In [1]:
```
%load_ext autoreload
%autoreload 2
```

In [2]:
```
from CSE142L.notebook import *
from notebook import *
# if you get something about NUMEXPR_MAX_THREADS being set incorrectly, d
```

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

Lab 3: The Memory Hierarchy

The next two labs will explore the impact of memory accesses on program performance. By the end of them, you should have clearer understanding of the critical impact that a program's memory behavior has on its performance. In particular, you'll learn about the concepts of:

1. Memory alignment
2. Thinking in cache lines
3. Working sets
4. The cache hierarchy
5. The impact of miss rate on performance
6. The role of the TLB in determining performance
7. Spatial locality
8. Temporal locality
9. Cache-aware optimizations
10. The impact of data structures on memory behavior

Along the way, we'll address several of the "interesting questions" we identified in the first lab, including:

- Why does increasing the size of array change `CPI`? And why does this change occur so quickly?
- How and why do the datatypes we use change `IC` and `CPI`?
- Why does the order in which the program performs calculations affect `CPI`?

~~This lab includes a programming assignment.~~

Check the course schedule for due date(s).

1 FAQ and Updates

- There are no updates, yet.

2 Additional Reading

If, after these two labs, you still thirst for practical knowledge about using memory effectively, you should should read this series of articles: What every programmer should know about memory (https://lwn.net/Articles/250967/). It's long but quite good. It's not required reading, but, for the programming assignments that say "make this code go as fast you can," everything it includes is fair game.

3 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 you to the menu of environments.

4 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**.

4.1 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)

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.

4.2 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.

5 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.

6 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.

6.1 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 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 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.

6.2 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 piazza.

6.3 Posting Answerable Questions on Piazza

If you want useful answers on 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 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 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 piazza. We will archive it and ask you to re-post your question with the information we need.

6.4 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 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.

6.5 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.

6.6 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 work, you'll need to fall back on working through Jupyter Notebook.

6.7 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.

6.7.1 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.

6.7.2 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")
```

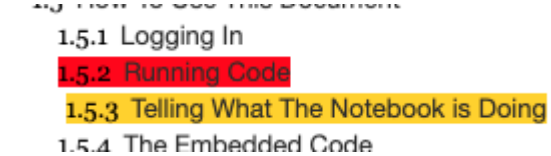
Code cells can also execute shell commands using the `!` operator. Try it below:

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

6.7.3 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:

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:



6.7.4 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.

6.7.5 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. If you get

```
You already have one or more jobs submitted or running.
a26fc9cc-ba36-4f49-89ea-1f36b16b5ea4    you@ucsd.edu    CREATED
 2022-10-07 23:46:18.709330+00:00    true
Do you want to cancel them and run this job? [y/N]:
```

You can run `cse142 job run --lab intro --take NOTHING true` and it should fix it.

9. If you get a big list of files that ends like this:

```
.cfiddle/builds/build/MORE_INCLUDES_-I_cse142L_CSE141pp-Tool-Moneta
_moneta__nibble/nibble_29.so
.cfiddle/builds/build/MORE_INCLUDES_-I_cse142L_CSE141pp-Tool-Moneta
_moneta__nibble/nibble_76.so
If you want to upload more than 200 files, pass '--input-file-count
-limit '.
```

It means you have too many files in your local directory. You can delete some of them or do what the error says and pass a large value to `--input-file-count-limit` (although that will make running jobs quite slow for you). A good candidate for deletion is your `.cfiddle` folder. You remove but you may need re-run some of your `build()` cells afterwards:

In [ ]:
```
#!rm -rf .cfiddle
```

10. If you get this

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

This means that cfiddle's `code()` function detected a change to the file mentioned (`foo.cpp` in the error above) and it is refusing to overwrite it, so it doesn't destroy your changes. This can happen, for example, if you've edited the file in VSCode. You can either

```
1. Delete the file (`rm foo.cpp`) and re-execute the cell.
2. Delete the file, and replace the argument to `code()` with the n
ew contents of the file, so you can keep editing in Jupyter noteboo
k.
3. Keep editing the file externally and replace the call to `code()
` with the file name.
```

6.7.6 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):

6.7.7 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).

6.7.8 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. Save frequently.

6.7.9 Saving Your Work and Making Sure Your Connected to the Server

In theory, Jupyter Notebook saves automatically. However, a few things can go wrong:

If your Datahub server shuts down, you can still edit your notebook, but you won't be able to save it.

You can tell your server has stopped if there's a red box in the upper right that says "Not Connected":

If this happens, you should stop working, restart your server and reload the lab.

In any case, it's a good idea to save frequently:

6.7.10 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.

7 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**.

8 Grading

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

Part	value
Reading quiz	3%
Jupyter Notebook	45%
Programming Assignment	50%
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.

9 New Tools

9.1 Measuring Cache Performance with Performance Counters

We'll use *performance counters* to measure cache performance on our bare metal machines in the cloud. We'll use the same tools we've used so far to measure `IC`, `CPI`, and `CT`.

Performance counters are hardware components in the CPU that measure performance-related events - like instructions executed, cache misses, branch mispredictions, and so on. In this lab, we will use performance counters extensively to characterize the behavior of programs.

Let's measure the cache behavior of `array` from the example above. Here's the same example again:

In [ ]:

```
#t = fiddle("array.cpp", function="array", run=["perf_count"], opt="-O3",
array = build(code(r"""
#include"cfiddle.hpp"
#include<iostream>
#include<cstring>
#include<util.hpp>

extern "C"
uint64_t* array(uint64_t size) {
    uint64_t * data = new uint64_t[size]();
    start_measurement();
    for(unsigned int i = 0; i < size; i++) {
        data[i] = -i;
    }
    end_measurement();
    return data;
}

"""), arg_map(OPTIMIZE="-O4", DEBUG_FLAGS="-g0"))
print(array[0].lib)
compare([array[0].source("array"),array[0].cfg("array")])
```

Question 1 (Completeness)

If `size == 8192`, how many total cache misses do think will occur during the execution of `array()`? (assume the cache is empty to begin with and that cache lines are 64 bytes).

Misses for size 8192:

Average misses per instruction in this loop:

Show Solution

9.2 Tensors

In this lab and future labs, we'll be using a data structure called a *tensor*. Tensors are a hot topic nowadays (you may have heard of the TensorFlow programming system from Google that accompanies their Tensor Processing Unit specialized processor), but at their heart they are not complicated: They are just multi-dimensional arrays.

We'll be using a 4-dimensional tensor data type called `tensor_t`. It's defined in `./tensor_t.hpp`. Here's the key parts of the source code:

In [ ]:
```
t = build("tensor_t.hpp")[0]
display(t.source(show=("START_CONSTRUCT", "END_CONSTRUCT")))
display(t.source(show=("START_GET", "END_GET")))
```

Note that `tensor_t` is a template, so we can have tensors of many different types: `tensor_t<int>`, `tensor_t<float>`, etc.

Although `tensor_t` is 4-dimensional, it stores its contents in a one-dimensional array of `T` called `data`. The `get()` method maps coordinates into that linear array and returns a reference to the corresponding entry. Since `get()` returns a reference, we can say things like:

```
tensor_t<float> t(10, 10, 10, 10);
t.get(1,2,3,4) = 4.0;
float s = t.get(1,2,3,4);
```

Question 2 (Correctness - 2pts)

Given the following `tensor_t` declarations and accesses, compute the total size of the tensor in element and bytes, and the index (starting at 0) of the corresponding data elements in the `data` array.

1. `tensor_t<uint32_t>(3,5,6,7)`
   A. Total elements:
   B. Total bytes:
   C. `get(1,0,0,0)`:
   D. `get(0,1,0,0)`:
   E. `get(0,0,1,0)`:
   F. `get(0,0,0,1)`:
2. `tensor_t<double>(2,4,8,16)`
   A. Total elements:
   B. Total bytes:
   C. `get(1,3,2,4)`:
   D. `get(2,2,1,7)`:

9.3 The Miss-Machine

We're going to need to generate some cache misses very efficiently in this lab. It's actually pretty hard to reliably create cache misses efficiently, especially if you want to out-smart modern cache hardware mechanisms like prefetchers. We also need to precisely control the amount of memory it

touches. You could use a random number generator, but even the simplest ones (like `fast_rand()`) are pretty computationally intensive. Instead, we are going to use a clever data structure that I don't think has an official name, so I'm going to call it the "miss machine".

The miss-machine is a circular linked list. The links are allocated in a block (of the size of memory footprint we want) and then formed into a circular linked list. Then, you set the `next` pointer so that the list jumps around at random, hitting every link, and eventually returning to the start. Then, if you want cache misses, you make the footprint bigger than your cache, and you just traverse the list for as long as you like. The sequence of addresses is (as near as the cache can tell) random, and the code required is extremely small -- just a few instructions are sufficient.

Here's an implementation. Read through the code and comments carefully.

In [ ]:

```
miss_machine = build(code(r"""
#include<cstdint>
#include<vector>
#include<algorithm>
#include<iostream>
#include"util.hpp"
#include"cfiddle.hpp"
#include <new>

#define BYTES_PER_CACHE_LINE 64

struct MM {
    struct MM* next;  // I know that pointers are 8 bytes on this machine
    uint8_t junk[BYTES_PER_CACHE_LINE - sizeof(struct MM*)]; // This forc
};

extern "C"
struct MM * miss(struct MM * start, uint64_t count) {
    for(uint64_t i = 0; i < count; i++) { // Here's the loop that does th
        start = start->next;
    }
    return start;
}


extern "C"
uint64_t* miss_machine(uint64_t footprint_bytes, uint64_t access_count) {

    const uint array_size = footprint_bytes/sizeof(struct MM);

    auto array = new struct MM[array_size]();

    // This is the clever part: 'index' is going to determine where the p
    std::vector<uint64_t> index;
    for(uint64_t i = 0; i < array_size; i++) {
        index.push_back(i);
    }
    // Randomize the list of indexes.
    std::random_shuffle(index.begin(), index.end());

    // Convert the indexes into pointers.
    for(uint64_t i = 0; i < array_size; i++) {
        array[index[i]].next = &array[index[(i + 1) % array_size]];
    }

    MM * start = &array[0];
    flush_caches();
    enable_prefetcher(0);
    start_measurement();
    start = miss(start, access_count);
    end_measurement();
    return reinterpret_cast<uint64_t*>(start); // This is a garbage value
}

"""), arg_map(OPTIMIZE="-O1", CXX_STANDARD="-std=gnu++17",DEBUG_FLAGS="-g
```

```
display(miss_machine[0].source())
```

Here's the CFG and assembly for `miss()`:

In [ ]:
```
display(miss_machine[0].cfg("miss"))
```

Question 3 (Completeness)

Based on this code, how many misses per instruction would you expect this code to produce? We'll call this metric `L1_MPI`. It's just the number of L1 cache misses divided by `IC`.

`L1_MPI`:

Show Solution

Here's the results (the `perf_counters` option specifies which performance counters to collect data for):

In [ ]:
```
miss_machine_data = run(miss_machine, "miss_machine",
                        arg_map(footprint_bytes=1024*1024, access_count=1(
                        perf_counters=["L1-DCACHE-LOAD-MISSES", "PERF_COU
```

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

A few things are notable: `L1_MPI` and `CPI` are the largest numbers we have seen to date. The miss machine works (But it could work better).

Question 4 (Optional)

Modify the fiddle above to get `L1_MPI` above 75%. Think about how you would modify the assembly to increase `L1_MPI`. How can you make that happen *without* manually editing the assembly?

10 Thinking in Cache Lines

One of the central issues in cache-aware programming (i.e., crafting your code to make efficient use of the memory hierarchy) is understanding "how much" memory your program (or part of it) is using and "how big" a particular data structure (e.g., an array) is, and how often your program accesses memory. It turns out there are a surprising number of ways to measure these things and the answers can be non-intuitive. Not only that, the most obvious ways to measure these characteristics are not the most useful -- if you're interested in fully-utilizing your memory system.

So let's take a look at three questions and see how to answer them in a cache-aware way.

10.1 How big is my data structure?

10.1.1 Primitive Types

Let's start simple: How many bytes do each of the primitive C++ data types occupy? Here's some code to check. It uses C's `sizeof` operator (it's not technically a function. Why?) that tells you how many bytes something occupies. The types listed are all the C++ primitive types.

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

extern "C"
void primitive_sizes() {
    std::cout << "\n";
    std::cout << "sizeof(char) = " << sizeof(char) << "\n";
    std::cout << "sizeof(short int) = " << sizeof(short int) << "\n";
    std::cout << "sizeof(int) = " << sizeof(int) << "\n";
    std::cout << "sizeof(long int) = " << sizeof(long int) << "\n";
    std::cout << "sizeof(long long int) = " << sizeof(long long int) << "
    std::cout << "sizeof(float) = " << sizeof(float) << "\n";
    std::cout << "sizeof(double) = " << sizeof(double) << "\n";
    std::cout << "sizeof(long double) = " << sizeof(long double) << "\n";
    std::cout << "sizeof(int8_t) = " << sizeof(int8_t) << "\n";
    std::cout << "sizeof(int16_t) = " << sizeof(int16_t) << "\n";
    std::cout << "sizeof(int32_t) = " << sizeof(uint32_t) << "\n";
    std::cout << "sizeof(int64_t) = " << sizeof(int64_t) << "\n";
    std::cout << "sizeof(int64_t*) = " << sizeof(int64_t*) << "\n";
    std::cout << "sizeof(void*) = " << sizeof(void*) << "\n";
}
"""))

with local_execution():
    run(sizeof, "primitive_sizes")
```

The types with a number in their names (like `uint64_t`) let you specify specific sizes (in bits) you'd like to use. They are defined in the `cstdint` header. The [C/C++ standard (https://en.wikipedia.org/wiki/C_data_types)](https://en.wikipedia.org/wiki/C_data_types) doesn't give specific sizes for 'int', 'long int', etc.

Instead, it places some constraints on what possible values might be. This is a design "bug" in the language. I don't know of any other modern languages that leave the bit widths of primitive types unspecified. One reason is that it hurts portability. The other, is that it makes cache-aware programming harder (as we shall see).

10.1.2 Structs

Let's try something more complicated. Look at the code below and answer this question. Then, run the code:

In [ ]:

```
structs = build(code(r"""
#include<cstdint>
#include<iostream>

struct struct_1 {
    uint32_t a;
};

struct struct_2 {
    uint32_t a;
    uint32_t b;
};

struct struct_3 {
    uint32_t a;
    uint8_t b;
};

struct struct_4 {
    uint64_t a;
    uint8_t b;
};

struct struct_5 {
    uint8_t a;
    uint8_t b;
};

struct struct_6 {
    uint64_t a;
    uint8_t b;
} ;

struct struct_7 {
    uint64_t a;
    uint8_t b;
    uint8_t c;
} ;

struct struct_8 {
    uint8_t a;
    uint64_t b;
    uint8_t c;
} ;


extern "C"
void struct_size() {
    std::cout << "\n";
    std::cout << "sizeof(struct_1) = " << sizeof(struct_1) << "\n";
    std::cout << "sizeof(struct_2) = " << sizeof(struct_2) << "\n";
    std::cout << "sizeof(struct_3) = " << sizeof(struct_3) << "\n";
    std::cout << "sizeof(struct_4) = " << sizeof(struct_4) << "\n";
    std::cout << "sizeof(struct_5) = " << sizeof(struct_5) << "\n";
    std::cout << "sizeof(struct_6) = " << sizeof(struct_6) << "\n";
    std::cout << "sizeof(struct_7) = " << sizeof(struct_7) << "\n";
```

```
    std::cout << "sizeof(struct_8) = " << sizeof(struct_8) << "\n";
}
"""))
```

Question 5 (Completeness)

Predict how many bytes of memory the structs below will occupy.

struct	sizeof()
struct_1	
struct_2	
struct_3	
struct_4	
struct_5	
struct_6	
struct_7	
struct_8	

In [ ]:

```
with local_execution():
    run(structs, "struct_size")
```

What's going on?!?!

What's going on is *data alignment*. If a memory address, A, is n-byte aligned, then $A\mod{} n = 0$. A particular value is "width-aligned" if it is aligned to its own size in bytes. So, a width-aligned `uint64_t` would reside at an address that is a multiple of 8 bytes.

In most architectures, width-aligned access is faster than non-width-aligned access. In some architectures, the ISA does not directly support non-width-aligned access (i.e., you can't load a 64-bit value from an address that is not a multiple of 8), because it requires extra hardware and complexity (e.g., if the architecture allows unaligned, multi-byte values, then a single load can access *two* cache lines instead of one). Instead, they require the compiler to implement these accesses with loads and shifts.

The strangeness in the outputs of the `sizeof` is a product of this. The C/C++ standards require that the members of a struct be stored *in the order they are declared*. Modern compilers "pad" members of the struct to enforce width-alignment. For this purpose, it's assumed that the struct starts address 0.

For example, consider `struct_8` above. It's laid out like so:

Byte	
0	`a`
1	unused
2	unused
3	unused
4	unused

Byte	
5	unused
6	unused
7	unused
8	b
9	b
10	b
11	b
12	b
13	b
14	b
15	b
16	c
17	unused
18	unused
19	unused
20	unused
21	unused
22	unused
23	unused

If this seems inefficient... it is. Or at least, it's a trade-off. It's better for performance this way and memory is plentiful. Really, though, the programmer should re-order the fields of the struct to allow for a more efficient layout. See if you can re-arrange the fields in `struct_8` to make it fit in 16 bytes.

10.1.3 Arrays

Compared to structs, arrays are pretty well-behaved. The size of an array is just the number of elements in the array multiplied by the size of the struct:

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

struct struct_8 {
    uint8_t b;
    uint64_t a;
    uint8_t c;
} ;


extern "C"
void array_size() {
    struct struct_8 _8[3];
    std::cout << "\n";
    std::cout << "sizeof(struct_8[3]) = " << sizeof(_8) << "\n";
}
"""))
with local_execution():
    run(array, "array_size")
```

10.2 How much memory does my code access?

Above, we measured the size of a data structure or array in bytes, and this makes sense for thinking about its size.

A related question is how much data does my program access. If you are interested in writing code that is cache-aware, thinking of data measured in bytes is not that useful. A better choice is to think about data measured in cache lines, because cache lines are the units of memory that the memory hierarchy transfers between caches.

So what is one cache line of memory? The seemingly obvious answer is that it is the number of bytes that a cache line holds. If that were the case, we could just divide the size of a structure by the cache line size. But there is more to it: Cache lines are width-aligned. That means that each cache line of memory starts at an address that is divisible by the cache line's size. This means that the number of cache lines a struct occupies *depends on its alignment*.

For example, let's assume our cache line size is 16 bytes, and `my_struct` is 16 bytes long. Here are two possible scenarios for `my_struct`. In the first, the beginning of `my_struct` is aligned to a cache line boundary, and `my_struct` occupies 1 cache line.

byte	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
cache line	0																1											
	my_struct																											

In the second scenario, `my_struct` is not aligned, and it occupies two cache lines:

byte	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

cache line	0								1							
		my_struct														

For the width-aligned case, if we access the whole struct, we will incur one compulsory cache miss. But in the second, non-aligned situation we will incur two.

We can modify the miss machine to illustrate this situation. The code is below. There are three differences:

1. I changed the name of the fields in `MM`. That's just to make the table below more readable.
2. `miss()` reads from two fields of `MM`. `a` at the beginning and `h` at the end.
3. We allocate `array` in a complicated way.

For `array`, instead of using `new`, we use `posix_memalign()` (https://man7.org/linux/man-pages/man3/posix_memalign.3.html) which lets you set the alignment of the allocated memory with its secord argument. We allocate `array` so that the elements it contains will be width-aligned. That is, `array % sizeof(MM) == 0`.

Having carefully, allocated aligned memory, we then unalign it:

```
array = reinterpret_cast<A*>(reinterpret_cast<uint64_t*>(array) + a
rg1);
```

The line above shifts `array` by `arg1` 8-byte words. So, if `arg1 = 4`, then `array % sizeof(MM) == 32`. You can read about `reinterpret_cast` (https://en.cppreference.com/w/cpp/language/reinterpret_cast), but you should use it sparingly (They gave it a hard-to-type name on purpose. Really!).

So, if we call this function with multiple values of `arg1`, we'll have something like this:

8-byte word	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
64-bytes cache line	00								01								02											
arg1 == 0	a	array[0]						h	a	array[1]						h	a	array[2]						h	a			
arg1 == 1		a	array[0]					h	a	array[1]						h	a	array[2]						h	a			
arg1 == 2			a	array[0]				h	a	array[1]						h	a	array[2]						h	a			
arg1 == 3				a	array[0]			h	a	array[1]						h	a	array[2]						h	a			
arg1 == 4					a	array[0]		h	a	array[1]						h	a	array[2]						h				
arg1 == 5						a	array[0]		h	a	array[1]					h	a	array[2]										
arg1 == 6							a	array[0]		h	a	array[1]				h	a	array										
arg1 == 7								a	array[0]			h	a	array[1]			h	a	ar									

When `arg1 % 8 = 0`, a `MM` occupies one cache line. When `arg1 % 8` is anything else, it occupies 2. And since `miss()` accesses `h`, we will access both lines.

Let's run the code with a range of `arg1` values.

In [ ]:

```
misaligned_machine = build(code(r"""
#include<cstdint>
#include<vector>
#include<algorithm>
#include<iostream>
#include"util.hpp"
#include"cfiddle.hpp"
#include <new>
#include <cassert>

struct MM {
    struct MM* next;
    uint64_t b;
    uint64_t c;
    uint64_t d;
    uint64_t e;
    uint64_t f;
    uint64_t g;
    uint64_t h;
};

extern "C"
struct MM * miss(struct MM * start, uint64_t count) {
    uint64_t sum = 1;
    for(uint64_t i = 0; i < count; i++) { // Here's the loop that does th
            sum += start->h;
            start = start->next;
    }
    return start + sum;
}

extern "C"
void miss_machine(uint64_t footprint_bytes, uint64_t access_count, uint o

    const uint array_size = footprint_bytes/sizeof(struct MM);

    struct MM * array = NULL;
    int r = posix_memalign(reinterpret_cast<void**>(&array), sizeof(struc
    assert(r == 0);
    array = reinterpret_cast<MM*>(reinterpret_cast<uintptr_t>(array) + of

    std::vector<uint64_t> index;
    for(uint64_t i = 0; i < array_size; i++) {
        index.push_back(i);
    }
    // Randomize the list of indexes.
    std::random_shuffle(index.begin(), index.end());

    // Convert the indexes into pointers.
    for(uint64_t i = 0; i < array_size; i++) {
        array[index[i]].next = &array[index[(i + 1) % array_size]];
    }

    MM * start = &array[0];
    flush_caches();
    enable_prefetcher(0);
```

```
    start_measurement();
    miss(start, access_count);
    end_measurement();
}

"""), arg_map(OPTIMIZE="-O1", DEGUG_FLAGS="-g0"))
```

Here's the assembly for `miss()`:

In [ ]:
```
display(misaligned_machine[0].cfg("miss"))
```

Question 6 (Completeness)

For what values of `offset` will `array` be cache line-aligned?

Based on the exercises about the miss machine and the code above, predict the `L1_MPI` for `miss()` when `array` is aligned and unaligned.

`offset` values:

aligned `L1_MPI`:

unaligned `L1_MPI`:

Show Solution

10.3 Language Support

The experiments above show that alignment and struct layout can affect the performance of toy programs, but do these kinds of details really make any difference in real code? They do!

How can we tell? Two ways:

1. We could go and write some high-performance systems and watch as alignment-related performance problems appear.
2. We can look at the tools that languages provide to deal with these issues.

#1 is lots of fun and I recommend it, but it takes a long time and we only have one quarter. But there are three programming assignments left...

Let's look at #2. For a long time, compiler writers and language designers have realized that memory layout and alignment is important, so they have provided support in the language to help programmers deal with this.

10.3.1 Struct Initialization In C

Since the early versions of C, you could initialize a struct like this:

```
struct Foo {
    char a;
    int b;
    char c;
};

Foo foo = {7,4,3};
```

Which will set `foo.a = 7`, `foo.b = 4`, and `foo.c = 3`. We'll call this "positional initialization" or PI. I think PI is super-unreadable, but it has an even bigger problem: If you change the layout of `Foo` to improve its cache efficiency, you have to update every static initialization of a `Foo` and there's no way for the compiler to tell if you missed one. So the C99 standard gave us "designated initialization":

```
Foo foo = {.a = 7, .b=4, .c=3};
```

This feature appeared as a gcc extension before it showed up in the C standard, and it's main use case was making easier to adjust struct layouts for memory-efficiency reasons. For instance, in the unaligned access example above, we could move `MM.h` to be near `MM.next` and reduce the effects of poor alignment.

10.3.2 C++ Object Alignment

Usually, the compiler can do a good job of aligning your structs, but there are cases where the programmer needs to force a structure to be aligned to a particular size. The old fashioned way to do this was by wrapping `malloc()` (the C version of `new`) to create a new memory allocation function that could return memory at the desired alignment. Writing such a function is a moderately interesting programming exercise.

Since 2001, C has had support for aligned allocation through `posix_memalign()`.

C++ has a more elegant solution and allows you to annotate the type with its alignment requirement using `alignas()`:

```
struct alignas(32) Foo {
    char a;
    int b;
    char c;
};
```

Which guarantees that all instance of `Foo` will be aligned to 32 bytes.

For both `posix_memalign()` and `alignas()`, the alignment value must be a power of 2. Other alignments are not generally useful.

11 Latency and Bandwidth

Latency and bandwidth are two fundamental measures of memory performance.

- Latency -- in seconds (or sometime cycles) -- measures the time it takes for a single memory access to complete.
- Bandwidth -- measured in bytes, megabytes, or gigabytes per second -- is the amount of data the processor can access per unit time.

Caches can improve both latency and bandwidth: The L1 cache has lower latency and higher bandwidth than the L2. The L2 out-performs the L3 on both metrics, and the L3 is better than DRAM on both metrics.

Let's start by taking some baseline measurements of the bandwidth and latency capabilities of our test machine. We'll start with DRAM.

11.1 Measuring DRAM Latency and Bandwidth

Before we measure DRAM's performance, we should be precise about what exactly we mean by "latency" and "bandwidth" for DRAM.

For DRAM latency, we mean the number of seconds it takes for a load instruction to retrieve data assuming that the load misses in *all* levels of the cache. This means that the memory request "goes all the way to memory" and has to come all the way back.

For DRAM bandwidth, we mean the maximum number of bytes the processor can read or write per second from DRAM rather than from any of the caches. This means that the only bytes that count for DRAM bandwidth are those read or written as part of a cache miss.

Measuring these two values is surprisingly difficult and would make a very good (and pretty challenging) programming assignment for a graduate-level architecture course. It's made more difficult in modern machines by the presence of multiple processors, multiple DRAM interfaces, multiple caches, and multiple cores. We'll discuss some of this complexity in more detail in future labs when we address the impacts of parallelism, but for now we will stick to the simplest version of these questions.

Given the difficulty of measuring these values, we will rely on Intel's Memory Latency Checker (https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html). It can perform a huge array of measurements, but we'll just do two:

In [ ]:
```
!cse142 job run --take NOTHING --lab caches 'mlc --bandwidth_matrix; mlc
```

The output is a little verbose, but you should see two numbers that look like measured values rather than documentation. They are both for 'Numa node' 0. The first number is the bandwidth in

MB/s. The second is latency.

Question 7 (Completeness)

Fill in the table below with the values you measured.

metric	value
Latency (ns)	
Latency @ max clock rate (cycles, for our processor)	
Latency @ min clock rate (cycles, for our processor)	
Bandwidth (GB/s)	

Show Solution

Let's see what kind of hardware is providing this kind of performance. To do this we'll use `dmidecode`, a utility to read the systems DMI table. The DMI table is populated by the system's BIOS (the software runs immediately after you turn the computer on) to describe the hardware available. DMI stands for "Desktop management interface" which is may be the least descriptive name for anything we'll use in the course. Everyone just calls it the "DMI table".

Anyway, `dmidecode` will dump all sort of interesting information about the system, but we'll pass `-t memory` to just ask it about the DIMM memory modules installed. We'll also use grep to filter out some extraneous information:

In [ ]:
```
!cse142 job run --take NOTHING --lab caches 'dmidecode -t memory > dmi_ou
!grep -v 'Not Specified\|Unknown\|None' dmi_output.txt
```

There's a lot of information here, but let's walk through it: Each entry in a DMI table has a type and a unique ID number or "Handle". Type 0x17 is a slot for a DIMM. Type 0x11 is a "memory array" (for me it shows up as Handle 0x0011). Our system has one memory array with 4 DIMM slots (Handles 0x0020-0x0023), but only one of them (0x0021 for me) is populated.

We can see that this module is 16GB in size, is manufactured by Micron, runs at a supply voltage of 1.2V, is DDR4, and runs at 2667MT/s.

MT/s stands for mega-transfers per second and measures how often data flows across the 64-bit memory bus.

Question 8 (Completeness)

What is the peak memory bandwidth of our system? What fraction of the peak bandwidth was `mlc` able to attain?

metric	value
Peak bandwidth	
`mlc` % of peak	

Show Solution

11.1.1 Is DRAM Fast?

60ns is an almost unimaginably short period of time, and 18GB/s seems like a lot of data, but we need to think about this relative to the processor. If you look back at the `CPI` measurements you collect for Lab 1, you'll probably find CPI values as low as 0.5. At 3.6GHz, that means our processor is executing around 7 billion instructions per second or, on average, one instruction every 0.166ns. So 60ns is about 360 instructions.

For bandwidth, the situation is not great either. On average about 20% of instruction access memory and every instruction must be loaded from memory to execute. The average length of an x86 instruction is about 2 bytes and for simplicity, let's assume all memory accesses are 4 bytes. That means that, on average, each instruction execute needs (on average) `2 + 0.2 * 4 = 2.8` bytes of memory.

Let's compare that to what our measurements show: 18GB/s divided by 7 Billion instructions per second is about 2.6 bytes per instruction. Not far off! However, recall from lab one that our processor actually has 6 cores! And the memory bandwidth from it's single memory channel will be shared among all of them.

We should also keep in mind that our machines are memory-poor: Campus bought them to be as cheap and compact as possible. So they are physically small (so they only have 4 memory slots) and campus only ordered one DIMM for each.

We will need a way to get higher bandwidth and lower latency.

11.2 Measuring Cache Latency and Bandwidth

Caches reduce latency and increase bandwidth. Let's refresh our memory about the machine we are running on and see how large this effect is.

11.2.1 The Caches we Have

Run the cell below remind yourself how large our L1, L2, and L3 caches are.

In [ ]:
```
!cse142 job run --take NOTHING  --force lscpu
```

The cache sizes are the *total* size for the whole CPU, which has 6 cores. So L1 caches are 32 kilobytes, the L2 caches (also one per core) are 256 kilobytes, and the single L3 is 12 megabytes.

11.2.2 Cache Latency

One cool thing we can do with the miss machine is use it to measure cache latency (and the cache size, if we didn't already know it). To do this, we'll run the miss machine with larger and larger memory footprints and then measure the latency. For footprints that fit in the L1 cache, the average latency of these accesses is the L1 cache latency. For arrays bigger than the L1 but that fit in the L2, it's the L2 cache latency. Likewise, for L3 and main memory.

```
The experiment is in the cell below.  `exp_range()` generates a seq
uence of footprint that increases exponentially.  It takes a little
while to run.
```

In [ ]:
```
stairs = run(miss_machine, "miss_machine", arg_map(footprint_bytes=exp_ra
                                                   access_count=10000
plotPE(df=stairs.as_df(), what=[("footprint_bytes", "ET")], logx=2, logy=
```

The "stair step" pattern you see is the jump in latency as data accesses start being served out of the slower and slower layers of the memory hierarchy. The steps occur roughly when we expect them to: 32kB (L1 to L2), 256kB (L2 to L3), and 12MB (L3 to DRAM). For larger sizes things get noisier so the "steps" aren't as crisp.

Here's the raw data:

In [ ]:
```
display(stair_steps.as_df())#[["size_bytes", "size_MB", "latency_ns"]])
```

Question 9 (Correctness - 4pts)

Based on this graph and the data, give the latencies for each of the caches.

Level of the memory hierarchy	latency (ns)	latency (Cycles)
L1		
L2		
L3		
Main memory		

Happily, both `mlc` and `lat_mem_rd` agree on the main memory latency.

We can also check how our measurements compare to the processor specifications. For the L1 cache, Intel has disclosed that the minimum latency for the L1 read is 4 cycles. That should match the value you computed above.

As far as I know, Intel has not released a similar value for the L2 and L3.

11.2.3 Cache Bandwidth

We don't have a handy tool for measuring cache bandwidth. Maybe we'll build one as a future programming assignment...

We do however, have some technical details about the processor, so we can calculate what the peak (maximum attainable) cache bandwidth should be.

Skylake processors (of which our CPU is an example) can execute two 64-byte loads per cycle and one 64-byte store per cycle. At 3.5GHz, this works out to a bandwidth of `2*64*3.5e9 = 417GB/s` for loads and `213GB/s` for stores for a total 625GB/s.

The L2 cache provide one 64-bytes load *or* store per cycle for `213GB/s.`

I don't have any information about the L3.

11.2.4 Are the Caches Fast?

They are certainly better than DRAM -- the L1's latency is `61/1.147 = 52.3` times lower than DRAM, and bandwidth (in our system with one bank of DRAM) is `625/18.8 = 33` times higher. Not only that, remember that we have six cores, so the total L1 cache bandwidth is `625GB/s * 6 = 3.7TB/s` .

However, the minimum load latency -- 4 cycles -- is still pretty high, considering that it's 4 times longer than the latency for a simple integer arithmetic operation (add, sub, etc.). That means that having lots of load instructions can have a significant impact on CPI -- remember that unoptimized code from Lab 2 with all accesses to local variables on the stack? The 4 cycle L1 latency is part of why `-O0` is so slow.

Caches are also slow compared to the register file. Skylake's register files have a total bandwidth (at 3.5GHz) of something like 5TB/s (although it's very hard to fully utilize it.) per core (or 30TB/s across all 6 cores). The register file latency is a little hard to quantify in a way that is comparable to cache latency, but 1/2 a cycle is a reasonable approximation.

The register file is also quite large: The total size is about 1kB. It's reasonable to think of the register file as a software-managed, "L0" cache.

12 Locality In Space and Time

Caches improve program performance by exploiting the fact that memory accesses are not random: Programs access memory in patterns and there is a lot of repetition in what programs do and a lot of similarity how programs behave at different times. This is not surprising since, programs are written languages with constructs like loops and function calls: These constructs naturally give rise to patterns because they cause the same code to execute repeatedly.

12.1 Spatial Locality

As you should recall from lecture, *spatial locality* is a property of a program (or part of a program) where it accesses memory locations nearby locations it has accessed recently. Let's see how spatial locality can affect performance.

Here's a simple test program that accesses a 1-dimensional tensor at different "strides". The "stride" of a sequence of accesses is just the distance between consecutive accesses. So, "stride 1" means accessing each element, and "stride 2" means accessing every other element. The outer loop ensures that the total number of accesses the loop perform remains the same, regardless of stride size).

Examine the code below, and then answer these question:

In [ ]:
```
stride = build(code(r"""
#include"tensor_t.hpp"
#include<cstdint>

extern "C"
void stride(uint size, uint64_t stride) {
    tensor_t<uint32_t> t(size,1,1,1);
    start_measurement();
    for(uint i = 0; i < stride; i++) {
        for(uint x = 0; x < size; x += stride) {
            t.get(x,0,0,0) = x;
        }
    }
    end_measurement();
}
"""), arg_map(OPTIMIZE="-O1", DEBUG_FLAGS="-g0"))

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

Question 10 (Completeness)

How many misses-per-instruction would you expect for `stride == 1` ? How about `stride == 8` ? Assume `size` is very large and that cache lines are 64 bytes.

`stride == 1` MPI:

`stride == 8` MPI:

`stride == 32` MPI:

Show Solution

Let's see if those predictions match reality. Run the cell below. It'll run the code above.

In [ ]:
```
stride_run = run(stride, "stride",
                      arg_map(size=1024*1024*128, stride=exp_range(1,64,2)
                      perf_counters=["L1-DCACHE-LOAD-MISSES",  "PERF_COUNT
```

In [ ]:
```
stride_data = PE_calc(stride_run.as_df())
display(stride_data[["stride", "L1_cache_misses", "L1_MPI"]])
plotPE(df=stride_data, lines=True, what=[("stride", 'L1_MPI')])
```

Question 11 (Completeness)

Do the measurements match our predictions? Does anything seem surprising about the results?

Of course, it's not really the misses we are worried about -- it's the impact on performance.

Recall what happens on a cache miss: Instead of accessing the data in the cache, the processor must go down the memory hierarchy. In the worst case, this means going to main memory, which can easily take 100s of cycles. Let's imagine it's 200 cycles. What will the impact be on CPI?

Question 12 (Correctness - 2pts)

If a cache miss increases memory instruction latency by 200 cycles, predict the ratio of CPI with a `stride == 1` and with `stride == 64` (based on the data above)?

Here's how it actually played out:

In [ ]:
```
display(stride_data)
plotPE(df=stride_data,lines=True, what=[("stride", 'L1_MPI'),("stride", '
```

There's no particular reason to expect these number to match your calculation in the question above. We don't know exactly how long an L1 cache miss takes. 200 cycles is a guess, but in both cases the large impact of cache performance on execution time should be clear.

12.2 Temporal Locality

We'll discuss temporal locality in the next lab.

13 Recap

In this lab, you have seen how important memory alignment is and how the compiler lays out structs to enforce it.
You have measured spatial locality and seen how its presence or absence can affect performance.

14 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.

14.1 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.

14.2 Reading Quiz

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

14.3 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.

14.4 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 [3]:
```
from IPython.display import IFrame
IFrame('https://docs.google.com/forms/d/e/1FAIpQLSdEyaIDy52FLLUzQEXoJJmz7
```

Out[3]: