In [1]:
```
%load_ext autoreload
%autoreload 2
from notebook 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 executables. 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 Pre-Lab Reading Quiz

Part of this lab is pre-lab quiz. The pre-lab quiz **has move to Canvas** so I can allow multiple attempts. It is due **before class on the day the lab is assigned**. It's not hard, but it does require you to read over the lab before class.

2.1 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 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)

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 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.

2.2 Taking the Quiz

You can find it here: https://canvas.ucsd.edu/courses/29567/quizzes (https://canvas.ucsd.edu/courses/29567/quizzes)

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

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

3 Browser Compatibility [...]

4 About Labs In This Class [...]

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

6 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.

7 A Note About The Examples

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

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

8 Common Errors

Known (but harmless) Bug There's a known, intermittent bug in the CFG generation tool. If you get this error:

```
    ^
Expected {'graph' | 'digraph'}  (at char 0), (line:1, col:1)
```

Just run the cell again (several times if needed), and it will eventually work. We are working on this bug, but it's non-deterministic.

9 New Tools

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

9.1 Fiddles

This lab includes some "fiddles" that allow you to write C++ code and then analyze and/or run it. Here's an example that runs the program then shows you the assembly for `main()` (give it a try, you can put whatever you want in the function.):

In [ ]:
```
fiddle("hello_fiddle.cpp", function="main", run=False, code=r"""
#include<iostream>
int main(int argc, char * argv[]) {
    std::cout << "hello world!\n";
    return 0;
}
""").asm
```

9.2 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 blocks* -- 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.

Take a look at this code then run the cell to see it's CFG:

In [ ]:
```
t = fiddle("if_ex.cpp", function="if_ex", run=False, remove_assembly=True
#include<cstdint>
#include<cstdlib>

extern "C"
uint64_t * if_ex(uint64_t * array,
         unsigned long int size) {
    if (size == 0) {
        return NULL;
    }
    return array;
}

int main() {
}
""")
display(t.source)
display(t.cfg)
```

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.

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 [ ]:

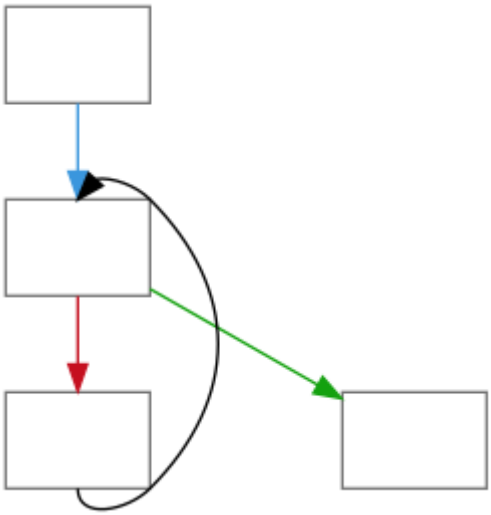
```
t = fiddle("if_else_if_else.cpp", function="if_else_if_else", remove_asse
#include<cstdint>
#include<cstdlib>

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 &array[size/3];
    } else {
        return array;
    }
}

int main() {
}
""")
display(t.cfg)
```

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 [ ]:

```
fiddle("mimic_1.cpp", "foo",code="""
extern "C"
int foo(int a);

int main(int argc, char * argv[]) {
    return foo(4);
}

extern "C"
int foo(int a) {
    return 0;
}

""", remove_assembly=True, run=False).cfg
```

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 [ ]:

```
t = fiddle("loop_if.cpp", function="loop_if", number_nodes=True, run=Fals
#include<cstdint>
#include<cstdlib>

extern "C"
uint64_t * loop_if(uint64_t * array,
           unsigned long int size) {
    uint64_t* t= NULL;
    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

}


int main() {
}
""")
compare([t.source, t.cfg])
```

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	

9.3 Call Graphs

CFGs shows 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 [ ]:

```
fiddle("cg1.cpp", gprof=True, function="one", code="""
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;
}
}
""", remove_assembly=True).call_graph
```

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 code you've written and what you think 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 [ ]:
```
t = fiddle("fiddle_sort.cpp", gprof=True, function="one", opt="-O0", code=
#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];
}

""", remove_assembly=False,  trim_addresses=True, trim_comments=True)
```

In [ ]:
```
display(t.call_graph)
```

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)

If you're curious, 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?

9.4 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 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 your information about any instruction.

9.4.1 The Basics

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

In [ ]:

```
t = fiddle("if_ex.cpp", function="if_ex", run=False, code=r"""
#include<cstdint>
#include<cstdlib>

extern "C"
uint64_t * if_ex(uint64_t * array,
         unsigned long int size) {
    if (size == 0) {
        return NULL;
    }
    return array;
}

int main() {
}
""")
compare([t.source, t.asm])
display(t.cfg)
```

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 assemble directives. They provide metadata about the instruction, functions, and symbols. They don't affect execution. We will mostly ignore them.

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 from 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).

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 path 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. Is 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.

Question 8 (Completeness)

How does the variable `size` appear in the assembly and the 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`?

9.4.2 A More Complex Example

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

In [ ]:

```
t = fiddle("loop_if.cpp", function="loop_if", run=False, code=r"""
#include<cstdint>
#include<cstdlib>

extern "C"
uint64_t * loop_if(uint64_t * array,
           unsigned long int size) {
    uint64_t* t= NULL;
    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

}


int main() {
}
""")
compare([t.source, t.asm])
display(t.cfg)
```

Question 9 (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.)?

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

9.4.3 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 [8]:
```
t = fiddle("loop_func.cpp", function="loop_func", number_nodes=True, run=
#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];

}


int main() {
}
""")
compare([t.source, t.asm])
display(t.cfg)
```

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 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 10 (Completeness)

Fill in the table below to compute how many total instructions and how many memory accesses occur when `loop_func()` executes (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 happy 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 and type into a unique symbol.

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

In [ ]:
```
fiddle("mangle.cpp", function="", demangle=False, code="""
#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;
}

int main(){}
""").asm
```

Question 11 (Completeness)

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

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

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 (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 some 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 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 static. This saves `mov` instructions and memory accesses. You can see it in action below:

In [10]:
```
unopt = fiddle("reg_assign.cpp", function="foo", run=False, code="""

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

int main(){
    return foo(1, 0);
}
""")
opt = fiddle("reg_assign.cpp", function="foo", run=False, opt="-O1")
display(unopt.source)
compare([unopt.asm, opt.asm], ["Unoptimized", "Optimized (-O1)"])
```

The code on the left is unoptimized. The code on the left 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 preform 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 12 (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 13 (Completeness)

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

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 [ ]:

```
unopt = fiddle("CSE.cpp", function="foo", run=False, code="""

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

int main(){
    return foo(1, 0);
}
""")
opt = fiddle("CSE.cpp", function="foo", run=False, opt="-O1")
display(unopt.source)
compare([unopt.asm, opt.asm], ["Unoptimized", "Optimized (-O1)"])
```

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 use `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 it's 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 14 (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 [ ]:
```
unopt = fiddle("LICM.cpp", function="foo", run=False, trim_addresses=True

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;
}

int main(){
    return foo(1, 0);
}
""")
opt = fiddle("LICM.cpp", function="foo", run=False, trim_addresses=True, 
display(unopt.source)
compare([unopt.asm, opt.asm], ["Unoptimized", "Optimized (-O1)"])
compare([unopt.cfg, opt.cfg], ["Unoptimized", "Optimized (-O1)"])
```

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 [ ]:
```
unopt = fiddle("SR.cpp", function="foo", run=False, trim_addresses=True, 
extern "C" int foo(register unsigned int a, register unsigned int b){
    return a *8;
}

int main(){
    return foo(1, 0);
}
""")
opt = fiddle("SR.cpp", function="foo", run=False, trim_addresses=True, op
display(unopt.source)
compare([unopt.asm, opt.asm], ["Unoptimized", "Optimized (-O1)"])
```

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 15 (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 [ ]:
```
unopt = fiddle("SR2.cpp", function="foo", run=False, trim_addresses=True,
extern "C" int foo(register unsigned int a, register unsigned int b){
    return a *8;
}
int main(){
    return foo(1, 0);
}
""")
opt = fiddle("SR2.cpp", function="foo", run=False, trim_addresses=True, o
display(unopt.source)
compare([unopt.asm, opt.asm], ["Unoptimized", "Optimized (-O1)"])
```

Question 16 (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 [ ]:
```
unopt = fiddle("CP.cpp", function="foo", run=False, trim_addresses=True, 

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

int main(){
    return foo(1, 0);
}
""")
opt = fiddle("CP.cpp", function="foo", run=False, trim_addresses=True, op
display(unopt.source)
compare([unopt.asm, opt.asm], ["Unoptimized", "Optimized (-O1)"])
```

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 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 [ ]:

```
unopt = fiddle("CP2.cpp", function="foo", run=False, trim_addresses=True,

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;
}

int main(){
    return foo(1, 0);
}
""")
opt = fiddle("CP2.cpp", function="foo", run=False, trim_addresses=True, o
display(unopt.source)
compare([unopt.asm, opt.asm], ["Unoptimized", "Optimized (-O1)"])
```

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

Question 17 (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 [23]:

```
unopt = fiddle("unroll1.cpp", function="foo", run=False, trim_addresses=T

extern "C" int foo(register unsigned int a, register unsigned int b){
    register unsigned int i, s = 0;
    for(i = 0; i < b*8; i++) {
        s+= i;
    }
    return s;
}

int main(int argc, char *argv[]){
    return foo(1, argc);
}
""")
opt = fiddle("unroll1.cpp", function="foo", run=False, number_nodes=True,
display(unopt.source)
compare([unopt.cfg, opt.cfg], ["Optimized but not unrolled", "Unrolled"])
```

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 18 (Correctness - 3pts)

If `b` is 4 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)?

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 19 (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 [24]:

```
opt = fiddle("unroll-not.cpp", function="foo", run=False, trim_addresses=

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

    i = 10000; // LOOP A
    while(i > 0) {
        i -= b;
        s+=i;
    }


    i = 8;     // LOOP B
    while(i > 0) {
        i -= b;
        s += i;
        if (i == 3)
            continue;
        if (i == 2)
            break;
    }

    return s;
}

int main(int argc, char *argv[]){
    return foo(1, argc);
}
""")
display(opt.source)
display(opt.cfg)
```

Question 20 (Completeness)

Simplify both of the loops above so that the compiler will unroll them. 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?

11.7 Combining Single-function Optimizations

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

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 [ ]:
```
unopt = fiddle("CP.cpp", function="div_loop", run=False, number_nodes=Tru
#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];
}

int main(){

    return div_loop(NULL, 12);
}
""")
opt = fiddle("CP.cpp", function="div_loop", run=False, number_nodes=True,
display(unopt.source)
compare([unopt.cfg, opt.cfg], ["Unoptimized", "Optimized (-O1)"])
```

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

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

Question 21 (Completeness)

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

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

Show Solution

11.8 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 [ ]:
```
fiddle("prologue.cpp", function="", run=False, opt="-Og", code="""

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

int main(){
    return sum(1,2);
}
""").asm
```

The body of `sum` is very simple: It should just be a single add instruction, but instead it has to return as well. The *call site* in `main` takes 3 instructions (the `ret` is part of the overhead for calling `main`, not `sum`).

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

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](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 22 (Optional)

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

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!

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 [26]:
```
loop_unopt = fiddle("inline1.cpp", function="main", run=False, remove_ass

extern "C" int inline __attribute__ ((used)) foo( register int a, registe
    return a + b;
}

int main(int argc, char * argv[]){
    register int i;
    register int s = 0;
    for(i = 0; i < argc; i++) {
        s += foo(i,i);
    }
    return s;
}
""")
loop_opt = fiddle("inline1.cpp", function="main", run=False, remove_assem
foo_unopt = fiddle("inline1.cpp", function="foo", run=False, remove_assem

compare([foo_unopt.source, foo_unopt.cfg], ["foo()", "No inlining"])
compare([loop_unopt.source, loop_unopt.cfg], ["loop()", "No inlining"])
compare([loop_opt.cfg], ["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 23 (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 [27]:
```
caller_unopt = fiddle("inline2.cpp", function="caller", run=False, remove_

extern "C" inline int 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(register int a) {
    return loop(20);
}

int main(){
    caller(1);
}
""")
caller_opt = fiddle("inline2.cpp", function="caller", run=False, remove_as
loop_unopt = fiddle("inline2.cpp", function="loop", run=False, remove_asse

compare([loop_unopt.source, loop_unopt.cfg], ["loop()", "No inlining"])
compare([caller_unopt.source, caller_unopt.cfg], ["caller()", "No inlinin
compare([caller_opt.cfg], ["caller() with inlining"])
```

Bye bye, function call! Bye Bye, loop!

Question 24 (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 [28]:
```
unopt = fiddle("sort_revisited.cpp", gprof=True, function="one", opt="-O0
#include<algorithm>
extern "C" int one(int a);

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

""", remove_assembly=False,  trim_addresses=True, trim_comments=True)
opt = fiddle("sort_revisited.cpp", gprof=True, function="one", opt="-O1 -
                remove_assembly=False,
                trim_addresses=True,
                trim_comments=True)
compare([unopt.call_graph, unopt.cfg],["Unoptimized call graphs", "Unoptir
compare([opt.call_graph, opt.cfg],["Optimized call graph", "Optimized CFG
```

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 `one()`, which now calls several other functions (e.g., `std::__introsort_loop()` and `std::__unguarded_linear_insert()`.
2. The CFG for `one()` 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 [29]:
```
!make clean
!make sort.exe sort_O3.exe sort_Og.exe
```

In [30]:
```
!cse142 job run --take "sort*.exe" --take '*.cfg' './sort.exe    --functi
!cse142 job run --take "sort*.exe" --take '*.cfg' './sort_O3.exe --functi
!cse142 job run --take "sort*.exe" --take '*.cfg' './sort_Og.exe --functi
```

In [31]:
```
df = render_csv("sort_0.csv").append(render_csv("sort_3.csv"))
```

In [32]:
```
display(df[["optimization", "IC", "CPI", "CT","ET"]])
plotPEBar(df=df,
          what=[("optimization", "IC"),
                ("optimization", "CPI"),
                ("optimization", "CT"),
                ("optimization", "ET")])
```

Question 25 (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 oft-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 our selves!

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 let's 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 note 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 [79]:

```
optimizations = "-Og -fno-inline"
heading(f"Compiled with {optimizations}")
static_unopt = fiddle("virt.cpp", function="static_call", run=False, numb
                      code="""
#include<cstdint>
#include"function_map.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;
    }
};

#define START_C extern "C" { // this just hides the braces from the edito
#define END_C  }


START_C


int static_call(uint64_t * array,
        uint64_t  size) {
    A a;
    register int sum = 0;

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

    return sum;
}

int virtual_call(uint64_t * array,
         uint64_t  size) {
    register A * a = new A();
    register int sum = 0;

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

    return sum;
}

END_C

#define METHOD_FUNC(n) FUNCTION("method", n)
METHOD_FUNC(virtual_call);
METHOD_FUNC(static_call);
""")
virt_unopt = fiddle("virt.cpp", function="virtual_call",  run=False, numb
```

```
!cse142 job run --lab compiler "./virt.exe --function ALL --size 10000000
```

In [76]:

```
compare([static_unopt.cfg,virt_unopt.cfg], ["Unoptimized Static call", "U
```

On the left, is `static_call()`. In block `n2` you can see the call to `A::foo(int)`.

`virtual_call()` is on the right. The structure is basically the same, but `n2` looks different. Instead of invoking the function via it's label, 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 `n2` of `virtual_call()` (you might have to double click to zoom in):

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).

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 [80]:

```
optimizations = "-O3"
static_opt = fiddle("virt.cpp", function="static_call",  run=False,  numb
virt_opt = fiddle("virt.cpp", function="virtual_call", run=False,  number_
!cse142 job run --lab compiler "./virt.exe --function ALL --size 10000000
```

In [78]:

```
compare([static_opt.cfg, virt_opt.cfg], ["Optimized Static call", "Optimi
```

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 the 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 `n6`.

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. The cells above collected data for both versions with and without optimizations.

In [90]:
```
unopt = render_csv("virt_unopt.csv")
unopt['optimizations'] = "unoptimized"
opt = render_csv("virt_opt.csv")
opt['optimizations'] = "optimized"
all = unopt.append(opt)
all["experiment"] = all["function"] + "-" + all["optimizations"]
display(all[["experiment", "IC", "CPI", "CT", "ET"]])
plotPEBar(df=all, what=[("experiment", "IC"), ("experiment", "CPI"), ("ex
```

Question 26 (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. 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 27 (Optional)

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

13 Practical Rules For Using Compiler Optimizations

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

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

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

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

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 `one()` in the previous section. If you set a break point on `std::sort()` , 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. 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 [ ]:
```
df = render_csv("sort_0.csv").append(render_csv("sort_3.csv")).append(ren
display(df[["optimization", "IC", "CPI", "CT","ET"]])
plotPEBar(df=df,
          what=[("optimization", "IC"),
                ("optimization", "CPI"),
                ("optimization", "CT"),
                ("optimization", "ET")])
```

Not a huge difference... 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.

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 an design meaningful experiments that let you measure their impact.

14 Limitations of the Compiler

Compiler optimizations can do a lot, but they are not magical. In particular, they cannot save you from choosing the wrong algorithm or data stucture for a particular task. Indeed, choosing the right algorithms and data structures can have even larger impacts on performance.

For instance, consider four well-known sorting algorithms `bubble_sort`, `insertion_sort`, `quick_sort`, and `stl:sort` (the implementation in the C++ `stl`. It could be any $O(n \log(n))$ sorting algorithm):

In [2]:
```
render_code("sort.cpp", show="bubble_sort")
render_code("sort.cpp", show="insertion_sort")
render_code("sort.cpp", show=("//START", "//END"))
render_code("sort.cpp", show="stl_sort")
```

```
// sort.cpp:24-41 (18 lines)
uint64_t* bubble_sort(uint64_t *list, uint64_t size)
{
        uint64_t temp;
        for(uint64_t i=0; i<size; i++)
        {
                for(uint64_t j=size-1; j>i; j--)
                {
                        if(list[j]<list[j-1])
                        {
                                temp=list[j-1];
                                list[j-1]=list[j];
                                list[j]=temp;
                        }
                }
        }
        CHECK(list);
        return NULL;
}
```

```
// sort.cpp:48-66 (19 lines)
uint64_t* insertion_sort(uint64_t *list, uint64_t size)
{
        for(uint64_t j=1;j<size;j++)
        {
                uint64_t key=list[j];
                uint64_t i = j-1;
                while(list[i] > key)
                {
                        list[i+1]=list[i];
                        if (i == 0)
                                break;
                        i=i-1;
                }
                list[i+1]=key;

        }
        CHECK(list);
        return NULL;
}
```

```
// sort.cpp:140-179 (40 lines)
//START
uint64_t partition(uint64_t *list, uint64_t p, uint64_t r)
{
        uint64_t pivot, index, exchange_temp;
        pivot = list[r];
        index = p - 1;
        for(uint64_t i = p; i < r; i++)
        {
                if(list[i] <= pivot)
                {
                        index++;
                        exchange_temp = list[i];
                        list[i] = list[index];
                        list[index] = exchange_temp;
                }
        }
        exchange_temp = list[r];
        list[r] = list[index+1];
        list[index+1] = exchange_temp;
        return index+1;
}

void quicksort_aux(uint64_t *list, uint64_t p, uint64_t r)
{
        uint64_t q;
        if(p<r)
        {
                q = partition(list, p, r);
                quicksort_aux(list, p, q-1);
                quicksort_aux(list, q+1, r);
        }
}

uint64_t* quick_sort(uint64_t *list, uint64_t size)
{
        quicksort_aux(list,0, size-1);
        CHECK(list);
        return NULL;
}
//END
```

```
// sort.cpp:181-186 (6 lines)
uint64_t * stl_sort(uint64_t * array,
                    uint64_t  size) {
        std::sort(&array[0], &array[size]);
        CHECK(array);
        return NULL;
}
```

Let's see how they perform sorting 64k elements:

In [ ]:
```
!make sort_O3.exe
!cse142  job run --take "sort*.exe" --take '*.cfg' './sort_O3.exe --funct
```

In [ ]:
```
plotPEBar("sorts.csv", what=[("function", "IC"), ("function", "CPI"), ("f
display(render_csv("sorts.csv", ["function", "IC", "CPI", "CT", "ET"]))
```

Question 28 (Correctness - 1pts)

What's the speedup of `quick_sort` over `bubble_sort` ?

I'm actually not very happy with this measurement. The runtime of `quick_sort` is too short for me to really think it's accurate -- I'd rather the runtime be in the 1-second range. But getting `quick_sort` to take 1 second requires me to increase the array size by 100x. Which would increase `bubble_sort` 's runtime by 10,000x, to about 30 hours. In any case, lesson is clear: Don't use bubble sort!

But it's not just a question of choosing among different approaches to the same problem. You need to make sure you are solving the problem you need to solve. Consider the problem of finding the smallest value in an array. Here's one implementation:

In [ ]:

```
fiddle("find_min.cpp", compile=False, code=r"""
#include"function_map.hpp"
#include<algorithm>
#include<cstdint>

// Reference implementation
uint64_t find_min_reference(uint64_t * a, uint64_t size) {
    std::sort(a, &a[size]);
    return a[0];
}

FUNCTION("find_min", find_min_reference); // Don't edit this.
////////////////////////////////////////
// Don't edit anything above this line //
////////////////////////////////////////

// Below you can implement as many different versions of the function as y
// To add another implementation, you need to do two things:
//
// 1. Create a function with the same signature (arguments and return typ
// 2. Call "FUNCTION" macro with "find_min" as the first argument and the
//
// You can see an example below with find_min_variant()

uint64_t find_min_solution(uint64_t * a, uint64_t size) {
    std::sort(a, &a[size]);
    return a[0];
}

FUNCTION("find_min", find_min_solution);

""")

!make find_min.exe
!cse142 job run  --take find_min.exe --take PE.cfg "./find_min.exe --func

plotPEBar("find_min.csv", what=[("function", "IC"),
                                ("function", "CPI"),
                                ("function", "CT"),
                                ("function", "ET")])
display(render_csv("find_min.csv", columns=["function", "IC", "CPI", "CT"
```

The question below is a simple programming assignment. You can do it in the fiddle above. The fiddle will save all the versions you write as `.fiddle_backups/find_min_0001.cpp` , `.fiddle_backups/find_min_0002.cpp` , etc.

Question 29 (Correctness - 5pts)

Modify `find_min_solution()` in the fiddle above to use a more appropriate, faster algorithm. What speedup does your version achieve relative to `find_min_reference()`? (Points for this question depend on the correctness of your code and the performance it achieves. Your target is 35x.).

Speedup:

15 Programming Assignment

The programming assignment for this lab is a warm up for the programming assignments in the next three labs. The assignment is to optimize the performance of the function below.

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 "Do Your Work Here" section. It has the key commands you'll need to evaluate and analyze your results.
3. When you are happy with your implementation, answer the questions in "Analyzing Your Implementation".
4. When you are all done, go to the Final Measurement section and follow those instructions.

15.1 The Function

In [3]:
```
render_code("sum.cpp", show="sum_of_locations")
```

```
// sum.cpp:1-39 (39 lines)
#include <cstdlib>
#include "archlab.hpp"
#include <unistd.h>
#include"omp.h"
#include"papi.h"
#include<algorithm>
#include<cstdint>
#include<unordered_set>
#include"function_map.hpp"



#define START_C extern "C" { // this just hides the braces from the edito
r, so it won't try to indent everything.
#define END_C  }


START_C

uint64_t sum_of_locations_solution(uint64_t *search_space, uint32_t searc
h_space_size, uint64_t* queries, uint32_t query_count)
{
        uint64_t r = 0;

        for(uint32_t i = 0; i < query_count; i++) {
                for(uint32_t j = 0; j < search_space_size; j++) {
                        if (search_space[j] == queries[i]) {
                                r += i;
                                break;
                        }
                }
        }
        return r;
}

// hello

FUNCTION("sum_impl", sum_of_locations_solution);

END_C
```

The function takes two arrays: one contains a list of unique integers (`search_space`) while the other (`queries`) has a list of query values. For each query, the function checks if the value is in

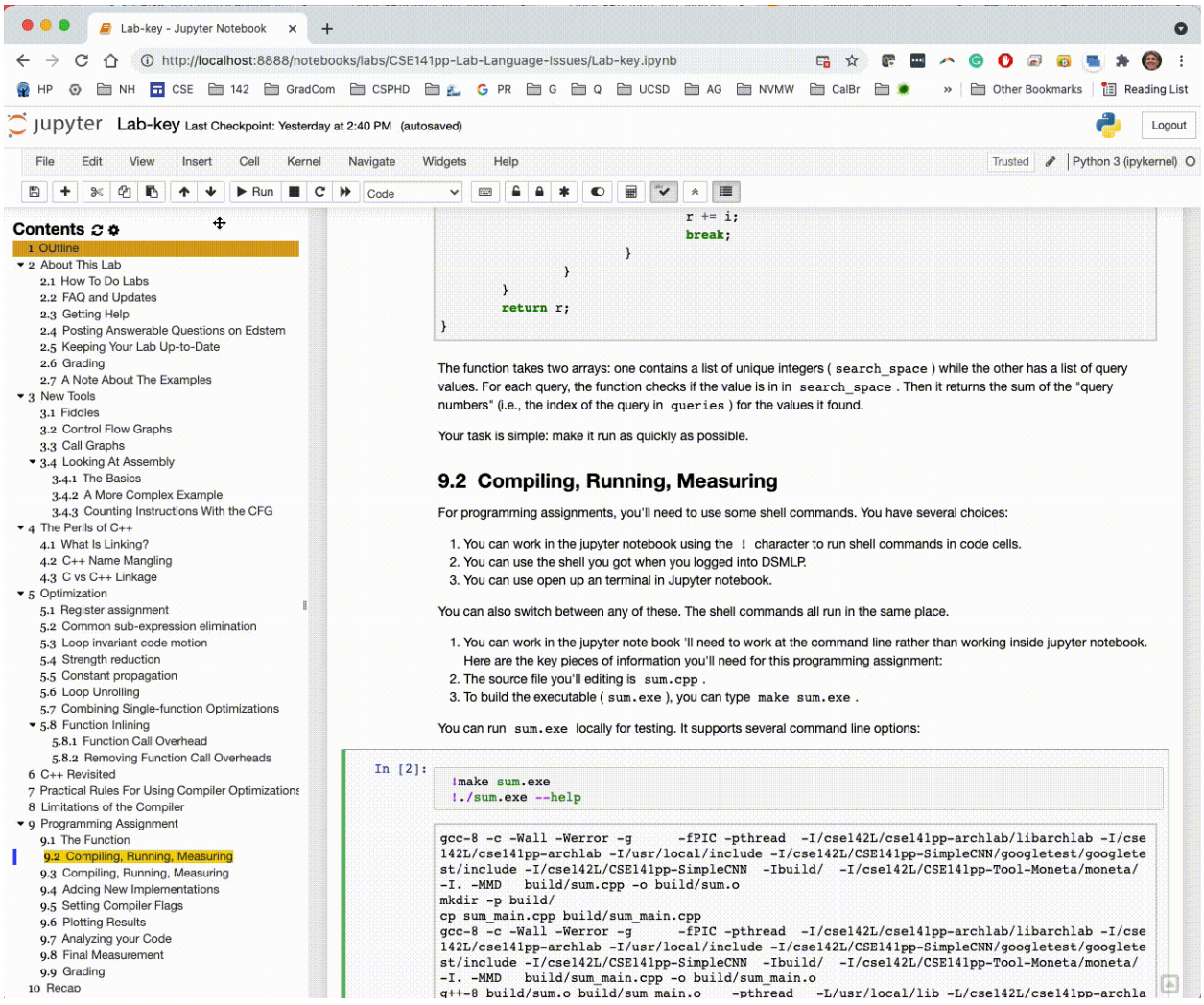
`search_space` . Then it returns the sum of the "query numbers" (i.e., the index of the query in `queries`) for the values it found.

Your task is simple: make it run as quickly as possible.

15.2 Compiling, Running, Measuring

For programming assignments, you'll need to use some shell commands. You have a couple choices:

1. You can work in the jupyter notebook using the `!` character to run shell commands in code cells. They are all provided below.
2. You can open up an terminal in Jupyter notebook.



You can also switch between any of these options. The shell commands all run in the same directory structure.

Here are the key pieces of information you'll need for this programming assignment:

1. The source file you'll editing is `sum.cpp` .
2. To build the executable (`sum.exe`), you can type `make sum.exe` .

You can run `sum.exe` locally for testing. It supports several command line options:

In [ ]:
```
!make sum.exe
!./sum.exe --help
```

The ones that might be helpful in this lab are:

- `--function` which will let you select an implementation to test.
- `--stats` which controls where your data goes.
- `--space-size` controls the search space size.
- `--queries` controls how many queries to run.
- `--seed` sets the random seed.
- `--function` determines which functions will run. You enter a subset of those listed above if you want or `ALL`
- `--stat-set` configures the performance counters. `PE.cfg` collects `IC`, `CPI`, `CT`, and `ET` data.

You can measure performance in the cloud with this command line (Note, I've set `--MHz 3500`. The default is 3501, but that will enable TurboBoost and make our performance unpredictable):

In [ ]:
```
!make sum.exe
!cse142 job run --lab compiler "./sum.exe --function ALL --MHz 3500 --sta
```

Note that the command line for `./sum.exe` is in quotes and the command line arguments for `sum.exe` go in quotes with it.

Before the `sum.exe` command line is the command to run `sum.exe` in the cloud.

If you run the command above on your starter code, it'll produce a `csv` file called `benchmark.csv` with two lines:

In [91]:
```
display(render_csv("benchmark.csv"))
```

The output of the program is in `result` and the execution time is in `ET`.

You can print the CSV file nicely at the command line:

In [92]:
```
!pretty-csv benchmark.csv
```

15.3 Adding New Implementations

The starter code for `sum.cpp` has one solution in it: `sum_of_locations_solution()`. The baseline (unoptimized) implementation is in `baseline_sum.cpp` and is called `sum_of_locations()`. `sum_of_locations_solution()` in `sum.cpp` should eventually

contain your best implementation, and that's what we will grade. You can add as many other implementations to `sum.cpp` as you'd like.

To add a new implementation, just create a new function with a new name (e.g., `sum_of_locations_new_version`) but same arguments and return value. Then add this immediately after the end of the function:

```
FUNCTION("sum_impl", sum_of_locations_new_version);
```

That will allow you to pass `--function sum_of_locations_new_version` to test your new version.

15.4 Setting Compiler Flags

There are two ways to set compiler flags for your code. When you turn you code into the autograder for credit, you'll need to add the compiler flags in `config.make`. Here's the initial contents:

In [4]:
```
render_code("config.make")
```

```
// config.make:1-3 (3 lines)
SUM_OPTIMIZE=
```

So, it will compile your code with no optimizations.

To edit `config.make`, you can open up a terminal and edit it with `emacs`, `vim`, or `pico`.

You can also pass `SUM_OPTIMIZE` to make. Run this cell and the `-finline-functions` appears in the command that compiles `sum.cpp`.

In [ ]:
```
!make clean
!make sum.exe SUM_OPTIMIZE=-finline-functions
```

15.5 Things To Try

Here are some suggestions on how to approach the lab.

1. Make sure to set appropriate compiler flags
2. Think about *what* `sum_of_locations()` is trying to achieve rather than *how* it is doing it.
3. If you had to achieve the same thing, how would you approach it?
4. How does `sum_of_locations()` use each of its data structures? What operations does it apply to each of them?
5. Is there a better choice of data structure?
6. The C++ standard template library is available for you to use.

You can assume that `queries` has lots of entries.

To achieve the target speedup you will need to modify the code/algorithm *and* apply compiler optimizations.

15.6 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 portion of the lab starting with C++ Revisited is especially useful in this regard.

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

15.6.1 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 that our experiments generate. Getting a dataframe from a CSV file is easy:

In [ ]:
```
df = render_csv("benchmark.csv")
```

`df` now contains the contents of `benchmark.csv` and we can inspect it like so:

In [ ]:
```
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", "IC", "CPI", "CT","ET"]])
```

We can concatenate multiple dataframes like so:

In [ ]:
```
df2 = render_csv("benchmark.csv")
df3 = df.append(df2)
display(df3[["function", "IC", "CPI", "CT","ET"]])
```

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

15.6.2 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 [ ]:
```
df = render_csv("benchmark.csv")
plotPEBar(df=df, what=[("function", "ET")])
plotPEBar(df=df, what=[("function", "ET"),("function", "IC")])
plotPEBar(df=df.append(df), what=[("function", "ET"),("function", "IC")])
```

15.6.3 Adding Tags to Experiments

In the first lab, you saw many examples of how we can run experiments with different parameter values with a single command line. For instance, here's a command line that runs your solution with multiple query list sizes:

In [ ]:
```
!cse142 job run --lab compiler "./sum.exe  --MHz 3500 --queries 1 10 100
```

In [ ]:
```
render_csv("queries.csv")
```

However, if you wanted to, for instance, compare the performance of the your code running under two different sets of compiler flags, that approach won't work because you need two different executable. You could do something like this:

In [ ]:
```
!make clean;
!make sum.exe SUM_OPTIMIZE=-finline-functions
!cse142 job run  --lab compiler "./sum.exe  --MHz 3500 --queries 1 --func
!make clean;
!make sum.exe SUM_OPTIMIZE=-funroll-loops
!cse142 job run  --lab compiler "./sum.exe  --MHz 3500 --queries 1 --func
```

And then display the results like this:

In [ ]:
```
inline = render_csv("inline.csv")
unroll = render_csv("unroll.csv")
both = inline.append(unroll)
display(both)
```

But there's no way to tell which lines came from which file! Oh no!

To solve this problem, you can pass `--tag`:

In [ ]:
```
!make clean;
!make sum.exe SUM_OPTIMIZE=-finline-functions
!cse142 job run  --lab compiler "./sum.exe  --MHz 3500 --queries 1 --func
!make clean;
!make sum.exe SUM_OPTIMIZE=-funroll-loops
!cse142 job run  --lab compiler "./sum.exe  --MHz 3500 --queries 1 --func
```

And then we can do:

In [ ]:
```
inline = render_csv("inline.csv")
unroll = render_csv("unroll.csv")
both = inline.append(unroll)
display(both)
```

And now we have the `opt` column with our tag values in it, and we can then plot the results and display a tidier table:

In [ ]:
```
plotPEBar(df=both, what=[("opt", "ET")])
display(both[["opt", "ET"]])
```

15.6.4 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 [ ]:
```
both["MHz_error"] = both["realMHz"]/both["cmdlineMHz"] - 1
display(both)
```

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 [ ]:
```
both["experiment-name"] = both["function"] + "-" + both["opt"]
```

Which is useful drawing graphs:

In [ ]:
```
plotPEBar(df=both, what=[("experiment-name", "ET")])
```

15.7 Do Your Work Here

This section is a workspace for you to do the assignment. It has the basic commands you'll need to get through it. You may find it useful to expand and add to this section. Go crazy!

15.7.1 Compile And Run

In [ ]:
```
!make sum.exe
!cse142 job run --lab compiler "./sum.exe --function ALL --MHz 3500 --sta
```

15.7.2 Graph Results

In [ ]:
```
df = render_csv("benchmark.csv")
display(df)
plotPEBar(df=df, what=[("function", "ET")])
```

15.7.3 Analyzing your Code With `fiddle`

In [ ]:
```
f = fiddle("sum.cpp", function="sum_of_locations_solution", build_cmd="ma
# display(f.asm) # uncomment to display the assembly.
display(f.cfg) # uncomment to display the cfg
# display(f.source) # uncomment to display the source
```

15.8 Analyzing Your Implementation

Complete these questions after you've completed your solution to programming assignment.

To achieve a good speedup on this lab you needed to apply both compiler optimizations and algorithm changes.
Optimizations can interact with each other in surprising ways. Let's see how each contributed to the overall speedup.

(The cell below should contain the code for your `sum_of_locations_solution()`, and the TAs will refer to it while grading. If your code isn't showing up here, rerun the cell. If it still doesn't show up, add a text cell and paste in your function.)

In [5]:
```
render_code("sum.cpp", show="sum_of_locations_solution")
render_code("config.make")
```

```
// sum.cpp:19-32 (14 lines)
uint64_t sum_of_locations_solution(uint64_t *search_space, uint32_t searc
h_space_size, uint64_t* queries, uint32_t query_count)
{
        uint64_t r = 0;

        for(uint32_t i = 0; i < query_count; i++) {
                for(uint32_t j = 0; j < search_space_size; j++) {
                        if (search_space[j] == queries[i]) {
                                r += i;
                                break;
                        }
                }
        }
        return r;
}
```

```
// config.make:1-3 (3 lines)
SUM_OPTIMIZE=
```

The rest of the lab is about measuring the contribution of compiler optimizations vs algorithmic optimizations. The result of the your investigation will go into the question below.

The first step is to collect the data.

The cell below will run four different versions of the experiment:

1. The `sum_of_locations` with compiler optimizations.
2. The `sum_of_locations` without compiler optimizations.
3. The `sum_of_locations_solution` with compiler optimizations.
4. The `sum_of_locations_solution` without compiler optimizations.

It uses tags to differentiate between the optimized and un-optimized versions.

In [ ]:
```
!make clean;
!make sum.exe
!cse142 job run  --lab compiler "./sum.exe  --MHz 3500 --function sum_of_
!make clean;
# This overrides config.make and forces no optimizations.
!make sum.exe SUM_OPTIMIZE=
!cse142 job run  --lab compiler "./sum.exe  --MHz 3500 --function sum_of_
```

Now you need to create a data frame that contains the results the experiment in form that we can plot. In particular you need to:

1. Open the `csv` file created by the command above and load them into two dataframes.
2. Combine them into one data frame
3. Compute a new column that has "function name-optimization level" in it (e.g., `sum_of_locations-noopt`).
4. Compute the speedup of all four implementations relative to `sum_of_location-noopt`.
5. Plot the resulting graph.

There are examples of the tools you'll need for this in the "Useful Tools" section above.

You must show your work. In the cell below, put the your code to do all of the above.

The resulting dataframe should look like this (but with data in it):

workload	ET	Speedup
sum_of_locations-noopt		
sum_of_locations-opt		
sum_of_locations_solution-noopt		
sum_of_locations_solution-opt		

Note:. You should *not* fill in the table above. You need use `display()` show your final data frame.

Question 30 (Correctness - 8pts)

Put your code to prepare the dataframe in the cell below. The last two lines should render your dataframe and plot `ET` as a bar graph. (You are allowed to hard-code numbers from your results into your code to compute speedup)

In [96]:
```
# Your code here
```

Question 31 (Correctness - 4pts)

Based on you data, did the compiler optimizations provide more speedup on `sum_of_locations()` or `sum_of_locations_solution()`?

Likewise, did algorithmic improvements provide more or less speedup with compiler optimizations enabled than with them disabled?

Question 32 (Optional)

Modify your dataframe manipulation code above to figure to not require hard-coding values for the speedup computation.

15.9 Final Measurement

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

Your grade is based on your speed up relative to the original version of `sum_of_locations()`. The target speedup is 35x. So your score is `your_speedup/35 * 100.0`. To get points, your code must also be correct. The autograder will run some tests on your code against random inputs to check it's correctness.

You can mimic exactly what the autograder will do with the command below.

After you run it, the results will be in `autograde/autogradecsv` rather than `./benchmark.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 `sum.cpp` and `config.make`.
2. Copies those files into a clean clone of the starter repo.
3. Builds `sum.exe` from scratch.
4. And then runs the command above with `--function sum_of_locations_solution sum_of_location`.
5. It then runs the `autograde.py` script to compute your grade.

In [ ]:
```
!cse142 job run --take sum.cpp --take PE.cfg --take config.make --lab com
!./autograde.py --submission autograde --results autograde.json
```

You can check the performance results like this:

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

And see the autograder's output like this:

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

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

All that's left is commit your code:

In [ ]:
```
!git commit -am "Solution to the lab."
!git push
```

If `git commit` tell you something like:

```
*** Please tell me who you are.

Run

git config --global user.email "you@example.com"
git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'prcheng@dsmlp-jupy
ter-prcheng.(none)')
Warning: Permanently added the RSA host key for IP address '140.82.
112.3' to the list of known hosts.
Everything up-to-date
```

Then you can do (but fill in your @ucsd.edu email and your name):

In [ ]:
```
!git config --global user.email "you@example.com"
!git config --global user.name "Your Name"
```

Subverting the autograder is an AI violation The autograder constrains how your code is run. It's not perfect, and there are probably ways to "hack" it give you more points than your solution deserves. Attempting to do so is an AI violation and will be treated accordingly.

16 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.

Question 33 (Optional)

What's something interesting or surprising you found in this lab by experiment with code in a `fiddle`? Paste the fiddle below and run it before you turn in your lab.

In [ ]:
```
# Your crazy fiddle here
```

17 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.

17.1 Reading Quiz

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

17.2 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 [5]:
```
!for i in 1 2 3 4 5; do echo Save your notebook!; sleep 1; done
```

Step 2: Run this command:

In [2]:
```
!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.

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.

17.3 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.

17.4 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 [6]:
```
from IPython.display import IFrame
IFrame('https://docs.google.com/forms/d/e/1FAIpQLScHbK7yLlixJqdYsRnpvLLT_
```

Out[6]:

CSE142L Lab Survey

swanson@eng.ucsd.edu Switch account

* Required

Email *

Your email

Student ID (to get credit) *

Your answer

Student ID (to confirm) *

Your answer

Which lab is this? *

Choose

How hard did you think this lab was? *

	1	2	3	4	5	
Very easy	○	○	○	○	○	Ve

How much did you learn from this lab? *

	1	2	3	4	5	
Very little	○	○	○	○	○	A gre

How interesting was this lab? *

	1	2	3	4	5	
Very boring	○	○	○	○	○	Very int

How many hours did you spend on this lab? *

Your answer

Name one thing you liked about the lab *

Your answer

Name another thing you liked about the lab *

Your answer

Name one thing we should change/improve about the lab *

Your answer