In [1]:
```
# This cell runs automatically.  Don't edit it.
from CSE142L.notebook import *
from notebook import *
from cfiddle import *
```

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

Lab 2: The Compiler

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 FAQ and Updates

- There are no updates, yet.

2 Using in the Correct Environment on DataHub

Use the right Datahub environment There is a different environment for each lab on DataHub, and you must use the correct environment when working on the corresponding lab.

To get into the right environment when you start a new lab, you should:

1. Connect to Datahub. If it takes you to the menu of environments, select the appropriate one.
2. Otherwise, click "Control Panel" in the upper right.
3. Then click "Stop my Server"
4. Click "Start my Server" which should take to the menu of environments.

3 Pre-Lab Reading Quiz

Part of this lab is a pre-lab quiz. The pre-lab quiz is on Canvas. It is due **on Wednesday before Section A meets** (check Canvas for the time). It's not hard, but it does require you to read over the lab before class. If you are having trouble accessing it, make sure you are **logged into Canvas**.

3.1 How To Read the Lab For the Reading Quiz

The goal of reading the lab before starting on it is to make sure you have a preview of:

1. What's involved in the lab.
2. The key concepts of the lab.
3. What you can expect from the lab.
4. Any questions you might have.

These are the things we will ask about on the quiz. You *do not* need to study the lab in depth. You *do not* need to run the cells.

You should read these parts carefully:

- Paragraphs at the top of section/subsections
- The description of the programming assignment
- Any other large blocks of text
- The "About Labs in This Class" section (Lab 1 only)
- The programming assignment description.

You should skim these parts:

- The questions.

You can skip these parts:

- The "About Labs in This Class" section (Labs other than Lab 1)
- Commentary on the output of code cells (which is most of the lab)
- Parts of the lab that refer to things you can't see (like cell output)
- Solution to completeness questions.

3.2 Taking the Quiz

You can find it here: https://canvas.ucsd.edu/courses/40763/quizzes (https://canvas.ucsd.edu/courses/40763/quizzes)

The quiz is "open lab" -- you can search, re-read, etc. the lab.

You can take the quiz 3 times. Highest score counts.

4 Browser Compatibility

We are still working out some bugs in some browsers. Here's the current status:

1. Chrome -- well tested. Preferred option. **Required for Moneta**
2. Firefox -- seems ok, but not thoroughly tested.
3. Edge -- seems ok, but not thoroughly tested.
4. Safari -- not supported at the moment.
5. Internet Explorer -- not supported at the moment.

At the moment, the authentication step must be done in Chrome. You usually *will not* have to re-authenticate between labs, so if things work OK for the first, things will probably work here.

5 About Labs In This Class

This section is the same in all the labs. It's repeated here for your reference.

Labs are a way to **learn by doing**. This means you *must* **do**. I have built these labs as Jupyter notebooks so that the "doing" is as easy and seamless as possible.

In this lab, what you'll do is answer questions about how a program will run and then compare what really happened to your predictions. Engaging with this process is how you'll learn. The questions that the lab asks are there for several purposes:

1. To draw your attention to specific aspects of an experiment or of some results.
2. To push you to engage with the material more deeply by thinking about it.
3. To make you commit to a prediction so you can wonder why your prediction was wrong or be proud that you got it right.
4. To provide some practice with skills/concepts you're learning in this course.
5. To test your knowledge about what you've learned.

The questions are graded in one of three ways:

1. "Correctness" questions require you to answer the question and get the correct answer to get full credit.
2. "Completeness" questions require you to answer (even if incorrectly) all parts of the question to get credit.
3. "Optional" questions are...optional. They are there if you want to go further with the material.

Some of the "Completeness" problems include a solution that will be hidden until you click "Show Solution". To get the most from them, try them on your own first.

Many of the "Completeness" questions ask you to make predictions about the outcome of an experiment and write down those predictions. To maximize your learning, think carefully about your prediction and commit to it. **You will never be penalized for making an incorrect prediction.**

You are free to discuss "Completeness" and "Optional" questions with your classmates. You must complete "Correctness" questions on your own.

If you have questions about any kind of question, please ask during office hours or during class.

5.1 How To Succeed On the Labs

Here are some simple tips that will help you do well on this lab:

1. Read/skim through the entire lab *before* class. If something confuses you, you can ask about it.
2. Start early. Getting answers on edstem/piazza can take time. So think through the lab questions (and your questions about them) carefully.
   A. Go through the lab once (several days before the deadline), do the parts that are easy/make sense
   B. Ask questions/think about the rest
   C. Come back and do the rest.
3. Start early. The DSMLP cluster gets busy and slow near deadlines. "The cluster was slow the night of the deadline" is not an excuse for not getting the lab done and it is not justification for asking for an extension.
4. Follow the guidelines below for asking answerable questions on edstem/piazza.

You may think to yourself: "If I start early enough to account for all that, I'd have to start right after the lab was assigned!" Good thought!

The Cluster Will Get Slow DSMLP and our cloud machines will get crowded and slow *before every deadline*. This is completely predictable. DSMLP can also get crowded due to deadlines in other courses. You need to start early so you can avoid/work around these slowdowns. Unless there's some kind of complete outage, we will not grant extensions because the servers are crowded.

5.2 Getting Help

You might run into trouble while doing this lab. Here's how to get help:

1. Re-read the instructions and make sure you've followed them.
2. Try saving and reloading the notebook.
3. If it says you are not authenticated, go to the the login section of the lab and (re)authenticate.

4. If you get a `FileNotFoundError` make sure you've run all the code cells above your current point in the lab.
5. If you get an exception or stack dump, check that you didn't accidentally modify the contents of one of the python cells.
6. If all else fails, post a question to edstem/piazza.

5.3 Posting Answerable Questions on Edstem/Piazza

If you want useful answers on edstem/piazza, you need to provide information that is specific enough for us to provide a useful answer. Here's what we need:

1. Which part of which lab are you working on (use the section numbers)?
2. Which problem (copy and paste the *text* of the question along with the number).

If it's question about instructions:

1. Try to be as specific as you can about what is confusing or what you don't understand (e.g., "I'm not sure if I should do *X* or *Y*.")

If it's a question about an error while running code, then we need:

1. If you've committed anything, your github repo url.
2. If you've submitted a job with `cse142` you *must* provide the job id. It looks like this: `544e0cf2-4771-43c3-86f8-1c30d7af601f`. With the id, we can figure out just about anything about your job. Without it, we know nothing.
3. The *entire* output you received. There's no limit on how long an edstem/piazza post can be. Give us all the information, not just the last few lines. We like to scroll!

For all of the above **paste the text** into the edstem/piazza question. Please **do not provide screen captures**. The course staff refuses to type in job ids found in screen shots.

We Can't Answer Unanswerable Questions If you don't follow these guidelines (especially about the github repo and the job id), we will probably not be able to answer your question on edstem/piazza. We will archive it and ask you to re-post your question with the information we need.

5.4 Keeping Your Lab Up-to-Date

Occasionally, there will be changes made to the base repository after the assignment is released. This may include bug fixes and updates to this document. We'll post on piazza/edstem when an update is available.

In those cases, you can use `./pull-updates` to pull the changes from upstream and merge them into your code. You'll need to do this at a shell. It won't work properly in the notebook. Save your notebook in the browser first.

Then, change to your lab directory and do

```
./pull-updates
```

Then, reload this page in your browser.

5.5 Writing Code Outside Jupyter Notebook

The code for some programming assignments could get pretty long. If you'd like, you can develop outside of Jupyter Notebook.

You can do this by removing the call to `code()` and replacing it with a file name. Then `build()` will use the source code in the file.

Don't overwrite your code: `code()` does some checks to try to avoid overwriting your code and will throw an exception if it found modifications to files it wrote earlier. This seems to work pretty well, but I wouldn't trust it, so commit often.

5.6 Using VSCode

You can also develop remotely using Microsoft VSCode. You can find instructions from campus about how to do this on Datahub under "Visual Studio (VS) Code" at this link: https://support.ucsd.edu/services?id=kb_article_view&sysparm_article=KB0032269&sys_kb_id=01322d481b5ed514d1b0a935604bcb7a (https://support.ucsd.edu/services?id=kb_article_view&sysparm_article=KB0032269&sys_kb_id=01322d481b5ed514d1b0a935604bcb7a

The TAs report that this works fine.

A few things to note:

1. That pages lists several ways of starting docker containers on the campus servers. The configuration for this class is a little unusual, and none of the other methods listed on that page have been tested for this class. I suspect they don't work, and we won't be fixing them.
2. You'll need to be on campus or on the campus VPN.
3. Using VSCode is not officially supported in this class. If it doesn't work for you, the TAs may be willing help you and you might have luck submitting a ticket to campus, but if you can't get it to wok, you'll need to fall back on working through Jupyter Notebook.

5.7 How To Use This Document

You will use Jupyter Notebook to complete this lab. You should be able to do much of this lab without leaving Jupyter Notebook. The main exception will be some parts of the some of the programming assignments. The instructions will make it clear when you should use the terminal.

5.7.1 Logging In

If you haven't already, you can go to the login section of the lab and follow the instructions to login into the course infrastructure.

5.7.2 Running Code

Jupyter Notebooks are made up of "cells". Some have Markdown-formatted text in them (like this one). Some have Python code (like the one below).

For code cells, you press `shift-return` to execute the code. Try it below:

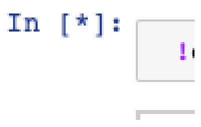
```
In [ ]: print("I'm in python")
```

Code cells can also execute shell commands using the `!` operator. Try it below:

```
In [ ]: !echo "I'm in a shell"
```

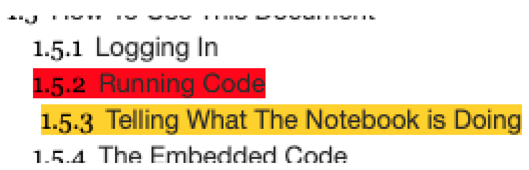
5.7.3 Telling What The Notebook is Doing

The notebook will only run one cell at a time, so if you press `shift-return` several times, the cells will wait for one another. You can tell that a cell is waiting if it there's a `*` in the `[ ]` to the left the cell:



```
In [*]: !
```

You'll can also tell *where* the notebook is executing by looking at the table of contents on the left. The section with the currently-executing cell will be red:

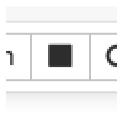


1.5.1 Logging In
1.5.2 Running Code
1.5.3 Telling What The Notebook is Doing
1.5.4 The Embedded Code

5.7.4 What to Do If Jupyter Notebook It Gets Stuck

First, check if it's actually stuck: Some of the cells take a while, but they will usually provide some visual sign of progress. If *nothing* is happening for more than 10 seconds, it's probably stuck.

To get it unstuck, you stop execution of the current cell with the "interrupt button":



You can also restart the underlying python instance (i.e., the confusingly-named "kernel" which is not the same thing as the operating system kernel) with the restart button:

Once you do this, all the variables defined by earlier cells are gone, so you may get some errors. You may need to re-run the cells in the current section to get things to work again.

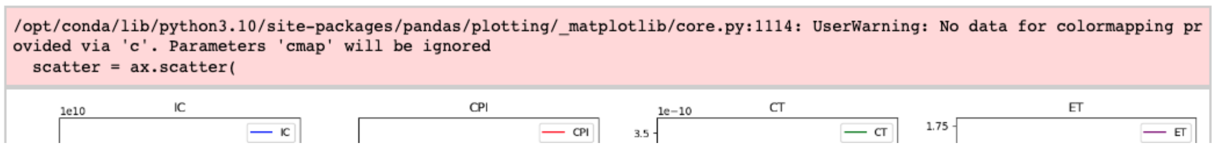
You can also try reloading the web page. That will leave Python kernel intact, but it can help with some problems.

5.7.5 Common Errors and Non-Errors

1. If you get `sh: 0: getcwd() failed: no such file or directory`, restart the kernel.
2. If you get `INFO:MainThread:numexpr.utils:Note: NumExpr detected 40 cores but "NUMEXPR_MAX_THREADS" not set, so enforcing safe limit of 8.`. It's not a real error. Ignore it.
3. If you get a prompt asking `Do you want to cancel them and run this job?` but you can't reply because you can't type into an output cell in Jupyter notebook, replace `cse142 job run` with `cse142 job run --force`. (see useful tip below.)
4. If you get an `Error: Your request failed on the server: 500 Server Error: Internal Server Error for url=http://cse142l-dev.wl.r.appspot.com/file`, trying running the job again.
5. Sometimes `cse142 job run` will just sit there and seemingly do nothing. Weirdly, interrupting the kernel (button above) seems to jolt it awake and cause it to continue.
6. These errors while display CFGs are harmless:

```
Cannot determine entrypoint, using 0x00002560.
Warning: run r2 with -e bin.cache=true to fix relocations in disassembly
Cannot determine entrypoint, using 0x00001140.
Warning: run r2 with -e bin.cache=true to fix relocations in disassembly
```

7. Warnings like this in pink about deprecated or ignored arguments are harmless:



```
/opt/conda/lib/python3.10/site-packages/pandas/plotting/_matplotlib/core.py:1114: UserWarning: No data for colormapping pr
ovided via 'c'. Parameters 'cmap' will be ignored
  scatter = ax.scatter(
```

7. If you get `http.cookiejar.LoadError: '/home/youruserrname/.djr-cookies.txt does not look like like a Netscape format cookies file.` remove the file and re-authenticate.
8. Problem with git clone...

5.7.6 Useful Tips

1. If you need to edit a cell, but you can't you can unlock it by pressing this button in the tool bar (although you probably shouldn't do this because it might make the lab work incorrectly. A

better choice is to copy and paste the cell, *and then* unlock the copy):

5.7.7 The Embedded Code

The code embedded in the lab falls into two categories:

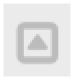
1. Code you need to edit and understand.
2. Code that you do not need to edit or understand -- it's just there to display something for you.

For code in the first category, the lab will make it clear that you need to study, modify, and/or run the code. If we don't explicitly ask you to do something, you don't need to.

Most of the code in the second category is for drawing graphs. You can just run it with shift-return to the see the results. If you are curious, it's mostly written with `Pandas` and `matplotlib` . These cells should be un-editable. However, if you want to experiment with them, you can copy *the contents* of the cell into a new cell and do whatever you want (If you copy the cell, the copy will also be uneditable).

Most Cells are Immutable Many of the cells of this notebook are uneditable. The only ones you should edit are some of the code cells and the text cells with questions in them.

Pro Tip The "carrot" icon in the lower right (shown below) will open a scratch pad area. It can be a useful place to do math (or whatever else you want).



5.7.8 Showing Your Work

Several questions ask you to show your work for calculations. We don't need anything fancy. Many of the questions ask you to compute something based on results of an experiment. Your experimental results will be different than others', so your answer will be different as well.

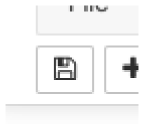
To make it possible to grade your work (and give you partial credit), we need to know where your answer came from. This is why you need to show your work. For instance this would be fine as answer to "On average, how many weeks do you have per lab?":

```
Weeks in quarter/# of labs = 10/5 = 2 weeks/lab
```

2 significant figures is sufficient in all cases, but you can include more, if you want.

If you are feeling fancy, you can use LaTex, but it's not at all required.

When it's appropriate, you can also paste in images. However, Jupyter Notebook is flaky about it. Saving your notebook frequently by clicking the disk icon seems to help:



5.7.9 Answering Questions

Throughout this document, you'll see some questions (like the one below). You can double click on them to edit them and fill in your answer. Try not to mess up the formatting (so it's easy for us to grade), but at least make sure your answer shows up clearly. When you are done editing, you can `shift-return` to make it pretty again.

A few tips, pointers, and caveats for answering questions:

1. The answers are all in github-flavored markdown (https://guides.github.com/features/mastering-markdown/) with some html sprinkled in. Leave the html alone.
2. Many answers require you to fill in a table, and many of the `|` characters will be missing. You'll need to add them back.
3. The HTML needs to start at the beginning of a line. If there are spaces before a tag, it won't render properly. If you accidentally add white space at the beginning of a line with an html tag on it, you'll need to fix it.
4. Text answers also need to start at the beginning of a line, otherwise they will be rendered as code.
5. Press `shift-return` or `option-return` to render the cell and make sure it looks good.
6. There needs to be a blank line between html tags and markdown. Otherwise, the markdown formatting will not appear correctly.

You'll notice that there are three kinds of questions: "Correctness", "Completeness", and "Optional". You need to provide an answer to the "Completeness" questions, but you won't be graded on its correctness. You'll need to answer "Correctness" questions correctly to get credit. The "Optional" questions are optional.

6 Logging In To the Course Tools

In the course you will use some specialized tools to let you perform detailed measurements of program behavior. To use them you need to login with your `@ucsd.edu` email address using the instructions below. **You need to use the email address that appears on the course roster. That's the email address we created an account for. In almost all cases, this is your `@ucsd.edu` email address.**

You'll probably only have to do this once this quarter, but if you get an error about not being authenticated, just re-authenticate. You can return to this notebook (or any other of the lab notebooks) to login at any time.

Here's what to do:

1. Enter your `@ucsd.edu` email address (without the '<>') in quotes after `login` below. It'll take a few seconds to load.
2. Click the google "G" login button below and login with your `@ucsd.edu` email address.
3. **Click the google button regardless of whether it says "sign in" or "signed in". Then be sure to select your `@ucsd.edu account` if it shows you multiple google acocunts**
4. You'll see a very long string numbers an letters appear above. Click "Copy it" to copy it.

Note: If it doesn't give you a choice about which account to log into and authentication fails, that means you are logged into a single Google account and that account is *not* your `@ucsd.edu` account. You'll have to log into your `@ucsd.edu` through Gmail or through Chrome's account manager and then try again.

Use Chrome The login process doesn't seem to work properly with Safari or Firefox. Use Chrome to login. You can use any of the other compatible browsers you want for the doing the rest of the lab, and it should be fine.

In [ ]:
```
login("<Your @ucsd.edu email address>")
```

Next step: Paste it below between the quote marks. Press `shift-return` .

In [ ]:
```
token("your_token")
```

It should have replied with

```
You are authenticated as <your email>
```

You are now logged in! Try submitting a job:

In [ ]:
```
!cse142 job run "echo Hello World"
```

If you see "Hello World", you're all set. Proceed with the lab!

Delete your token from the above cell (`token("...")`). Because your token is essentially your username and password combined, you should treat it like a password or ssh private key. **Sharing your token with another student or possessing another student's token is an AI violation.**

7 Grading

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

Part	value
Reading quiz	3%
Jupyter Notebook	70%
Programming Assignment	25%
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) 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.

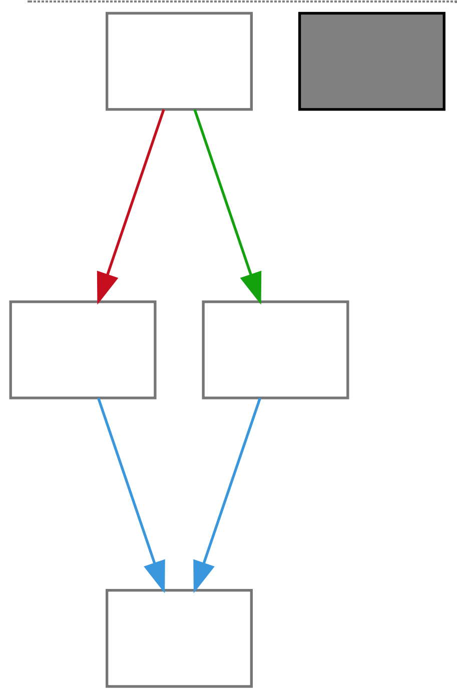
CFiddle can generate CFGs for most functions. Take a look at this code then run the cell to see it's CFG:

In [8]:
```
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"))

compare([if_ex[0].source("if_ex"),
         if_ex[0].cfg("if_ex", remove_assembly=True)])
```

```
100%                                    1/1 [00:00<00:00, 61.08it/s]
```

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

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



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 gray 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 here.
```

How many paths through the code are there?

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

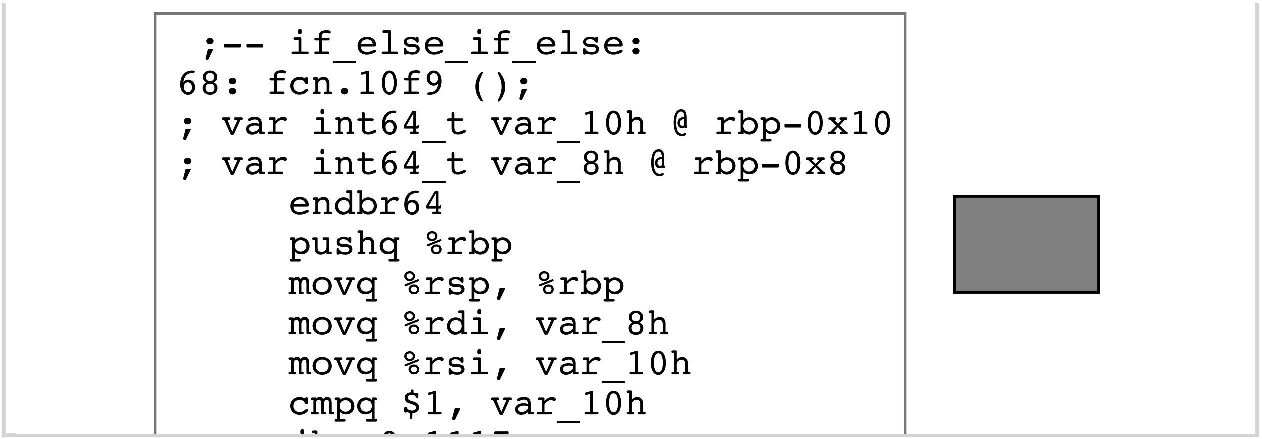
In [3]:

```
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;
    }
}
"""))[0].cfg("if_else_if_else")
```

```
100%                                                1/1 [00:00<00:00, 17.31it/s]
```

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

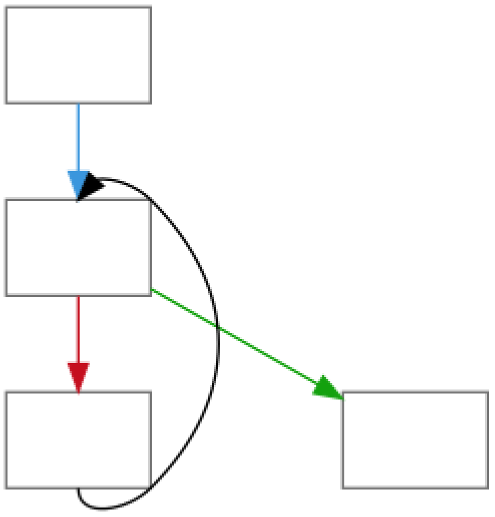
Out[3]:



```
 ;-- if_else_if_else:
68: fcn.10f9 ();
; var int64_t var_10h @ rbp-0x10
; var int64_t var_8h @ rbp-0x8
     endbr64
     pushq %rbp
     movq %rsp, %rbp
     movq %rdi, var_8h
     movq %rsi, var_10h
     cmpq $1, var_10h
```

Question 2 (Correctness - 2pts)

In the fiddle below, modify `foo()` so that its CFG looks like this (you can edit the code as many times as you'd like):



In [ ]:

```
build(code(r"""
extern "C"
int foo(int a) {
    return 0;
}
"""))[0].cfg("foo", remove_assembly=True)
```

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 [ ]:
```
loop_if = build(code(r"""
#include<cstdint>
#include<cstdlib>

extern "C"
uint64_t loop_if(uint64_t array, unsigned long int size) {
    uint64_t t= 0;
    int k = 0;
    for(uint i = 0; i < size; i++) {
        if (i-size != 0) {
            k = 4;    //  L1
        } else if (i+size != 0) { // L2
            k = 5;
        }
    }
    return t + k; // L3

}

"""))

compare([loop_if[0].source("loop_if"), loop_if[0].cfg("loop_if", number_n
```

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 here.
```

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

In [ ]:

```
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 fiddle 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 [ ]:

```
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 fiddle 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

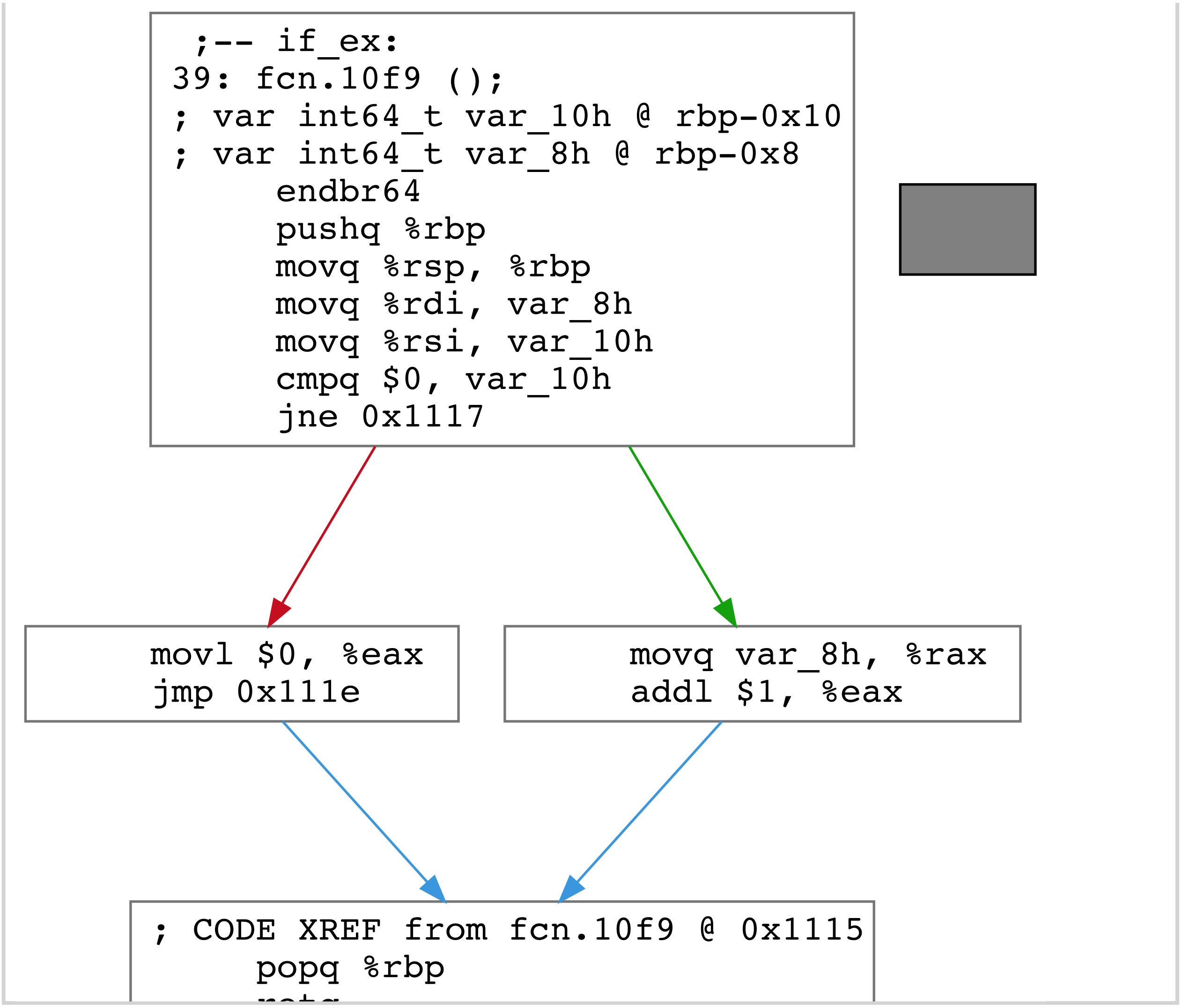
We'll start simple by revisiting the `if_ex()` function we saw earlier (run the cell):

In [4]:
```
compare([if_ex[0].source("if_ex"), if_ex[0].asm("if_ex")])
display(if_ex[0].cfg("if_ex"))
```

```
int if_ex(uint64_t array, unsigned
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
```

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



```
 ;-- if_ex:
39: fcn.10f9 ();
; var int64_t var_10h @ rbp-0x10
; var int64_t var_8h @ rbp-0x8
     endbr64
     pushq %rbp
     movq %rsp, %rbp
     movq %rdi, var_8h
     movq %rsi, var_10h
     cmpq $0, var_10h
     jne 0x1117
```

```
     movl $0, %eax
     jmp 0x111e
```

```
     movq var_8h, %rax
     addl $1, %eax
```

```
; CODE XREF from fcn.10f9 @ 0x1115
     popq %rbp
```

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.

Why does the CFG contain pseudo-assembly instead of the actual assembly? -- Because that's what the CFG drawing tool provides. (As an aside, the tool we use to draw the CFGs is called Redare2 (https://rada.re/n/) and it's a really powerful reverse engineering tool. It's really cool, but it has a really terrible interface).

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 [ ]:

```
if_ex = build("./if_ex.cpp", build_parameters=arg_map(OPTIMIZE=["-O0", "-(
compare([x.cfg("if_ex") for x in  if_ex], [f"OPTIMIZE = {x.get_build_paran
```

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:
    addl    $1, -4(%rbp)
.L2:
    movl    -4(%rbp), %eax
    cmpl    -20(%rbp), %eax
    jl    .L5
    movl    -8(%rbp), %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

9.3.2 A More Complex Example

Here's the assembly for `loop_if()`. Run the cell. (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.)

In [ ]:

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

extern "C"
uint64_t loop_if(uint64_t array, unsigned long int size) {
    uint64_t t= 0;
    int k = 0;
    for(uint i = 0; i < size; i++) {
        if (i + size != 0) { //  L1
            k = 4;
        } else if (i+size == 0) { // L2
            k = 5;
        }
    }
    return t + k; // L3

}

"""), build_parameters=arg_map(OPTIMIZE="-g0"))
analyze(loop_if[0], "loop_if")
```

Question 10 (Correctness - 2pts)

What instruction implements these parts of the `loop_if()` (just copy and paste it from the assembly and don't be afraid to google. Include the registers)?

	Instruction
The add on line `L1`	
The comparison to zero on line `L2`	

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 by looking at the assembly.

In [ ]:
```
loop_func = build(code(r"""
#include<cstdint>
#include<cstdlib>

extern "C"
uint64_t *loop_func(uint64_t  *array, unsigned long int size) {
    uint64_t s = 0;
    for(uint i = 0; i < 10; i++) {
        s += array[i];
    }
    return array + s;
}

"""), arg_map(OPTIMIZE="-g0"))
analyze(loop_func[0], "loop_func")
```

We are primarily interested in two things:

1. How many instructions execute *in total* (that's just `IC` from the performance equation).
2. How many instructions access memory.

Counting instructions is simple: You can count up the number of instructions in each basic block and multiply it by the number of times the basic block executes.

Determining which instructions access memory is also pretty simple except for one wrinkle.

The basic rules are that:

1. In x86 assembly, if the instruction uses an addressing mode with parentheses (e.g., `-8(%rbp)` or `(%rax)`) then it accesses memory.
2. In pseudo-assembly, operands like `[var_8h]` or `[rax*8]` are *also* memory accesses.
3. `push` and `pop` instructions are memory accesses.

The exception is the `lea` family of instructions (`leaq`, `leal`, etc.). These are "load effective address" and they just compute the address and store the *address* into the destination register. So when you count memory accesses you should ignore `lea*` instructions.

Simple, right! :-)

For this question, recall that "dynamic instructions" is the number of instructions that the processor executes and "static instructions" is the number of instructions the compiler generates.

Question 11 (Completeness)

Fill in the table below to compute how many total instructions and how many memory accesses occur when `loop_func()` executes. You only need to fill out the last two columns on the bottom row. (Hint: It is not necessary to trace through exactly what each instruction does. Instead

think about how control will flow through the CFG and just count how many times each block must execute. Also don't be afraid to google the instruction names).

basic block	# of times the block executes	static instruction count	static memory instruction count	dynamic instruction count	dynamic memory instruction count
n0					
n1					
n2					
n3					
			Total		

10 The Perils of C++

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!

The solution that C++ compilers have adopted is called *name mangling*. Name mangling is a deterministic, standardized way to convert *any* function type into a unique symbol.

Let's see what it does. Run the cell and answer the question:

In [ ]:

```
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 12 (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 [ ]:

```
!echo _Z3foof | c++filt
!echo _Z3fooRKSt3mapINSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
```

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 fiddle 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 use it in the examples, because it makes it easier to refer to the functions.

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 register rather than on the stack. This saves `mov` instructions and memory accesses. You can see it in action below:

In [5]:

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

```
100%                                     2/2 [00:00<00:00, 28.25it/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 lots of parentheses (on instructions other than `lea`) it's probably not optimized.

Question 13 (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)?

You can also encourage the compiler to put a variable into a register with the `register` keyword. This is not a good practice in real code since modern compilers do register assignment automatically. We'll use this trick in some of our examples to highlight the impact of individual optimizations.

Question 14 (Completeness)

Go ahead and try it on the fiddle above by writing `register int a` in the argument list for `foo()`. What changed?

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 [ ]:
```
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_build_pa
```

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 [ ]:
```
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.get_build_
```

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 [ ]:
```
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 effective address. Using `leal` in this way is a very common idiom in x86 assembly, because most x86 instructions overwrite one argument. `lea` , however, does not.

Question 15 (Completeness)

Play around with the fiddle 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 [ ]:
```
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 [ ]:
```

optimization_example(code(r"""
extern "C" int foo(register unsigned int a, register unsigned int b){
    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.

Question 16 (Completeness)

Modify the fiddle above so that the `0` in the `leal` s first argument becomes a 4.

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

In [ ]:
```
optimization_example(code("""
extern "C" int foo(register unsigned int a, register unsigned int b){
    return a *8;
}
"""), function="foo")
```

Question 17 (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 [ ]:
```
optimization_example(code("""

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

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

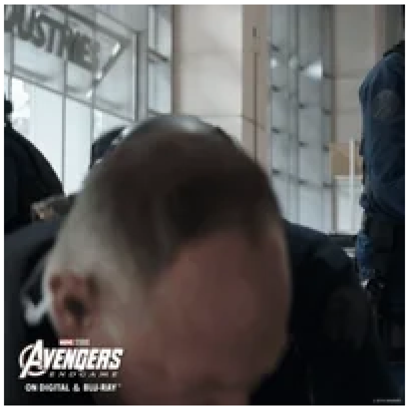
Again, the compiler is doing multiple things at once, but the constant propagation is visible: In the unoptimized code, it moves `$4` into both `%r12d` and `%ebx` and then adds both those register to `%eax` on the next two lines. In the optimized code it's folded both `4` s into the `8` in the `leal` 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 [ ]:
```

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 18 (Optional)

Play around with the fiddle 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 constructs 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 [ ]:

```
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*8; i++) {
        s+= array[i];
    }
    return s;
}
""", file_name="foo.cpp"), function="foo", OPTIMIZE= ["-O0", "-Og -funrol
```

On the left, the loop body is unoptimized blocks `n1` and `n2` . They get merged and replicated into optimized blocks `n1` and `n2` (on the right).

Question 19 (Correctness - 3pts)

If `b` is 16 and CPI 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)? Assume the branch in `n0` of the optimized code is taken.

The example above is simplified since the loop bound is `8*b`. From experience, I know that gcc likes to unroll loops 8 times, so this makes the number of iterations work out nicely. Replace the loop bound with `b` and try running it again.

Question 20 (Completeness)

After you replaced `8*b` with `b`, why did the compiler add the additional basic blocks that appeared in the optimized CFG? For an (optional) challenge, explain how they work.

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

In [ ]:

```
optimization_example(code(r"""

extern "C" int foo(register unsigned int a, register unsigned int b, unsi
    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", "-O1 -funroll-loops"])
```

Question 21 (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 preventing the compiler from unrolling the loop? Can it it unroll loops that use the `break` keyword? `continue`? `if-else`? Non-constant stride?

11.7 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 [ ]:
```
optimization_example(code(r"""
#include<cstdint>
#include<cstdlib>
extern "C" uint32_t div_loop(uint64_t * array,  unsigned long int size) {
#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-inline"])
```

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 22 (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

11.8 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.

11.8.1 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 [ ]:
```
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)](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`, 1x `endbr64`, 1 x `ret`, and 1x `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/A has a little more detail.

If you're curious, use the fiddle 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 23 (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?

11.8.2 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 [ ]:

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

int k = 0;
extern "C"
int inline __attribute__ ((used)) foo( register int a, register int b) {
    if (a + k)
        return b+k;
    else
        return 2 *k;
}

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", "-O2 -g0"]))

compare([overhead[0].source("foo"), overhead[0].cfg("foo")], ["foo()", "N
compare([overhead[0].source("loop"), overhead[0].cfg("loop")], ["loop()",
compare([overhead[1].cfg("loop")], ["loop() with inlining"])
```

When the compiler is finished with `loop()` it contains all the code that was in `foo()`. It also no longer contains a function call at all.

Question 24 (Completeness)

Based on the change in instruction count (IC), how much speedup does inlining provide in this case?

Show Solution

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 [ ]:

```
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 sum = 0;
    for(int i = 0; i < 10; i++) {
        sum += the_loop(20);
    }
    return sum;
}

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

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

Bye bye, function call! Bye Bye, loops!

Question 25 (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 [ ]:
```
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_success=True)

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

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

sort = build("./stl_sort.cpp", build_parameters=arg_map(OPTIMIZE=["-O0 -g

compare([sort[0].cfg("stl_sort"), sort[1].cfg("stl_sort")], ["Without inl
```

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.

There's a large impact on performance as well. The cell below runs a very similar code to the fiddle above, and this cell takes a while...:

In [ ]:
```
sort = build("./stl_sort.cpp",
             build_parameters=arg_map(OPTIMIZE=["-O0", "-Og", "-O3"], DEBU

sort_run = run(sort, function="sort_harness", arguments=arg_map(size=1000
```

In [ ]:
```
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 26 (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. 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. When a member function is called from a variable of the parent class, C++ determines at run time whether to call the parent version or the child version of the function

In the fiddle 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.)

Run the cell, and we'll look at the assembly.

In [ ]:

```
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=["-Og -fno-
```

In [ ]:

```
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, but this time the call is in `n3` . 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 left 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 [ ]:
```
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 [ ]:
```
virtual_data = run(virt, function=["static_call", "virtual_call"],
                   arguments=arg_map(size=100000000)).as_df()
virtual_data["experiment"] = virtual_data["function"] + "-" + virtual_data
virtual_df = PE_calc(virtual_data)
display(virtual_df)
plotPEBar(df=virtual_df, what=[("experiment", "IC")])
```

Question 27 (Correctness - 2pts)

How much speedup does the static call to `foo()` provide over the virtual call to `foo()` "unoptimized" case? How about in the optimized case?

Speedup in unoptimized case:

Speedup in the optimized case:

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 28 (Optional)

How much speedup do optimizations provide for the `virtual_call()` ? What did the compiler do to achieve this?

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 [ ]:
```
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")], ["values()",
```

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

Question 29 (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 [ ]:
```
noalias = build(code(r"""
#include"cfiddle.hpp"

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

    *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 [ ]:
```
side_effect = build(code(r"""
#include"cfiddle.hpp"

extern "C" void something() {
}

extern "C" void pointers(int * __restrict__ c, int *__restrict__ a, int *
    *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=["-O3"],
compare([side_effect[0].asm("pointers")], [html_parameters(side_effect[0]
```

Question 30 (Completeness)

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

Show Solution

There we go! The compiler can optimize again.

However, inlining can only do so much. If `something()` were defined in another file or, even worse, in a library. The compiler would have to assume that `something()` had side effects.

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)](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 `stl_sort()` 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 [ ]:
```
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

As I mentioned early in the class (and as you will need for the programming assignment), one goal of this class is to give you better intuition about what the compiler does to your code. To help with that, let's take a look at common C++ constructs to see what kind of assembly they generate with and without optimizations. You'll probably want to read this section carefully while working on the programming assignment.

15.1 Trivial Function

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

extern "C" void foo() {
}
"""), arg_map(OPTIMIZE=["-O0", "-O3"], DEBUG_FLAGS="-g0 -Wno-unused"))
compare([x.asm("foo") for x in foo], [f"OPTIMIZE = {x.get_build_parameter
```

15.2 Local Variable Access

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

extern "C" void foo() {
        int i = 0;
        i = 4;
}

extern "C" void bar() {
        register int i = 0;
        i = 4;
}

"""), arg_map(OPTIMIZE=["-O0", "-O3"], DEBUG_FLAGS="-g0 -Wno-unused"))
```

Without optimizations, every non- `register` access is a load or a store: The `movl` instructions store 0 and the 4 to `i`. The optimizer realizes the whole thing is pointless and just returns.

In [ ]:
```
compare([x.asm("foo") for x in foo], [f"OPTIMIZE = {x.get_build_parameters
```

With `register` the loads and stores for `i` go away, but there are hidden memory operations in `pushq` and `popq`. The optimizer just returns.

In [ ]:
```
compare([x.asm("bar") for x in foo], [f"OPTIMIZE = {x.get_build_parameters
```

15.3 Array Access

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

extern "C" uint foo(register  uint * array) {
    return array[4];
}

"""), arg_map(OPTIMIZE=["-O0", "-O3"], DEBUG_FLAGS="-g0 -Wno-unused"))
```

Note: A reminder than in x86 the first 6 function arguments are passed in %rdi, %rsi, %rdx, %rcx, %r8, and %r9.

To access an array, we need to compute the address of the element. In general, it looks like `array_address + sizeof(element) * index`. In this case, that works out to `4*4 = 16` and you can see that constant 16 shows in the `movl`. The addressing mode it uses (`n(%rax)`) adds `n` to `%rax` and uses that as the effective address.

Without `register` on the declaration of `array`, the compiler emits several more instructions. Try modifying the above cell and see if you can figure out what it's doing.

With optimizations, there's a single load instruction.

In [ ]:
```
compare([x.asm("foo") for x in foo], [f"OPTIMIZE = {x.get_build_parameters
```

15.4 Complex Array Access

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

extern "C" uint foo(register  uint * array, register uint64_t i, register
    return array[i * 5 + k];
}

"""), arg_map(OPTIMIZE=["-O0", "-O3"], DEBUG_FLAGS="-g0 -Wno-unused"))
```

Here the address calculation is more complicated. Without optimizations, the compiler dutifully implements the calculations. The most interesting part is

```
movq    %rcx, %rax
salq    $2, %rax
addq    %rcx, %rax
```

Which multiplies `%rcx` by 5 (how?).

With optimizations, the compiler relies on the `leaq` with a complex addressing mode to do most of the work. The fully general version is:

```
k(%r1,%r2,l) = %r1 + l*%r2 + k
```

Where `l` must be a power of two.

So the first `leaq` does `i * 4 + i` to get `5*i`. The `addq` does the `+k`, and then the `movl`, takes the result which is in `%rax`, multiplies it by 4 (which is `sizeof(uint)`) and adds it to `array` which is in `%rdi`. It use that as the effective address and loads the value into `%eax`.

In [ ]:
```
compare([x.asm("foo") for x in foo], [f"OPTIMIZE = {x.get_build_parameters
```

15.5 `for` Loops

In [ ]:

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

extern "C" uint foo( uint64_t i,  uint64_t k) {
uint64_t sum = 0;
    for (uint64_t a = 0; a < i; a++) {
        sum += a;
    }
    return sum;
}

"""), arg_map(OPTIMIZE=["-O0", "-O3"], DEBUG_FLAGS="-g0 -Wno-unused"))
```

Note: We are shifting to looking at CFGs as the code constructs get more complex. Remember that our CFG generate pseudo-assembly rather than real assembly. In particular, each thing that looks like `var_20h` represents a load or store.

Here, the unoptimized code pushes several things onto the stack (`pushq`) and initializes `sum` and `a` with `movl` . The loop body is just two `addq` to increment `a` and add it to `sum` .

The optimizer can't actually help much here aside from removing the `pushq` s and `popq` s.

In [ ]:

```
compare([x.cfg("foo") for x in foo], [f"OPTIMIZE = {x.get_build_parameter
```

15.6 Array Access in a Loop

In [ ]:

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

extern "C" uint foo(register  uint * array, register uint64_t i, register
    register int sum = 0;
    for (register uint64_t a = 0; a < i; a++) {
        sum += array[i * 5 + a];
    }
    return sum;
}

"""), arg_map(OPTIMIZE=["-O0", "-O2"], DEBUG_FLAGS="-Wno-unused"))
```

Without optimization, the address calculation occurs with each loop iteration and (like above) it doesn't use `leal` . It also increments `a` (which lives in `%rbx`) using the last `addq` .

With optimizations, the `i*5` (which is loop-invariant), is moved above the loop (can you spot it? Remember that the x86 calling convention puts `i` is `%rsi`).

In [ ]:
```
compare([x.cfg("foo") for x in foo], [f"OPTIMIZE = {x.get_build_parameter
#compare([x.asm("foo") for x in foo], [f"OPTIMIZE = {x.get_build_paramete
#compare([x.cfg("bar") for x in foo], [f"OPTIMIZE = {x.get_build_paramete
```

Another trick it uses is to get rid of `a` altogether by rewriting the loop bounds check to operate on the address we are accessing rather than an index variable.

To see this:

```
; initially array is in `rdi` and i is in rsi
foo:
.LFB39:
    .cfi_startproc
    endbr64
    testq    %rsi, %rsi
    je    .L4
    leaq    (%rsi,%rsi,2), %rdx ; rdx = i + 2 *i = 3 * i
    leaq    (%rsi,%rsi,4), %rax ; rax = i + 4 *i = 5 * i -- lets ca
ll %eax "address"
    leaq    (%rdi,%rdx,8), %rcx ; rcx = array + (3*i * 8) <-- This
 is one past the last last element of the array the loop will acces
s.  We'll call this 'last_address'.
    leaq    (%rdi,%rax,4), %rax ; rax = array + 5*i * 4 <-- This is
the first element of the array the loop will access
                                ;  The steps above look non-intutiv
ie, but 3*i*8 - 5*i*4 = 4*i, which is the number of bytes between t
he first last and last access.
    xorl    %edx, %edx          ; sum = 0
    .p2align 4,,10
    .p2align 3
.L3:
    addl    (%rax), %edx        ; sum += *address
    addq    $4, %rax            ; address += 4
    movl    %edx, %r8d          ; I'm not sure why it does this.
    cmpq    %rax, %rcx          ; does address == last_address?
    jne    .L3                     ; if not, go again
    movl    %r8d, %eax
    ret
    .p2align 4,,10
    .p2align 3
.L4:
    xorl    %r8d, %r8d
    movl    %r8d, %eax
    ret
    .cfi_endproc
```

We can try to mimic the optimizer and do this by hand:

In [ ]:

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

extern "C" uint foo(register  uint * array, register uint64_t i, register
    register int sum = 0;
    register uint *end = &array[6*i];
    for (register uint* a = &array[5*i]; a < end; a++) {
        sum += *a;
    }
    return sum;
}

"""), arg_map(OPTIMIZE=["-O0", "-O2"], DEBUG_FLAGS="-Wno-unused"))

compare([x.cfg("foo") for x in foo], [f"OPTIMIZE = {x.get_build_parameters
```

That worked pretty well!: The first basic block is much larger but the loop body is much more compact.

Before we get too pleased with ourselves, though, you should note that this is just with `-O2` . Change the optimization flag to `-O3` and see what it does.

15.7 Calls to Small Functions

In [ ]:

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

extern "C" uint64_t f(register uint64_t a, register uint64_t b) {
    return 3*a + 5*b;
}

extern "C" void foo(register uint64_t * array, register uint64_t  i, regis
    array[0] = f(i,k);
    array[1] = f(i,i);
    array[2] = f(i,7);
    array[3] = f(11,13);
}

"""), arg_map(OPTIMIZE=["-O0", "-O2 -fno-semantic-interposition"], DEBUG_I
```

In [ ]:

```
compare([x.asm("foo") for x in foo], [f"OPTIMIZE = {x.get_build_parameters
compare([x.asm("f") for x in foo], [f"OPTIMIZE = {x.get_build_parameters(
```

The unoptimized code simply calls `f()` four times with the arguments passed to it, and that takes quite a few instructions, especially when you add in the many instructions the unoptimized version of `f()` uses to perform an two multiplies and an add.

With optimizations, the compiler generates three different implementations that take advantage of the specific values passed to `f()` , but it complicates things a little by reordering the instructions. Below, I've re-arranged the instructions to make it clearer

```
; f(i,k)
leaq    (%rdx,%rdx,4), %rdx ; k = k * 5
addq    %rax, %rdx          ; k += i*3
movq    %rdx, (%rdi)        ; array[0] = k*5 + i*3

; f(i,i)
salq    $3, %rsi            ; i *= 8
movq    %rsi, 8(%rdi)       ; array[1] = i*8

; f(i,7)
leaq    (%rsi,%rsi,2), %rax ; %rax  = i * 3
addq    $35, %rax           ; rax += 35
movq    %rax, 16(%rdi)      ; array[2] = i*3 + 35

; f(11,13)
movq    $98, 24(%rdi)       ; array[3] = 98
```

Each version is simpler than the last, and the `f(i,i)` and `f(11,13)` are both shorter than the fully general version.

Strange optimization options : You might notice the `-fno-semantic-interposition` optimization flag. It's a good a example of the ~300 optimization flags that gcc provides. It's there to ensure that the compiler inlines functions correctly. I need it because cfiddle compiles your code into a shared library (i.e., a `.so` file) that requires with `-fPIC` (which means compile as position-independent code, which is what's required for dynamic loading.) compiler option. How did I know to add this option? I looked at the assembly and saw that `f()` was not being inlined like I thought it should be. Then I did some experiments and eventually posted to [Stack Overflow (https://stackoverflow.com/questions/73900598/why-does-fpic-hinder-inlining)](https://stackoverflow.com/questions/73900598/why-does-fpic-hinder-inlining).

15.8 Other Features

Question 31 (Completeness)

In the two cells below, write code that illustrates an interesting construct in C or C++ other than the ones presented above. What's interesting about what the compiler generated?

Language feature in the first cell:

What's interesting about the assembly:

Language feature in the second cell:

What's interesting about the assembly:

In [ ]:

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

extern "C" void foo() {
    // your language feature here
}

"""), arg_map(OPTIMIZE=["-O0", "-O3 -fno-semantic-interposition"], DEBUG_I
compare([x.asm("foo") for x in foo], [f"OPTIMIZE = {x.get_build_parameters
```

In [ ]:

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

extern "C" void foo() {
    // your language feature here
}

"""), arg_map(OPTIMIZE=["-O0", "-O3 -fno-semantic-interposition"], DEBUG_I
compare([x.asm("foo") for x in foo], [f"OPTIMIZE = {x.get_build_parameters
```

16 Programming Assignment: Can You Beat the Compiler?

This Programming Assignment Is Much More Involved than Lab 1's: Budget your time accordingly.

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 [ ]:
```
render_code("matmul.hpp")
```

And the rest of the implementation:

In [ ]:
```
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 [ ]:

```
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 it.
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 all jus
// implementation above, but when you are done, they should be specialize
// are called by the go() function in the next cell.
extern "C" void matrix_product_1(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)));
            }
        }
    }
}

extern "C" void matrix_product_2(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)));
            }
        }
    }
}

extern "C" void matrix_product_3(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)));
            }
        }
    }
}

extern "C" void matrix_product_4(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)));
            }
        }
    }
}

extern "C" void matrix_product_5(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)));
            }
        }
    }
}
""", file_name="matmul_solution.cpp"), build_parameters=arg_map(MORE_SRC=
```

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 [6]:
```
build("matmul.cpp")[0].source(show=("START_TESTS", "END_TESTS"))
```

```
100%                                                    1/1 [00:00<00:00, 12.91it/s]
```

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.
- 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.
- Using better algorithms.

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. Repeat.

To be clear, you *are not* required to apply the optimizations in the order listed (but it's a good idea). It is also ok if the difference between two versions includes multiple optimizations. However, the more cleanly you can separate the effects of the optimizations you apply, the easier it will be to answer questions later in the lab.

16.2 Running the Code

You can compile and run the code like so:

In [ ]:
```
matmul = build("matmul_solution.cpp", build_parameters=arg_map(OPTIMIZE=[
matmul_run = run(matmul, function=[f"test_matrix_product_{k}" for k in ra
```

And look at the results like so:

In [ ]:
```
PE_calc(matmul_run.as_df())
```

And checkout the assembly

In [ ]:
```
matmul[0].asm("matrix_product_1")
```

Ideally, you'd also be able to look at the CFG. However, the CFG generator seems to fail on most versions of the matrix product. I'm not sure why. I would not rely on it for this PA.

In [ ]:
```
# matmul[0].cfg("matrix_product_1")  This probably won't work.
```

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`).

16.4 The Test Suite

The lab provides a comprehensive test suite for your implementation. The code in is `run_test.cpp` . It use GTest (https://github.com/google/googletest) which is a very powerful C++ testing tool from Google. Because of this, the tests run outside of cfiddle.

You can build the tests with:

In [ ]:
```
!make run_tests.exe
```

NOTE: When you build run_tests.exe, it'll use the version of `matmul_solution.cpp` it finds in your lab directory. The contents of this file will reflect the last time you executed the cell above with the `build()` command in it, so if you make changes there, you'll need to re-run the cell.

Then you can run the tests like this (they take a little while to run):

In [ ]:
```
!./run_tests.exe
```

The tests are in 5 groups, each of which tests one of the five functions you'll write.

16.4.1 Running the Test Suite

The test suite is meant to help keep you on the right track as you go through the assignment. When you make a change to your code, I would:

1. Run all the tests. If they all pass, great!
2. If some fail, run the tests for the `matrix_product_*()` function you're working on.
3. Once you find a particular test case that fails, you can run and debug just that test (see below).

When a test fails, `run_tests.exe` will print out a description of the test that failed:

```
[----------] Global test environment tear-down
[==========] 16 tests from 6 test suites ran. (15925 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 16 tests, listed below:
[  FAILED  ] MatmulTests_5.matrix_product_5
[  FAILED  ] MatmulTestsHidden.hidden
[  FAILED  ] matrix_product/MatmulTests_1.matrix_product/0, where G
etParam() = (1, 1, 1, 1)
[  FAILED  ] matrix_product/MatmulTests_1.matrix_product/1, where G
etParam() = (10, 20, 30, 10)
[  FAILED  ] matrix_product/MatmulTests_1.matrix_product/2, where G
etParam() = (32, 20, 10, 10)
[  FAILED  ] matrix_product/MatmulTests_1.matrix_product/3, where G
etParam() = (30, 30, 30, 10)
[  FAILED  ] matrix_product_2/MatmulTests_2.matrix_product_2/0, whe
re GetParam() = (1, 1)
[  FAILED  ] matrix_product_2/MatmulTests_2.matrix_product_2/1, whe
re GetParam() = (10, 10)
[  FAILED  ] matrix_product_2/MatmulTests_2.matrix_product_2/2, whe
re GetParam() = (32, 10)
[  FAILED  ] matrix_product_2/MatmulTests_2.matrix_product_2/3, whe
re GetParam() = (30, 10)
[  FAILED  ] matrix_product_3/MatmulTests_3.matrix_product_3/0, whe
re GetParam() = (1, 1)
[  FAILED  ] matrix_product_3/MatmulTests_3.matrix_product_3/1, whe
re GetParam() = (2, 11)
[  FAILED  ] matrix_product_3/MatmulTests_3.matrix_product_3/2, whe
re GetParam() = (4, 12)
[  FAILED  ] matrix_product_3/MatmulTests_3.matrix_product_3/3, whe
re GetParam() = (6, 13)
[  FAILED  ] matrix_product_4/MatmulTests_4.matrix_product_4/0, whe
re GetParam() = (1)
[  FAILED  ] matrix_product_4/MatmulTests_4.matrix_product_4/1, whe
re GetParam() = (2)

16 FAILED TESTS
```

Each test has a name (e.g., `matrix_product_3/MatmulTests_3.matrix_product_3/2`). To run the tests just for `matrix_product_1()` you can do:

In [ ]:
```
!./run_tests.exe --gtest_filter=*MatmulTests_1*
```

To run one test do something like:

In [ ]:
```
!./run_tests.exe --gtest_filter=matrix_product/MatmulTests_1.matrix_produ
```

You can also get a list of all the tests with

In [ ]:
```
!./run_tests.exe --gtest_list_tests
```

16.4.2 Debugging With the Test Suite

If you want to debug your test, open a shell and do

```
$ gdb run_tests.exe
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/license
s/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from run_tests.exe...
```

Then set a breakpoint on `matmul_product_1` (or whatever function you want to debug):

```
(gdb) b matmul_product_1
Breakpoint 1 at 0x2f922: file matmul_solution.cpp, line 9.
```

and then run the executable with `--gtest_filter` to run the test you want to debug (this output is from an earlier lab, but you get the idea):

```
(gdb) run --gtest_filter=simple_tests/NibbleTestFixtureSmall.simple
_tests/0
Starting program: /cse142L/labs/CSE141pp-Lab-The-PE/run_tests.exe -
-gtest_filter=simple_tests/NibbleTestFixtureSmall.simple_tests/0
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_d
b.so.1".
Note: Google Test filter = simple_tests/NibbleTestFixtureSmall.simp
le_tests/0
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from simple_tests/NibbleTestFixtureSmall
[ RUN      ] simple_tests/NibbleTestFixtureSmall.simple_tests/0
1

Breakpoint 1, nibble_search (query=15 '\017', targets=std::vector o
f length 1, capacity 1 = {...}) at nibble_solution.cpp:9
9    extern "C" uint64_t nibble_search(uint8_t query, const std::ve
ctor<uint16_t> & targets) {
(gdb)
```

And debug away!

16.5 Useful Tools

The tools we'll use in the labs to manipulate and plot data are based on two widely-used software packages: Pandas (https://pandas.pydata.org/) and matplotlib (https://matplotlib.org/). They are powerful software packages with complex interfaces. Most of the tools you'll use directly are defined in `notebook.py` . These are the same functions that I've used above plot results, so this Jupyter Notebook is full of examples of how they work.

The documentation below provides an introduction to the tools in `notebook.py` .

16.5.1 Data Frames

Data frames are a fancy sort of 2-dimensional array. They have rows of data and named columns -- very much like the CSV files. In this class we get them by calling `as_df()` on the result of `run()` (If this fails, go run the cells in Limitations of the Compiler:

In [ ]:
```
df = matmul_run.as_df()
display(df)
```

The numbers down the side are the "index" and the bold names at the top of the columns are the column names.

We can "slice" the data frame as well to extract a subset of its columns. We do this by passing an array of column names in square braces (`[]`):

In [ ]:
```
display(df[["function", "ET"]])
```

We can concatenate multiple dataframes like so:

In [ ]:
```
import pandas as pd
df2 = matmul_run.as_df()
df3 = pd.concat([df,df2])
display(df3)
```

Note that `concat` returns a new dataframe (`df3`) rather than modifying `df` .

16.5.2 Computing Useful Values

The `PE_calc()` function computes the terms of the performance equation:

In [ ]:
```
pe = PE_calc(df)
display(pe)
```

16.5.3 Plotting Results

Matplotlib (https://matplotlib.org/) is an extremely flexible and powerful plotting tool that we could spend a whole quarter on. Fortunately, `notebook.py` has some convenient helper functions in it that make it easy graph data in data frames.

One of those helpers in `plotPEBar` . The cell below, creates a dataframe and then plots several bar charts. The `what` parameter is a list of tuples (pairs of values in `(` and `)`). The first value in each tuple is what ends up on the x-axis. The second item of the tuple is plotted on the y axis. You can plot as many bar charts as you want and it'll plot all the entries in the dataframe:

In [ ]:
```
plotPEBar(df=pe, what=[("function", "ET")])
plotPEBar(df=pe, what=[("function", "ET"),("function", "IC")])
plotPEBar(df=pd.concat([df,df]), what=[("function", "ET"),("function", "I
```

16.5.4 Adding Tags to Experiments

Sometimes it can be useful to add extra columns to the output of your experiments. You can do this by passing `TAG="something"` to `build()` :

In [ ]:
```
my_code = build(code(r"""
#include"cfiddle.hpp"
extern "C" void foo(uint64_t length) {
    start_measurement();
    for(volatile uint64_t i  = 0; i < length; i++) {
    }
    end_measurement();
}
"""), build_parameters=arg_map(OPTIMIZE=["-O3", "-O1"], TAG="my compile ta
```

Then you can run it:

In [ ]:
```
my_run = run(my_code, function="foo", arguments=arg_map(length=[10, 10000
```

And the tag will appear in the output:

In [ ]:
```
display(my_run.as_df())
```

If you want to compare to implementations, you can do this:

In [ ]:
```
my_code_nonvolatile = build(code(r"""
#include"cfiddle.hpp"
extern "C" void foo(uint64_t length) {
    start_measurement();
    for(uint64_t i  = 0; i < length; i++) {// without `volatile`
    }
    end_measurement();
}
"""), build_parameters=arg_map(OPTIMIZE=["-O4", "-O1"], TAG="not volatile
```

In [ ]:
```
my_run = run(my_code+ my_code_nonvolatile, function="foo", arguments=arg_
```

In [ ]:
```
my_run_data = PE_calc(my_run.as_df())
display(my_run_data)
```

16.5.5 Doing Math on Data Frames

Data frames also make it easy to do math on your data. For instance, can compute the average error between `cmdlineMHz` and `realMHz` :

In [ ]:

```
my_run_data["MHz_error"] = my_run_data["realMHz"]/my_run_data["requestedMH
display(my_run_data)
```

There's a lot going on there. The division is a "vector" or "element-wise" operation: the result for each row is computed from the values in that row. However, the `-1` is a "scalar" (single value), so it's subtracted from all the results.

You can also do math on strings:

In [ ]:

```
my_run_data["experiment-name"] = my_run_data["function"] + "-" + my_run_da
```

Which is useful drawing graphs:

In [ ]:

```
plotPEBar(df=my_run_data, what=[("experiment-name", "ET")])
```

16.6 Analyzing Your Solution

You just spent significant effort optimizing `matrix_product()` for different call sites. Let's see how you did compared to the compiler.

The cell below compiles and runs your code and the baseline version with and without optimizations.

In [ ]:

```
starter_unoptimized = build("matmul_starter.cpp", build_parameters=arg_map
starter_optimized = build("matmul_starter.cpp", build_parameters=arg_map(
solution_unoptimized = build("matmul_solution.cpp", build_parameters=arg_m
solution_optimized  = build("matmul_solution.cpp", build_parameters=arg_ma
analysis_data = run(starter_unoptimized +
                    starter_optimized +
                    solution_unoptimized +
                    solution_optimized, function=[f"test_matrix_product_{
```

Here's the results:

In [ ]:

```
analysis_df = PE_calc(analysis_data.as_df())
display(analysis_df)
```

Now we'll use [pivot table (https://pandas.pydata.org/docs/reference/api/pandas.pivot_table.html)](https://pandas.pydata.org/docs/reference/api/pandas.pivot_table.html) (Which are awesome!) to rearrange the data into a more useful format:

In [ ]:

```
perf_compare=pd.pivot_table(analysis_df, values="ET", columns="descriptio

perf_compare["the speedup of your manual optimizations alone"] = perf_com
#perf_compare["the speedup of the compiler's optimizations alone"] =
#perf_compare["the combined speedup of your manual optimizations and the
#perf_compare["the speedup your optimizations provide in addition to the
display(perf_compare)
```

Question 32 (Correctness - 3pts)

Modify the cell above to compute speedups that measure:

1. **the speedup of your manual optimizations alone (done for you).**
2. **the speedup of the compiler's optimizations alone.**
3. **the combined speedup of your manual optimizations and the compiler's optimizations.**
4. **the speedup your optimizations provide in addition to the compiler's (i.e., what's the speedup from your code changes if the the compiler optimizations are already enabled).**

Question 33 (Correctness - 2pts)

Based on the data above, were you able better-optimize any of the five functions? If you did, which ones? Why do you think you beat the compiler on these functions, but not others?

Question 34 (Correctness - 2pts)

In the real world, you should, of course, compile with optimizations enabled. Did the optimizations you made make things easier for the compiler's optimizer so that the combination of you _and_ the compiler did better than the compiler alone? For which functions was this the case?

Question 35 (Correctness - 5pts)

Based on the data above select:

1. **A function (other than `matrix_product_5`) for which your optimizations improved performance beyond what the compiler achieved on its own.**
2. **If there was no such function, the function (other than `matrix_product_5`) for which you achieved the greatest speedup.**

Copy the code for that function into the first cell below. Then, in the following cell, provide another version of the same function that achieves the same performance but has as few changes as possible.

> What optimizations actually made a difference with `-O3` enabled?

Here are some things to consider as your approach this problem:

1. If you saved versions of your code as you applied optimizations, running them might provide some insight.
2. Don't assume that all the optimizations you applied are necessary. For instance, if you have three versions with optimizations A, $A + B$, and $A + B + C$, and the last version is the best, don't assume that optimizations A and B are necessary. Perhaps C is solely responsible for the benefit when the compiler is optimizing too.

In [ ]:

```
# Your fully-optimized version of the function
final_code = build(code(r"""
#include"cfiddle.hpp"
#include"matmul.hpp"
extern "C" void matrix_product_1(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)));
                        }
                }
        }
}

// Comment out the empty implmentation of the function you implemented abo

// extern "C" void matrix_product_1(Matrix * c, Matrix * a, Matrix * b) {
extern "C" void matrix_product_2(Matrix * c, Matrix * a, Matrix * b) {}
extern "C" void matrix_product_3(Matrix * c, Matrix * a, Matrix * b) {}
extern "C" void matrix_product_4(Matrix * c, Matrix * a, Matrix * b) {}
extern "C" void matrix_product_5(Matrix * c, Matrix * a, Matrix * b) {}


""", file_name="all_optimizations.cpp"), build_parameters=arg_map(OPTIMIZI
```

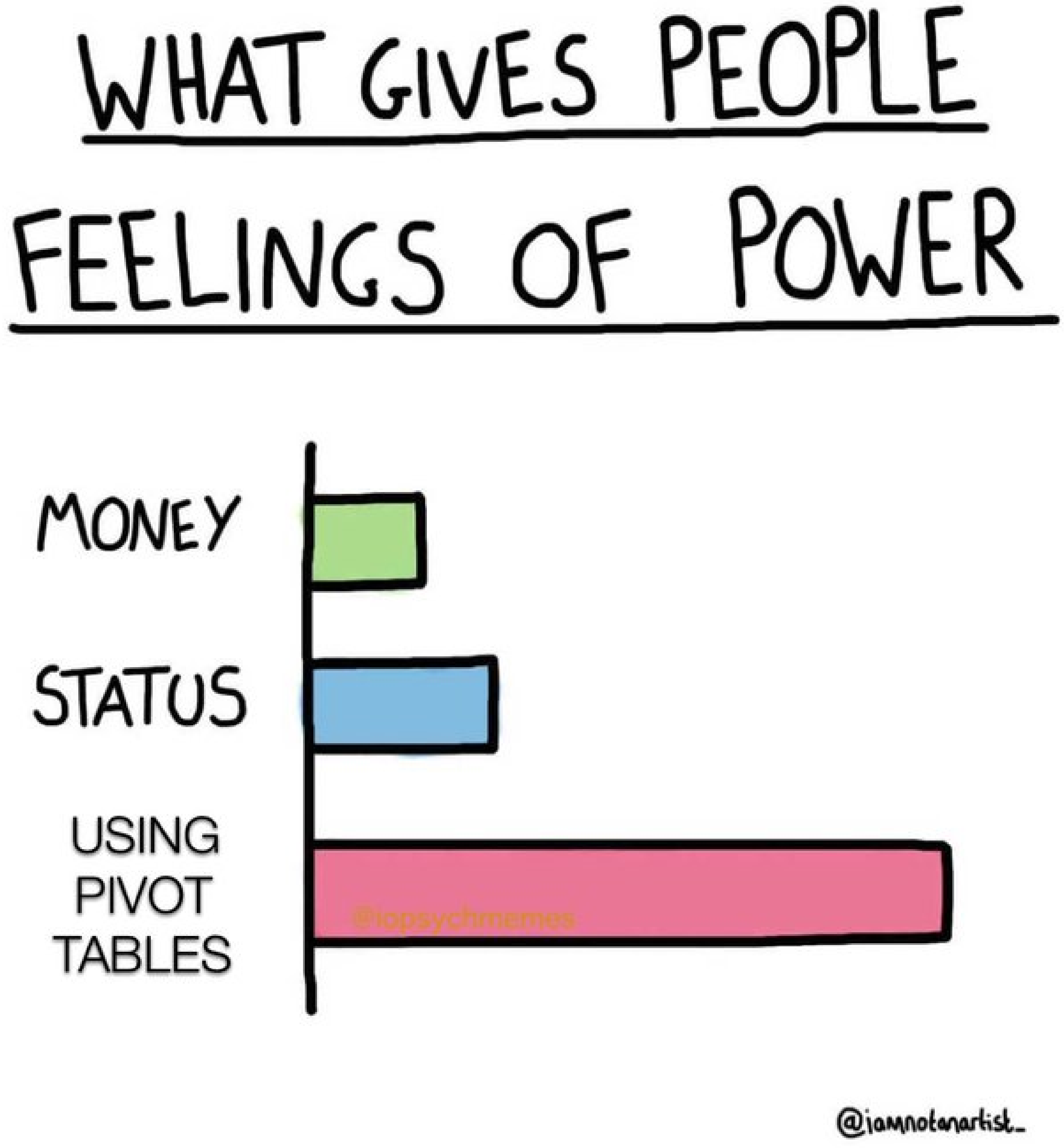
In [ ]:

```
# A version of the same function with minimal set of optimizations that a
# Or, to put it another, what's the smallest set of changes you can make 
#
# If you saved versions of your implementation as you added each optimiza
# performance got better.
good_performance_with_minimal_changes = build(code(r"""
#include"cfiddle.hpp"
#include"matmul.hpp"
extern "C" void matrix_product_1(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)));
                        }
                }
        }
}

// Comment out the empty implmentatiox of the function you implemented abo

// extern "C" void matrix_product_1(Matrix * c, Matrix * a, Matrix * b) {
extern "C" void matrix_product_2(Matrix * c, Matrix * a, Matrix * b) {}
extern "C" void matrix_product_3(Matrix * c, Matrix * a, Matrix * b) {}
extern "C" void matrix_product_4(Matrix * c, Matrix * a, Matrix * b) {}
extern "C" void matrix_product_5(Matrix * c, Matrix * a, Matrix * b) {}

""", file_name="minimal_optimizations.cpp"), build_parameters=arg_map(OPT
```

You can the code to compare the performance. The results should show that the two implementations have nearly the same performance.

In [ ]:

```
# Be sure to adjust the name of the function to match the version you've 
selective_optimization = run(final_code + good_performance_with_minimal_ch
selective_optimization_data = selective_optimization.as_df()
display(selective_optimization_data)
pd.pivot_table(selective_optimization_data, values="ET", columns="functio
```



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

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

And run the autograder

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

And see the autograder's output like this:

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

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

For each lab, there are two different assignments on gradescope:

1. The lab notebook.
2. The programming assignment.

There's also a pre-lab reading quiz on Canvas and a post-lab survey which is embedded below.

18.1 The CSE142L Emergency Lab Submission Form

We do not accept late submissions. However, sometimes things go wrong at submission time. To accommodate this, we have the Emergency Lab Submission Form (https://docs.google.com/forms/d/e/1FAIpQLSdPhzCyLgjmtzwF8frQ1Vrz_zHPaKurlcOf1mWMbAL3ja
It allows us to deal with submission problems in a fair and uniform way.

Here's the process:

1. If you are having trouble submitting, commit your work, and fill out this form *before the deadline*. THERE WILL BE NO EXCEPTIONS GRANTED.
2. The commit has you provide for your github repo must be dated before the deadline.
3. You can continue to try to submit via the normal gradescope.
4. If you aren't able to successfully submit via gradescope, then submit a regrade request during the regrade period.
5. We will review the contents of your github repo, the gradescope submission URLs, and the job IDs you provide.
6. If there was some problem with the infrastructure, you can receive up to full credit. If there was a problem on your side (e.g., not generating the PDF properly), you can earn up to 90% credit.

We will not address these issues on Piazza or via email.

18.2 Reading Quiz

The reading quiz is an online assignment on Canvas. It's due before the class when we will assign the lab.

18.3 The Note Book

You need to turn in your lab notebook and your programming assignment separately.

After you complete the lab, you will turn it in by creating a version of the notebook that only contains your answers and then printing that to a pdf.

Step 1: Save your workbook!!!

In [ ]:
```
!for i in 1 2 3 4 5; do echo Save your notebook!; sleep 1; done
```

Step 2: Run this command:

In [ ]:
```
!turnin-lab Lab.ipynb
!ls -lh Lab.turnin.ipynb
```

The date in the above file listing should show that you just created `Lab.turnin.ipynb`

Step 3: Click on this link to open it: ./Lab.turnin.ipynb (./Lab.turnin.ipynb)

Step 4: Hide the table of contents by clicking the

Step 5: Select "Print" from *your browser*'s "file" menu. Print directly to a PDF.

Step 6: Make sure all your answers are visible and not cut off the side of the page.

Step 7: Turn in that PDF via gradescope.

Print Carefully It's important that you print directly to a PDF. In particular, you should *not* do any of the following:

1. **Do not** select "Print Preview" and then print that. (Remarkably, this is not the same as printing directly, so it's not clear what it is a preview of)
2. **Do not** select `Download as-> PDF via LaTex. It generates nothing useful.

Once you have your PDF, you can submit it via gradescope. In gradescope, you'll need to show us where all your answers are. Please do this carefully, if we can't find your answer, we can't grade it.

18.4 The Programming Assignment

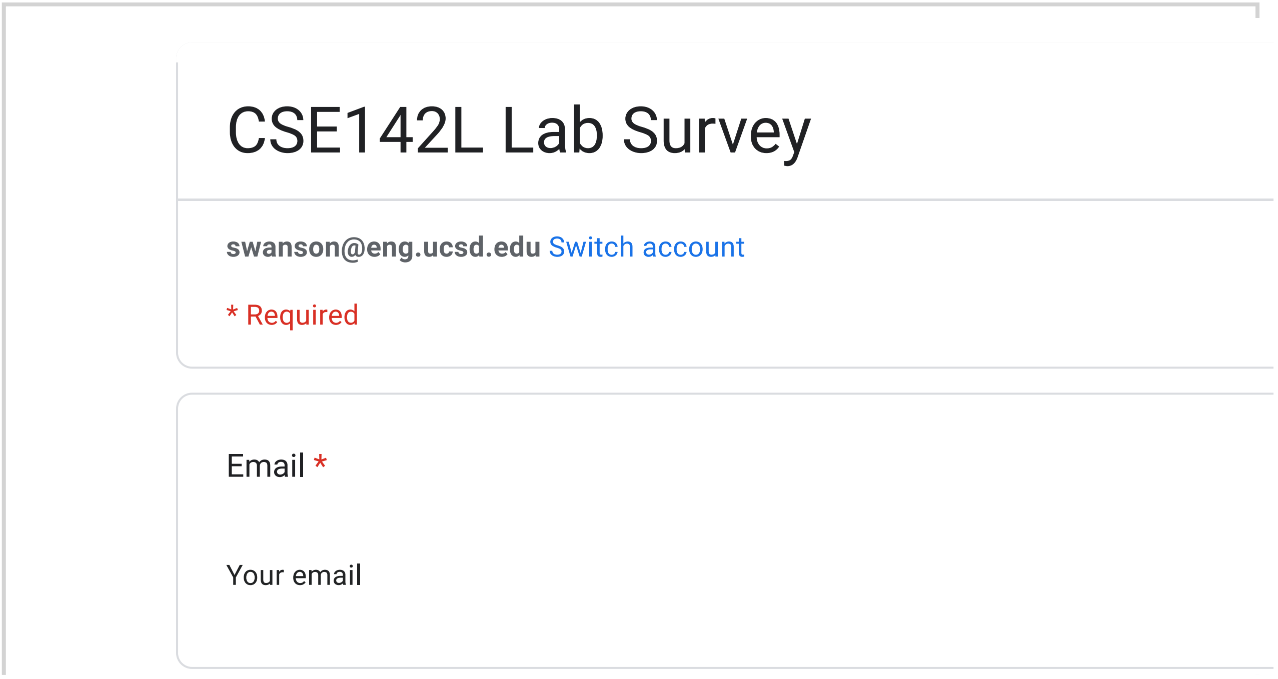
You'll turn in your programming assignment by providing gradescope with your github repo. It'll run the autograder and return the results.

18.5 Lab Survey

Please fill out this survey when you've finished the lab. You can only submit once. Be sure to press "submit", your answers won't be saved in the notebook.

In [7]:
```
from IPython.display import IFrame
IFrame('https://docs.google.com/forms/d/e/1FAIpQLSdEyaIDy52FLLUzQEXoJJmz7
```

Out[7]:



CSE142L Lab Survey

swanson@eng.ucsd.edu Switch account

* Required

Email *

Your email