In [4]:

```
%load_ext autoreload
%autoreload 2
from notebook import *
# if get something about NUMEXPR_MAX_THREADS being set incorrectly, d
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

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

3 Pre-Lab Reading Quiz

Part of this lab is a pre-lab quiz. The pre-lab quiz **has moved to Canvas** so I can allow multiple attempts. It is due **before class on the day the lab is assigned**. It's not hard, but it does require you to read over the lab before class. If you are having trouble accessing it, make sure you are **logged into Canvas**.

3.1 How To Read the Lab For the Reading Quiz

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

1. What's involved in the lab.
2. The key concepts of the lab.
3. What you can expect from 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.

3.2 Taking the Quiz

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

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

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

4 Browser Compatibility [...]

5 About Labs In This Class [...]

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

7 Grading

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

Part	value
Reading quiz	3%
Jupyter Notebook	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.

8 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 that 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 the same code we looked at in the last lab:

In [ ]:

```
t = fiddle("stride.cpp", function="stride", name="spatial", opt="-O1"
code=r"""
#include"pin_tags.h"
#include"CNN/tensor_t.hpp"
#include"function_map.hpp"
#include<cstdint>

extern "C"
uint64_t* stride(uint64_t * data, uint64_t size, uint64_t arg1) {
    tensor_t<uint32_t> t(size,1,1,1, (uint32_t *)data);
    TAG_START("init", t.data, &t.as_vector(t.element_count()), true);

    for(uint i = 0; i < arg1; i++) {
        for(uint x = 0; x < size; x+=arg1) {
            t.get(x,0,0,0) = x;
        }
    }

    TAG_STOP("init");
    return data;
}

FUNCTION(one_array_1arg, stride);
""")
compare([t.source, t.cfg])
```

We are going to run it again 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.

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`. Assume we run `stride()` 10,000 times with the same values of `data` and `size`.

Cache misses for size = 1024:

Cache misses for size = 4096:

Cache misses for size = 16384:

Show Solution

Question 2 (Completeness)

How well do your predictions match the results?

Show Solution

Moneta has a simple cache simulator built into that can estimate how a cache will behave and let us visualize the results. It models a fully-associative cache. Run the next two cells to see.

In [ ]:
```
t = fiddle("stride.cpp", function="stride", name="temporal", run=["mo
           cmdline=f"--size 1024 4096 16384 --arg1 16 --iters 1",
           perf_cmdline="--stat-set L1.cfg --MHz 3500 --calc misses_p
```

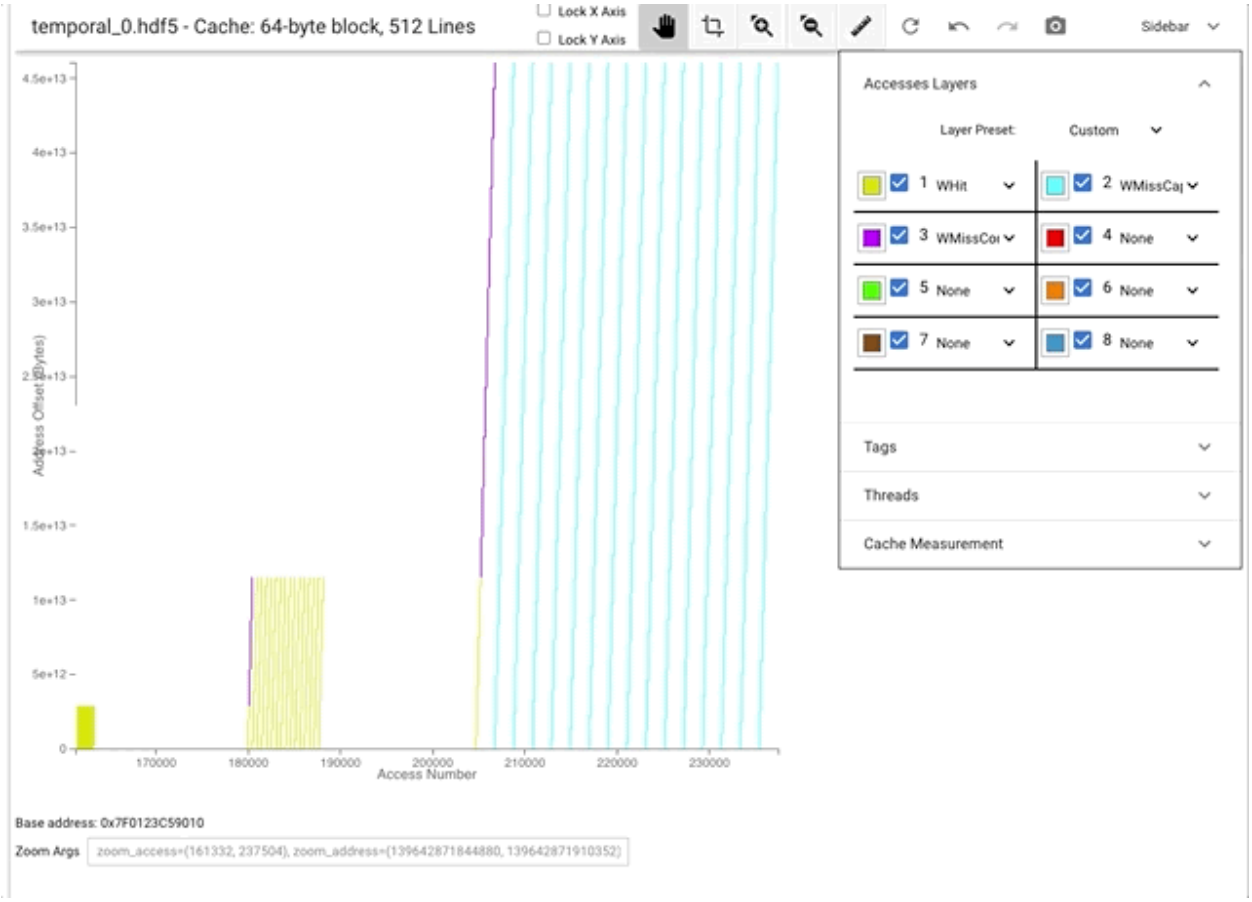
In [ ]:
```
show_trace("./temporal_0.hdf5", show_tag=["init0","init1","init2"], l
```

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

The green line on the y-axis shows the size of the L1 cache (32kB), and 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 probably blurred together for 1024), and if you zoom in you can see that the stride is 64 bytes (try it!). As you can see, when the region of memory is larger than the cache, the whole block turns gold instead of purple.

This is a good chance to use Moneta's measurement tool: Click on the ruler icon above the Moneta graph and use it to select the the middle block of accesses. A display on the right will open up with some statistics about that region of the plot:



The "Cache Measurement" heading provides information about the cache that Moneta is simulating. Under "current selected area", you can see that the selected area contains accesses to 256 distinct cache lines. You can also see the cache miss rate for that area.

Question 3 (Correctness - 2pts)

How many distinct cache lines are accessed with `size = 16384` ? Use the measurement tool to check.

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

Question 4 (Completeness)

How will increasing the stride to 64 affect the number of misses when `size = 16384` ?

Show Solution

Question 5 (Correctness - 2pts)

Why did increasing the stride size reduce the cache miss rate even though `size` remained the same?

9 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).

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()`. The crazy `&(*a), &(*a)+ 1` is just to set the bounds for the Moneta tag so we know where the `set` lives in memory. Run the cell, and it'll open up the resulting trace in Moneta.

9.1 One Data Structure [...]

10 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.

10.1 Capacity and Compulsory Misses [...]

10.2 Conflict Misses

Question 6 (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 early 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 [ ]:

```
from notebook import *

t = fiddle("conflict.cpp", code="""
#include"pin_tags.h"
#include"CNN/tensor_t.hpp"
#include"function_map.hpp"
#include<cstdint>

extern "C"
uint64_t* conflict(uint64_t * data, uint64_t size, uint64_t arg1) {
    tensor_t<uint32_t> t(size,1,1,1, (uint32_t *)data);
    TAG_START("init", t.data, &t.as_vector(t.element_count()), true);

        for(uint x = 0; x < size; x+=arg1) {
            t.get(x,0,0,0) = x;
        }
    TAG_STOP("init");
    return data;
}

FUNCTION(one_array_1arg, conflict);
""")

stride16 = fiddle("conflict.cpp",
           function="conflict", opt="-O1",
           run=["perf_count"], name="stride16",
           cmdline=f"--size {16*64} --arg1 16  --iters 10000",
           perf_cmdline="--stat-set L1.cfg --MHz 3500")
stride1024 = fiddle("conflict.cpp",
           function="conflict", opt="-O1",
           run=["perf_count"], name="stride1024",
           cmdline=f"--size {1024*64} --arg1 1024 --iters 10000",
           perf_cmdline="--stat-set L1.cfg --MHz 3500")

df = render_csv(["stride16.csv", "stride1024.csv"])
```

In [ ]:

```
display(df)
plotPEBar(df=df, what=[("arg1", "L1_MPI"),("arg1", "L1_cache_misses")
```

Question 7 (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 product of bad luck: 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. As the example above shows, however, it is not hard to construct programs that are this unlucky. We have a term for these access patterns: We say they are "pathological".

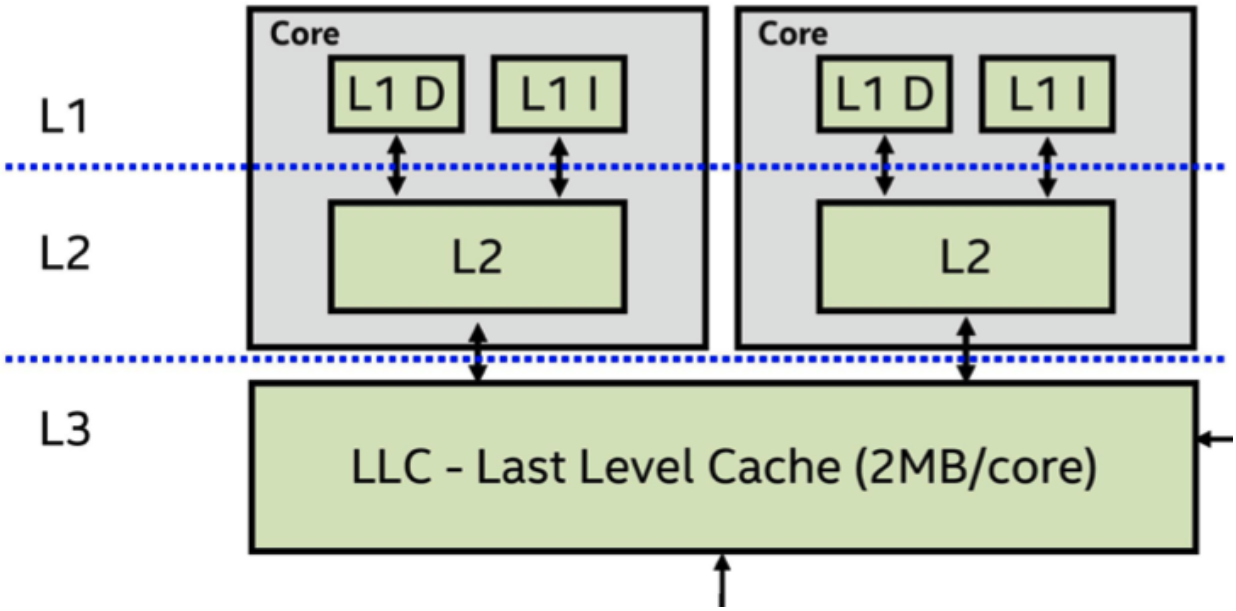
By definition, pathological access patterns are rare, so we don't spend too much time worrying about them. But they can crop up and it's a good idea to be aware of the possibility.

Question 8 (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?

11 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. We may look at the L3 in more details when we study multi-core. For now, we will take a look at the L2. The L2 is 256kB and is 8-way set associative.

The code below is similar to the `stride` function we used in the prevous 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 L2 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.

Question 9 (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`:

L2 critical `size`:

In [ ]:

```
space = lambda x: " ".join(map(str, x))
L2 = fiddle("L23.cpp", function="L23", name="L2", run=["perf_count"],
            code=r"""
#include"pin_tags.h"
#include"CNN/tensor_t.hpp"
#include"function_map.hpp"
#include<cstdint>

extern "C"
uint64_t* L23(uint64_t * data, uint64_t size, uint64_t arg1) {
    tensor_t<uint32_t> t(size,1,1,1, (uint32_t *)data);
    TAG_START("init", t.data, &t.as_vector(t.element_count()), true);

    for(uint i = 0; i < (1 << 20)/size; i++) {
        for(uint x = 0; x < size; x+=arg1) {
            t.get(x,0,0,0) = x;
        }
    }

    TAG_STOP("init");
    return data;
}

FUNCTION(one_array_1arg, L23);
""",
            cmdline=f"--size {space([2**i for i in range(4, 20)])} --a
            perf_cmdline="--stat-set L2.cfg --MHz 3500")

L1 = fiddle("L23.cpp", function="L23", name="L1", run=["perf_count"],
            cmdline=f"--size {space([2**i for i in range(4, 20)])} --a
            perf_cmdline="--stat-set L1.cfg --MHz 3500")
```

In [ ]:

```
data = render_csv("L1.csv")
L2_data = render_csv("L2.csv")
data[["L2_MPI", "L2_cache_misses"]] = L2_data[["L2_MPI", "L2_cache_mi
display(data[["function","size", "arg1", "IC", "CPI", "L1_MPI","L1_ca
plotPE(df=data, logx=2, logy=10, combined=True, lines=True, what=[ ("
```

Question 10 (Completeness)

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

L1 prediction correct?:

L2 prediction correct?:

As always, we are more interested in performance than MPI. Let's see how CPI behaved:

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

Question 11 (Correctness - 2pts)

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

2MB to 1MB?:

512kB to 128kB?:

512kB to 32kB?:

32kB to 8kB?:

12 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 [ ]:
```
render_code("TLB.cpp", show=("//START", "//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:

In [ ]:
```
!make build/TLB.so
do_cfg("build/TLB.so", symbol="sym.MM_4096ul__miss_MM_4096ul____MM_40
```

Question 12 (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 13 (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 14 (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.

13 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: 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!).

13.1 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 [ ]:

```
 x_outside = fiddle("renest.cpp", function="x_outside", name="x_outsi
                code=r"""
#include"pin_tags.h"
#include"CNN/tensor_t.hpp"
#include"function_map.hpp"
#include<cstdint>

extern "C"
uint64_t* x_inside(uint64_t * data, uint64_t size, uint64_t arg1) {
    tensor_t<uint32_t> t(size/arg1,arg1,1,1, (uint32_t *)data);

    TAG_START("x_inside", t.data, &t.as_vector(t.element_count()), tr

    for(uint y = 0; y < arg1; y++) {
        for(uint x = 0; x < size/arg1; x++) {
            t.get(x,y,0,0) = x;
        }
    }

    TAG_STOP("x_inside");

    return data;
}

FUNCTION(one_array_1arg, x_inside);

extern "C"
uint64_t* x_outside(uint64_t * data, uint64_t size, uint64_t arg1) {
    tensor_t<uint32_t> t(size/arg1,arg1,1,1, (uint32_t *)data);

    TAG_START("x_outside", t.data, &t.as_vector(t.element_count()), t

    for(uint x = 0; x < size/arg1; x++) {
        for(uint y = 0; y < arg1; y++) {
            t.get(x,y,0,0) = x;
        }
    }

    TAG_STOP("x_outside");

    return data;
}

FUNCTION(one_array_1arg, x_outside);
""",
            cmdline=f"--size {1024*1024} --arg1 {1024*4}", perf_cmdli

x_inside = fiddle("renest.cpp", function="x_inside", name="x_inside",

                  cmdline=f"--size {1024*1024} --arg1 {1024*4}", perf_
```

In [ ]:

```
show_trace("./x_inside_0", show_tag=["x_inside"], layer_preset=["x_in
```

In [ ]:
```
1 show_trace("./x_outside_0", show_tag=["x_outside"], layer_preset=["x_
```

Amazingly, those two plots contain exactly the same number of 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: With `x` on the inside, the program marches linearly through memory. With the `x` loop outside, it takes large strides through the array. In particular, it doesn't access the same 64 byte cache line again until long after it has been evicted from the cache.

Question 15 (Completeness)

Use Moneta's measurement tool to measure the cache miss rate for both versions of the code.

version	hit rate
x loop inside	
x loop outside	

Question 16 (Completeness)

What value of `--arg1` should result in a very high (e.g., > 95%) hit rate, even with the `x` loop on the outside? Try to reason through the correct value before running any experiments.

Show Solution

13.2 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 spatial 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.

13.2.1 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 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 [ ]:
```
display(IFrame("https://www.youtube.com/embed/ulKbLD6BRJA", width=560
```

The code is below. Run the cell to generate a Moneta trace for a 16kB source and 4kB kernel:

In [ ]:
```
render_code("convolution.cpp", show=("//START", "//END"))
```

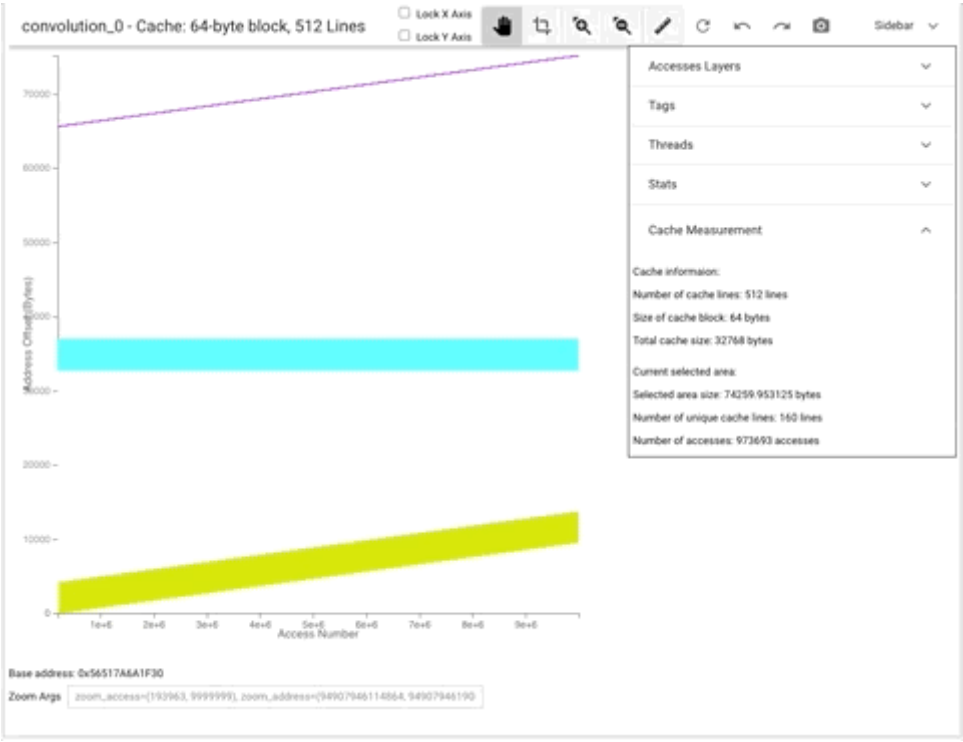
Run the cell below to take a look at the trace.

In [ ]:
```
fiddle("convolution.cpp", function="convolution", analyze=False,
       name="convolution", run=["moneta"], tagged_only=False,
       opt="-O3",
       cmdline=f"--size {4*1024} --size2 {1024} --size3 {4*1024}  --t
show_trace("./convolution_0", show_tag=["source", "kernel","target"])
```

You should see three strips of color. 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.

Let's see how much of the cache the program is taking up. Use the measuring tool to select a narrow, vertical band from top to bottom and spans 1M memory accesses. Check the number of cache lines being touched and the cache hit rate. This is the a rough measure of the working set of

the algorithm. You can drag the measured area around to see how the size of the working set changes (or doesn't).



Question 17 (Completeness)

Based on measurements with the ruler tool, how big (in cache lines) is the working set for this computation? Does it vary throughout the computation?

13.2.2 Memory Behavior in 1-D Convolution

The measurement tool can measure the working set of this computation: if we take a narrow, vertical slice of the trace, that tells us how many cache lines the program is using during that period. If that working set is larger than the cache, than it's likely we will have poor performance.

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()`. 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.

In [ ]:
```
1 do_cfg("build/convolution.so", symbol="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%)? What fraction of the execution is spent on the outer loop?

	Inner loop	Outer loop (excluding the inner loop)
Instructions		
Loads		
Stores	# KEY 1	0

	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

13.2.3 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 [ ]:
```
render_code("convolution.cpp", show="do_convolution_new_loop")
```

In [ ]:
```
fiddle("convolution.cpp", function="convolution_new_loop", analyze=Fa
            cmdline=f"--size {4*1024} --size2 {1024} --size3 {4*1024}
fiddle("convolution.cpp", function="convolution_new_loop", analyze=Fa
        cmdline=f"--size {32*1024} --size2 {8*1024} --size3 {32*1024}
```

Here's the trace on as small data set. It looks just like it did in the original version.

In [ ]:
```
show_trace("./convolution_new_loop_small_0", show_tag=["source", "ker
```

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 that poor datahub's brain can handle). 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.

In [ ]:
```
show_trace("./convolution_new_loop_big_0", show_tag=["source", "kerne
```

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 [ ]:
```
render_code("convolution.cpp", show="do_convolution_tiled")
```

We'll run it first, with a small data set so you can see what the renesting does. Here it is on a small data set:

In [ ]:
```
fiddle("convolution.cpp", function="convolution_tiled", analyze=False
            cmdline=f"--size {4*1024} --size2 {1024} --size3 {4*1024}

show_trace("./convolution_tiled_little_0", show_tag=["source", "kerne
```

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'. If you click on "tags" on the right, you'll see a list of the tags. CLick the magnifying glass next to "kernel" and you'll zoom into the kernel.

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.

If you zoom way in, you'll see that the code repeatedly accesses the whole chunk, updating the sum stored in each entry of tensor.

Rerun the cell above or press the "undo" arrow in the viewer to zoom back out.

`target`

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

`source`

Zoom in on `source`. AT first, it looks like `target`, but if you zoom in further you'll see that the purple lines are "thick": The code accesses each element of the tensor repeatedly during 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 executino. The "sawtooth" pattern would continue if we could trace more accesses.

In [ ]:
```
fiddle("convolution.cpp", function="convolution_tiled", analyze=False
       cmdline=f"--size {32*1024} --size2 {8*1024} --size3 {32*1024}
show_trace("./convolution_tiled_big_0", show_tag=["source", "kernel",
```

Question 20 (Completeness)

Use the ruler tool to measure the working set size by measuring 1M memory operations. What's the cache hit rate?

Let's run the code again with twice as much data. This time, the trace won't even cover 1 chunk:

In [ ]:
```
convolution = fiddle("convolution.cpp", function="convolution_tiled",
            cmdline=f"--size {64*1024} --size2 {16*1024} --size3 {64*


show_trace("./convolution_tiled_0", show_tag=["source", "kernel", "ta
```

Question 21 (Completeness)

Use the ruler tool to measure the working set size by measuring 1M instructions. For larger data, what's the cache miss rate?

So the hit rate doesn't change with the input size and it's very, very high!

Let's compare the performance of the original (un-tiled) version with our tiled version on the same data set. In theory, the tiled version should be much faster because it has better temporal locality:

In [ ]:
```
!make clean
fiddle("convolution.cpp", function=["convolution", "convolution_tiled
            cmdline=f"--size {64*1024} --size2 {16*1024} --size3 {64*
#fiddle("convolution.cpp", function="convolution_tiled", analyze=Fals
#            cmdline=f"--size {64*1024} --size2 {16*1024} --size3 {64
```

In [ ]:
```
render_csv("convolution.csv", columns=["function", "size", "size2", "
```

Hmmm....

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

Ugh! IC rose by 1.3x! Crud. Let's look at the assembly. On the left is the untiled version. On the right, it is tiled. (It might help to open the images below in separate tabs or download and print them. That's what I had to do.)

In [ ]:
```
1 compare([do_cfg("build/convolution.so", symbol="sym.do_convolution"),
```

Question 22 (Correctness - 6pts)

Find the innermost loop body in both CFGs. 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
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. To put it another way: The higher loop overhead increased IC more than better locality reduce CPI.

Use this fiddle to answer the question below:

Question 23 (Completeness)

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

In [ ]:

```
convolution = fiddle("convolution_question.cpp", function="convolution
                code=r"""
#include"pin_tags.h"
#include"CNN/tensor_t.hpp"
#include"function_map.hpp"
#include<cstdint>

extern "C"
void do_convolution_question(const tensor_t<uint32_t> & source,
                  const tensor_t<uint32_t> & kernel,
                  tensor_t<uint32_t> & target, int32_t tile_size) {

    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 + tile_siz
                target.get(i,0,0,0) += source.get(i + j,0,0,0) * kerne
            }
        }
    }


}

extern "C"
uint64_t* convolution_question(uint64_t * source_space, uint64_t sourc
                    uint64_t * kernel_space, uint64_t kernel_size,
                    uint64_t * target_space, uint64_t _target_size,
                    int32_t tile_size) {
    tensor_t<uint32_t> source(source_size,1,1,1, (uint32_t *)source_sp
    tensor_t<uint32_t> kernel(kernel_size,1,1,1, (uint32_t *)kernel_sp
    uint64_t target_size = source_size - kernel_size;
    tensor_t<uint32_t> target(target_size,1,1,1, (uint32_t *)target_sp
    TAG_START("source", source.data, &source.as_vector(source.element_
    TAG_START("kernel", kernel.data, &kernel.as_vector(kernel.element_
    TAG_START("target", target.data, &target.as_vector(target.element_

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

    TAG_STOP("source");
    TAG_STOP("kernel");
    TAG_STOP("target");
    return target_space;
}

FUNCTION(convolution, convolution_question);

""",
cmdline=f"--size {64*1024} --size2 {16*1024} --size3 {64*1024}  --tile
data = render_csv("convolution.csv").append(render_csv("build/convolut
data[["function", "size", "size2", "size3", "IC", "CPI", "CT", "ET", "
```

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

13.2.4 Discussion

Let's take a moment to think about why that was so hard. Let's address two 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.

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". 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, study the code to understand the cause, and made changes to try to improve things. This is one part of how you should approach the programming assignment.

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, 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.

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)

13.2.5 Tuning Tile Size

Shockingly, we aren't done with 1D convolution yet! What about the `tile_size`? What's the right value?

We could try to do some math to figure out exactly how big it should be, but at this point, you're tired and I 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 can't do 1, 2, or 4 because they are not multiples of 8)

In [ ]:
```
fiddle("convolution.cpp", function="convolution_tiled_split", analyze
            cmdline=f"--size {64*1024} --size2 {16*1024} --size3 {64*

```

In [ ]:
```
display(render_csv("tile_size.csv", columns=["tile_size", "IC", "CPI"
data = render_csv("convolution.csv").append(render_csv('convolution_t
data = data.append(render_csv("convolution_tiled_split.csv")).append(
plotPE("tile_size.csv", what=[("tile_size", "IC"),("tile_size", "CPI"
```

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.

Here's all our data so far:

In [ ]:
```
display(data[["function", "tile_size","IC", "CPI", "CT", "ET", "L1_MP
```

Turing `tile_size` gave us an additional 0.32/0.27 = 1.14x. 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 [ ]:
```
render_code("convolution.cpp", show="do_convolution_tiled_fixed_tile"
```

In [ ]:
```
fiddle("convolution.cpp", function="convolution_tiled_fixed_tile", an
            cmdline=f"--size {64*1024} --size2 {16*1024} --size3 {64*
data = render_csv("convolution.csv").append(render_csv('convolution_t
data = data.append(render_csv("convolution_tiled_split.csv")).append(
display(data[["function", "tile_size","IC", "CPI", "CT", "ET", "L1_MP
```

That's worth another 3% and brings up the total speedup (all from reduced `IC`) to 0.47/0.26 = 1.74x.

Ok. We are now done with 1-D convolutions.

14 Programming Assignment

For your programming assignment in this lab you'll be optimizing a specialized version of matrix multiply. The input to your program will be a square matrix, M, (stored in a `tensor_t`) and a power, p, and your job is to compute M^p as quickly as possible.

The expression M^p means M multiplied by itself, p times, where multiplication is normal matrix multiplication.

This computation has a variety of applications. For instance, you can use this algorithm to evaluate Markov Chains.

For this assignment we'll be computing on `tensor_t<uint64_t>`. Many applications would use `float` or `double`, but that problem is harder due numerical instability (https://en.wikipedia.org/wiki/Numerical_stability) issues. You don't need to worry about integer overflow in this assignment. It will happen a lot, and it's consider the "correct" behavior for the purposes of this lab.

14.1 Reference Code

The reference implementation is in `matexp_reference.hpp`:

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

Read through the code and comments to make sure you understand what the code is doing.

14.2 Detailed Requirements

The requirements for the lab are pretty simple:

1. M will be square and it's width/height will be less than 2048.
2. p will be less than or equal to 1024.
3. p will be greater than or equal to 0.
4. Like `matexp_reference`, your function need to be a template function, but you can assume that `T` is always `uint64_t`.
5. Values in M can be any `uint64_t` value.
6. Your output must match the output of the code in `matexp_reference.hpp`.
7. Your implementation should go in `matexp_solution.hpp`. The starter version is just a copy of `matexp_reference.hpp`.
8. The baseline for this lab includes `-O3` optimizations. You can change them if you wish.

14.3 Running the Code

The driver code for the lab is in `matexp_main.cpp` and `matexp.cpp`. `matexp_main.cpp` is mostly command line processing (take a look if you want). `matexp.cpp` is what actually calls your code:

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

It defines four functions:

- `matexp_reference_c` Calls the starter code with `size` x `size` matrix and `power`.
- `matexp_solution_c` Calls your code with `size` x `size` matrix and `power`.
- `bench_reference` Runs benchmarks we will use for grading for the starter code.
- `bench_solution` Runs benchmarks we will use for grading for your code.

To invoke these, you can build and run `matexp.exe`:

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

`matexp.exe` takes several command line parameters:

In [ ]:
```
!./matexp.exe --help
```

The notable ones are:

1. `--size` -- set the size of the matrix to multiply.
2. `--power` -- set the power to raise it to.
3. `--p1` to `--p5` -- set parameters (see below.)
4. `--function` what functions to run.
5. `--seed` set the random seed.
6. `--stats-file` sets where statistics should go.

The first five of these can take multiple values and `matexp.exe` will run all combinations and they will end up in `stats.csv` :

In [ ]:
```
!./matexp.exe --function matexp_reference_c matexp_solution_c --size
render_csv("stats.csv", columns=["function", "size", "power", "seed"]
```

And, of course, we run it all in the cloud (we've added `--stat-set L1.cfg` to gather cache and TLB statistics):

In [ ]:
```
!make matexp.exe
!cse142 job run --lab caches2 "./matexp.exe --stat-set L1.cfg --funct
render_csv("stats.csv", columns=["function", "size", "power",  "seed"
```

14.4 Setting Parameters

One of the key parts of this lab is setting parameters (e.g., tiling sizes), and the `matexp.exe` has support for this built in vias the `--p1` -- `--p5` command line options and function parameters.

You can use these for whatever you'd like:

1. Setting tile sizes.
2. Selecting among different implementations.
3. Whatever else.

Their default value is 1.

Just like `--size` and `--power` , you can multiple values and `matexp.exe` will run all combinations. For example:

In [ ]:
```
!cse142 job run --lab caches2 "./matexp.exe --stat-set L1.cfg --funct
```

In [ ]:
```
render_csv("stats.csv", columns=["function", "seed", "size", "power",
```

Tips for Using Parameters

1. Running multiple values of multiple parameters can result in a lot of experiments... sometimes too many.
2. Jobs in the cloud are limited to 5 minutes, so you need to limit the number of tests per job.
3. That said, exploring a wide space of parameter settings can be an effective way to optimize your code.

14.5 The Test Suite

> **NOTE:** You normally will not need to run `run_tests.exe` in the cloud. It'll work fine, but it takes longer which will slow your work down. The test suite is about *correctness* not performance.

The lab provides a comprehensive test suite for your implementation. The code in is

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

14.5.1 Test Suite Details

You can list all the tests with:

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

The first group of tests is (under `MatexpTests`), contains four simple tests that call `do_simple_diag_test()` and `do_simple_offdiag_test()`:

In [ ]:
```
render_code("run_tests.cpp", show=("//START1", "//END1"))
```

These two functions take a diagonal or off-diagonal matrix and raise them to a power. It's easy to calculate the correct results for these computations, so they make good tests.

These tests are:

1. `one_test` runs the two functions above for a given size.
2. `simple_tests` runs the two functions above for a set of small test cases.
3. `simple_random_tests` runs the two functions for 10 randomly generate test cases.

The final test in this group (`randomize_tests`) calls `do_test()` which compares the output `matexp_reference.hpp` with `matexp_solution.hpp` for randomized test cases.

The second group of tests (under `MatexpTests/MatexpTestFixture`) calls `do_test` with a bunch of test cases of various sizes.

14.5.2 Test Suite Command Line Options

You can limit the tests that `run_tests.exe` runs with a command line argument. So you can run `MatexpTests.simple_tests`:

In [ ]:
```
!./run_tests.exe --gtest_filter=MatexpTests.simple_tests
```

Or `MatexpTests.one_test`:

In [ ]:
```
!./run_tests.exe --gtest_filter=MatexpTests.one_test
```

`run_tests.exe` also takes all the same arguments as `matexp.exe`: `--size`, `--power`, and `--p1` to `--p5`. It will run all the tests for those values. For instance:

In [ ]:
```
!./run_tests.exe --gtest_filter=MatexpTests.one_test --size 1 2 --pow
```

14.6 Things To Try

There are two main challenges I see in this lab:

1. Make matrix multiplication fast, primarily by improving it's memory behavior.
2. Applying matrix multiplication efficiently to compute M^p.

The benchmarks are structured to evaluate your solution's success on both of these challenges.

14.6.1 Tiling Matrix Multiplication

The obvious approach to improving cache performance is tiling and renesting. You saw an example of this with 1-D convolution, and the principle is the same, but the problem is a little more complex because there is an extra loop.

There are two ways to approach this task and you should try to apply both at once:

1. You should think about the data access pattern of matrix multiply in terms of temporal and spatial locality.
   A. How can you maximize spatial locality?
   B. Don't forget to consider all three matrices.
   C. How large can the tiles while still fitting in the cache?
2. You should try different tiling schemes:
   A. Different ways to split and renest the three loops.
   B. Different tile sizes (`--p1` to `--p5` are provided for this purpose)
3. Don't forget about loop overhead.

14.6.2 Raising to a Power

Computing M^p can done more efficiently than multiplying M by itself p times (which is what the reference code does). By way of a hint, remember that:

$$M^{p+q} = M^p M^q$$

As you work on this part of the problem, I suggest practicing with integers first. I found it useful to code my solution with integers and test it and then rewrite it for matrices.

14.6.3 Using the Test Suite

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

1. Run all the tests. If they all pass, great!
2. If some fail, run `simple_tests` and then `simple_random_tests`.
3. Once I find a particular test case that fails, I'd use `one_test` to run just that configuration while debugging.

If and when you make use of parameters (`--p1` etc.), I'd try out the values of interest with `./run_tests.exe` before bothering to run code in the cloud.

One debugging tip: `tensor_t.hpp` includes support for the `<<` operator so you can say

```
std::cerr << my_tensor
```

This can be very helpful when debugging.

14.7 Useful C++

There are few things in C++ (or GCC extension to C++, really) that might be useful in this lab.

First, you can prevent inlining of a particular function by declaring it like so:

```
void __attribute__((noinline)) matexp_solution(...)
```

This can make it easier to debug, because you can set a breakpoint on the function and it'll work like you expect.

Second, you can turn on arbitrary optimizations for particular functions like so:

```
#pragma GCC push_options
#pragma GCC optimize ("unroll-loops")

void your_function() {
}

#pragma GCC pop_options
```

14.8 Do Your Work Here [...]

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

14.9 Tools

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

14.9.1 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:

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.

14.9.2 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 `Makefile` is set up to build assembly files (ending in `.s`) in the `build` directory. All the assembly for `SolutionAllocator` and `ReferenceAllocator` (and a whole bunch of other stuff) will be in `Allocator.s`. It's quite long: my version is 44,262 lines, so searching through it by hand is daunting. To make matters worse, all the function names are mangled.

One solution to this is to `c++filt` to demangle the names and the use `grep` to find the symbols of interest (the `^` matches the beginning of the line`):

In [3]:
```
1 !make build/matexp.s
2 !c++filt < build/matexp.s | grep '^void matexp_solution<uint64_t>'
```

```
g++-8 -S -c -Wall -Werror -g  -O3    -fPIC -pthread  -I/cse142L/cse141pp-
archlab/libarchlab -I/cse142L/cse141pp-archlab -I/usr/local/include -I/us
r/local/include -I/cse142L/CSE141pp-SimpleCNN  -Ibuild/  -I/cse142L/CSE14
1pp-Tool-Moneta/moneta/  -I. -MMD  -std=gnu++11   -g0 build/matexp.cpp -o
build/matexp.s
```

You can see that there are several different versions of each method, one for each set of template parameters. Unless you're doing something very sophisticated with your implementations, the assembly will all be basically the same.

You can now render the assembly here with:

In [ ]:
```
!make build/Allocator.s
render_code("build/Allocator.s", show="SolutionAllocator<MissingLink,
```

14.9.3 Looking at the CFG

Control flow graphs are easier to interpret than the assembly, but getting them for C++ functions is also a little complicated. The tool that our CFG generator is built on uses its own name mangling scheme internally. To get the names it uses for your functions you can use the command below. We pass the executable to `cfg` along with `--filter` which takes a string to search for. If you leave out `--filter` you will all 2890 symbols in the executable.

In [ ]:
```
!make clean
!make matexp.exe
```

In [ ]:
```
!cfg matexp.exe --filter matexp_solution --list
```

There's a one-to-one correspondence between these names the names we saw earlier. You can render a CFG like so:

In [ ]:
```
do_cfg("matexp.exe", symbol="sym.matexp_solution_c")
```

14.9.4 Profiling

Profiling can be valuable tool in figuring out where your code is spending time.

To profile your allocator, you need to recompile it with profiling enabled:

NOTE: Don't forget to rebuild without `GPROF=yes` . BUilding in support for gprof will slow down your code a bit.

In [ ]:
```
!make clean matexp.exe GPROF=yes
```

You'll need to profile one type of benchmark at a time. just run one of the lines below at a time.For good accuracy, you should profile in the cloud.

In [ ]:
```
# Profile in the cloud
!cse142 job run --lab caches2 --force "./matexp.exe --MHz 3500 --func
```

Either way, you get `gmon.out` which you can process with `gprof` to get something you can read (sort of):

In [ ]:
```
1 !gprof ./matexp.exe
```

The output is a big for Jupyter Notebook. In a terminal you can do:

```
gprof matexp.exe | less -S
```

Which will let you look at the file without wrapped lines.

14.9.5 Debugging

Your code will certainly have errors in it, and you'll need to debug. Unfortunately, the Linux debugger `gdb` doesn't work inside the note book. If you want to use it, you can do so at the terminal:

```
swanson-dev-192:/cse142L/labs/CSE141pp-Lab-Caches gdb alloc_main.ex
e
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/license
s/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show c
opying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from alloc_main.exe...done.
(gdb) run --size 1024  --function allocator_bench_solution
Starting program: /cse142L/labs/CSE141pp-Lab-Caches/alloc_main.exe
 --size 1024  --function allocator_bench_solution
warning: Error disabling address space randomization: Operation not
permitted
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_d
b.so.1".
registering function: allocator_bench_starter
registering function: allocator_bench_solution
registering function: allocator_microbench_starter
registering function: allocator_microbench_solution
Loading Native engine.
registering env: alloc_test
Running allocator_bench_solution
[Inferior 1 (process 20279) exited normally]
(gdb)
```

A few notes about debugging:

1. Start with `allocator_microbench_solution`, it's much simpler.
2. Start with `alloc()`, since you have to `alloc()` before you can `free()`.
3. Turn down `--size`. It's set large so it runs long enough to a good timing measurement.

The code for `alloc_main()` is a little complicated, so it can be hard to get the debugger to stop in your code. Instead, set a breakpoint on the function you want to debug:

```
bash$ gdb alloc_main.exe
(gdb) break allocator_microbench_solution
(gdb) run --function allocator_microbench_solution --size 1024
(gdb) list
```

which will give you something like this:

```
$ gdb alloc_main.exe
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/license
s/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show c
opying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from alloc_main.exe...done.
(gdb) break allocator_microbench_solution
Breakpoint 1 at 0x1d9f6: file build/Allocator.cpp, line 179.
(gdb) run --function allocator_microbench_solution --size 1024
Starting program: /cse142L/labs/CSE141pp-Lab-Caches/alloc_main.exe
 --function allocator_microbench_solution --size 1024
warning: Error disabling address space randomization: Operation not
permitted
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_d
b.so.1".
registering function: allocator_bench_starter
registering function: allocator_bench_solution
registering function: allocator_microbench_starter
registering function: allocator_microbench_solution
Loading Native engine.
registering env: alloc_test
Running allocator_microbench_solution
Breakpoint 1, allocator_microbench_solution (count=1024, seed=37359
28559) at build/Allocator.cpp:179
179         run_microbench<SolutionAllocator<uint[4], 8>>(count, see
d);
(gdb) list
174     FUNCTION(alloc_test, allocator_microbench_starter);
175
176
177     extern "C"
178     uint64_t * allocator_microbench_solution(uint64_t count, uin
t64_t seed) {
179         run_microbench<SolutionAllocator<uint[4], 8>>(count, see
d);
180         return NULL;
```

```
181     }
182     FUNCTION(alloc_test, allocator_microbench_solution);
(gdb)
```

There's a pretty good `gdb` tutorial here (https://www.cs.cmu.edu/~gilpin/tutorial/).

14.9.6 Tracing With Moneta

Note: I found Moneta to be most useful for the `miss_machine` benchmark, but not as much for the rest of the programming assignment. If you don't find it useful, don't use it.

Note: The commands below rely on some tracing commands in `ChunkAlloc.cpp`. This means you won't be able to use them until you've replace `posix_memalign()` with `alloc_chunk()` in `SolutionAllocator.hpp`.

Note: The settings below assume optimizations are turned on. Otherwise, the benchmark generates so many memory accesses, that the trace is too large to load.

In [ ]:
```
!make alloc_main.exe
mtrace --memops 1000000 -- ./matexp.exe --size 60 --power 2 --functio
```

In [ ]:
```
show_trace("./trace_0",  show_tag=["dst", "B", "A"])
```

14.10 Final Measurement

When you are done, make sure your best allocator is called `matexp_solution()` in `matexp_solution.hpp`. Your code will be invoked with `--p1` to `--p5` equal to 1, so you'll need to "bake in" the optimal values for those parameters.

Then you can submit your code to the Gradescope autograder. It will run the commands given above and use the `ET` values from `autograde.csv` to assign your grade.

Your grade is based on your speed up relative `matexp_reference.hpp` on three benchmarks.

For each of them, there's a target speedup given in the table. You get a score for each benchmark between 0 and 33.3, and the overall score is the sum of these scores. For each function, the score is compute as `your_speedup/target_speedup * 33.3`.

For this lab, you don't get extra credit for beating the targets. This will help ensure that your design in balanced: You much do well at all 3 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.exe` to check it's correctness.

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

After you run it, the results will be in `autograde/autograde.csv` rather than `./bench.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 `matexp_solution.hpp` and `config.make`.
2. Copies those files into a clean clone of the starter repo.
3. Builds `matexp.exe` from scratch.
4. And then runs the commands the benchmarks.
5. It then runs the `autograde.py` script 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 [ ]:
```
!make matexp.exe
!cse142 job run --take matexp_solution.hpp --take config.make --lab c
```

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

You can check the performance results like this:

In [ ]:
```
!./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['capped_score']), 2)}")
```

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

And see the autograder's output like this:

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

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 [ ]:

```
1
2 !git config --global user.email "you@example.com"
3 !git config --global user.name "Your Name"
```

15 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, and you got to apply those concepts to computing M^p. 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.

16 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.

16.1 If You Have Trouble

If it's near the deadline and you are having trouble turning in any part of your lab, you can fill out this form: https://forms.gle/ThHjESfbZRqqztXUA (https://forms.gle/ThHjESfbZRqqztXUA) to let us know what's going on and provide us access to the work you have done prior to the deadline.

> **NOTE:** Filling out the form above *before* the deadline is the *only* mechanism available to receive credit without turning in the assignment on time.

If it's more than a day before the deadline, you can reach out via Piazza and hopefully we can get it sorted out.