In [3]:
```
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 4: The Memory Hierarchy (Part II)

This lab is a continuation of the previous lab. While that lab focused on the basics of cache-aware programming and spatial locality, this lab will focus more on temporal locality and how you can modify your programs to maximize it.

As a reminder, between these two labs, 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 you want to learn *a lot* more about optimizing matrix multiply, try this paper:
https://www.cs.utexas.edu/~flame/pubs/GotoTOMS_revision.pdf
(https://www.cs.utexas.edu/~flame/pubs/GotoTOMS_revision.pdf)

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")
```

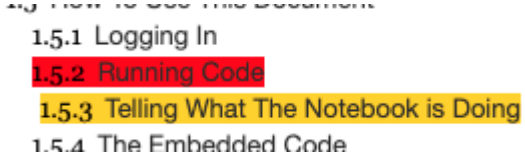
Code cells can also execute shell commands using the `!` operator. Try it below:

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 [5]:
```
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.

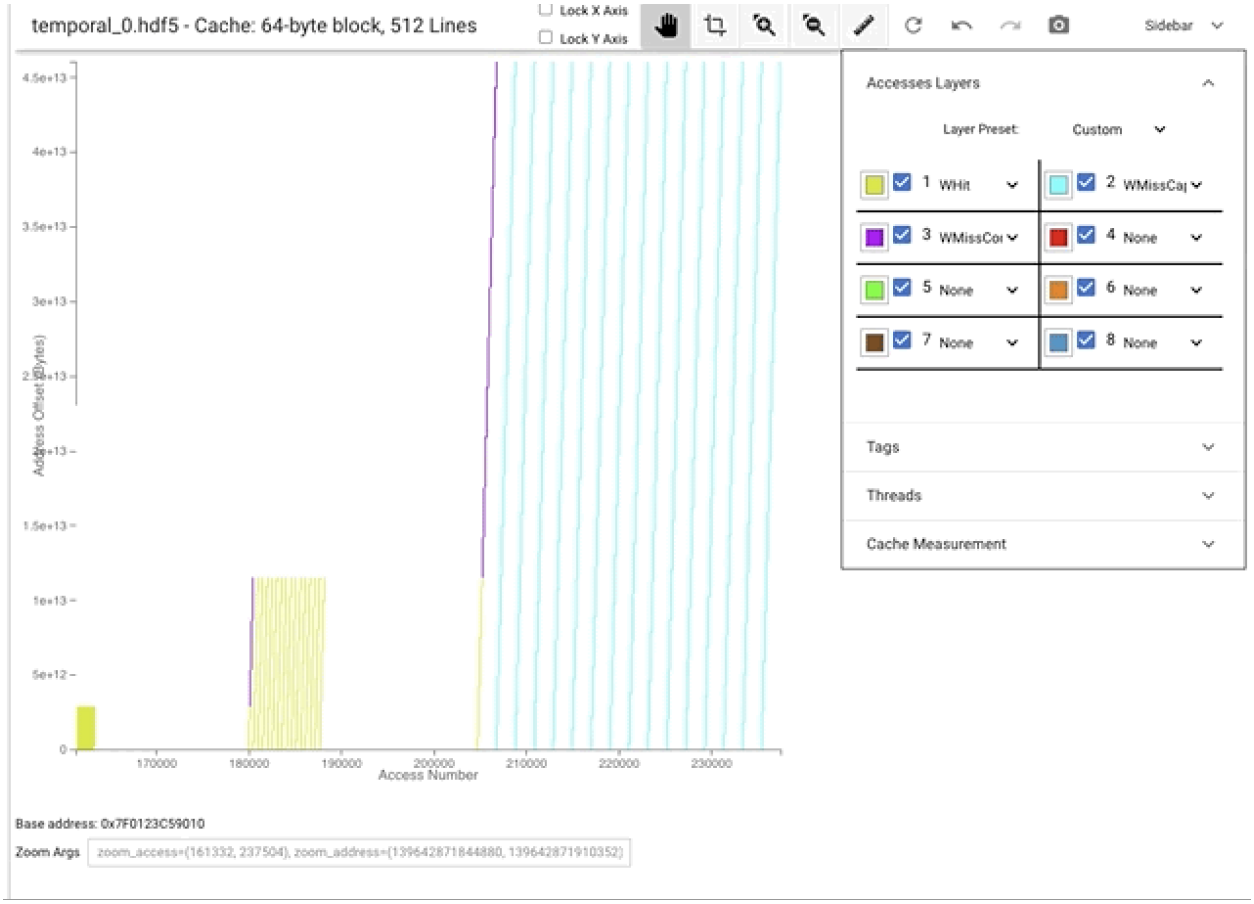
You'll follow the directions at the end of the lab to submit the lab write up and the programming assignment through gradescope.

Please check gradescope for exact due dates.

9 New Tools

The only new tool you'll be using in this lab is a new kind of graph that visualizes a "trace" of memory accesses made by a program. Traces get used a lot in computer system analysis. In this case, it's a memory trace which is just a list of all of memory accesses a program makes.
We generate the trace using [Intel's Pin binary instrumentation tool (https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html)](https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html), which can do all kinds of cool things. In our case, it injects code into the running executable that writes the address of each load and store to a file along with some metadata (e.g., which thread made the access, whether it was read or write).

Here's an example:



The horizontal axis is the "access number". The first memory access in the program has access number 1, the second has access number 2, and so on. The vertical axis is the relative address. The colors mean different things in different plots.

You might ask "Why are these graphs janky screen captures?" Well, the graphs were rendered with a very cool Jupyter Notebook extension called Moneta that some of the first students to take 142L wrote as a group independent study during the spring of 2020 (the first quarter of the pandemic). It was an awesome interactive tool that let you pan and zoom and measure memory traces. However, it suffered from two key flaws:

1. It was entirely too much for datahub to handle. It used vast amounts of memory and would routinely crash students' virtual machines.
2. It relied on some obsolete python libraries.

Moneta II is in the works to address these problems, but it's not ready yet. So, for now, we have screen caps.

10 Temporal Locality

In the last lab, we examined the notion of spatial locality in detail. Now, we will turn to temporal locality.

Temporal locality exists when a program accesses the same memory multiple times within a short time. Caches exploit temporal locality by holding on to data that has been accessed recently. If the processor accesses it again, the cache can provide it very quickly.

With spatial locality, it was pretty easy to predict the cache miss rate for a simple loop that performs stride-based accesses (see below). With temporal locality it is harder because of associativity and conflicts. Before we dive into that, let's have quick refresher about how caches work (if this is fuzzy, go back and the review the slides and/or readings).

When a memory operation (load or store) accesses a memory location, A, the cache breaks A's address into three parts:

tag	index	offset
the remaining bits	`log2(# of associative sets)`	`log2(cache line size)`

Together, the tag and the index of A are a unique name (or number) for the cacheline-sized (and cacheline size-aligned) piece of memory that contains A. The index of A tells the cache which associative set might contain that cache line.

The cache can then check that set to see if A is present. If it is, it's a hit. If not, it's a miss, and the cache will choose one of the lines in the set to evict to make room for A's cache line.

There are two important things to note:

1. A's cacheline is in the cache if and only if, it is in the associative set corresponding to its index (it can never be in another associative set).
2. There are many, many other cache lines that also "live" in A's associative set.

The L1 data cache in our processor is 32kB, with 64-byte lines, and it's 8-way set associative. So, there are 32,768/64 = 512 cache lines arranged in 512/8 = 64 associative sets. If the machine has 16GB of memory, it has 256-Million cache lines of main memory. So, there are about 4 million cache lines that "live" in each associative set. Clearly, there is plenty of opportunities for conflicts.

To see how temporal locality plays out in practice, here's some code that should look familiar from the last lab:

In [ ]:

```
stride = build(code(r"""
#include"util.hpp"
#include"tensor_t.hpp"
#include<cstdint>

extern "C"
void stride(uint size, uint64_t stride_size, uint reps) {
    tensor_t<uint32_t> t(size,1,1,1);
    flush_caches();
    start_measurement();
    for(uint k = 0; k < reps; k++) {
        for(uint i = 0; i < stride_size; i++) {
            for(uint x = 0; x < size; x += stride_size) {
                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")])
```

We are going to run it with a fixed stride of 16 elements (64 bytes -- our cache line size) and we will vary `size` between 1024 and 16,384 (16 * 1024). This corresponds to region of memory between 4kB and 128kB. Setting the stride to the cache line size ensures that our access stream has very little *spatial* locality, since every access will refer to a different cache line. The `flush_caches()` function invalidates the contents of all the caches -- a useful step for getting clear readings.

Question 1 (Completeness)

Given the conditions described above, estimate the *number of cache misses* that will occur for `size = 1024`, `size = 4096`, and `size = 16384`.

Cache misses for size = 1024:

Cache misses for size = 4096:

Cache misses for size = 16384:

Show Solution

Run the cells below to see how your prediction played out

In [ ]:
```
stride_data = run(stride,  function="stride", arguments=arg_map(stride_si
                  perf_counters=["L1-DCACHE-LOAD-MISSES",  "PERF_COUNT_HW_
# compare([t.source, t.cfg]) # Uncomment this line to see the code again.
```

In [ ]:
```
stride_data_calc = PE_calc(stride_data.as_df())
display(stride_data_calc)
plotPE(df=stride_data_calc,lines=True, what=[ ('size', 'CPI'), ("size", '
```

Question 2 (Completeness)

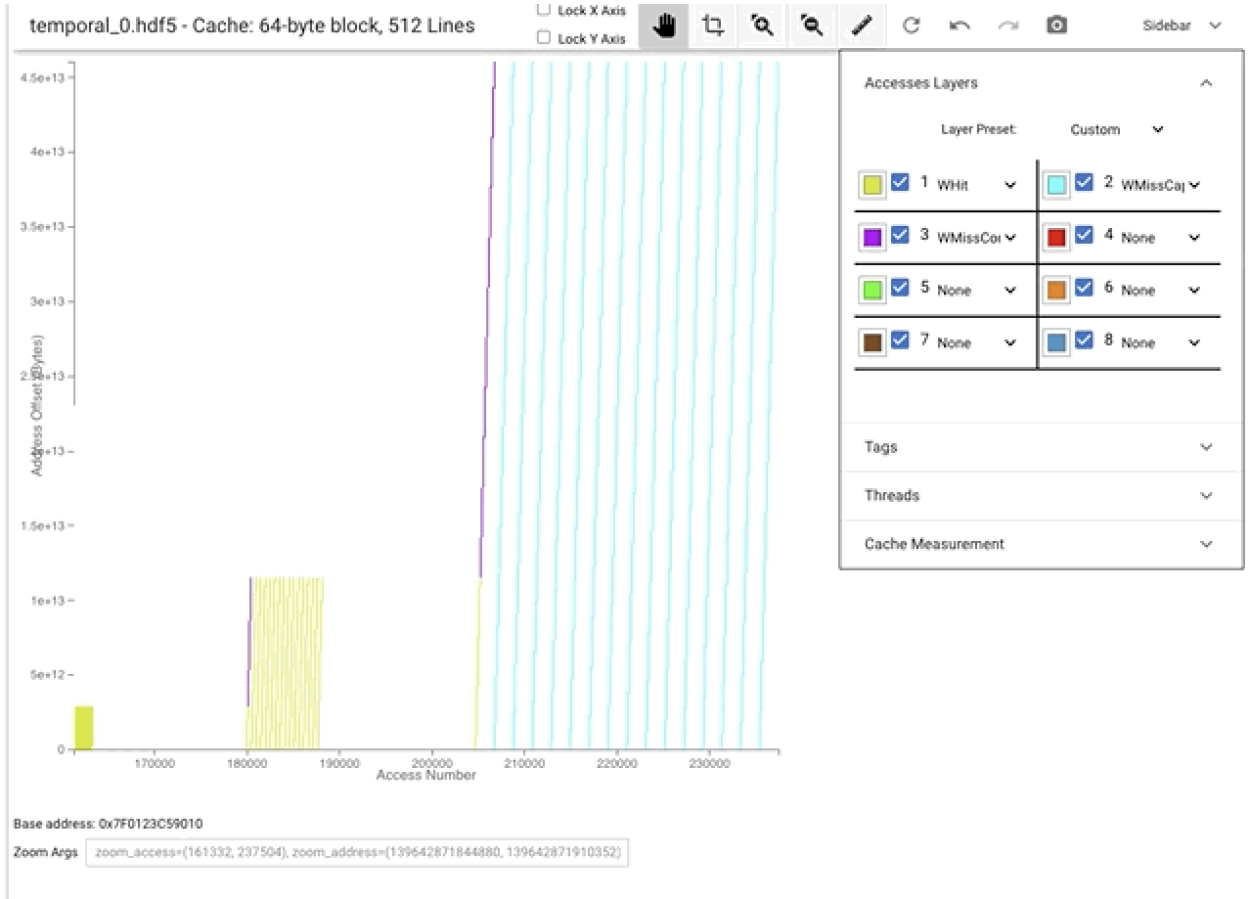
How well do your predictions match the results?

Show Solution

Question 3 (Optional)

Comment out the `flush_caches()` line in the code above. What happens? Why?

Here's a Moneta plot of what's going on during the execution of `stride()` 's inner loops for `size = 1024` , `size = 4096` , and `size = 16k` .



In the plots, compulsory misses are purple, hits are gold, and capacity misses are light blue.

The large blocks are the accesses for `size = 1024`, `size = 4096`, and `size = 16k`.

In each block, each upward-slanting line is one iteration through the outer loop (they are blurred together for 1024). As you can see, when the region of memory is larger than the cache, the whole block turns blue instead of gold.

Let's try another experiment and increase the stride by 4x to 64 element (256 bytes) while using `1024`, `4096`, and `16384` for `size`.

11 Working Sets

In lecture you heard about the "working set" of an application, and the notion of a working set is deeply tied to temporal locality. The working set is the *portion of memory that the program is currently using*. The connection between working sets and temporal locality lies in the word "currently" since that refers to a period of time. In essence, the working set is the set of cache lines that a program accesses repeatedly over a period of time.

One thing to note: Without reuse, there can be no temporal locality. A single access to a cache line has no temporal locality.

Generally speaking, there will be fewer cache misses (and performance will be faster) if the working set fits in the L1 cache (or failing that, in the L2 cache).

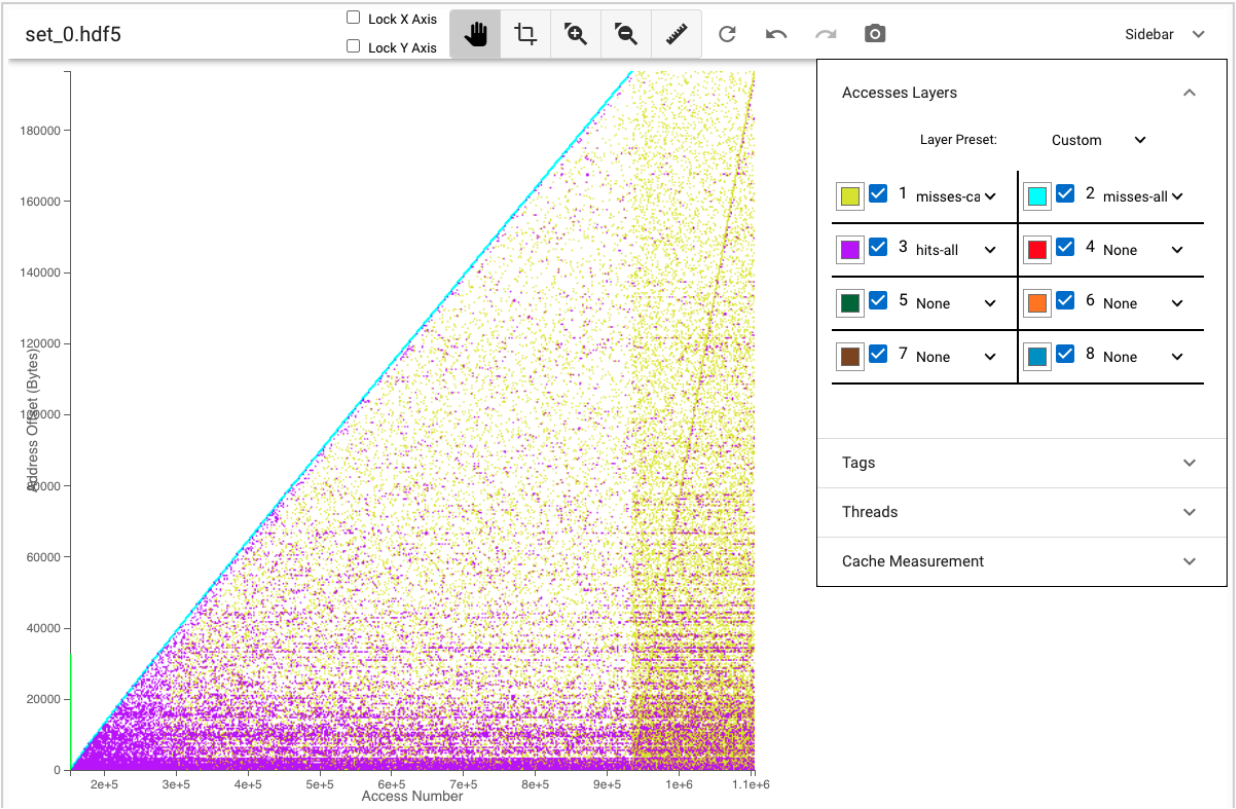
11.1 Examining `std::set`

To illustrate how working set size influences cache behavior, we'll use the `set` container object from the C++ standard template library. Internally, `set` is implemented as a red-black binary tree. The code below creates an `std::set` and then fills it with 4096 pseudo-random (and non-repeating) `uint64_t` values using `insert()` and then performs a bunch of queries with `find()`.

In [ ]:

```
cpp_set = build(code(r"""
#include"pin_tags.h"
#include"cfiddle.hpp"
#include<set>
#include<cstdint>

extern "C"
void working(uint64_t size) {
    auto s = new std::set<uint64_t>();
    uint64_t seed = 1;

    TAG_START("build", (void*)-1, 0, true);
    for(uint x = 0; x < size; x++) {
        auto t = fast_rand(&seed);
        s->insert(t);
        auto a = s->find(t);
        TAG_GROW("build",  &(*a), &(*a)+ 1);
    }
    TAG_STOP("build");

    seed = 1;
    start_measurement();
    TAG_START("search", (void*)-1, 0, true);
    for(uint x = 0; x < size; x++) {
        auto a = s->find(fast_rand(&seed));
        TAG_GROW("search", &(*a), &(*a)+ 1);
    }
    TAG_STOP("search");
    end_measurement();

    TAG_START_ALL("delete", false);
    delete s;
    TAG_STOP("delete");
}
"""))
cpp_set[0].source("working")
```

Here's what the memory trace looks like



What you are looking at is the region of the heap that the C++ standard library is allocating to hold the set. Since, it's a tree-based structure, it's made up of many small objects that get allocated with `new`. The heap is allocating space starting at a low address and working upward -- hence the diagonal.

The color key is:

- Compulsory misses are light blue
- hits are purple.
- Conflict misses are gold.

Question 4 (Completeness)

Approximately how many bytes does the `set` occupy? What's the ratio of bytes occupied to the number of values the `set` contains? How many bytes would be needed to store the same elements in an array? What is the light blue line along the top slope of the triangle?

Bytes for the `set`:

Bytes per element in the `set`:

Bytes to store elements:

What is the blue line?:

Show Solution

In the code, we created two tag: `build` and `search`. The graph shows both of them. `Build` is the big triangle. `search` is the rectangle on the right that's a little darker yellow.

Also, recall that the green line on the vertical axis is the size of the cache that Moneta is modeling.

Question 5 (Correctness - 2pts)

At the beginning of the build portion of the experiment what is miss rate? When does it start to climb? Why?

Miss rate at the beginning?:

WHen does it start to rise?:

Why does it climb?:

Question 6 (Completeness)

Where are hits more concentrated on the graph? What part of the data structure do you think they are accessing (think carefully about how `std::set` is implemented and accessed)?

Where are they concentrated?:

What part of the data structure are they accessing? Be as specific as you can.:

Show Solution

12 The Three C's

Recall from lecture (or review the slides) that we can classify cache misses into types (known as "The Thee C's"):

1. **Compulsory**: These misses occur because the processor has not accessed this cache line before.
2. **Capacity**: These occur because the program is accessing more memory than the cache can hold (i.e., it's working set is bigger than the cache).
3. **Conflict**: These occur because a given cache line of memory can only live in one of the associative sets of the cache.

12.1 Capacity and Compulsory Misses

The our investigation of spatial locality, temporal locality, and working sets illustrated compulsory and capacity misses.

Question 7 (Completeness)

Look back over the Moneta graphs in previous sections of this lab and grab screen captures of the parts of the graphs that illustrate compulsory and capacity misses. Paste them below and describe why the illustrate each kind of miss.

Compulsory

Capacity

Show Solution

12.2 Conflict Misses

Let's try to produce some conflict misses. In the last lab, we used a miss machine to generate lots of misses. They were mostly capacity misses (i.e., we accessed too many cache lines), and the miss machine let us produce lots of seemingly random accesses really fast. For conflict misses, we need something different: Highly-organized misses placed precisely.

The necessary ingredient for lots of conflicts misses is many memory accesses that will map to the same associative set in the cache. If we access many of these cache lines, the associative set will "overflow" and that will causes misses.

Question 8 (Completeness)

Assume our 32kB cache with 64-byte lines and 8-way associativity and 64-bit addresses. Given an address A, how can we compute a new address, B, that will map to the same associative set but is not part of the same cache line as A? Given an index, i, into an array, how can we compute the index of another element, j, that will conflict with the first?

How do you compute `B` ?

How do you compute `j` ?

Show Solution

Let's see if your formula worked. We'll run `stride()` from earlier with a stride of 16 and 1024. In the experiment below we set `size` so that we cover 64 strides worth of the memory, since both strides are larger than cache line, each execution of the loop will touch 64 cache lines.

In [ ]:
```
display(stride[0].source("stride"))

conflict_run = run(stride, "stride", arg_map(size=16*64, stride_size=16, 
                   perf_counters=["L1-DCACHE-LOAD-MISSES",  "PERF_COUNT_HW
conflict_run2 = run(stride, "stride", arg_map(size=1024*64, stride_size=1
                    perf_counters=["L1-DCACHE-LOAD-MISSES",  "PERF_COUNT_H
```

In [ ]:
```
conflict_data = PE_calc((conflict_run + conflict_run2).as_df())
display(conflict_data)
```

In [ ]:
```
plotPEBar(df=conflict_data, what=[("stride_size", "L1_MPI"),("stride_size
```

That worked pretty well! we got a lot more L1 cache misses.

Question 9 (Completeness)

Based on our analysis above, what do you think will happen with if the stride is one cache line longer (1040 bytes) or or one cache line shorter (1008 bytes)? Why?

Stride 1008:

Stride 1040:

Show Solution

The main lesson here is that conflict misses are largely a product of bad luck if the working set is smaller than the cache: It may happen that for a particular cache capacity, associativity, and line size, that many cache lines in the application's working set happen to map to the same associative set.

Fortunately, in modern processors caches are pretty highly-associative (our is 8-way) and at that level of associativity conflict misses are not a huge problem. If you working set is smaller than your cache's capacity, you'd have to be very unlucky to have enough cache lines land in the same associative set to cause many conflict misses. If you working set is larger than your cache, you're going to have misses regardless. As the example above shows, it is not hard to construct programs with small working sets that are unlucky. We have a term for these access patterns: We say they are "pathological".

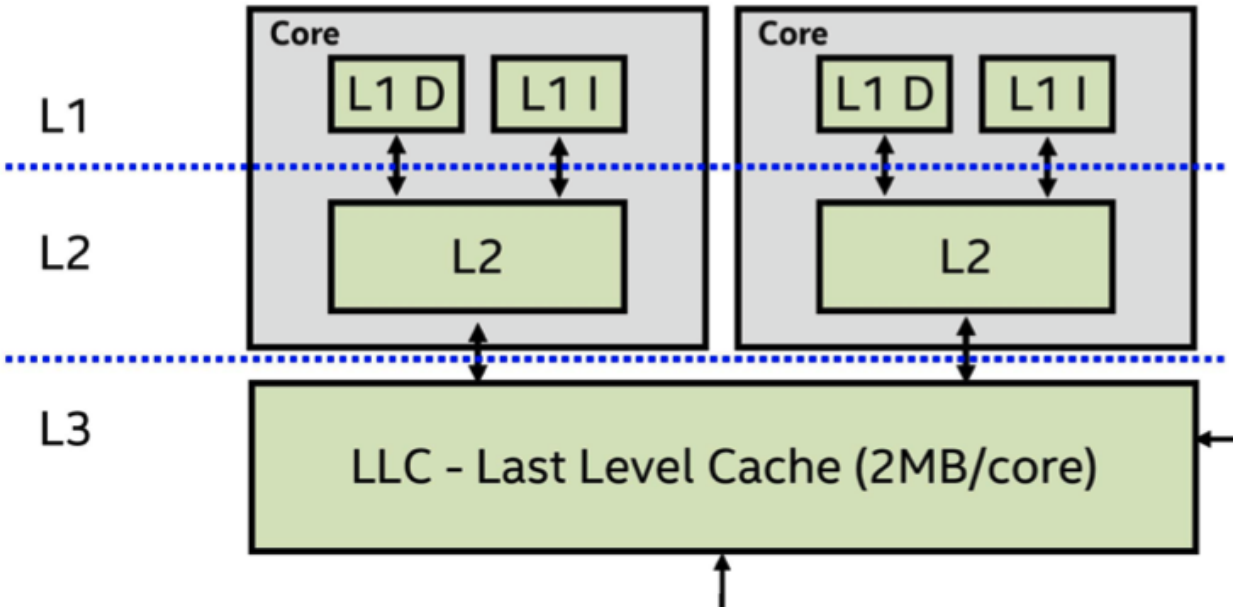
By definition, pathological access patterns are rare, so we don't spend too much time worrying about them. But they can crop up and it's a good idea to be aware of the possibility.

Question 10 (Optional)

Consider the implementation of `tensor_t` described earlier in the lab. Accessing a tensor column-wise produces strided accesses which could lead to conflict misses if the dimensions of the tensor are "unlucky". Why is this so? What constitutes "unlucky" dimensions? How could you modify `tensor_t` make it (mostly) immune to "unlucky" dimensions?

13 The L2 and L3 Caches

So far in these two labs, we have focused on the L1 cache, but our machine also has L2 and L3 caches. Here's how they are organized:



As a reminder, the L1 is 32kB, 8-way set associative, with 64-byte lines. So, there are 512 cache lines divided into 16 associative sets.

The L1 and L2 are private to each core while the L3 is shared among all the cores on the CPU. The L2 is 256kB and is 8-way set associative. The L3 is 2MB per core, but it's shared across all the cores. Our machine has six cores.

On our machine, the L3 is the "last level cache" or LLC.

The code below is similar to the `stride` function we used in the previous lab. The change is that the outer loop is setup so we do the same number of memory accesses for all values of `size` (This is why we divide by `size`). Our goal is to measure the L1 and L3 MPI as size increases. The CPU's performance counters don't let us collect L1 and L2 statistics at the same time, so we have to run the experiment once for each cache.

Why don't we measure L2 MPI? Well, there aren't great performance counters for the L2 on our machine.

Question 11 (Completeness)

As `size` increases, the miss rate for the L1 and L2 will rise. At value of `size` would you expect to see significant increases in L1 and L2 MPI?

L1 critical `size`:

L3 critical `size`:

Let's see how we do:

In [ ]:
```
L23 = build(code(r"""
#include"cfiddle.hpp"
#include"tensor_t.hpp"
#include<cstdint>

extern "C"
void L23(uint64_t size, uint64_t stride_size) {
    tensor_t<uint32_t> t(size,1,1,1);
    start_measurement();
    for(uint64_t i = 0; i < (1 << 27)/size; i++) {
        for(uint x = 0; x < size; x+=stride_size) {
            t.get(x,0,0,0) = 0;
        }
    }
    end_measurement();
}

"""), arg_map(OPTIMIZE="-O1"))

L1 = run(L23, "L23", arg_map(size=exp_range(16,1024*1024*128,2), stride_s
         perf_counters=["L1-DCACHE-LOAD-MISSES",  "PERF_COUNT_HW_INSTRUCT

L3 = run(L23, "L23", arg_map(size=exp_range(16,1024*1024*128,2), stride_s
             perf_counters=["LLC-LOAD-MISSES",  "LLC-STORE-MISSES", "PERF_
```

In [ ]:
```
display(PE_calc(L1.as_df()))
display(PE_calc(L3.as_df()))
plotPE(df=PE_calc(L1.as_df()), what=[("size", "L1_cache_misses"), ("size"
plotPE(df=PE_calc(L3.as_df()), what=[("size", "L3_cache_misses"), ("size"
```

In [ ]:
```
!cse142 job run --lab caches2 showevtinfo
```

Question 12 (Completeness)

Do the data match your prediction? If not, how did it differ?

L1 prediction correct?:

L3 prediction correct?:

The graphs also include CPI, which should, in theory, be the same since we just ran the same code

twice with different performance counters. You can see the small increase when the L1 hit raises increases, and the larger bump with misses in the L3.

In [ ]:
```
plotPE(df=data, logx=2, combined=True, lines=True, what=[ ('size', 'CPI')
```

Question 13 (Correctness - 2pts)

Based on this data, how much speedup could you expect from reducing your working set size (in bytes) from...

8MB to 4MB?:

2MB to 512kB?:

2MBkB to 128kB?:

128MB to 32kB?:

14 The TLB

The three levels of on-chip caches set the number of *cache lines* the processor can quickly access. As you heard in 142, though, there is another kind of cache in the processor: the TLB. Instead of data, the TLB caches the translations from virtual addresses to physical addresses, and its size sets the number of *pages* your program can access quickly.

Here's what our processor has:

1. 64 entries for 4kB pages (256kB total)
2. An L2 TLB with 1024 entries (8-way set associative; 4MBs total @ 4kB pages).
3. 32 entries for 2MB pages (64MB total).
4. 4 entries for 1GB pages (4GB total).

This is a little more complicated than what you heard about in 142. First off, there is an L1 TLB *and* an L2 TLB. If we think of the L1 TLB as cache for memory translations, then the L2 TLB is exactly analogous to the L2 cache: If the processor has a TLB miss in the L1 TLB, it can look in the L2 TLB. One important point: memory address translation *always* happens at the L1 cache because *all* the caches are physically tagged. This means that the L2 TLB *has nothing to do with the L2 Cache*.

The L2 TLB can cover 4MBs worth of 4kB pages of virtual address space. If you are using more pages than that, you'll get TLB misses and your performance will suffer.

Here's a fun idea!: Let's use a miss machine to measure the L1 TLB miss latency.

The the code below is version of our miss machine code from the last lab but with a few changes:

1. It has a template-configurable link size (`BYTES`).
2. We allocate the `MM` links in array that 4096-byte aligned.
3. We use `madvise()` (https://man7.org/linux/man-pages/man2/madvise.2.html) to prevent us from using 2MB pages, which Linux will automatically use when it can. We'll come back to that.
4. We can set the *total size* of the miss machine *in bytes* with the `size` parameter. It should be a multiple of `BYTES` .

Read through the code to make sure it makes sense.

In [4]:
```
tlb = build("TLB.cpp", arg_map(OPTIMIZE="-O1"))
tlb[0].source(show=("//START", "//END"))
```

```
100%                                                1/1 [00:00<00:00, 21.87it/s]
```

Out[4]:
```
//START
#include<cstdint>
#include<cstdlib>
#include<vector>
#include<algorithm>
#include"cfiddle.hpp"
#include<iostream>
#include <sys/mman.h>
#include<cstring>

template<size_t BYTES>
struct MM {
    struct MM* next;  // I know that pointers are 8 bytes on this machin
e.
    uint64_t junk[BYTES/8 - 1]; // This forces the struct MM to take a up
a whole cache line, abolishing spatial locality.
};


template<class MM>
MM *  __attribute__ ((noinline))  miss(MM * start, uint64_t access_count)
{
    for(uint64_t i = 0; i < access_count; i++) { // Here's the loop that
 does this misses. It's very simple.
        start = start->next;
    }
    return start;
}

template<size_t BYTES>
uint64_t* TLB(uint64_t size, uint64_t access_count) {
    struct MM<BYTES> * array = NULL;
    int r =  posix_memalign(reinterpret_cast<void**>(&array), 4096, size
);
        if (r == -1) {
                std::cerr << "posix_memalign() failed.  Exiting: " << str
error(errno) << "\n";
                exit(1);
    }

    r = madvise(reinterpret_cast<void*>(array), size, MADV_NOHUGEPAGE);
        if (r == -1) {
                std::cerr << "madvise() failed.  Exiting: " << strerror(e
rrno) << "\n";
                exit(1);
    }

    std::cout << "array alignment is " << (reinterpret_cast<uintptr_t>(ar
```

```
ray) % 4096) << "\n";
    std::cout << "array size is " << size/BYTES << " element; " << size <
< "B\n";

    // This is clever part  'index' is going to determine where the point
ers go.  We fill it consecutive integers.
    std::vector<uint64_t> index;
    for(uint64_t i = 0; i < size/BYTES; 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 < size/BYTES; i++) {
        array[index[i]].next = &array[index[(i + 1) % (size/BYTES)]];
    }

    MM<BYTES> * start = &array[0];
    start_measurement();
    start = miss(start, access_count);
    end_measurement();
    return reinterpret_cast<uint64_t*>(start); // This is a garbage valu
e, but if we don't return it, the compiler will optimize out the call to
 miss.
}
//END
```

There are two parameters we need to set: The size of `MM` (`BYTES` in the code above) and the `size`.

Here's what the `miss()` function looks like for `BYTES = 4096`. It should be familiar from Lab 3 (note that we have use the mangled name):

In [ ]:
```
tlb[0].cfg("MM_4096ul__miss_MM_4096ul____MM_4096ul___unsigned_long_")
```

Question 14 (Completeness)

Using the code above, what values of `BYTES` and `size` should we run the miss machine with to measure the L1-TLB miss latency? (The fact that there are two experiments listed is a hint that you'll need to run two different experiments.)

	BYTES	size
Experiment 1		
Experiment 2		

Show Solution

Question 15 (Optional)

The measurement above is for a miss to the L1 TLB. Perform a different experiment to measure the L2 TLB miss latency. This is harder than it appears at first.

Question 16 (Optional)

The measurements above are based on 4kB pages, but we can also use 2MB "huge pages". Repeat the experiment above to determine whether 2MB TLB entries can also reside in the L2 TLB.

A few notes:

1. This one is a little involved. You'll need to significantly tweak the experiments we did above.
2. Whether 2MB TLB entries can be in the in the L2 TLB is not clearly specified in any documents I have found, so I don't know the answer.
3. To get the system to use 2MB huge pages, remove the call to `madvise()` in `TLB.cpp` and ask `posix_memalign()` to give 2MB-aligned memory.
4. Look in `TLB.cpp` for examples of how to change `BYTES`. `TLB_2M()` is a good starting point.

15 Optimizing For Locality

Minimizing cache misses is critical for maximizing performance because, as you have seen, even a small number of misses can inflate `CPI` and `ET`. As a result, programmers who are concerned about performance often spend a lot of effort optimizing there code to reduce misses.

Below, we'll take a look a two common optimizations that aim to increase locality: Loop reordering and tiling.

In the compiler lab, you explored several other optimizations that compilers apply very effectively. While there are compilers that apply these (and other) locality optimizations, many do not, and even when they do, these locality optimizations do not work as effectively when applied automatically, so

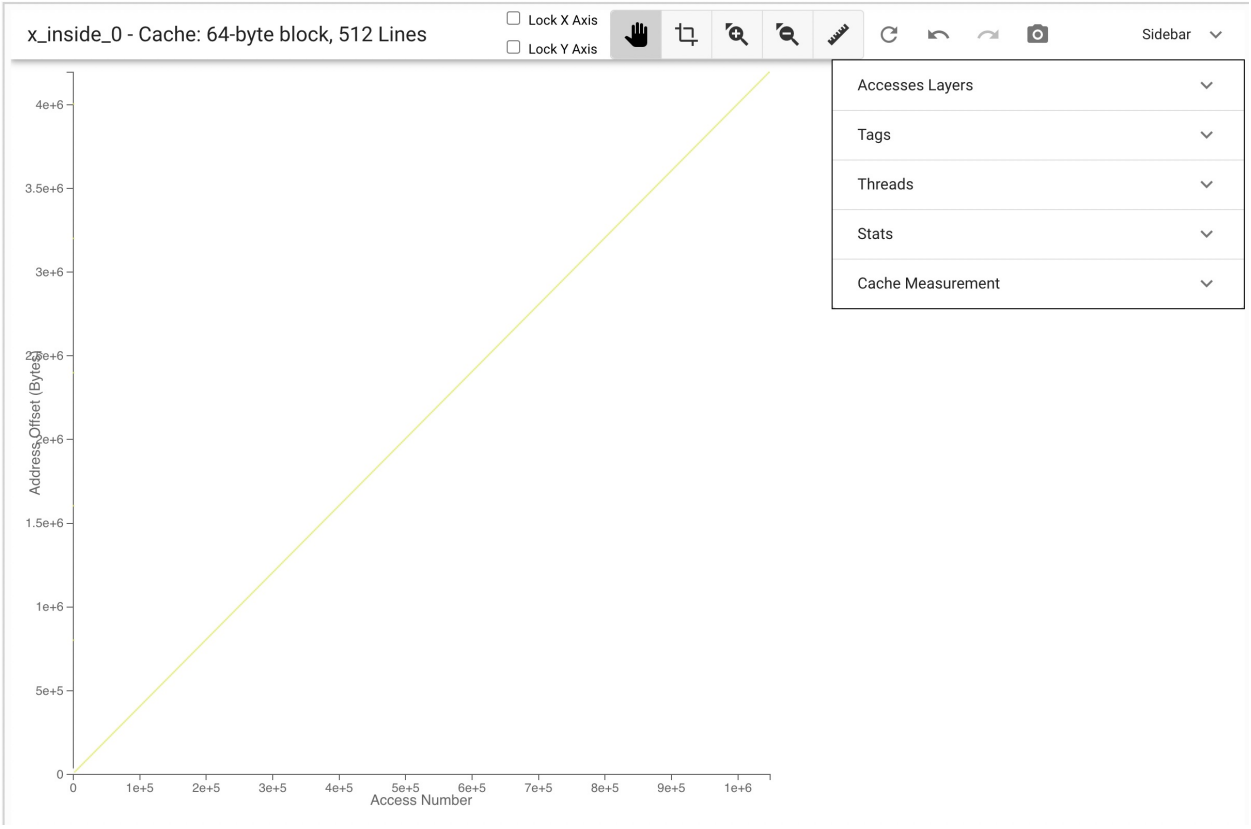
performance-obsessed programmers often apply locality optimizations by hand (but, of course, only when profiling and Amdahl's law demonstrates it's potentially profitable!).

15.1 Loop Renesting

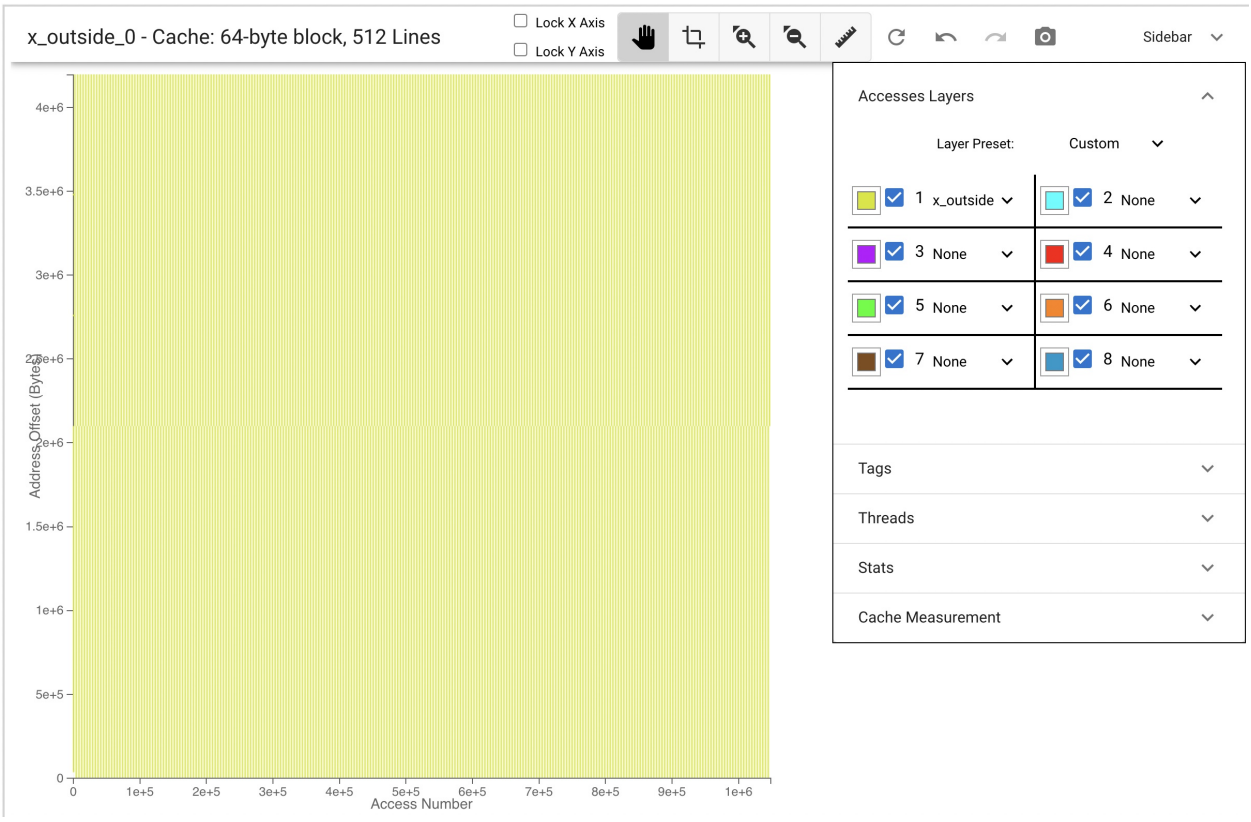
Loop reordering or "re-nesting" is an optimization that changes the order in which loops are nested to improve locality. For instance, consider the code below. It initializes a 2D tensor, but it does it twice: The first time, the loop for `x` is on the outside of the loop nest. The second time, `x` is on the inside.

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

extern "C"
void x_inside(uint64_t size, uint64_t rows) {
    tensor_t<uint32_t> t(size/rows,rows,1,1);
    disable_prefetcher();
    flush_caches();
    start_measurement();
    for(uint y = 0; y < rows; y++) {
        for(uint x = 0; x < size/rows; x++) {
            t.get(x,y,0,0) = x;
        }
    }
    end_measurement();

}


extern "C"
void x_outside(uint64_t size, uint64_t rows) {
    tensor_t<uint32_t> t(size/rows,rows,1,1);
    disable_prefetcher();
    flush_caches();
    start_measurement();
    for(uint x = 0; x < size/rows; x++) {
        for(uint y = 0; y < rows; y++) {
            t.get(x,y,0,0) = x;
        }
    }
    end_measurement();

}

"""), arg_map(OPTIMIZE="-O1"))
renest[0].source()
```

Here's the memory trace for `x_inside`:



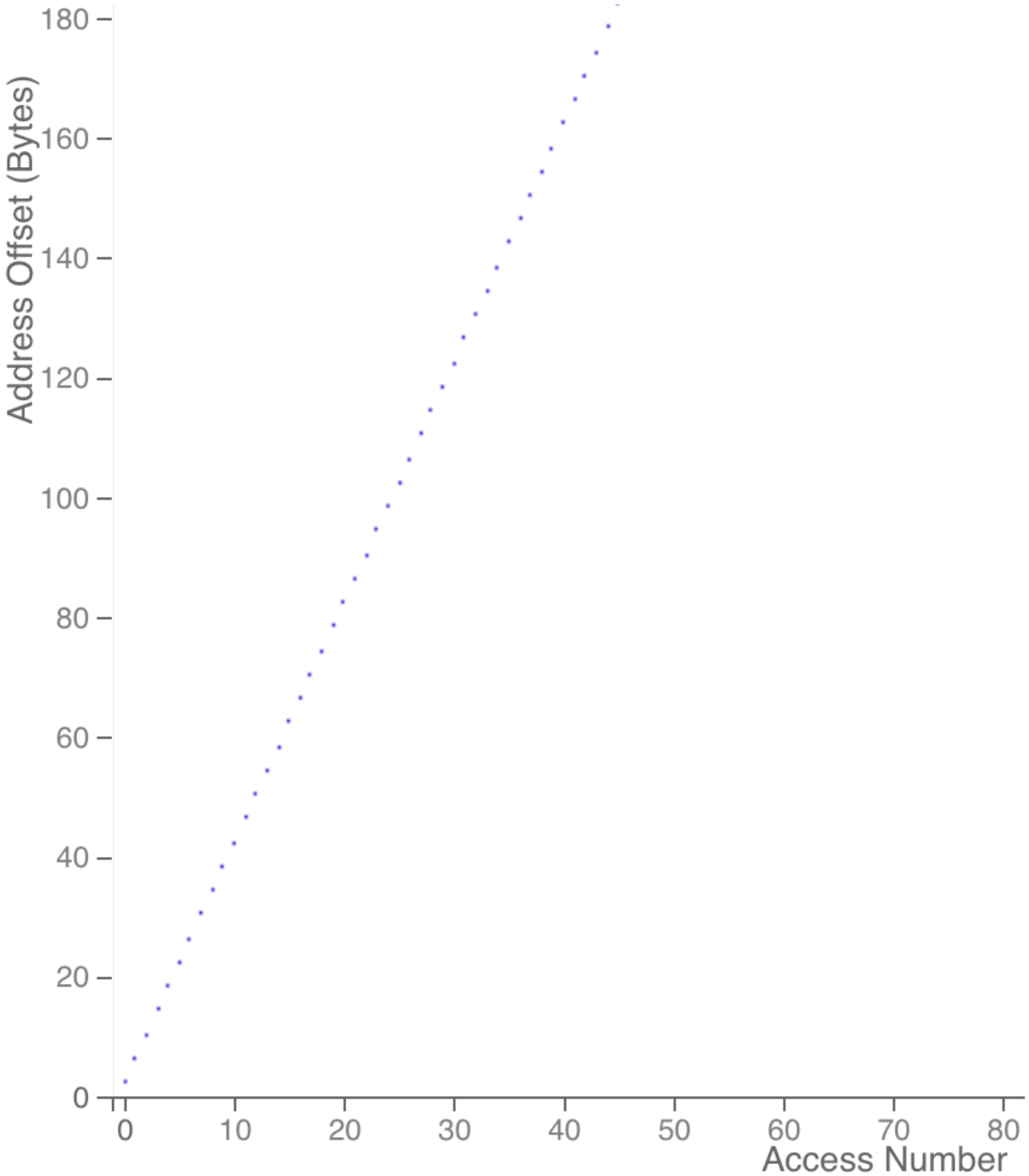
And here it is for `x_outside()`:



Remarkably, those two plots contain exactly the same memory accesses, they just distributed differently through time.

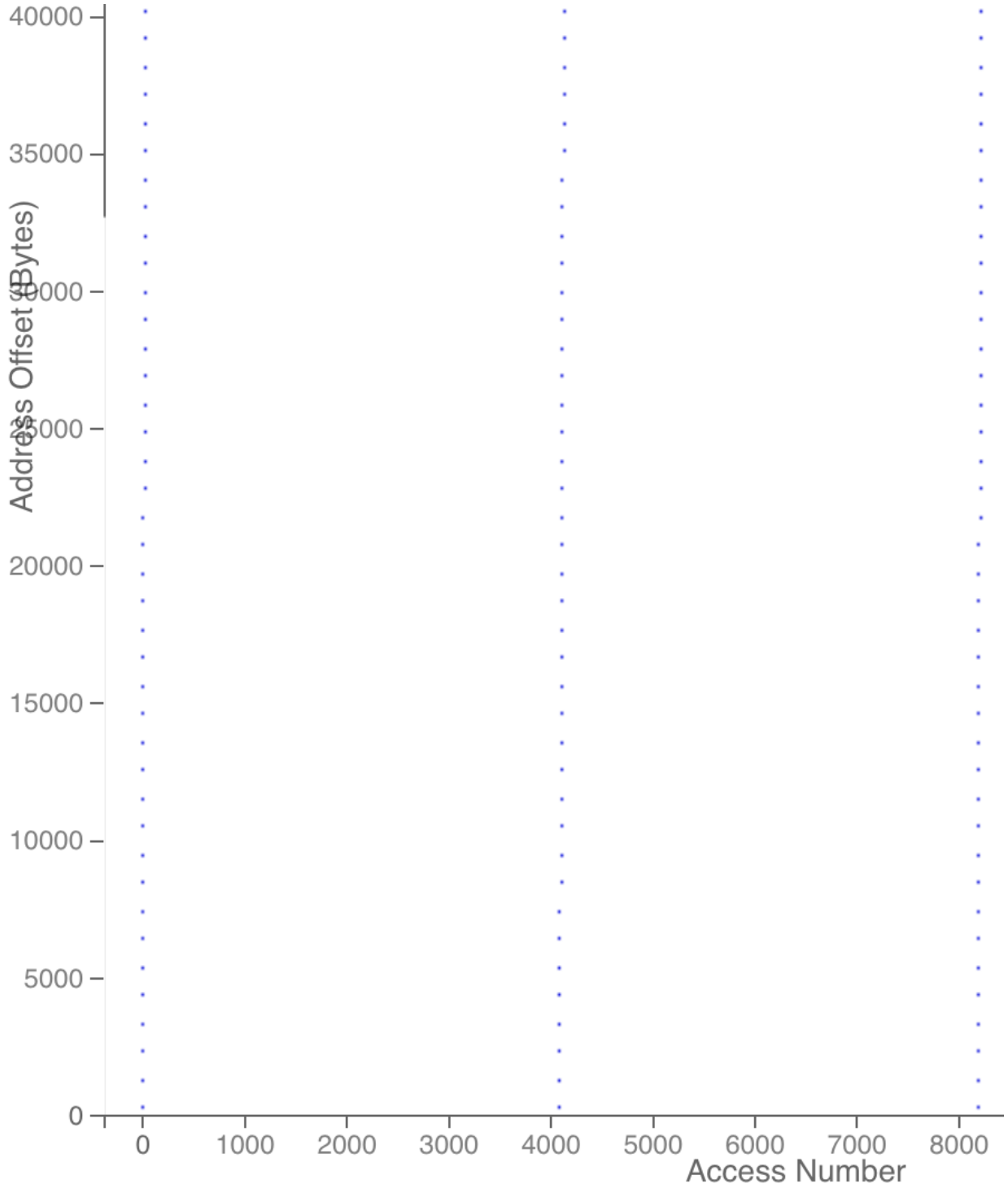
Recall from our earlier discussion of `tensor_t`, that incrementing the first argument to `get()` corresponds to moving to the next element in the underlying array of data. In the code above, `x` is the first argument to `get()`, so putting the `x` loop inside leads to better spatial locality.

You can see this reflected in the traces more clearly if we zoom in: With `x` on the inside, the program marches linearly through memory:



See how both the range of the horizontal and vertical axes are both quite small. This means that the accesses close in space and time.

With the `x` loop outside, it takes large strides through the array, and accesses to the addresses are spread out over a long time (note th much larger range on both axes:



In particular, it doesn't access the same 64 byte cache line again until long after it has been evicted from the cache.

Question 17 (Completeness)

What value of the `rows` argument to `x_outside()` should result in a very high (e.g., > 95%) hit rate? Try to reason through the correct value before running any experiments.

Show Solution

15.2 Loop Tiling

Renesting loops can improve spatial locality, but it is generally less effective for improving temporal locality. There are two criteria that must be met in order to exploit temporal locality:

1. The cache line must be re-used.
2. The re-use must occur before the cache line is evicted by other cache lines coming in the cache.

This second condition has a direct connection to working set size: If the working set size of a piece of code is too large, it is likely that parts of it will be evicted before they are accessed again, making it hard for the processor to exploit the temporal *and* spatial locality.

Our goal, then, is to shrink the working set so that it fits in the cache and we can exploit the resulting locality.

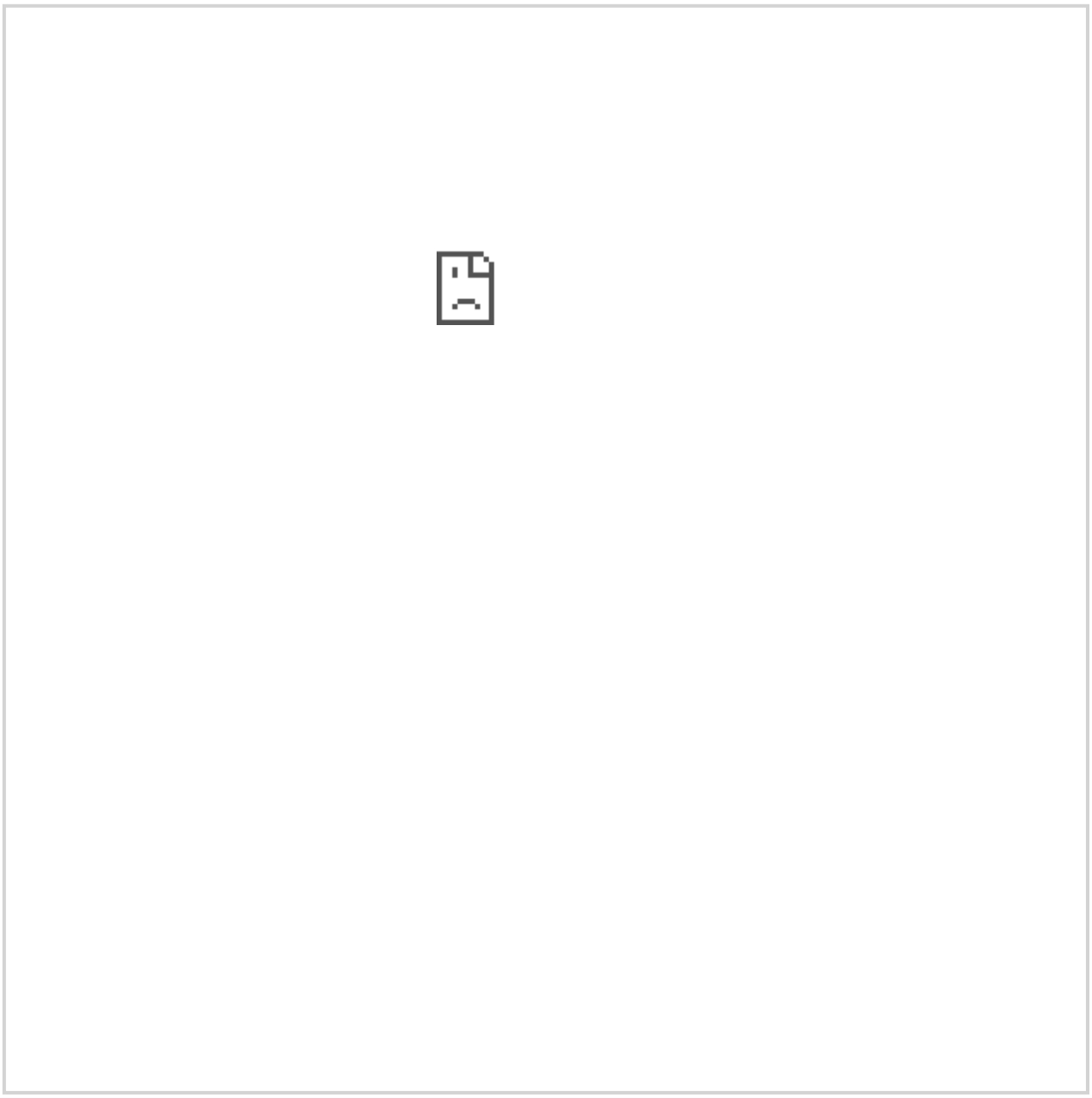
As an example, let's consider a 1-dimensional convolution.

15.2.1 1-D Convolution

One-dimensional convolution a simple algorithm and a fundamental building block for many signal processing systems. The inputs are two 1-dimensional arrays (we will use `tensor_t<uint32_t>`) that we will call the `source` and the `kernel` and it produces a third array called the `target`. The `kernel` is usually much smaller than the `source` and the `length(target) = length(source) - length(kernel)`.

Conceptually, we compute the entries of `target` by "sliding" `kernel` along `source` and computing the dot product at each position. Here's a video that illustrates:

In [6]:
```
display(IFrame("https://www.youtube.com/embed/ulKbLD6BRJA", width=560, he
```



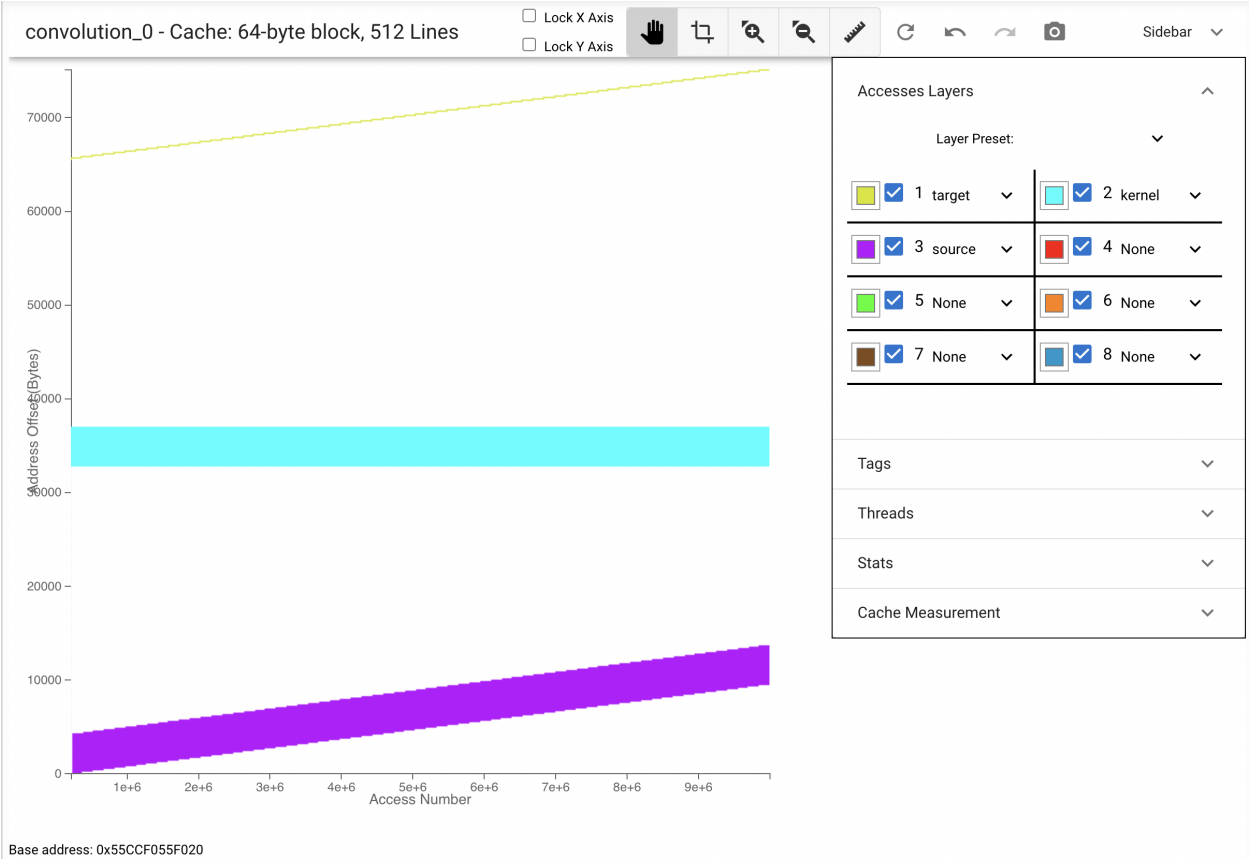
The code for simple implementation is below:

In [7]:
```
convolution = build("convolution.cpp", arg_map(OPTIMIZE="-O3", DEBUG_FLAG
convolution[0].source(show=("//START", "//END"))
```

```
100%                                                    1/1 [00:00<00:00, 35.42it/s]
```

Out[7]:
```
//START
extern "C"
void do_convolution(const tensor_t<uint32_t> & source,
                    const tensor_t<uint32_t> & kernel,
                    tensor_t<uint32_t> & target) {
        disable_prefetcher();
        flush_caches();
        start_measurement();
        for(register int32_t i = 0; i < target.size.x; i++) {
                for(register int32_t j = 0; j < kernel.size.x; j++) {
                        target.get(i,0,0,0) += source.get(i + j,0,0,0) *
kernel.get(j,0,0,0);
                }
        }
        end_measurement();
}

extern "C"
void convolution(uint64_t source_size, uint64_t kernel_size, int32_t tile
_size) {
        tensor_t<uint32_t> source(source_size,1,1,1);
        tensor_t<uint32_t> kernel(kernel_size,1,1,1);
        uint64_t target_size = source_size - kernel_size;
        tensor_t<uint32_t> target(target_size,1,1,1);
        // TAG_START("source", source.data, &source.as_vector(source.elem
ent_count()), true);
        // TAG_START("kernel", kernel.data, &kernel.as_vector(kernel.elem
ent_count()), true);
        // TAG_START("target", target.data, &target.as_vector(target.elem
ent_count()), true);


        // Here's the the key part:
        do_convolution(source, kernel, target);

        // TAG_STOP("source");
        // TAG_STOP("kernel");
        // TAG_STOP("target");

}

//END
```

And here's a trace of it running with a 16kB source and 4kB kernel:



The yellow line is `target`, the light blue band is `kernel`, and the `source` is in purple. The slow shift upward of `source` shows the sliding slice of `source` that `kernel` is being dot-producted (dot-produced?) with.

It's not easy to tell from the trace, the working is around 150 cache lines or so -- no problem for our processor. The size of the working set remain roughly constant, although the cache lines that make it up change as the computation moves along source and target.

15.2.2 Memory Behavior in 1-D Convolution

We can see from the trace that there is quite a bit of reuse: The code reads `kernel` over and over again, and there's a lot of overlap between the parts of `source` that the program accesses. There is *a lot* of temporal locality, and we should be able to use it.

Here's the assembly for `do_convolution()` (our CFG rendered can't handle it :-(). A few notes:

1. `%rdi` points to `source`
2. `%rsi` points to `kernel`
3. `%rdx` points to `target`
4. Note how constant propagation and inlining have turned all those calls to `get()` into very simple code.
5. The inner loop body starts an `.L41`

In [ ]:
```
convolution[0].asm("do_convolution")
```

Question 18 (Correctness - 2pts)

How many loads does the inner loop perform per iteration? How many stores? (Recall that `op r1, r2` in x86 means `r2 = r1 op r2`). If `source` has 4096 element and `kernel` has 512 element, what's the dynamic instruction count for each loop (to within 10% -- you don't need to be perfectly exact. Show your work.)? What fraction of the execution is spent on the outer loop?

	Inner loop	Outer loop (excluding the inner loop)
Instructions		
Loads		
Stores		

	Inner loop	Outer loop (excluding the inner loop)
Dynamic Instruction count		
% of IC		

Question 19 (Completeness)

Compute (rather than measure as you did above) the size of the working set of this computation? What would it be if `kernel` were 32kB, and `source` was 128kB. Roughly estimate the cache hit rate in each case.

Working set for 16kB source and 4kB kernel:

Estimated hit rate:

Working set for 128kB source and 32kB kernel:

Estimated hit rate:

Show Solution

15.2.3 Tiling 1-D Convolution

We are going to "tile" the execution of this function to reduce the number of cache misses for large data sets. Tiling works by breaking up the execution of a set of nested loops into smaller parts with smaller working sets. If the resulting working set fits in the cache, the number of cache misses should drop significantly.

We'll perform the tiling in two steps:

1. We'll "split" a loop into two nested loops.
2. Then we'll renest the resulting loops.

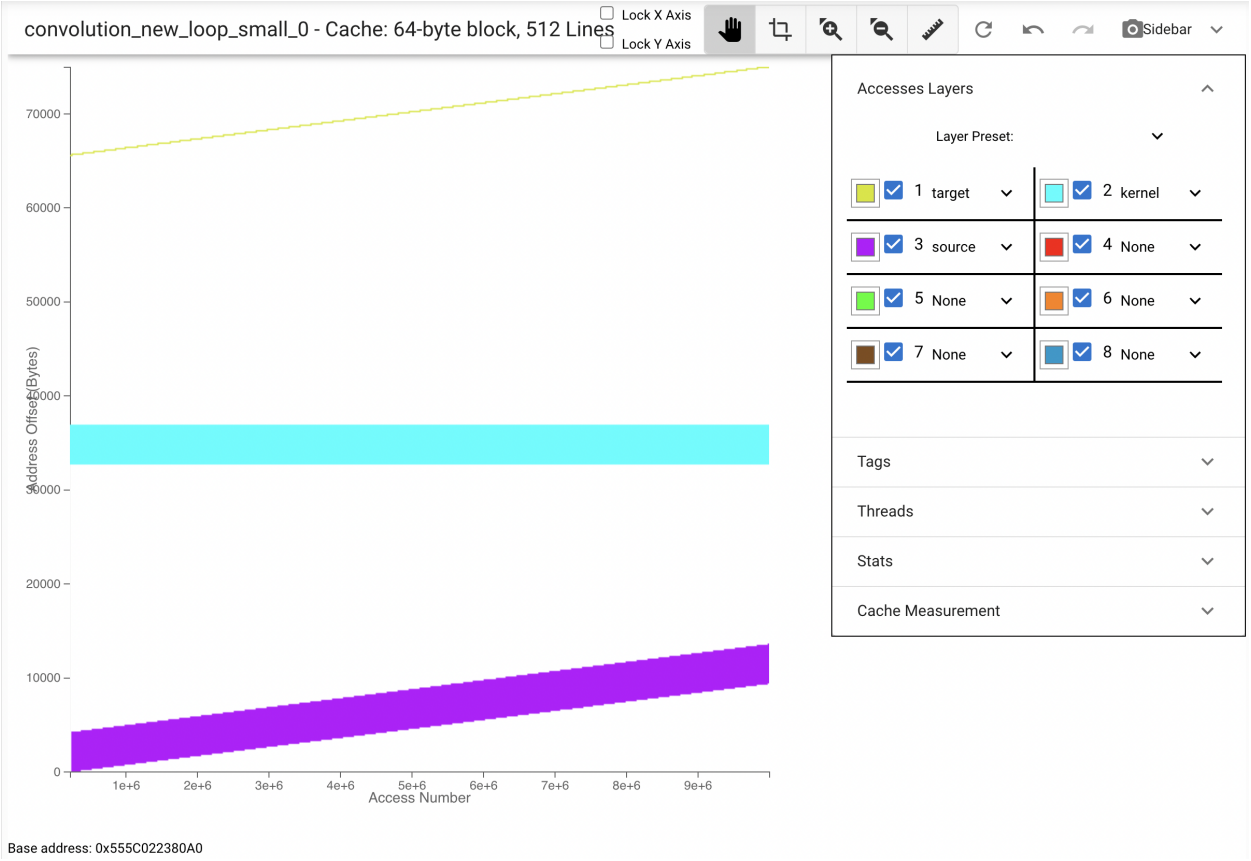
To split a loop, we will break the loop into fixed-size chunks. The outer loop will iterate over the chunks, and the inner loop will iterate over the elements within a chunk. This first transformation *has no effect* on the order in which computation occurs.

Here's version of the convolution code with the `kernel` loop split into chunks of size `tile_size` (we'll set `tile_size` to 64 for now):

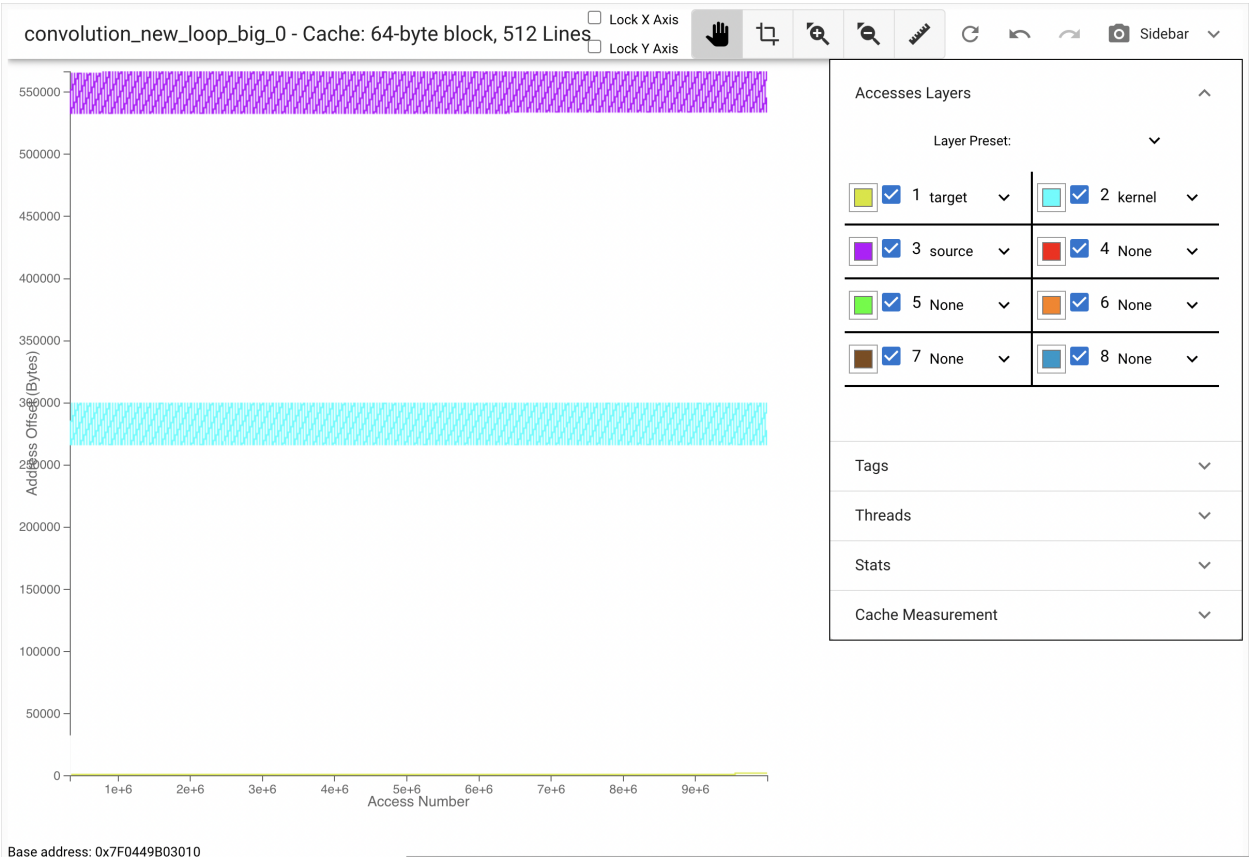
In [8]:
```
convolution[0].source("do_convolution_new_loop")
```

Out[8]:
```
void do_convolution_new_loop(const tensor_t<uint32_t> & source,
                             const tensor_t<uint32_t> & kernel,
                             tensor_t<uint32_t> & target, int32_t tile_si
ze) {

        start_measurement();
        for(int32_t i = 0; i < target.size.x; i++) {
                // We create a new loop variable jj that advanced one chu
nk at a time.
                for(int32_t jj = 0; jj < kernel.size.x; jj += tile_size)
{
                        // We iterate over the chunk.  The more complicat
ed termination
                        // condition keeps ups from running off the end o
f the arry
                        for(int32_t j = jj; j < kernel.size.x && j < jj +
tile_size; j++) {
                                target.get(i,0,0,0) += source.get(i + j,0
,0,0) * kernel.get(j,0,0,0);
                        }
                }
        }
        end_measurement();


}
```

Here's the trace for the 4kB source and 1kB kernel. Nothing changes:



And here's the trace for the big data set



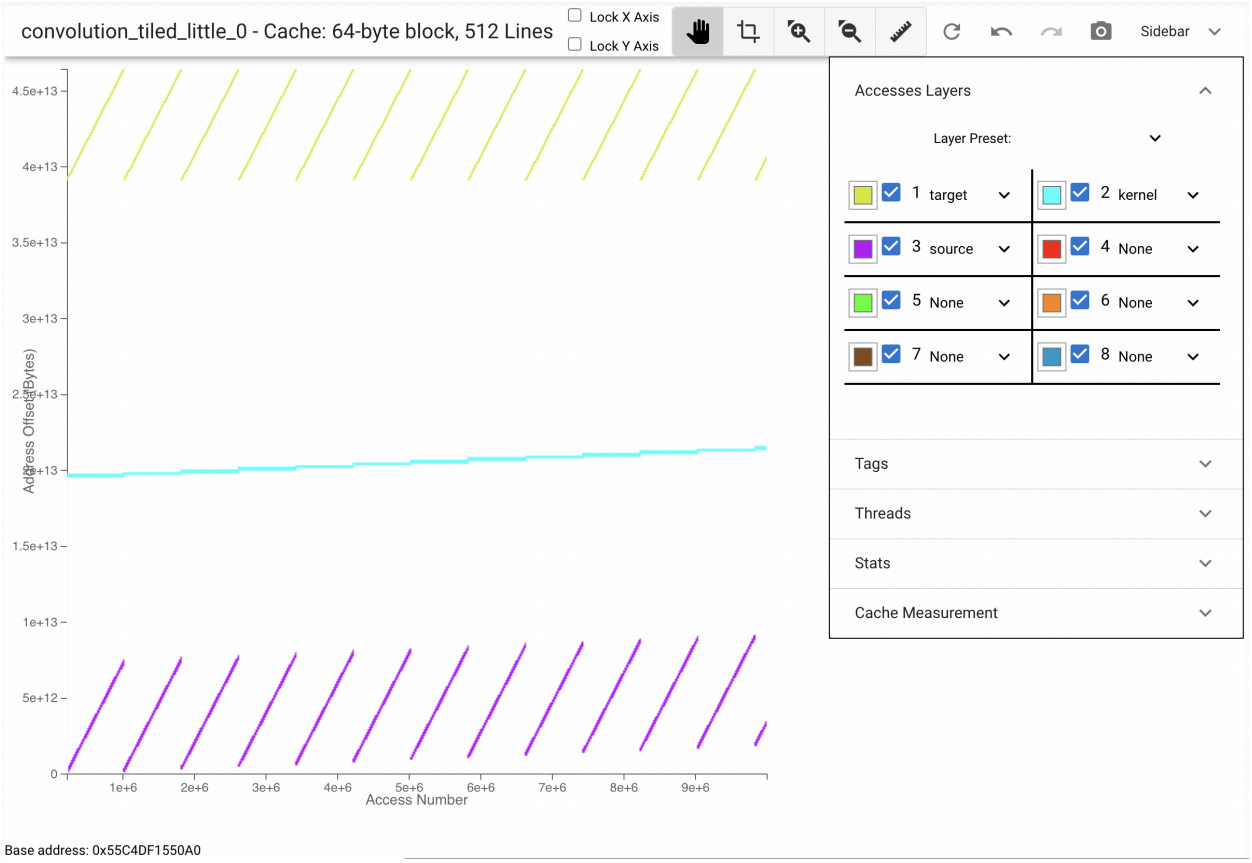
Here it is the same code on a much larger data set that doesn't fit in the L1. The slop of the lines is less pronounced because are only seeing the first 10 million memory accesses (which is all Moneta

can display at once). You might notice that the data structures are laid out differently (i.e., the color stripes are in a different order) in this plot than the one above. This is the memory allocator putting things wherever it wants. The color scheme is the same across all these plots.

The next step is to renest the split loop. Here's the code. The only change is that we swapped the order of the `jj` loop and the `i` loop. Now, we will run the whole algorithm for 2048 elements of the `kernel` at a time.

In [ ]:
```
convolution[0].source("do_convolution_tiled")
```

Here's this code running on the small data set:

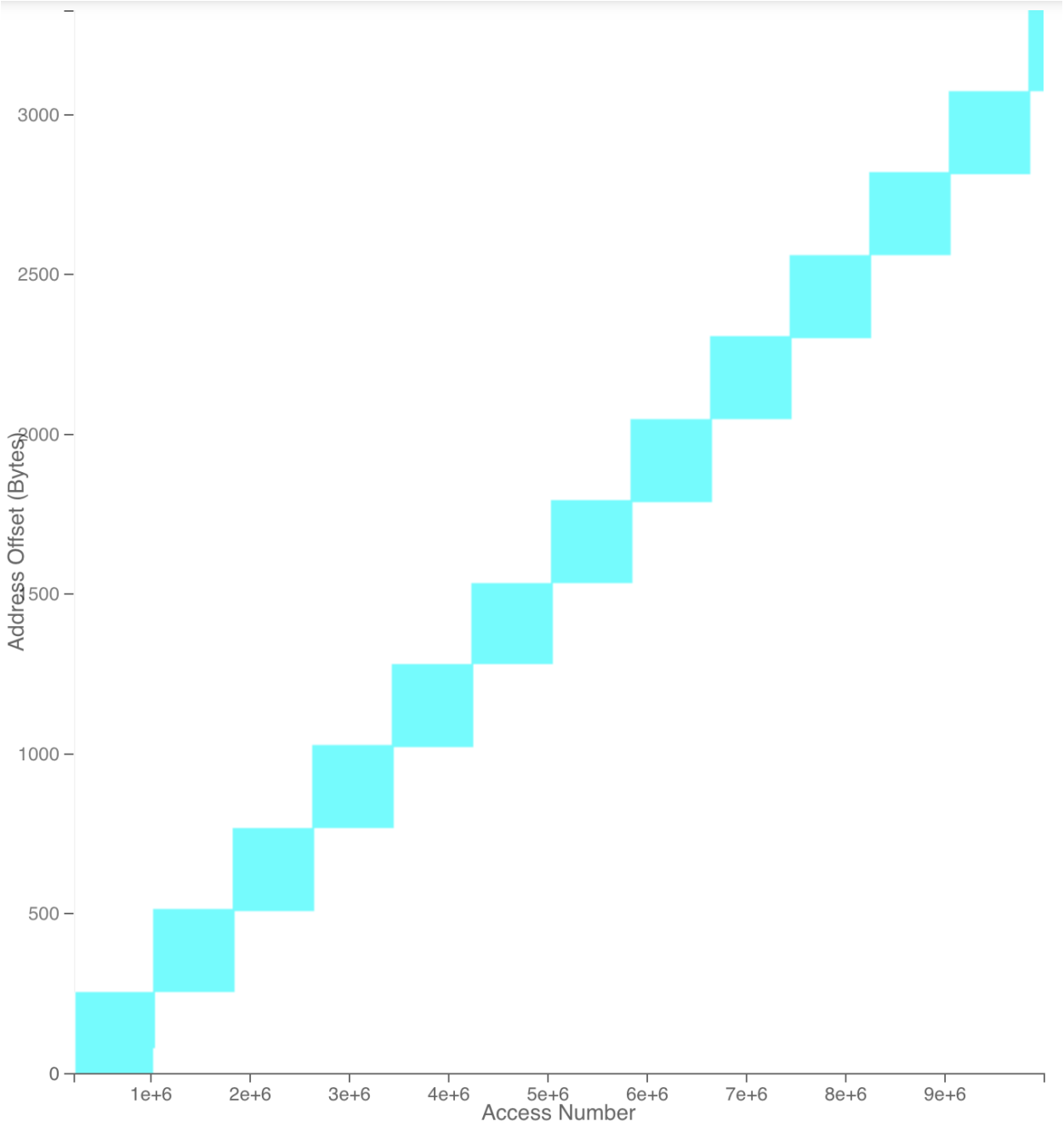


Well, that's different!

Let's take a look at the access pattern for each of the tensors:

`kernel`

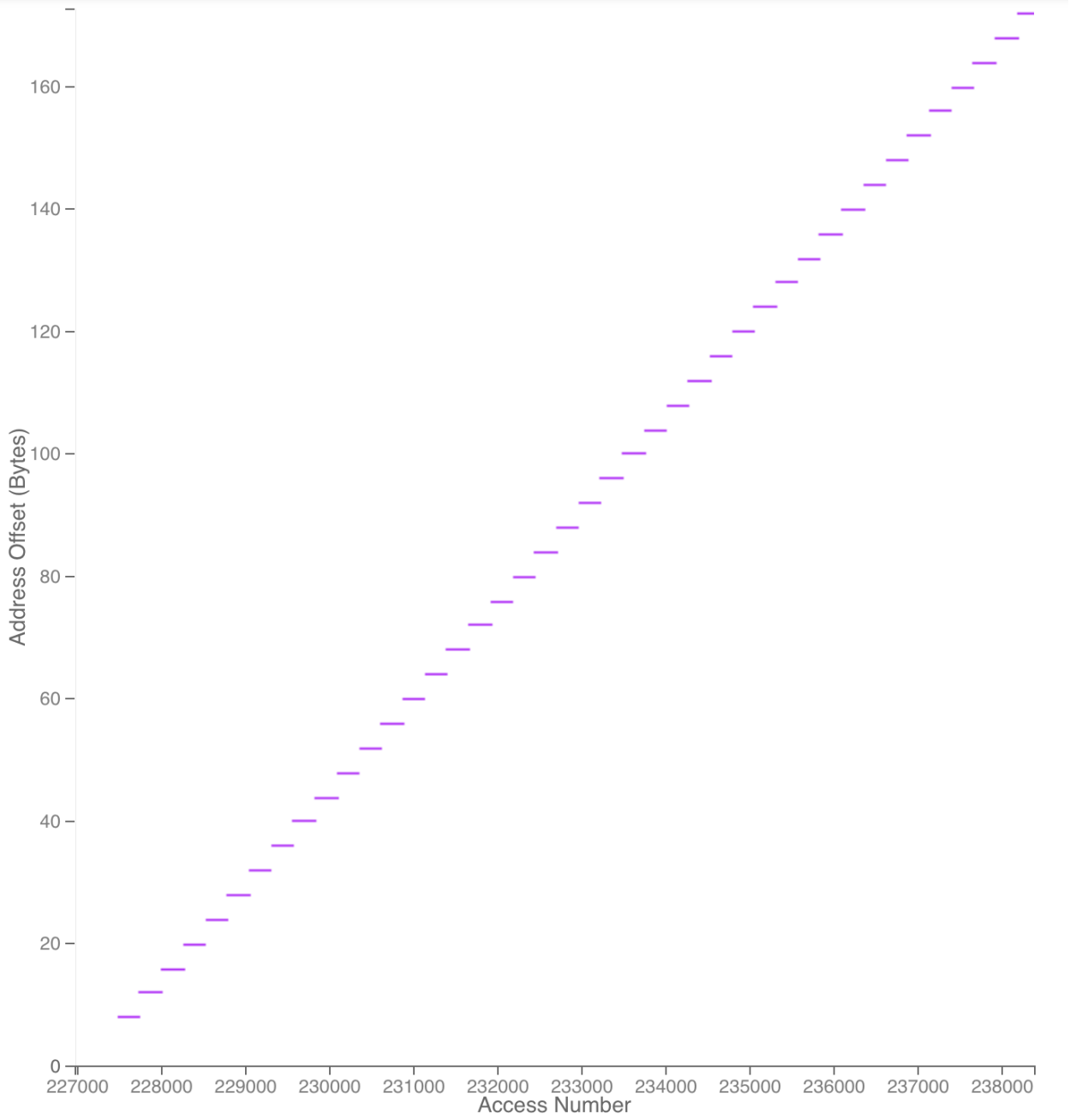
The kernel is in light blue and labeled with tag 'kernel'. Let's zoom in:



The chunking structure shows up as a "stair step" pattern. Each of the light blue blocks shows the portion of the `kernel` tensor the code is processing for each chunk.

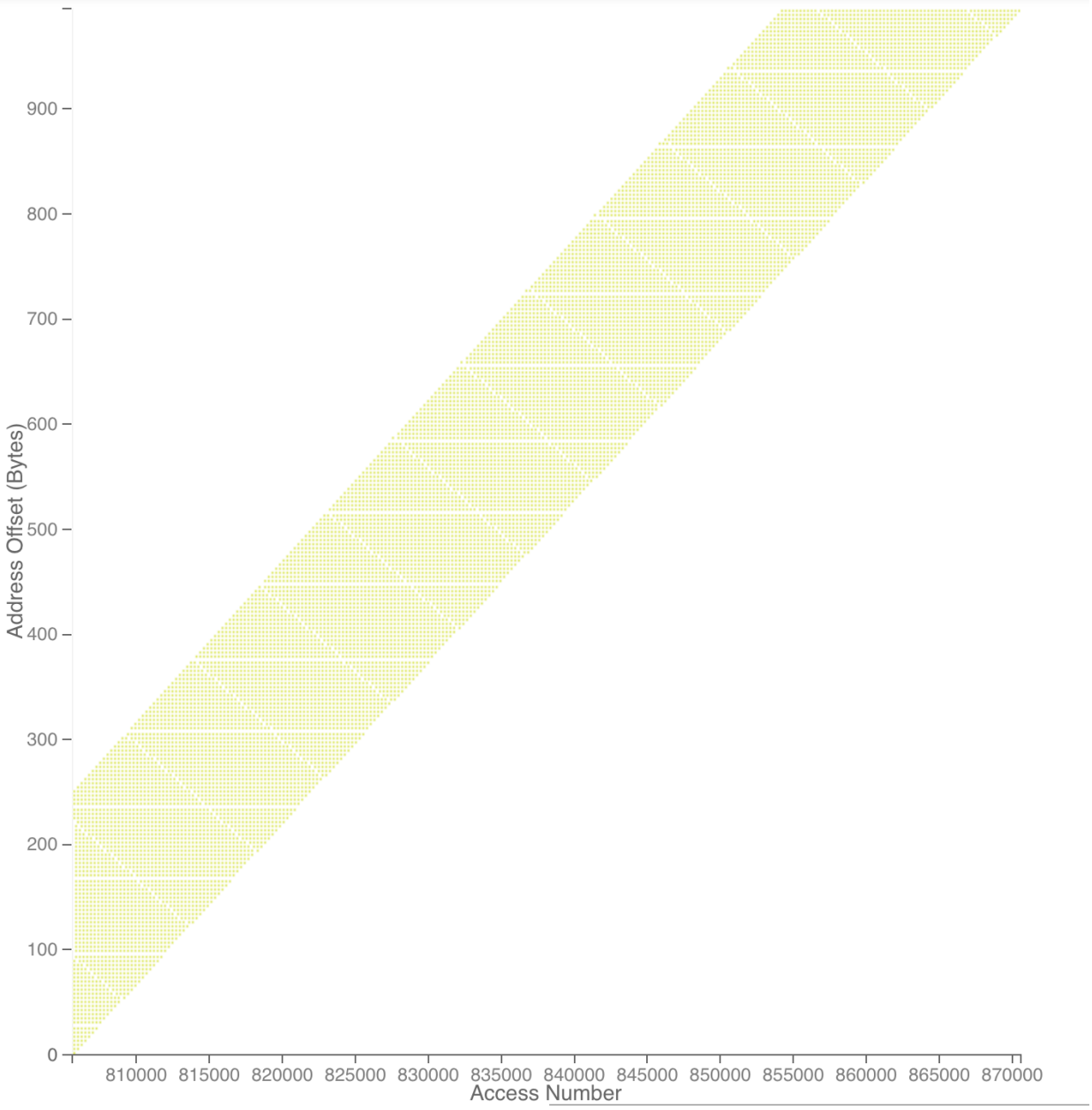
`target`

Zooming waaay in on `target` and we can see individual accesses. You'll see that for each chunk, we make a linear pass over `target`, accessing each element once.



`source`

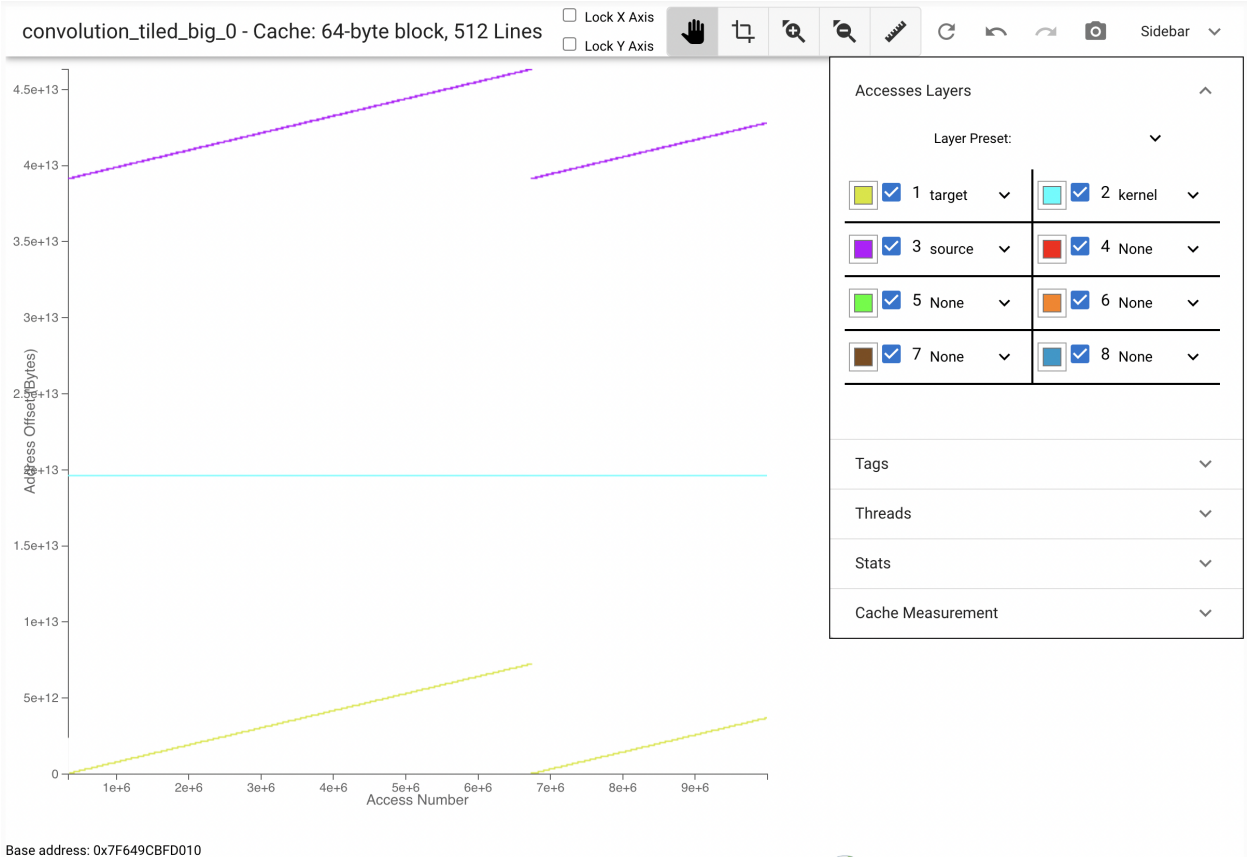
Zooming in on `source`:



The thick stripe is the code accessing each element of the tensor repeatedly for each chunk.

`target` and `source` both exhibit reuse -- one of the two criteria necessary to leverage spatial locality. The second criteria is that the working set fits in the cache.

Let's run it on a data set that won't fit in the L1. The trace shows just 1.5 or so chunks worth of execution. The "sawtooth" pattern would continue if we could trace more accesses.



Let's see how the cache is doing. We'll compare the untiled version and the tiled version on two large datasets:

In [ ]:

```
tiled_run = run(convolution,
                function=["convolution", "convolution_tiled"],
                arguments=arg_map(source_size=32*1024, kernel_size=8*
                          arg_map(source_size=4*32*1024*4, kernel_siz
                perf_counters=["L1-DCACHE-LOAD-MISSES",  "PERF_COUNT_
```

In [ ]:

```
tiled_data = PE_calc(tiled_run.as_df())
display(tiled_data)
```

There are two things to note:

1. Tiling reduced `L1_MPI` by two orders of magnitude (about 100x)
2. For the tiled version, the `L1_MPI` and `CPI` change very little (~1%) as the data set size increased. This is because the tile size (rather than the total data size) sets the working set.

We would hope to see a large boost in performance with tiling. We get a boost, but is it at as large as you'd expect?

Question 20 (Correctness - 2pts)

Consider the speedup provided by each term of the performance equation for the largest data set. Show your work. What contributes to and detracts from achieving a larger, overall improvement in performance?

IC:

CPI:

CT:

Hmmm....

What happened? L1_MPI is low, CPI went down, but ET went up!

So how can we get IC under control?

Let's look at the assembly. On the left is the untiled version. On the right, it is tiled.

In [ ]:
```
compare([convolution[0].asm("do_convolution"),
         convolution[0].asm("do_convolution_tiled")], ["not tiled", "tile
```

Question 21 (Completeness)

Give the labels that mark the beginning and end of the inner loop for the tiled code.

Show Solution

Question 22 (Correctness - 3pts)

For the inner loop body of each version, fill out the table below assuming that `kernel` is 64kB (16k entries) and `source` is 512kB (64k entries).

	# of times executed	static insturctions	dynamic instructions

	# of times executed	static insturctions	dynamic instructions
untiled			
tiled			

Ratio of Tiled IC/Untiled IC:

In the tiled code, there are some extra instructions in the inner loop. They are due to the more complex loop completion condition required by tiling. This eats into our performance!

Question 23 (Completeness)

Modify the cell below (it's the same code as `convolution_tiled()`) to reduce loop overhead and achieve speedup with tiling.

In [ ]:

```
convolution_question = build(code(r"""
#include"cfiddle.hpp"
#include"tensor_t.hpp"
#include"util.hpp"
#include<cstdint>
#include<cassert>

extern "C"
void do_convolution_tiled(const tensor_t<uint32_t> & source,
                          const tensor_t<uint32_t> & kernel,
                          tensor_t<uint32_t> & target, int32_t tile_size)
        disable_prefetcher();
        flush_caches();

        start_measurement();
        for(int32_t jj = 0; jj < kernel.size.x; jj += tile_size) {  // Mov
                for(int32_t i = 0; i < target.size.x; i++) {
                        for(int32_t j = jj; j < kernel.size.x && j < jj +
                                target.get(i,0,0,0) += source.get(i + j,0,
                        }
                }
        }
        end_measurement();
}

extern "C"
void convolution_tiled(uint64_t source_size,  uint64_t kernel_size, int32_
        tensor_t<uint32_t> source(source_size,1,1,1);
        tensor_t<uint32_t> kernel(kernel_size,1,1,1);
        uint64_t target_size = source_size - kernel_size;
        tensor_t<uint32_t> target(target_size,1,1,1);

        do_convolution_tiled(source, kernel, target, tile_size);
}
""", file_name="convolution_question.cpp"), build_parameters=arg_map(OPTIM

cq_run = run(convolution_question, function="convolution_tiled", arguments



cq_data = PE_calc(cq_run.as_df())
display(tiled_data) # this is the data from above for comparison
display(cq_data)
```

Show Solution

Question 24 (Completeness)

Go back up to the fiddle you worked on earlier. Try to reduce loop overhead due to unrolling. Feel free to start with the code for `convolution_tiled_unrolled()`.

Show Solution

Question 25 (Optional)

Take what you've learned about loop unrolling and see if you can speedup the `baseline` implementation *without* tiling. Can you beat the `tiled-split` version?

Question 26 (Optional)

I'm divided on whether the `if` around the two versions of the inner loop is elegant or ugly. Can you find a more elegant way to express the loop bound that allows the compiler to unroll the loop effectively? Do you think it makes the performance characteristics of the code more or less maintainable? ("maintainable" in this case means that it is unlikely that someone later will inadvertently change the code in a way the causes the compiler to no longer unroll the loop correctly)

15.2.4 Tuning Tile Size

Shockingly, we aren't done with 1D convolution yet! Like a zillion cells ago, we picked 64 as the value for `tile_size` ? Is that a good choice?

We could try to do some math to figure out exactly how big it should be, but at this point, you're tired and so am I. Let's just check experimentally. We'll run it with powers of 2 from 8 to 8k and see what's best. (We'll skip 1, 2, or 4 because they are not multiples of 8 -- our unrolling factor). This take a while.

In [ ]:
```
tuning = run(convolution,
             function="convolution_tiled_split",
             arguments=arg_map(source_size=4*32*1024*4,
                               kernel_size=8*1024*4,
                               tile_size=exp_range(8,8192,2))
             perf_counters=["L1-DCACHE-LOAD-MISSES", "PERF_COUNT_HW_INSTR
```

In [ ]:
```
tuning_data = PE_calc(tuning.as_df())
display(tuning_data)
```

In [ ]:
```
plotPE(df=tuning_data, what=[("tile_size", "IC"),("tile_size", "CPI"),("t
```

We want to minimize `ET`. There's pretty broad minimum area where the `IC` (why does increasing `tile_size` reduce IC?) is pretty low and `CPI` is not too high due to `L1_MPI` shooting up when `tile_size` gets big enough to blow out the L1 cache. 512 looks like a good value.

In [ ]:
```
display(tiled_data) # this is the data from above for comparison
display(ca2_data)
display(tuning_data)
```

Turing `tile_size` gave us an additional 6.4/5.9 = 1.08x. This is mostly from reducing `IC`. Interestingly, we also dropped `L1_MPI` by a factor of ~10, although it was already very low, so the impact is negligible.

There's one more things we'll try before call it a day: We can help the compiler a bit more by converting `tile_size` to a constant:

In [ ]:
```
convolution[0].source("do_convolution_tiled_fixed_tile")
```

We'll measure it below.

15.2.5 Final Measurement

Let's re-run all versions to see how performance improved:

In [ ]:
```
# Versions with naive tile size
to_run = arg_map(executable=convolution,
                 function=["convolution",
                           "convolution_tiled",
                           "convolution_tiled_unrolled",
                           "convolution_tiled_split"],
                 arguments=arg_map(source_size=4*32*1024*4,
                                   kernel_size=8*1024*4,
                                   tile_size=64),
                run_options=arg_map(MHz=3500))
#tuned tile size
to_run +=arg_map(executable=convolution,
                 function=["convolution_tiled_split",
                           "convolution_tiled_fixed_tile"],
                 arguments=arg_map(source_size=4*32*1024*4,
                                   kernel_size=8*1024*4,
                                   tile_size=512),
                run_options=arg_map(MHz=3500))

everything = run_list(to_run,
                      perf_counters=["L1-DCACHE-LOAD-MISSES", "PERF_COUNT_
```

In [ ]:
```
everything_data = PE_calc(everything.as_df())
everything_data["speedup"] = everything_data.iloc[0]["ET"]/everything_data
display(everything_data)
```

In [ ]:
```
plotPEBar(df=everything_data, what=[("function", "IC"),("function", "L1_M
```

Overall, we got about 2.13x speedup!

Question 27 (Correctness - 2pts)

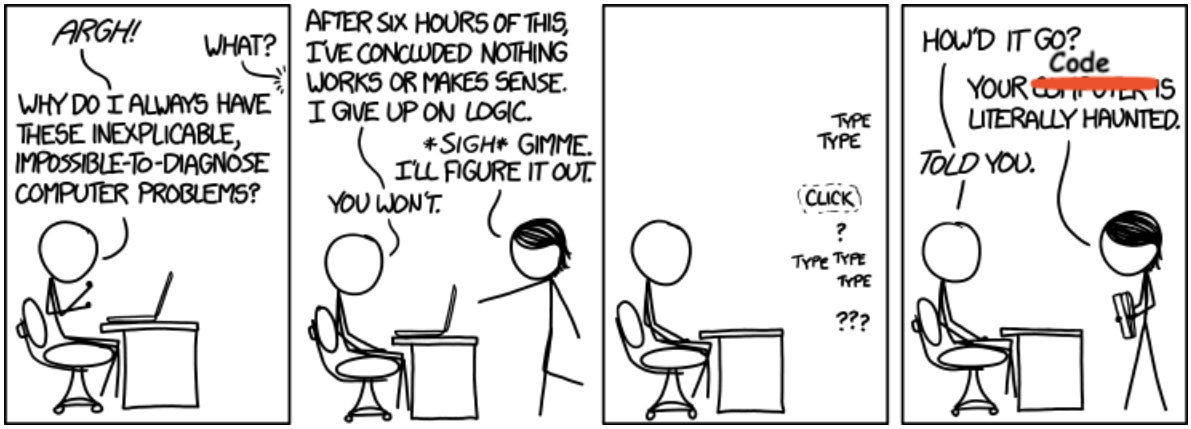
Which optimization provided the largest incremental speedup (i.e., the biggest speedup relative to the version before it)? Why do you think this is?

Question 28 (Optional)

Do the following:

1. In `convolution.cpp`, search for `CURSED`
2. Turn the `0` to a `1`
3. Recompile `convolution.cpp` by re-running the `build()` command at the top of this section
4. Re-run the experiment above.

What changed? Why? How can you fix it?



15.2.6 Discussion

Let's take a moment to think about why that was so complicated. Let's address three questions.

First question Why was it so hard to realize significant performance improvements for 1-D convolution by reducing cache misses?

There are two main reasons: First, the inner loop of convolution is very small, so the extra loop overhead from tiling really killed us. If the loop body had been larger, the relative impact of the loop overhead would have been smaller. This is a good example of Amdahl's Law.

Second, there was no loop-carried dependence between loads: None of the loads in one iteration needed to finish before the loads in the next iteration could begin. This means there was a lot of instruction level parallelism (ILP, which you are just learning about in 142) and, in particular, there is a lot of memory parallelism. This mean that multiple loads (and probably multiple cache misses) ran in parallel. When multiple long-latency operation run in parallel, we say their latency is "hidden".

If the loads were dependent, the impact of the cache misses would have been larger, so reducing them would have had bigger impact. We'll talk about that more in Lab 5.

Second question What lessons should you take away from this example?

The most important lesson is about the process: I made small changes, measured their impact, studied the code (and the assembly) to understand the cause, and made changes to try to improve things. In the real world, I would have run regressions tests repeatedly as well.

A secondary lesson is the "trick" with the `if` statement to make the common case faster. You should *not* just decide to apply that trick to every loop you encounter because it makes the code harder to read, but it is a good tool to have. Like all manual optimizations, though, it should only be applied when you have data to suggest it'll work.

Third Question Couldn't the compiler do this for me?

Maybe. Some commercial compilers can do loop tiling automatically, and we'll see that gcc can do something similar in the next lab. I don't believe gcc has a tiling optimization pass.

Fourth Questions Should you tile by hand? As with all optimizations, the answer depends on whether the code is slow. If this had been a real program, we would have

1. Made sure we were compiling with `-O3`.
2. Profiled to determine that `convolution()` was an performance-critical piece of code.
3. Looked at the assembly to see what the compiler was doing.
4. Check the compiler man page for flags that might help (e.g., is there `-ftile-loops`?).
5. Then, consider tiling by hand.

Getting good speedup from tiling is not especially easy. I originally intended this section of the lab to be much shorter, but, as you've seen, my initial efforts were disappointing.

Final Observation It's worth noting that in this example, we spent roughly equal amounts of time figuring out how to reduce the number of cache misses (which reduced CPI) and then mitigating the negative impacts on IC due to increased loop overhead. This illustrates that implementing some optimizations incurs a cost that limits the impact of the optimization. We then have to optimize the optimization to reduce that cost. We will see this again in Lab 5, as we try to use multiple processors to speed up code without the overhead spoiling the party.

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

16 Programming Assignment

Your programming assignment for this lab is to implement a memory allocator. You've used memory allocators a lot: in C++ you access them via `new` and `delete`. In C, it's `malloc()` and `free()` (which `new` and `delete` call internally). The default implementation of `malloc()` and `free()` are general purpose: they are reasonably fast and can allocate data of any size. However, many applications need to allocate many, many objects very quickly, so a general-purpose allocator is too slow. In these cases, it makes sense to implement a specialized memory allocator because they can be much faster.

In this lab, you'll be implementing a specialized memory allocator that has these basic characteristics:

1. It can only allocate objects of a single type (and therefore a single size).

2. It will ensure those objects are aligned to a configurable size.
3. The allocator will be an object that can allocate and deallocate objects using a simple interface (see below).
4. When the allocator is deleted, it will automatically deallocated all objects that are still allocated.

Here's the reference implementation:

In [ ]:
```
render_code("ReferenceAllocator.hpp")
```

First, note that it's a template class that takes two parameters:

- `T` : the type it will allocate. You can assume the size of `T` no larger than 4096 bytes.
- `ALIGNMENT` : The alignment of the allocations it will make. Alignment size can be powers up 2 between 8 and 4096 bytes (8, 16, 32, 64, etc.).

The allocator has just four methods:

- `ReferenceAllocator()` : the constructor.
- `alloc()` allocates and returns an instance of `T` .
- `free()` deallocates `p` , a previously allocated instance of `T` .
- `~ReferenceAllocator()` : The destructor for the allocator. It needs to clean up all the memory the allocator manages.

It also defines `ReferenceAllocator::ItemType` , which let's us access the type the allocator allocates and `ReferenceAllocator::Alignment` which gives us access to the alignment size.

The implementation above just relies on the standard library's alignment-aware interface to the general-purpose allocator. This means it's not optimized to exploit the fact that we only need to allocate a single size of object, and this is where the big optimization opportunities lie.

Your allocator will not rely on any of the normal memory-allocating library calls. Instead will use a very simple allocator to allocate memory in bulk from the operating system. The interface is called `ChunkAlloc` :

In [ ]:
```
render_code("ChunkAlloc.hpp")
render_code("ChunkAlloc.cpp")
```

`ChunkAlloc` gets memory the same way `malloc()` does: By calling the `mmap()` system call which stands for "memory map". `mmap()` is a great tool and can do many things. You can read about it here (https://man7.org/linux/man-pages/man2/mmap.2.html), but it's not necessary for the lab. What is important for our purposes is that `alloc_chunk()` will return a 128kB region of memory that is 4kB-aligned.

16.1 Detailed Requirements

You're going to build your own version of `ReferenceAllocator` called `AlignedAllocator`. You'll find a copy of `ReferenceAllocator.hpp` in `AlignedAllocator.hpp`. Do your work there.

Now that you understand the basics of how `ReferenceAllocator` works, here's the detailed list of requirements for `AlignedAllocator`:

1. All addresses that `AlignedAllocator::alloc()` returns must be aligned to `ALIGNMENT` so `addr % ALIGNMENT == 0`.
2. All addresses that `AlignedAllocator::alloc()` returns must point to at least `sizeof(T)` bytes of memory.
3. `AlignedAllocator::alloc()` needs to set all bytes of memory that the instance of `T` will occupy to zero before constructing `T`.
4. `AlignedAllocator::alloc()` needs to construct an instance of `T` in the memory using the in-place `new` operator (see below).
5. `AlignedAllocator::alloc()` cannot return the same pointer twice unless the pointer has been deallocated with `AlignedAllocator::free()` first.
6. After the destructor completes, `AlignedAllocator` must have called `free_chunk()` for every chunk it allocated with `alloc_chunk()`. i.e., `get_allocated_chunks()` must return 0.
7. You are free to use the STL data structures for the internals of `AlignedAllocator`, but `alloc()` may not return any memory that is a part of an STL container.
8. The only mechanism you can use to allocate memory is `alloc_chunk()/free_chunk()`. No calls to `malloc()` (or other functions from the standard library that allocate raw memory) or `new` (other than the "in place" version.) to allocate the space `AlignedAllocator` will return.
9. Your allocator must recycle: If memory is returned to it via `free()`, your allocator should reallocate that memory before requesting new memory via `alloc_chunk()`. This prevents your allocator from continually allocating new memory, which is very fast, but not a realistic solution.

`ReferenceAllocator` already satisfies all of these except the last two: `ReferenceAllocator` uses `posix_memalign()` which is forbidden in your solution. With respect to recycling, I'm not sure how `posix_memalign()` works internally, so I'm not sure how it recycles, but it does something reasonably efficient. You'll find that removing `posix_memalign()` and meeting the above criteria will require you to rewrite most of the code in `AlignedAllocator.hpp`.

16.2 Evaluation

Your implementation will be evaluated based on correctness and performance. Your implementation of `AlignedAllocator` must pass the tests in `run_tests.cpp` which cover the requirements listed above. There will be hidden test cases.

The performance portion is based on the average score over several benchmarks in `Allocator.cpp`. Details of the grade calculation are given under "Final Measurement" below.

The code for the benchmarks is in `Allocator.cpp`. The code is below. The key functions are `bench()`, `microbench()`, `exercise()`, and `miss_machine()`. Here's what they do:

- `microbench()` just calls `alloc()` many times and records the execution time. Then it does the same for `free()`.
- `exercise()` allocates a bunch of objects, deletes some at random, allocates some more, deletes some at random, etc. This puts your allocator into a "warmed up" or "well-used" state to approximate how it would behave in a long-running program.
- `bench()` measure the execution time of `exercise()`.
- `miss_machine()` measures the speed of a specialized version of the miss machine that tests how well your allocator manages spatial locality (more below)

The `*_solution` functions near the bottom are wrappers to run each benchmark. Pay attention to where `start_measurement()` and `end_measurment()` are called.

In [ ]:
```
render_code("Allocator.cpp")
```

16.3 `miss_machine()`

`miss_machine()` is the most interesting of the three benchmark functions because it addresses one of the subtle problems that can arise with a memory allocator.

Because the memory allocator controls the layout of a program's memory, it can have a strong impact on how a program accesses memory. For instance:

1. If the allocator is poorly designed (or unlucky) it might arrange memory so that there are many conflict misses.
2. Depending on how effectively the allocator reclaims space that is freed, it may waste space.
3. How the allocator places objects can affect which objects are close enough to benefit from spatial locality.

The `miss_machine()` function in `Allocator.cpp` explores the third issue. Here's the code again:

In [ ]:
```
build("Allocator.cpp")[0].source(show=("//BEGIN", "//END"))
```

Here's what the benchmark does.

1. It create an allocator and warms it up with `exercise()`.
2. It builds a miss machine out of links allocated one-at-a-time from your allocator. This is different than description of the miss machine given earlier in the lab: In that case we allocated the links in an array, guaranteeing good spatial locality.
3. Then it measures the execution time of a call to `do_misses()`, which traverses the miss machine.

Read through the code and comments carefully to understand it.

We are going to run `miss_machine` with `count = 4096`. Since each `MissingLink` is 8 bytes, 4096 links *should* fit in the L1 cache, since `4096*8 = 32*1024`. Let's see how the allocators do:

In [ ]:
```
alloc = build("Allocator.cpp", arg_map(OPTIMIZE="-O3"))
miss_machine_run = run(alloc, function=["miss_machine_solution"],
                       arguments=arg_map(link_count=4096, access_count=10
                       perf_counters=["L1-DCACHE-LOAD-MISSES", "PERF_COUN
                      )
```

In [ ]:
```
miss_machine_data = PE_calc(miss_machine_run.as_df())
display(miss_machine_data)
```

I get an `L1_MPI` of 0.2. Let's take a look at the CFG for `do_misses()`:

In [ ]:
```
alloc[0].cfg("do_misses")
```

As you can see, there are only 5 instructions and one of them is an access. The load which accounts for 0.2 of the instructions in the loop body. Since MPI of 0.2, means that the miss rate for the `movl` in the loops is around 100%.

Your task is to speed up `do_misses()` and the best way to do that is to reduce MPI by allocating `MissingLinks` in such a way that you maximize spatial locality across the miss machine.

A couple of things to keep in mind:

1. This benchmark is only 12.5% of your grade, and your work on the other benchmarks will get you most of those points. It's also the hardest part. Work on it last.
2. A locality-aware allocator is probably more complex than a non-locality-aware allocator. If you add too much complexity, it will reduce performance on the other benchmarks.
3. Remember that the key to spatial locality is to fit your data (in this case the links in the miss machine) into the smallest possible number of cache lines possible.

16.4 Useful C++

C++ likes to keep you safe by not letting you convert pointers to integers or letting you convert pointers from one type to another. However, memory allocators need to break the rules occasionally to transform untyped bytes into objects. The main tool for this is `reinterpret_cast<>()` and the "in place" `new` operator. `ReferenceAllocator` provides an example of how to use both mechanisms.

16.4.1 `reinterpret_cast<>()`

`reinterpret_cast<>()` let's you change a values from one type to another as long as they are the same size. So you can do this:

```
char *x;
int *y = reinterpret_cast<int *>(x);

// or

void * x = alloc_chunk();
T * t = reinterpret_cast<T*>(x);
```

If you are familiar with C's `(T*)(x)` casting syntax, `reinterpret_cast` is equivalent, but preferred because it's easier to spot in code.

A related tool is `uintptr_t`, which is an unsigned integer that is the same size as a pointer. So you can increment a pointer by one byte doing this:

```
int * x = new int;
uintptr_t n = reinterpret_cast<uintptr_t>(x);
n += 1;
x = reinterpret_cast<int *>(n);
```

Note that this is different than;

```
int *x = new int;
x++;
```

Since, under C++'s pointer arithmetic rules, if you increment a pointer of type `T*`, it actually increases the address by `sizeof(T)` (i.e., 4 bytes for an `int`).

16.4.2 In-Place `new`

If you call `new T`, C++ will allocate some memory to hold a new instance of `T` and then run `T`'s constructor on it. But what if *you* want to decide where to construct the new instance?

In that case you can say something like this:

```
void * p = alloc_chunk();
new (p) T;
```

To construct a new instance of `T` "in place" in the memory pointed to by `p`. Or, if you wanted to initialize an instance of `T` starting at the 11th byte after `p`, you could do this:

```
uintptr_t n = reinterpret_cast<uintptr_t>(p);
n += 11;
void *q = reinterpret_cast<void*>(n);
new (q) T;
```

Why would anyone ever want to do that?

16.5 How To Do This Lab

Here's some tips about how to approach this lab.

16.5.1 The Lifecycle of a Memory Allocator

Here are the basic steps that your memory allocator must accomplish over it's lifetime.

1. Initialize itself and its internal data structures.
2. Respond to calls to `alloc()`
   A. If the allocator has recycled objects, initialize one and return it.
   B. Otherwise, if the allocator has new (not recycled) memory on hand, initialize one object's worth and return it.
   C. Otherwise, if the allocator has no memory on hand, call `alloc_chunk()`, and goto 2.2.
3. Respond to calls to `free()`
   A. Store the object somewhere so it can be recycled.
4. On destruction, called `free_chunk()` to deallocate all the memory you allocated with `alloc_chunk()`.

16.5.2 Thing to Try

Here are some ideas about how to get started:

1. The main difference between what you'll implement in `AlignedAllocator` and `posix_memalign()` is that `AlignedAllocator` only needs to allocate objects of a single size. You can exploit this fact to improve performance.
2. `alloc_chunk()` let's you allocate enough space to store many objects. Since the objects are all the same size and alignment, you can calculate where each of instance of `T` will reside with the chunk.
3. You need to recycle. So, think about how you can efficiently store `free()`ed memory while it's waiting to be re-`alloc()`ed.
4. You will need to choose the right data structures and algorithms to achieve good performance. Think about what each data structure *needs* to do and what operations are most important to performance. Use the STL! (In past labs, I noticed many students implementing things that the STL already provides. Don't re-invent -- or debug -- the wheel!)

Your overall score is based on your allocator's performance across eight benchmarks. This might seem daunting, but the performance of the benchmarks is very correlated: If you speed up your allocator for one of them, it will get faster for many of the others.

With that in mind, start with the simplest ones and go from there. I'd proceed in this order:

1. The `microbench` function.
2. The `bench` function.
3. The `miss_machine` function.

16.6 Do Your Work Here

Below are the key commands you'll need to make progress on the lab. Your solution should go in `AlignedAllocator.hpp`:

16.6.1 Compiling and Running

You can compile and the benchmarks locally using this command. This is only useful for debugging. Performance running locally is not very meaningful:

In [ ]:
```
alloc = build("Allocator.cpp", arg_map(OPTIMIZE="-O3", MORE_SRC="ChunkAll
```

In [ ]:
```
to_run = arg_map(executable=alloc,
             function=["allocator_bench_solution"],
             arguments=arg_map(count=128*1024*16, seed=42),
             run_options=arg_map(MHz=3500))
to_run += arg_map(executable=alloc, function=["allocator_microbench_solut
              arguments=arg_map(count=1600000, seed=42),
              run_options=arg_map(MHz=3500))
to_run += arg_map(executable=alloc, function=["miss_machine_solution"],
              arguments=arg_map(link_count=4096, access_count=100000000,
              run_options=arg_map(MHz=3500))

alloc_run = run_list(to_run,
                     perf_counters=["L1-DCACHE-LOAD-MISSES",  "PERF_COUNT
```

In [ ]:
```
alloc_run_data = PE_calc(alloc_run.as_df())
display(alloc_run_data)
```

If you want to just run your solution in `AlignedAllocator.hpp` , remove the "starter" functions from the list of functions run above. You can run them in the cloud by removing `with local_exeuction():` .

16.6.2 Drawing Graphs

In [ ]:
```
plotPEBar(df=alloc_run_data, what=[("tag", "ET"), ("tag", "L1_MPI")])
```

16.6.3 Running the Regressions

There's a test suite that your code must pass and that the autograder will run. You can run it like this:

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

16.7 Tools

These are some tools you might find useful as you optimize your implementation. I encourage you to give some of them a try.

16.7.1 Debugging Regressions

If a regression fails, `run_tests.exe` will tell you which test failed. Here are some tips for debugging. First, get a list of the tests:

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

One of them will be the test that failed. Then you can debug in gdb (at a terminal again):

```
bash$ gdb run_tests.exe
(gdb) run --gtest_filter=<name_of_failing_test> --gtest_break_on_fa
ilure
```

The `--gtest_filter` just runs one test. and `--gtest_break_on_failure` will stop drop you into the debugger if the error occurs.

16.7.2 Looking At Assembly

As you learned in the previous lab, name mangling makes it a little tricky to inspect the details of what the compiler does to C++ code, especially when it uses templates. So let's see how we can track down the assembly for for your implementation.

The cell below filters through assembly for the compiled version of `Allocator.cpp` to find the methods in `AlignedAllocator`:

In [ ]:
```
print("\n".join([l for l in alloc[0].raw_asm().split("\n") if l.startswitl
```

You can see that there are several different versions of each method, one for each set of template parameters.

Then, you can pass one of those lines to `alloc[0].asm()` to see the assembly. For example:

In [ ]:
```
alloc[0].asm("AlignedAllocator<unsigned char [3], 16ul>::~AlignedAllocato
```

If you'd prefer you write it to a file:

In [ ]:
```
with open("out.s", "w") as out:
    out.write(alloc[0].raw_asm("AlignedAllocator<unsigned char [3], 16ul>
!cat out.s
```

16.7.3 Debugging

Using the debugger in this lab is the same as the other labs. Please check the documentation in those labs.

16.8 Final Measurement

When you are done, make sure your best allocator is called `AlignedAllocator` in `AlignedAllocator.hpp`. Then you can submit your code to the Gradescope autograder. It will run the commands given above and compute your grade.

Your grade is based on your speed up relative `ReferenceAllocator` on the eight benchmarks.

For each of them, there's a target speedup given the table below.

You don't get extra credit for beating the targets. This will help ensure that your design in balanced: You must do well at all 8 benchmarks to do well on the lab.

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

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

After you run it, the results will be in `autograde/bench.csv`, `autograde/microbench.csv`, and `autograde/miss_machine.csv` rather than `./bench.csv`, `./microbench.csv`, and `miss_machine.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 `AlignedAllocator.cpp`.
2. Copies those files into a clean clone of the starter repo.
3. Runs your code using `run_bench.py`
4. It then runs `autograde.py` to compute your grade.

Running the cell below will do the same thing as the Geradescope autograder. And the cell below shows the name and target speedups for each benchmark. This takes 1-2 minutes to run.

In [ ]:
```
!cse142 job run --take AlignedAllocator.hpp  --lab caches2-bench --force
```

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

You can check the performance results like this:

In [ ]:
```
#run it locally
!./autograde.py --submission autograde --results autograde.json

from autograde import compute_all_scores
df = compute_all_scores(dir="autograde")
display(df)
print(f"total points: {round(sum(df['score']), 2)}")
```

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

You can also inspect the autograder's output.

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"
```

17 Recap

This lab completes our tour of (single processor) memory systems. It explored what's required to exploit temporal locality and when it does and does not exist. It also looked at other key components of the memory hierarchy: The lower-level caches and the TLB. Finally, it developed an optimized version of 1-D convolution using tiling and renesting. You should now be well-prepared for the next lab, where we will explore (among other things) how multiple processors further-complicate the performance of the memory hierarchy.

18 Turning In the Lab

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

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

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

18.1 The CSE142L Emergency Lab Submission Form

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

Here's the process:

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

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

18.2 Reading Quiz

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

18.3 The Note Book

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

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

Step 1: Save your workbook!!!

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

Step 2: Run this command:

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

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

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

Step 4: Hide the table of contents by clicking the

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

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

Step 7: Turn in that PDF via gradescope.

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

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

18.4 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 [ ]:
```
from IPython.display import IFrame
IFrame('https://docs.google.com/forms/d/e/1FAIpQLSdEyaIDy52FLLUzQEXoJJmz7
```