast_rand()?**: `fast_rand()` is an extremely fast and extremely poor random number generator. Higher-quality random number generators are slow enough that their execution can easily overwhelm anything we are trying to measure.

Another advantage is that `fast_rand()` consumes a current random seed and produces a new one. This gives us very good control over what random numbers it produces (which, if you think about, is an oxymoron...), which makes it good for testing: With the same seed, we'll always get the same values.

16.1 Applying the Optimizations

You can apply any optimizations that you'd like (subject to the list of constraints above). Here's a good list and good order to think try applying them:

- Function inlining.
- Register assignment.
- Common sub-expression elimination.
- Constant propagation.
- Loop invariant code motion.
- Strength reduction.
- Loop unrolling.

Later in this lab, we will ask you compare the performance of the code with different optimizations applied. To facilitate that, I suggest the following algorithm for doing this lab:

1. Pick an optimization.
2. Apply it to all 5 versions of matrix product. Optimizations will apply differently depending on the assumption you can make for each version.
3. Make the regressions pass.
4. Save a copy with a descriptive name like `00_matmul_solution_inlining.cpp` and `01_matmul_solution_register_assignment.cpp`.
5. Check your performance. If you meet the targets for some of the versions, you can stop optimizing them.
6. Repeat.

To be clear, you *are not* required to apply the optimizations in the order listed (but it worked

About the canary: You might notice `canary.cpp` in the example above and the performance results below. Sometimes our servers get slow for unknown reasons. The canary is piece of code with known performance. The autograder runs it to make sure the machine is performing well. The performance of the canary has no effect on your score.

16.2 Run And Measure the Code [...]

This section contains some cells with useful code to compile, run, and analyze your code. You can rearrange or modify them as you see fit.

16.3 Things To Try

Here are some suggestions on how to approach the lab.

1. Apply the optimizations suggested above.
2. Work in baby steps and run the regressions *a lot*. If you make a bunch of big changes and something goes wrong, debugging will be almost impossible. Make incremental, correct changes (as verified by `run_tests.exe`). Make use of the filter options on `run_tests.exe` to reduce testing time.
3. Check your performance against the targets to potentially do less work. For instance, if your `matrix_product_2()` implementation meets the performance targets for `matrix_product_3()`, you can just copy and paste it. Check performance with the cells under "Final Measurement and Grading".

16.4 The Test Suite [...]

16.5 Useful Tools [...]

16.6 Analyzing Your Solution [...]

16.7 Final Measurement and Grading

When you are done, make sure your best solution is in `matmul_solution.cpp`. Then you can submit your code to the Gradescope autograder. It will run the commands given above and use the `ET` values from `autograde.csv` to assign your grade.

Your grade is based on your speed up relative to the original version of `matmul_solution.cpp` in the lab. The target speedups are visible in the output of the autograder cell below.

You don't get extra credit for beating the target.

To get points, your code must also be correct. The autograder will run the regressions in `run_tests.exe` to check it's correctness.

You can mimic exactly what the autograder will do with the command below. You can run the cell below to list them and the target speedups.

After you run it, the results will be in `autograde/autograde.csv` rather than `./autograde.csv`. This command builds and runs your code in a more controlled way by doing the following:

1. Ignores all the files in your repo except `matmul_solution.cpp`.
2. Copies those files into a clean clone of the starter repo.
3. Builds and runs `run_tests.exe` with the hidden tests enabled.
4. Runs your code using `run_bench.py`.
5. It then runs the `autograde.py` script to compute your grade.

Running the cell below does just what the Gradescope autograder does. And the cell below shows the name and target speedups for each benchmark. This takes 1-2 minutes to run.

> **Only Gradescope Counts** The scores produced here **do not** count. Only gradescope counts. The results here should match what Gradescope does, but I would test your solution on Gradescope well-ahead of the deadline to ensure your code is working like you expect.

> **There are hidden test cases** The autograder will run some tests that you can't see. So it's possible that the cells below will pass, but gradesceope will fail.

In [109]:
```
!cse142 job run --take matmul_solution.cpp --lab compiler-bench -
```

In [110]:
```
render_csv("autograde/autograde.csv")
```

And run the autograder

In [115]:
```
#run the autograder locally
from autograde import compute_all_scores
df = compute_all_scores("autograde/autograde.csv", "targets.csv")
display(df)
print(f"total points: {round(sum(df['score']), 2)}")
```

The "score" column contains the number of points you'll receive.

And see the autograder's output like this:

In [116]:
```
render_code("autograde.json")
```

Most of it is internal stuff that gradscope needs, but the key parts are the `score`, `max_score`, and `output` fields.

All that's left is commit your code:

In [ ]:
```
!git add matmul_solution.cpp
!git commit -m "Yay! I finished the lab!"
!git push --force
```

17 Recap

This lab has illustrated a range of common compiler optimizations that improve program performance. We've seen how optimizations can work in isolation and, often more important, how one optimization (I'm looking at you, inlining) can often unlock additional opportunities to apply further optimizations. We've also seen how optimizations are especially important for C++ and other languages where small functions are common. Finally, we quantified the impact of optimizations using the performance equation.

18 Turning In the Lab [...]