In [91]:
```
%load_ext autoreload
%autoreload 2
from notebook import *
hist_size=10000000
# 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 5: Parallelism

Or, if we are being honest, mostly it's Caches Part III

Modern computers exploit parallelism in many ways:

1. They can execute multiple threads at once.
2. They can execute instructions in parallel.
3. They can handle multiple memory requests at once.

We are going to look at each of these kinds of parallelism, but we'll spend the most time on threading, since it's the form of parallelism that's most apparent to the programmers and the one that takes the most effort to exploit. Not surprisingly, we will find that much of what makes parallel code fast and slow has to do with how it uses memory.

In particular we are going to study:

1. Instruction level parallelism.
2. Memory-level parallelism.
3. Thread-level parallelism.
    A. How it works.
    B. How to use it.
    C. Cache Coherence
    D. Synchronization
    E. False sharing
4. Hyperthreading
5. The OpenMP extensions for C/C++

This lab includes a programming assignment.

Check the course schedule for due date(s).

4 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**.

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 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/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.

5 Browser Compatibility [...]

6 About Labs In This Class [...]

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

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 Instruction Level Parallelism

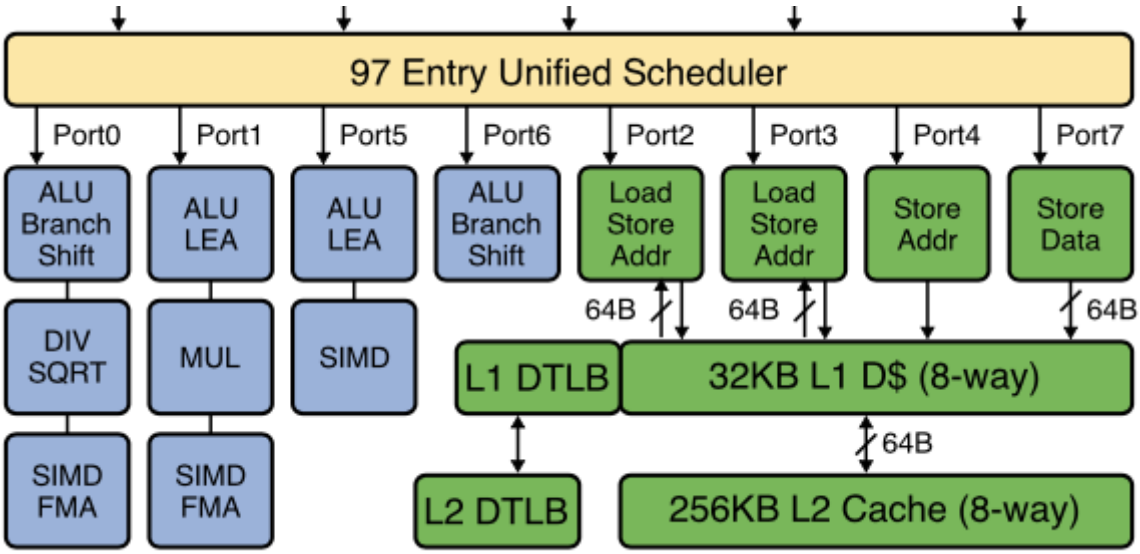
As you learned in CSE142, modern processors can exploit parallelism between instructions by fetching a large "window" of instructions and dynamically identifying instructions that can execute at the same time. The algorithm it uses is pretty amazing and the fact that processors can do this at over 3GHz is mind-blowing.

[Meme Redacted For Space Reasons]

But how well does it work? How much ILP can we get? Where does ILP come from?

9.1 Let's Look at Our Machine

Here's a cartoon of our processor's functional units.



The blue units on the left can all execute basic arithmetic operations ("ALU") and two of them can execute our old friend `lea`. One interesting note about this diagram: There are 2 load unit and one unit each for store addresses and store data. This means that processor can issue 2 loads but only one store per cycle.

Question 1 (Completeness)

Based on this image, what's the minimum achievable CPI for ALU operations on our CPU? For load operations? For all operations?

Minimum possible ALU CPI:

Minimum possible load CPI:

Minimum possible total CPI:

Show Solution

Question 2 (Completeness)

Based on this design, do you think the designers are more concerned about achieving parallelism between memory operations or arithmetic (non-memory) operations?

Question 3 (Optional)

Why does it "execute" the data and address for a store separately?

9.2 Accounting For Branches

In this section we are going look at instruction dependencies and critical paths quite closely, following what you learned in CSE142. Branches and comparison operations complicate things a little bit. For the most part we will ignore comparison instructions and branch instruction when calculating critical paths. This is because the processor speculates on branches, this means that these instructions don't have much impact on the performance of the processor's out-of-order execution engine. Since the processor only uses them to detect mis-speculations, it never has to wait on them.

We do, however, count those instructions when calculating *how many* instructions have executed. While comparisons and branches don't contribute latency to the execution, they do eventually have to execute, so the processor's branch predictions can be checked.

9.3 Experiments [...]

Here's a simple function (run the code to see it). I've turned off optimizations and use `register` to get `i` into a register.

Study the code and pay particular attention to the data dependences between the instructions in the inner loop body.

9.4 Dynamic Loop Unrolling [...]

`wide_5()` is not a very realistic function, since it has an artificially huge amount of ILP. But ILP can arise in more natural ways as well.

Below is a function that represents a common pattern: A loop body that computes a value based on the loop index and stores it somewhere. In this case, the computations is a long chain of additions. Study it and answer the question below.

10 Memory-level Parallelism

One of the key themes in this class is that it's really all about memory, so it is with ILP as well: Parallelism among arithmetic instructions is fine, but it's the parallelism among memory operations that really pays off.

In fact memory parallelism is so important that it has its own name: Memory Level Parallelism (MLP).

Let's take a look at a new and improved miss machine and use it to explore MLP.

10.1 Miss Machine++

Our new miss machine is much like the old one except it's wrapped up in a class so we can easily create more than one of them and pass them around as objects. It also has support for performing stores in addition to loads as it walks around the links. Look over the code and make sure you understand it.

In [93]:

```
render_code("MissMachine.hpp")
```

```
// MissMachine.hpp:1-70 (70 lines)
#pragma once
#include<cstdint>
#include<vector>
#include<algorithm>
#include<cassert>
struct MissingLink {
        struct MissingLink* next;  // I know that pointers are 8 bytes on
this machine.
        uint64_t data;
        MissingLink(): next(NULL) {}
};

extern "C"
struct MissingLink * do_load_misses(struct MissingLink * start, uint64_t
count) {
        for(uint64_t i = 0; i < count; i++) { // Here's the loop that doe
s this misses. It's very simple.
                start = start->next;
        }
        return start;
}

extern "C"
struct MissingLink * do_store_misses(struct MissingLink * start, uint64_t
count) {
        for(uint64_t i = 0; i < count; i++) { // Here's the loop that doe
s this misses. It's very simple.
                start = start->next;
                start->data = i;
        }
        return start;
}

class MissMachine {
public:
        size_t link_count;
        size_t link_bytes;
        uint8_t *data;
        MissingLink * dummy; // this is here to prevent the compiler from
optimizing out the  calls to do_load_misses()
        std::vector<MissingLink*> links;

        MissMachine(size_t link_bytes, size_t total_bytes): link_count(to
tal_bytes/link_bytes), link_bytes(link_bytes) {
```

```
                assert(total_bytes > link_bytes);
                assert(link_bytes >= sizeof(void*));
                data = new uint8_t[link_bytes * link_count];
                for(unsigned int i = 0; i < link_count; i++) {
                        links.push_back(new (reinterpret_cast<MissingLink
*>(reinterpret_cast<uintptr_t>(data) + i * link_bytes)) MissingLink);
                }
        }

        void make_links() {
                for(unsigned int i = 0; i < link_count -1; i++) {
                        links[i]->next = links[i + 1];
                }
                links[link_count -1]->next = links[0];
        }

        inline MissingLink * load_miss(size_t count, size_t start = 0) {
                return dummy = do_load_misses(links[start], count);
        }

        inline MissingLink * store_miss(size_t count, size_t start = 0) {
                assert(link_bytes >= sizeof(MissingLink));
                return dummy = do_store_misses(links[start], count);
        }

        void shuffle(uint64_t seed =0xDEADBEEF) {
                std::random_shuffle(links.begin(), links.end());
        }

        ~MissMachine() {
                delete [] data;
        }
};
```

10.2 Parallel Miss Machines

We will use `MissMachine` to create parallel chains of loads much like we built parallel chains of adds in the previous section.

The code below accomplishes this by creating a miss machine, and then spreading four pointers uniformly along the circular chain of links. Then we traverse the chain at all four points in the chain in parallel. Here's the code and the assembly (sorry, the CFG tool breaks on this code for some reason). The key basic block is the three instructions after `.L196`, `al` and `bl` are in `rax` and `rbx`.

```
.L196:
        movq    (%rax), %rax
        movq    (%rbx), %rbx
        subl    $1, %edx
        jne     .L196
```

In [ ]:

```
1 miss_machines = fiddle(fname="miss_machines.cpp", function="sample", 
2 compare([miss_machines.source, miss_machines.asm])
```

Question 4 (Correctness - 4pts)

What's the critical path (in cycles) for the inner loop above? And what's the estimated CPI. Recall that the latency for a cache hit on our machine is 4 cycles. Assume the loads all hit.

1. **critical path**:
2. **CPI**:

10.2.1 L1 Hits [...]

10.2.2 L1 Misses [...]

Let's see about hiding the latency of some L1 misses that hit in the L2. To do this, we'll just expand the size of our miss machine to fit in the L2 but not the L1. THe L2 is 256kB, so 128kB should do it. We'll also increase the link size to 64 bytes so we don't have any cache line reuse.

10.2.3 L2 Misses [...]

Same drill: We'll increase the miss machine size so it fit's in the L3 but not the L2: The L3 is 12MB, so 2MB should be enough.

11 Thread Performance

ILP and MLP exist within a single core, and the degree of parallelism is limited by the number of instructions the CPU can issue per cycle. To get more parallelism, we need to use more CPUs, and for that we'll need to create threads.

A thread is a flow of control through your program that runs on a processor. Every program has at least one thread, and by creating multiple threads you can spread the work of your program across many cores, hopefully improving it's performance.

Making programs fast via multi-threading is an extremely deep and complex area. We could easily spend an entire quarter studying techniques for creating, managing, and using threads, and most universities (including UCSD) offer several courses on this topic (Start with Operating Systems and then take the graduate Parallel Computation course). Indeed, some people have devoted their entire careers to the topic.

All this effort is for good reason: the amount of ILP and MLP that individual cores can utilize has been roughly constant over last decade and shows no signs of improving much. Making matters worse, clock speeds are growing very slowly. That means that adding cores is the main way that computer are getting faster, but that only works if we can use threads effectively.

But, it's week 8, and we don't have time for all of that. Instead, we are going to take a whirlwind tour of how you can create threads, how to make them communicate with one another, why the underlying hardware can make that hard and/or slow, and what you can do about it.

To start, let's see how many cores we have:

In [ ]:
```
1 !cse142 job run --lab parallel --force 'lscpu'
```

The key lines are `Socket(s): 1` and `Core(s) per socket: 6`. A "socket" is place on a motherboard to stick a physical CPU. We have 1, so all our cores live on one chip. That chip has 6 "cores". A core is complete processor pipeline.

You might notice that it also says `CPU(s): 12`. This would be better phrased as "logical cores". It's twice the number of actual (physical) cores because each of the cores can run two threads at once via Hyperthreading. These cores are numbered 0-11.

By convention, the lower half of the core numbers cover one logical core on each physical core. The top half of the numbers are the second logical cores. So, logical core 0 and logical core 6 are on the same physical core.

For now, we are going to stick to logical cores 0-5. We'll return to the upper 6 logical processors later in the lab.

11.1 Spawning Threads

The first step to using threads is to create some. C++ has pretty good threading facilities. The key is the `std::thread` object that represents a running thread. To start a thread, you create an `std::thread` object and pass it a function to call and the arguments you'd like to pass to the function.

The `std::thread`'s `join()` method waits for the thread to complete.

Here's some code that runs three threads that print out some numbers. The cell below will run the code three times separated by "FINISHED EXECUTION". Pay close attention to the output of each run.

In [ ]:

```
t = fiddle("threads.cpp", function=["threads", "threads", "threads"],
code=r"""
#include"function_map.hpp"
#include<cstdint>
#include<thread>

void go() {
    for(int i = 0; i < 10; i++)
        std::cerr << i << "\n";
}

extern "C"
uint64_t* threads(uint64_t threads, uint64_t * data, uint64_t size, u
    std::thread t1(go);  // Create a thread to run go().  Pass no arg
    std::thread t2(go);
    std::thread t3(go);

    t1.join(); // wait for t1 to finish.
    t2.join();
    t3.join();
    std::cerr << "FINISHED EXECUTION\n";
    return data;
}
FUNCTION(one_array_2arg, threads);
""", run=["perf_count"])
```

See how the order of the number changes? This is because the relative execution rate of each thread is different. Also, the threads take turns writing to standard output and the order they go in is non-deterministic.

This non-determinism is the bane of multi-threaded debugging: Imagine if your bug only occurred for one of the very many possible orderings of the operations in your threads? Your bug might occur just 1 in 100 (or 1000 or 10,000) times you run the program.

11.2 Measuring Thread Behavior

Of course, we will want to measure the performance and behavior of our threads. Things get a little tricky here, because of a limitation of our performance counter measurement library. It can only measure performance counters of the main thread of the program.

For the experiments we are going to run this is not a big problem: All our threads will be doing the same thing at the same time, so measuring one is a good as measuring any other. In some cases, though, we will need to *estimate* aggregate values across all the cores/threads. For instance, if we want to estimate the *total* number of instructions execute by all threads, we'll multiply the single-thread IC we measure for one thread by the thread count.

Here's a simple threaded program that runs one miss chain in each of several threads.

In [ ]:

```
t = fiddle("threads.cpp", function="threads", analyze=False,  opt="-O
code=r"""
#include"function_map.hpp"
#include<cstdint>
#include<thread>
#include"MissMachine.hpp"

void go(MissMachine * machine, uint64_t arg2) {
    machine->load_miss(arg2);
}

extern "C"
uint64_t* threads(uint64_t thread_count, uint64_t * data, uint64_t si
    MissMachine a( arg1, size);
    a.make_links();

    std::thread **threads = new std::thread*[thread_count-1]; // Allo
    for(unsigned int i = 0; i < thread_count-1; i++) {
        threads[i] = new std::thread(go, &a, arg2); // create the thr
    }
    go(&a, arg2); // So will this thread, hence the -1
    for(unsigned int i = 0; i < thread_count-1; i++) { // wait for ev
        threads[i]->join();
        delete threads[i]; // cleanup
    }
    delete threads; // cleanup.

    return data;
}
FUNCTION(one_array_2arg, threads);
""", run=["perf_count"],
           cmdline=f"--size 4096 --arg1 8 --arg2 100000000 --thread 1
           perf_cmdline="--stat-set L1.cfg --MHz 3500")

```

In [ ]:

```
display(render_csv("build/threads.csv", columns=["thread", "size", "a
plotPE("build/threads.csv", lines=True, what=[("thread", "IC")])
```

Question 5 (Correctness - 1pts)

With 6 threads how many total instructions were executed?

Total IC:

11.3 Thread Communication with Volatile Variables and Locks

In order for threads to work together, they must share variables: *Sharing* means that more than one thread is reading and/or writing to the variable during the same period of time. Threads working together *must* share some information, otherwise they cannot make progress together on a common goal. For example, imagine expecting 8 people in sealed rooms, who have never met, to make progress on a single task -- it is not possible.

There are two separate problems we need to solve. The first is *how* to share data and the second is how to share it in a coordinated and reliable way.

11.3.1 Sharing Data Between Threads with `volatile`

The code below declares a global variable, initializes it to zero, and then provides `wait()` to wait for it to change. It also provides `signal()` to update the global variable. You could imagine that two threads could use these two function to coordinate in a simple way: Thread `T1` could call `wait()` to wait for another thread, `T2`, to do something. When `T2` is done, it could call `signal()` to let `T1` know it has done it.

In [ ]:

```
t = fiddle("not_shared.cpp", function="wait", analyze=True,  opt="-O3
code=r"""

int flag = 0;
extern "C"
void wait() {
    while(flag);
}

void signal() {
    flag = 1;
}
""", run=None)
t.cfg
```

As written and compiled with optimizations, the compiler does what we would expect: it checks `flag` once. If it's non-zero (i.e., it evaluates to `true`), it returns. Otherwise it loops infinitely. This will clearly not work as a thread communication mechanism: Unless `T2` calls `signal()` *before* `T1` calls `wait()`, `T1` will never get the message.

The compiler is assuming that `flag` *will not change*. This is a valid assumption for the compiler to make because, by default, variables in C and C++ are considered to the thread-private -- only the current thread will access them. That's not what we want for thread communication.

We can fix this by declaring `flag` as `volatile`. `volatile` tells the compiler that the variable is shared and so it might change at any time, which dramatically reduces the number of optimizations it can apply. Let's see what it does now:

In [ ]:

```
t = fiddle("not_shared.cpp", function="wait", analyze=True,  opt="-O3
code=r"""

volatile int flag = 0;
extern "C"
void wait() {
    while(flag);
}

void signal() {
    flag = 1;
}

""", run=None)
t.cfg
```

Now, `wait()` checks `flag` every time, and our communication mechanism will work. Or will it...

11.3.2 Memory Ordering [...]

Here's a slightly more complicated example that actually uses our communication mechanism: instead of waiting on a fixed value for flag, `wait()` will wait for a configurable value. The threads take turns waiting on one another, and set `other_value` each time. They also check whether `flag` and `other_value` match, and tell us if they don't.

11.3.3 Locks

A *lock* or *mutual exclusion variable (mutex)* is a small data structure that can be *locked* and *unlocked*. We'll use C++'s `std::mutex`.

If a thread calls `lock()` on a mutex that is not currently locked, the thread "holds" the lock and starts executing a region of code called a "critical section". At the end of the critical section, it calls `unlock()` to release the lock.

If a thread, `T`, calls `lock()` on a mutex that *is* currently locked, it will wait until the thread that holds it calls `unlock()`. Then, `T` gets the lock and can proceed.

The result is that, at any time, only one thread is executing inside the critical section.

Internally, locks are implementing using some kind of flag (similar to `flag` in our example above) and some of those fences I mentioned above.

Here's an example. In this code, we have a shared variable `shared`. Several threads are going to work together to increment `shared` 600,000 times. If we run with n threads, each will do 600,000/n increments.

If `do_lock` is `true`, they will use `yes_locks()` which protects each lock with an increment. Otherwise, they will use `no_locks()` which doesn't.

In [ ]:

```
t = fiddle("lock_demo.cpp", function="lock_demo", analyze=False,  opt
code=r"""
#include"function_map.hpp"
#include<cstdint>
#include<thread>
#include<mutex>

std::mutex lock;
volatile int shared = 0;
extern "C"
void yes_locks(uint64_t id, int count) {
    for(int i= 0; i < count; i++){
        lock.lock();
        shared++;
        lock.unlock();
    }
}

extern "C"
void no_locks(uint64_t id, int count) {
    for(int i= 0; i < count; i++){
        shared++;
    }
}

extern "C"
uint64_t* lock_demo(uint64_t thread_count, uint64_t * data, uint64_t 
    shared = 0;
    std::thread **threads = new std::thread*[thread_count];
    bool do_locks = (arg2 != 0);
    for(unsigned int i = 0; i < thread_count - 1; i++) {
        if (do_locks) {
            threads[i] = new std::thread(yes_locks, i, arg1/thread_co
        } else {
            threads[i] = new std::thread(no_locks, i, arg1/thread_cou
        }
    }
    if (do_locks) {
        yes_locks(thread_count - 1, arg1/thread_count);
    } else {
        no_locks(thread_count - 1, arg1/thread_count);
    }
    for(unsigned int i = 0; i < thread_count - 1; i++) {
        threads[i]->join();
    }
    std::cerr << "do_lock: " << do_locks << "; " << "thread count: " 
    return data;
}
FUNCTION(one_array_2arg, lock_demo);
""", run=["perf_count"],
           cmdline=f"--arg1 600000 --arg2 1 0 --thread 1 2 3 4 5 6",
           perf_cmdline="--stat-set L1.cfg --MHz 3500")

```

As you can see from the output, we get the right answer when we use locks, and the wrong answer when we don't (except for 1-thread case).

Here's the assembly for `no_locks()`.

In [ ]:
```
1 do_cfg("build/lock_demo.so", symbol="no_locks")
```

Question 6 (Completeness)

Based on the assembly/CFG above, and assuming multiple threads are running at once, explain how `shared` ends up being computed incorrectly without locks and how adding locks prevents it.

Show Solution

So locks are fine for correctness (which is very important), but what about performance.

Here's a bunch of data and graphs about the performance of the code above with and without threads.

In [ ]:
```
1  from notebook import *
2  df = render_csv("build/lock_demo.csv")
3  df["label"] = df["thread"].apply(lambda x: f"{x} threads;" ) + " " + (
4  df["Total IC"] = df["IC"] * df["thread"]
5  df["IC per increment"] = df["IC"]/(df["arg1"]/df["thread"])
6  df["Cycles per increment"] = df["cycles"]/(df["arg1"]/df["thread"])
7  df["L1 Misses Per Increment"] = df["L1_cache_misses"]/(df["arg1"]/df[
8  df["locks"] = df["arg2"]
9  display(df[["thread", "locks", "IC per increment", "CPI", "ET", "Cycl(
10 plotPEBar(df=df, include_numbers=False, what=[("label", "IC per incre
```

Question 7 (Correctness - 3pts)

Answer the questions below:

- How much does adding locks slow down the single thread case in terms of cycles per increment?
- How much does adding a second thread slow down each increment in terms of cycles per increment?

- How many cycles does it take to take and release a lock?

There are two main things to take away from this data:

1. Just taking and releasing locks is expensive -- even if there's only one thread. This overhead comes mostly from increased instruction count. In addition, fences are expensive: up to 33 cycles on our machine.
2. When there is more than one thread competing for the lock, things get even worse. This overhead comes from increase cache misses.

Both of these are noteworthy. The extra instructions in the locks are an example of the overhead involved in improving performance. We saw this before with loop tiling: Splitting and renesting loops can improve performance but it also increases instruction count, so some of the improved performance goes to paying for that overhead. It the same thing here: Taking and releasing a lock is code we didn't have to run before and that doesn't contribute to the useful work our program is doing. There is no free lunch.

The cache misses are a bigger problem and, interestingly, they are due to kind of cache miss we have not seen before in this class.

11.4 Cache Coherence and the 4th C

In the last two labs we've seen the Capacity, Conflict, and Compulsory misses, but there is a 4th kind of miss that only occurs in multi-processing systems: Coherence Misses.

As you learned in CSE142, cache coherence is how the processor keeps multiple caches synchronized, so that the processors all see the same value for a given address.

To refresh your memory, the key point of coherence are:

1. Coherence operates on cache lines, like all things in the memory hierarchy.
2. Multiple processors can have a copy of the a cache line in their cache if they are *only* reading from it.
3. Only a single processor may have a copy of the cache line if it is writing to it.

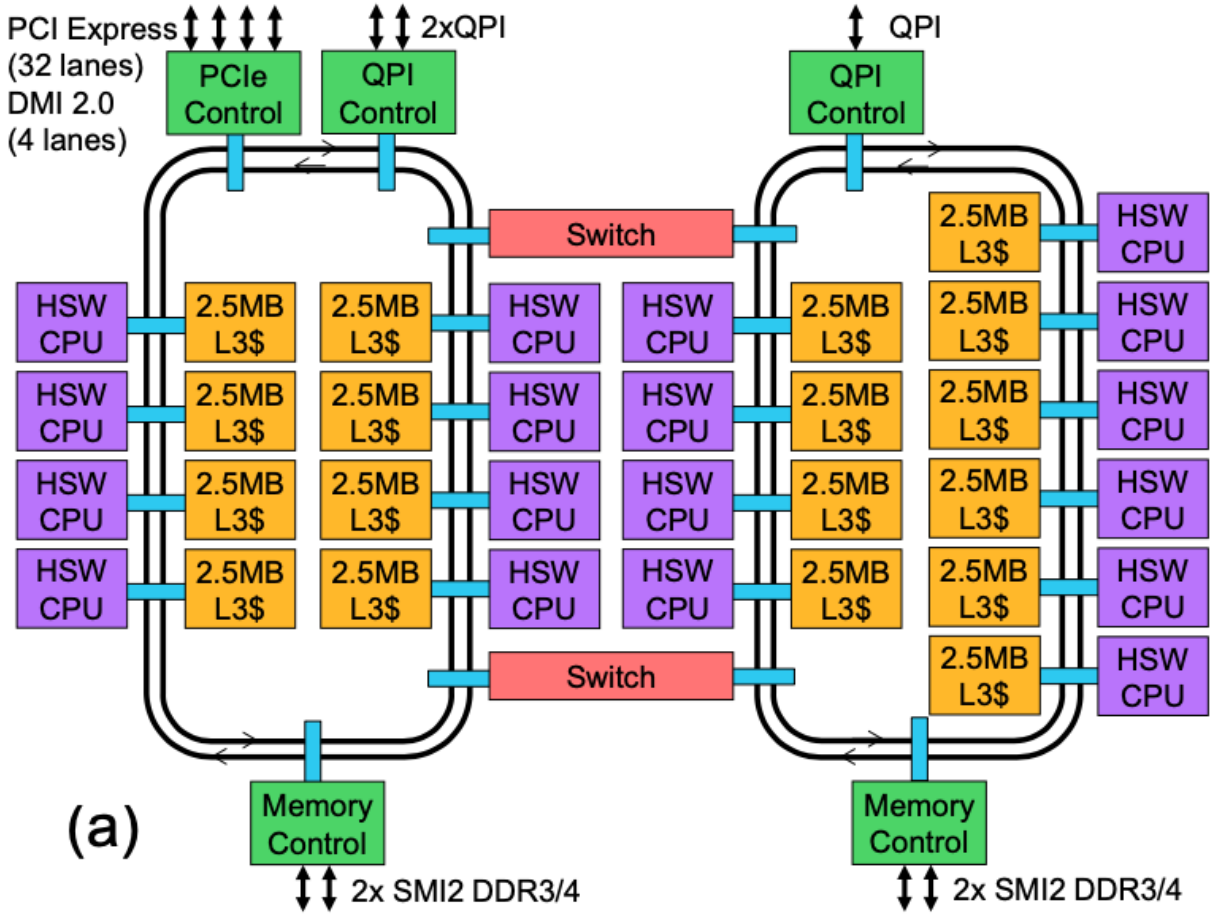
Enforcing #2 and #3 is expensive: When a processor wants to update a cache line, it has to tell all the processors that have a copy of it to *invalidate* their copy. This means removing it from the cache.

When a memory access would have been a hit, but it is a miss because the cache was invalidated, we call that a *coherence miss*.

Satisfying cache misses is multiprocessors is also more complicated due to coherence: If a load misses in the cache it has to check the other caches to see if they have a copy. If they do, then that copy is more up-to-date than what is in main memory, so that is where the cache line needs to be loaded from. This is called a *cache-to-cache* transfer.

The invalidations and cache-to-cache transfers are implemented by sending message between the caches over an on-chip network.

Here's an example of an on-chip network for an 18-core Intel Haswell processor:



Sending messages across such a network can be pretty expensive and take quite a long time.

11.4.1 Coherence Performance

Here's a simple program to measure the cost of communicating between cores. We are going to run two threads either on the same core or on two different cores. The only trick is that we control which thread runs where. You can look at `threads.hpp` to see how `bind_to_core()` works.

One thing to keep in mind: When the two threads are running on the same core, they will have to take turns, so that single core will be doing twice as much work than each core in the two-core case.

So, in an ideal world, the two-core case would be twice as fast as the one-core case.

In [ ]:

```
t = fiddle("coherence.cpp", function="coherence", analyze=False, opt=
code=r"""
#include"function_map.hpp"
#include<cstdint>
#include<thread>
#include<mutex>
#include"threads.hpp"
#include"pthread.h"

std::mutex lock;
volatile int shared = 0;
void go(uint64_t id,int count) {
    for(int i= 0; i < count; i++){
        lock.lock();
        shared++;
        lock.unlock();
    }
}

extern "C"
uint64_t* coherence(uint64_t thread_count, uint64_t * data, uint64_t 
    shared = 0;
    std::thread other (go, 1, arg1);
    bind_to_core(other, arg2);

    bind_to_core(pthread_self(), arg3);
    go(0, arg1);
    other.join();
    return data;
}
FUNCTION(one_array_2arg, coherence);
""", run=["perf_count"],
           cmdline=f"--arg1 10000000 --arg2 0 1   --arg3 0" ,
           perf_cmdline="--stat-set L1.cfg --MHz 3500")

```

In [ ]:

```
df = render_csv("build/coherence.csv")
df["other_core"] = df["arg2"]
df["this_core"] = df["arg3"]
df["label"] = df["arg2"].astype(str) + " to " + df["arg3"].astype(str
df["IC per increment"] = df["IC"]/df["arg1"]
df["Cycles per increment"] = df["cycles"]/df["arg1"]
display(df[["thread", "size", "arg1", "this_core", "other_core", "arg
plotPEBar(df=df,  what=[("label", "CPI"), ("label", "ET"), ("label",
```

In this case, two cores is not better than one: Even with two processors to do the work, execution slows down by a wide margin!

The underlying problem is all the coherence misses necessary as `shared` 's cache line "ping pongs" between the two cores.

11.4.2 Reducing Sharing

Since frequent sharing seems to hurt performance, a good way to improve performance in concurrent programs is to reduce sharing. In fact, a central tenant of multi-threaded programming to reduce sharing as much as possible.

Here's a version of the code above with two changes:

1. Most of the sharing removed: Each thread works on its own variable and we add them up at the end.
2. We don't use locks any more.

In [ ]:

```
t = fiddle("false_sharing.cpp", function="false_sharing", analyze=Fal
code=r"""
#include"function_map.hpp"
#include<cstdint>
#include<thread>
#include<mutex>
#include"threads.hpp"
#include"pthread.h"

volatile int shared = 0;
volatile int not_shared_0 = 0;
volatile int not_shared_1 = 0;

void go_0(uint64_t id,int count) {
    for(int i= 0; i < count; i++){
        not_shared_0++;
    }
}

void go_1(uint64_t id,int count) {
    for(int i= 0; i < count; i++){
        not_shared_1++;
    }
}

extern "C"
uint64_t* false_sharing(uint64_t thread_count, uint64_t * data, uint6
    shared = 0;
    std::thread other (go_0, 1, arg1);
    bind_to_core(other, arg2);

    bind_to_core(pthread_self(), arg3);
    go_1(0, arg1);
    other.join();
    shared = not_shared_0 + not_shared_1;
    return data;
}
FUNCTION(one_array_2arg, false_sharing);
""", run=["perf_count"],
           cmdline=f"--arg1 10000000 --arg2 0 1   --arg3 0" ,
           perf_cmdline="--stat-set L1.cfg --MHz 3500")

```

Question 8 (Completeness)

How much difference in performance do you expect to see between running both threads on one core vs. running them on two cores?

In [ ]:

```
df = render_csv("build/false_sharing.csv")
df["other_core"] = df["arg2"]
df["this_core"] = df["arg3"]
df["label"] = df["arg2"].astype(str) + " to " + df["arg3"].astype(str
df["IC per increment"] = df["IC"]/df["arg1"]
df["Cycles per increment"] = df["cycles"]/df["arg1"]
display(df[["thread", "size", "arg1", "other_core", "arg3", "IC", "CP
plotPEBar(df=df,  what=[("label", "CPI"), ("label", "ET"), ("label",
```

Interesting. That helped some, but not as much we'd like: Ideally, we'd get 2x speedup with 2 cores, but instead we only get about 1.4x.

Question 9 (Completeness)

Add a single line to the code above to get the 2x performance improvement we seek. (Hint: The memory system thinks in cache lines and so should you).

Show Solution

11.4.3 Non-Uniform Memory Access

Multiple processors complicates the notion of memory latency as well.

If you look closely at the on-chip network above, you can notice that there are two memory controllers -- one on each of the side of the on-chip network. This mean that, depending on where a core is in relation to the DRAM it's trying to access, the memory latency will be different. Similarly, depending on which part of the L3 cache a cache line landed, the latency of an L2 cache might vary.

This effect is called non-uniform memory access (NUMA). NUMA effects are even greater if a computer has multiple sockets -- some memory requests will have to go between chips while others are "local".

It'd be great to measure NUMA effects on our machine, but our machine only has 6 cores, 1 socket, and one (active) memory controller, so there's not enough non-uniformity to measure reliably.

12 Example: Histogram [...]

13 Amdahl's Law and Imperfect Speedup

In our histogram example, we got 6x speedup with 6 threads. Speeding up a program by p with p processors is called *linear speedup* and it's always the goal of multi-threaded. It's hard to achieve, however, because of Amdahl's Law.

So far, in this class we've talked about Amdahl's Law as it applies optimizations: The more widely applicable an optimization, the larger it's benefit. Originally, however, Amdahl's Law was just about parallel computation. In particular, if we have p processors, Amdahl's Law bounds the maximum speedup, S, we can achieve:

$$S \leq \frac{1}{x/p + (1-x)}$$

Where x is the fraction of the program that can parallelized across all p processors. For the histogram example, $x = 1$, so $S \leq p$, and our implementation achieved this upper bound.

Usually, however, $x < 1$, so $S < p$.

To see this, let's parallelize merge sort. The implementation below divides the array in half and recursively calls merge sort on each sub array. To parallelize it, we will spawn a thread to sort each sub-array. As the sub arrays get smaller and smaller, two things will happen:

1. We will spawn an enormous number of threads.
2. Each of them will do very little work.

Neither of these is good, so we'll use `threshold` to control the minimum size for which we will spawn a thread. If the sub-array is smaller than `threshold`, we'll just do all the work in the current thread.

This means that if the array is of size `threshold` $\times 2^k$, we'll use a maximum of 2^k threads.

You'll notice that there are no locks. This is because there's not actually any sharing: Only one thread is ever working on any one part of the array at a time. (The attentive reader will recall that I said that sharing is necessary to coordinate threads. Here, the sharing and coordination happen before the threads are created and after we `join()` them).

In [ ]:

```
t = fiddle("merge_sort.cpp", function="merge_sort", analyze=False, op
code=r"""
//  From https://codereview.stackexchange.com/questions/87085/simple-
#include"function_map.hpp"
#include<cstdint>
#include<thread>
#include<math.h>
#include<mutex>
#include"threads.hpp"
#include"pthread.h"

void merge(uint64_t *list, int64_t p, int64_t q, int64_t r)
{
//n1 and n2 are the lengths of the pre-sorted sublists, list[p..q] an
    int64_t n1=q-p+1;
    int64_t n2=r-q;
//copy these pre-sorted lists to L and R
    uint64_t * L = new uint64_t[n1+1];
    uint64_t * R = new uint64_t[n2+1];
    for(int64_t i=0;i<n1; i++)
    {
        L[i]=list[p+i];
    }
    for(int64_t j=0;j<n2; j++)
    {
        R[j]=list[q+1+j];
    }


//Create a sentinal value for L and R that is larger than the largest
//element of list
    uint64_t largest;
    if(L[n1-1]<R[n2-1]) largest=R[n2-1]; else largest=L[n1-1];
    L[n1]=largest+1;
    R[n2]=largest+1;

//Merge the L and R lists
    int64_t i=0;
    int64_t j=0;
    for(int64_t k=p; k<=r; k++)
    {
        if (L[i]<=R[j])
        {
            list[k]=L[i];
            i++;
        } else
        {
            list[k]=R[j];
            j++;
        }
    }
    delete L;
    delete R;
}

```

```
void merge_sort_aux(uint64_t *list, int64_t p, int64_t r, int64_t thr
{
    if(p<r)
    {
        int64_t q=floor((p+r)/2);
        if (r - p > threshold) {
            std::thread left(merge_sort_aux, list,p,q, threshold);
            std::thread right(merge_sort_aux, list,q+1,r, threshold);
            left.join();
            right.join();
        } else {
            merge_sort_aux(list,p,q, threshold);
            merge_sort_aux(list,q+1,r, threshold);
        }
        merge(list,p,q,r);
    }

}

extern "C"
uint64_t* merge_sort(uint64_t thread_count, uint64_t *list, uint64_t 
    merge_sort_aux(list, 0, size - 1, arg1);
    return list;
}
FUNCTION(one_array_2arg, merge_sort);

""", run=None)
```

Question 10 (Completeness)

What is x for parallel merge sort? Can you bound speedup for 4 processors, a threshold of 1024, and a total array size of 4096? If you can't get a precise answer try to estimate or provide an upper bound the value of x.

Show Solution

Given our implementation, we can't directly control the number of threads we are using, but by setting `threshold` to smaller values, we'll use more threads. In particular, if we have `size == threshold * 2^k` , we will use a peak of `size/threshold` threads, that's what `approx threads` measures.

Let's run it:

In [ ]:
```
t = fiddle("merge_sort.cpp", function="merge_sort", analyze=False, op
           cmdline=f"--size {8*1024*1024} --arg1 {8*1024*1024} {4*102
           perf_cmdline="--stat-set L1.cfg --MHz 3500")
```

In [ ]:
```
df = render_csv("build/merge_sort.csv")
df["i"] = df.index
df["threshold"] = df["arg1"]
df["approx threads"] = df["size"]/df['threshold']
df["speedup"] = df.iloc[0]["ET"]/df["ET"]
display(df[["size", "threshold", "approx threads", "ET", "speedup"]])
plotPE(df=df, lines=True, what=[("approx threads", "speedup")])
```

Speedup looks ok, but it's not "linear speedup". From 1 thread to 2, we get 2x, but from 2 to 4, we only get 1.56x. From 4 to 8, we get 1.29 (although, in fairness, we only have 6 cores).

Unfortunately, this how it usually goes: Linear speedup is very hard to achieve in practice, because it requires reducing communication and serialized computation to nearly zero. We were able to do that with the histogram, but sorting (and most other things) are more complex.

And it's not just that x is less than 1. There is also usually some overhead from locks (which was absent in our histogram) which further degrades performance.

14 Hyperthreading

Earlier in the lab, we saw that our CPU *says* it has 12 processors even though it really only has 6. The 6 extra, logical processors are due to hyperthreading, a clever trick that allows two threads to run *at exactly the same time* on the same processor.

The catch is that the two threads running on the same core compete with each other for resources. This works fine if the two threads need different resources, but if they need the same resources, performance will suffer.

Let's see how it does for our two parallel programs. Here's the histogram with threads turned up to 12:

In [ ]:
```
t = fiddle("histogram.cpp", function="run_private2_histogram",  name=
           cmdline=f"--size {hist_size} --thread 1 2 3 4 5 6 7 8 9 10
           perf_cmdline="--stat-set L1.cfg --MHz 3500")
```

In [ ]:
```
hist_data = render_csv(["histogram_unthreaded.csv", "histogram_hyper.
#hist_data["ET"] = hist_data["ET"]/hist_data["thread"].apply(lambda x
hist_data["speedup"] = hist_data.iloc[0]["ET"]/hist_data["ET"]
hist_data["Total IC"] = hist_data["IC"]*hist_data["thread"]
hist_data["Total cache misses"] = hist_data["L1_cache_misses"]*hist_d
display(hist_data[hist_columns])
plotPE(df=hist_data, lines=True, what=[("thread", "speedup")])

```

For thread counts 7-12 some of the threads are sharing a core. As you can see, there's a hit to performance when we go to 7 cores. This because the execution time of the whole program is the execution time of the *slowest* thread. In this case, that is either thread 1 or thread 7, since they share a core.

Adding more hyperthreads improves performance but not as much as adding more full processors. For instance, doubling the number of logical processors from 6 to 12 only yield only about 1.3x speedup, while going from 3 to 6 gave us 2x.

Still, 1.3x is pretty good, really, since it didn't require any additional hardware.

And here's merge sort:

In [ ]:
```

t = fiddle("merge_sort.cpp", function="merge_sort", analyze=False, op
           cmdline=f"--size {8*1024*1024} --arg1 {8*1024*1024} {4*102
           perf_cmdline="--stat-set L1.cfg --MHz 3500")
```

In [ ]:
```
df = render_csv("build/merge_sort.csv")
df["i"] = df.index
df["threshold"] = df["arg1"]
df["approx threads"] = df["size"]/df['threshold']
df["speedup"] = df.iloc[0]["ET"]/df["ET"]
display(df[["size", "threshold", "approx threads", "ET", "speedup"]])
plotPE(df=df, lines=True, what=[("approx threads", "speedup")])
```

The story is pretty similar: From 2 to 4 threads gives 1.6x, while going from 4 to 8 only gives us 1.24.

15 OpenMP

Using locks and threads to parallelize code is notoriously tricky. I chose the histogram and merge sort examples intentionally because they are pretty simple. Fortunately, the compiler can help us a great deal for certain kinds of code.

Compilers have had good success automatically parallelizing code for well-behaved loops. "Well-behaved" means loops with loop bounds that don't change and that increment their index variables by fixed amounts. Many programs fall into this category, including matrix multiplication and our histogram code. Merge sort, by contrast, does not.

OpenMP (Open Multi-Processing) is a widely-used and widely-supported set of extensions for C/C++ (and Fortran, if you're into that) that let the programmer guide the compiler to parallelize loops.

The extensions take the form of `#pragma` s. OpenMP has many of them, but we will use three:

- `#pragma omp parallel for`
- `#pragma omp critical`
- `#pragma omp simd`

`omp parallel for` tells the compiler to parallelize the following loop.

`omp critical` tells the compiler that the next block of code should be treated as a critical section.

`omp simd` tell the compiler to try to vectorize the loop.

For instance, here's the OpenMP version of our histogram.

In [ ]:
```
render_code("histogram.cpp", show=("//START_OPENMP", "//START_OPENMP"
```

This code is roughly equivalent to the course-grain locking version we wrote earlier. Internally, OpenMP divides the iterations of the outer loop into chunks, and sends them threads to execute them. Pretty slick! It create a single lock to protect the histogram update. Probably not so slick.

Let's see how it does:

In [ ]:
```

os.environ["OPENMP"]="yes"
t = fiddle("histogram.cpp", function="run_openmp_histogram",  name="h
           cmdline=f"--size 10000000 --thread 1 2 3 4 5 6",
           perf_cmdline="--stat-set L1.cfg --MHz 3500")
```

In [ ]:
```
hist_data = render_csv(["histogram_unthreaded.csv", "histogram_openmp
#hist_data["ET"] = hist_data["ET"]/hist_data["thread"].apply(lambda x
hist_data["speedup"] = hist_data.iloc[0]["ET"]/hist_data["ET"]
hist_data["Total IC"] = hist_data["IC"]*hist_data["thread"]
hist_data["Total cache misses"] = hist_data["L1_cache_misses"]*hist_d
display(hist_data)
plotPE(df=hist_data, lines=True, what=[("thread", "speedup"), ])
```

That's awful! But not unexpected. One lock means no concurrency and a shared histogram array

means lots of sharing. We should have known better!

Here's a better OpenMP version:

In [ ]:
```
render_code("histogram.cpp", show=("//START_OPENMP_PRIVATE", "//START_
```

We did the blocking ourselves this time. Splitting the loop gives us a chance to declare a local set of histogram counters inside the loop. These counters are local to the loop body they live in. Since the iterations will be running in different threads, each thread gets their own histogram. We initialize them and record our data in the loop. Then, at the end, we grab the lock once and merge in our updates. If we set `arg1` to 1000, this should mean 1000x few locks and unlocks, and there will be much less frequent sharing.

Let's see how this version does:

In [ ]:
```

t = fiddle("histogram.cpp", function="run_openmp_private_histogram",
           cmdline=f"--size {hist_size} --thread 1 2 3 4 5 6 --arg1 1
           perf_cmdline="--stat-set L1.cfg --MHz 3500")
```

In [ ]:
```
hist_data = render_csv(["histogram_unthreaded.csv", "histogram_openmp_
#hist_data["ET"] = hist_data["ET"]/hist_data["thread"].apply(lambda x
hist_data["speedup"] = hist_data.iloc[0]["ET"]/hist_data["ET"]
hist_data["Total IC"] = hist_data["IC"]*hist_data["thread"]
hist_data["Total cache misses"] = hist_data["L1_cache_misses"]*hist_d
display(hist_data[hist_columns])
plotPE(df=hist_data, lines=True, what=[("thread", "speedup"), ])

```

Pretty good: 5.95x speedup with 6 threads (there's some variation from run to run.). And the code is much simpler with OpenMP! Sounds good to me.

Especially, if I had to do PA.

15.1 SIMD Parallelism in OpenMP

OpenMP also support SIMD parallelism, which uses vector instructions to improve performance. We discussed SIMD during the 142L lecture last week. Review the podcast, if you missed it.

Applying SIMD with OpenMP is pretty easy: You just put `#pragma omp simd` before the loop body. The loop needs to very regular (e.g., constant stride and no branches), and there's no guarantee that the compiler will be able to vectorize it.

Here's an example of using SIMD parallelism for dot product.

In [ ]:

```
t = fiddle("dp.cpp", function="dp", analyze=False, opt="-O3",
code=r"""
#include"function_map.hpp"
#include<cstdint>

extern "C"
uint64_t vsum(uint64_t *a, uint64_t* b, uint64_t * c, uint64_t len)
{
    uint64_t s = 0;
    for(unsigned int i = 0; i < len; i++) {
        c[i]=a[i]+b[i];
    }
    return s;
}

extern "C"
uint64_t vsum_simd(uint64_t *a, uint64_t* b, uint64_t * c, uint64_t l
{
    uint64_t s = 0;
#pragma omp simd
    for(unsigned int i = 0; i < len; i++) {
        c[i]=a[i]+b[i];
    }
    return s;
}


extern "C"
uint64_t* dp(uint64_t threads, uint64_t * list, uint64_t size, uint64_
    if(arg1 == 0) {
        list[0] = vsum(list, &list[size/3], &list[size*2/3], size/3);
    } else if(arg1 == 1){
        list[0] = vsum_simd(list, &list[size/3], &list[size*2/3], siz
    }
    return list;
}
FUNCTION(one_array_2arg, dp);


""", run=["perf_count"],
           cmdline=f"--size 10000000 --thread 1 --arg1 0 1",
           perf_cmdline="--stat-set L1.cfg --MHz 3500")
```

In [ ]:

```
df = render_csv("build/dp.csv")
df["opt"] = df["arg1"].apply(lambda x: ["no SIMD", "SIMD"][x])
df[["opt", "IC", "CPI", "CT", "ET"]]
```

1.16x speedup for one line of code is pretty good! Note IC dropped but CPI and CT went up a bit!

Here's the assembly:

In [ ]:
```
1 compare([do_cfg("build/dp.so", symbol="vsum"), do_cfg("build/dp.so", s
```

It's the `dq` suffixes and the `xmm` registers in the big basic block on the right that show we are using vectors.

This code is not taking full advantage of our machine, however. By default, gcc only uses the mmx instructions that can use 128-bit vectors. Our processor has AVX, which supports 256-bit. We can turn this on with `-march=native` :

In [ ]:
```
1
2 !rm -f build/dp.o build/dp.so
3 t = fiddle("dp.cpp", function="dp", analyze=False, opt="-O3 -march=na
4           cmdline=f"--size 10000000 --thread 1 --arg1 0 1",
5           perf_cmdline="--stat-set L1.cfg --MHz 3500")
```

In [ ]:
```
1 df = render_csv("build/dp.csv")
2 df["opt"] = df["arg1"].apply(lambda x: ["no SIMD", "SIMD"][x])
3 df[["opt", "IC", "CPI", "CT", "ET"]]
```

Speedup is the same, but IC dropped significantly while CPI went up dramatically.

Here's the assembly:

In [ ]:
```
1 compare([do_cfg("build/dp.so", symbol="vsum"), do_cfg("build/dp.so", s
```

Now it's using `ymm` registers, with are 256 bits.

All-in-all, SIMD is a mixed bag on our machine, it seems, but it doesn't seem to hurt much and can improve performance a bit.

16 Programming Assignment

Your programming assignment in this lab is an extension of your work on the last lab. This time, you'll be parallelizing your implementation of matrix exponentiation. You can C++ threads, pthreads, or OpenMP. The lab assume you'll use OpenMP. If you use one of the others, the course staff will be less able to help you out.

You are free to use your (and only your) solution to the previous lab as a starting point for this lab.

Most of this section is a verbatim copy of the PA for Lab 4. The sections with new content have " (New for Lab 5)" at the end of their names.

Obviously, the performance targets have been increased.

16.1 Performance Variability (New for Lab 5)

> **Multithreaded Performance is Variable**: Multithreading introduces variability in performance. You will only get credit for performance numbers recorded on gradescope, you may need to run your gradescope job more than once to get meaningful measurements.

Running code on multiple processors introduce performance variability -- some times up to 20% or more. In the real world, you'd either tolerate this or spend a lot of time trying to fix it. Neither of those works well in a lab, because we have limited time and you need a grade that reflects the quality of your work rather than whether a run was lucky or unlucky.

To address this problem, the lab includes does two things:

1. It runs your code 6 times and takes the average. This is implemented in the `Makefile`.
2. It includes a "canary" program with known performance.

The canary runs before your code and if the performance of the canary is too low, the autograder will reject the run. This means that it is much more likely that your gradescope submissions will fail and you may need to submit several times to get a good measurement.

With the canary filtering out slow runs, performance variation is less this 5%. So there is some marginal value in submitting multiple times in the hopes of getting a slightly good score.

> **Budget time for multiple submissions**: Having to resubmit to gradscope due to canary failure is totally predictable, and you have been warned. Plan to submit (and resubmit as necessary) well-ahead of the deadline.

16.2 Reference Code (New For Lab 5)

The reference implementation is in `matexp_reference.hpp`. It's basically the same as for Lab 4, but it demonstrates `wall_time()` (see below) and it makes the tensor copy explicit:

In [99]:
```
render_code("matexp_reference.hpp")
```

```
// matexp_reference.hpp:1-93 (93 lines)
#ifndef MATEXP_REFERENCE_INCLUDED
#define MATEXP_REFERENCE_INCLUDED
#include <cstdlib>

#include <unistd.h>
#include<cstdint>
#include"function_map.hpp"
#include"tensor_t.hpp"
#include"pin_tags.h"
#include<archlab.hpp>


template<typename T>
void  __attribute__((noinline,optimize("Og"))) mult_reference(tensor_t<T>
&C, const tensor_t<T> &A, const tensor_t<T> &B)
{
        // This is just textbook matrix multiplication.

        for(int i = 0; i < C.size.x; i++) {
                for(int j = 0; j < C.size.y; j++) {
                        C.get(i,j) = 0;
                        for(int k = 0; k < B.size.x; k++) {
                                C.get(i,j) += A.get(i,k) * B.get(k,j);
                        }
                }
        }
}


// A simple function to copy the contents of one tensor into another.
template<typename T>
void copy_matrix_ref(tensor_t<T> & dst, const tensor_t<T> & src) {
        for(int32_t x = 0; x < dst.size.x; x++)
                for(int32_t y = 0; y < dst.size.y; y++)
                        dst.get(x,y) = src.get(x,y);
}

template<typename T>
void __attribute__((noinline,optimize("Og"))) matexp_reference(tensor_t<T
> & dst, const tensor_t<T> & A, uint32_t power,
                        // parameters you can use for whatever purpose you
 want (e.g., tile sizes)
                        int64_t p1=1,
                        int64_t p2=1,
```

```
                    int64_t p3=1,
                    int64_t p4=1,
                    int64_t p5=1) {
        // Tags for moneta

        TAG_START("dst", dst.start_address(), dst.end_address(), false);
        TAG_START("A", A.start_address(), A.end_address(), false);

        // In psuedo code this just
        //
        // dst = I
        // for(i = 0..p)
        //    dst = dst * A

        // Start off with the identity matrix, since M^0 == I
        // The result will end up in dst when we are done.
        for(int32_t x = 0; x < dst.size.x; x++) {
                for(int32_t y = 0; y < dst.size.y; y++) {
                        if (x == y) {
                                dst.get(x,y) = 1;
                        } else {
                                dst.get(x,y) = 0;
                        }
                }
        }


        // wall_time() returns the time since and has microsecond accurac
y.
        // Substract times to get latency.
        double started = wall_time();
        for(uint32_t p = 0; p < power; p++) {
                // Copy dst, since we are going to modify it.  We constru
ct B
                // to be the same size as dst, but pass false to avoid ze
roing
                // it, since we are about to overwrite it.
                tensor_t<T> B(dst.size, false);
                copy_matrix_ref(B, dst);
                TAG_START("B", B.start_address(), B.end_address(), false
);
                mult_reference(dst,B,A); // multiply!

                TAG_STOP("B");

        }
        double finished = wall_time();

        std::cerr << "That took " << finished - started << " seconds\n";
```

```
        TAG_STOP("dst");
        TAG_STOP("A");

}

#endif
```

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

16.3 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`.

16.4 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:

16.5 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`, `--power`, and `--thread`, you can multiple values and `matexp.exe` will run all combinations. For example:

16.6 The Test Suite [...]

Tests are great: Tests are about the best thing ever (although writing them is a hassle). If you run the tests consistently, you can worry *much* less about correctness. Make small incremental changes to your code, run the tests after each change and enjoy the warm glow of happiness when they pass!

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 `run_test.cpp` . `run_tests.exe` also takes the `--p*` arguments so you can run the regressions with different parameter settings. This is a *good* idea.

For this lab, the autograder runs the tests with multiple threads.

You can build the tests with:

16.7 OpenMP (New For Lab 5)

OpenMP is automatically turned for your code in this lab, so the `#pragma` command we used for the histogram should work fine.

If you want to compile at the command line, you'll need to either do

```
export OPENMP=yes
```

once each time you log in or invoke `make` like so:

```
make matexp.exe OPENMP=yes
```

each time you build.

16.7.1 Key Commands

The three `#pragma` s you'll need for this lab are

1. `#pragma omp parallel for` for parallelizing loops.
2. `#pragma omp critical` for parallelizing loops.
3. `#pragma omp simd` for vectorizing loops

These are the only three used in the solution used to set the performance targets.

These three blog posts provide a good introduction to these commands:

- http://jakascorner.com/blog/2016/04/omp-introduction.html (http://jakascorner.com/blog/2016/04/omp-introduction.html)
- http://jakascorner.com/blog/2016/05/omp-for.html (http://jakascorner.com/blog/2016/05/omp-for.html)

- http://jakascorner.com/blog/2016/07/omp-critical.html (http://jakascorner.com/blog/2016/07/omp-critical.html)

They are required reading.

These articles provide some more advanced topics that might be useful:

- http://jakascorner.com/blog/2016/06/omp-for-scheduling.html (http://jakascorner.com/blog/2016/06/omp-for-scheduling.html)
- http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html (http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html)
- http://jakascorner.com/blog/2016/07/omp-default-none-and-const.html (http://jakascorner.com/blog/2016/07/omp-default-none-and-const.html)

There's an enormous amount of bad information online about OpenMP.

16.7.2 Looking at OpenMP Assembly

If you look at the assembly for OpenMP programs, you'll find that your loop body has been replaced with a function call. OpenMP does this so it can tell it's worker threads to call the function to perform one iteration of your loop.

16.7.3 Caveats for our Tools

First, `gprof` doesn't work on with multi-threaded programs. You can use it for single-threaded runs, though.

Second, Moneta's cache model is not multithread-aware, so the cache hit/miss numbers for multithreaded programs are not meaningful.

Moneta may also show more threads that you might be expecting. OpenMP threads seem to be Thread 0 and 13 and above. If you see some threads that don't seem to be doing anything, that's not surprising or concerning.

Finally, our performance counting code only collects data for one thread. For OpenMP code this is ok: all the threads do basically the same thing. But you'll notice, for instance, that if a loop runs in 4 threads, the measure instruction count will go down by ~1/4 (assuming multi-threading didn't add a lot of overhead).

16.7.4 Controlling the Number of Threads

By default the autograder will run your code with 12 threads. If you want to use a different number in your final run, you can call

```
omp_set_num_threads(thread_count);
```

in your function. You should call it before the first OpenMP `#pragma`.

I can control thread count during development with `--thread`.

16.8 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.

16.8.1 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 tile size be 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. Debugging tiling
   A. Debugging tiling can be tricky.
   B. Start with small matrices and small tile sizes.
   C. Try multiple small tile sizes (pass `--p*` to the regressions)
4. Don't forget about loop overhead.

There's a nice Wikipedia page (https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm) about matrix multiplication. It covers the theory behind implementing it effectively. The content is good, but don't assume that theory and practice will match.

16.8.2 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.

16.8.3 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.

16.8.4 Non-Deterministic Tests (New for Lab 5)

With threading, comes non-deterministic bugs. This means that the tests may fail only occasionally for your code. If this seems to be happening, a good strategy is to just run them repeatedly and confirm that it's the case.

It's not a bug in the test suite, you have a thread synchronization error.

16.8.5 General Tips (New for Lab 5)

In the examples we saw that loop tiling and OpenMP pragmas can work well together. This carries through to how you should figure out what to parallelize. It's worth your time to try parallelizing different loops and changing how your loops are nested to accommodate that.

A few tips:

1. At this points you have many tools available to you -- `omp parallel for`, `omp simd`, compiler optimizations, tiling. The number of combinations is enormous. I suggest applying them in this order (from largest impact-per-effort to smallest):
   B. tiling (last lab)
   C. `omp parallel for`
   D. `omp simd`
   E. Fiddling with other compiler options/per-function compiler options.
   F. Crazy stuff like intrinsics for better SIMD performance.
2. While tiling only applied to loops with reuse (because temporal locality requires reuse), `parallel for` can apply to loops without reuse. Same for `omp simd`.
3. `omp parallel for` implicitly does tiling, since it divides the iterations of the parallel loop across several cores.
4. Nesting parallel for loops with OpenMP is not usually a good idea (although it should work). Start by picking one loop to parallelize.
5. You want to parallelize an outer loop, so that the threads are working on large pieces of computation and need to synchronize less.
6. Pay close attention to whether iterations of your parallel loop are writing to the same locations. If so, you'll need a `omp critical` to ensure correct updates.

This last point can be tricky. If I have this code:

```
#pragma omp parallel for
for(int i = 0; i < 10; i++) {
    for(int j = 0; j < 10; j++) {
        for(int k = 0; k < 10; k++) {
            X(i,j) += Y(k,j);
```

The only store is the assignment to `X(i, j)`. Since `i` is the index of the parallel loop, I know that no other thread will be updating `X(i,j)`, since no other thread will have the same value of `i`.

However, in this code:

```
#pragma omp parallel for
for(int i = 0; i < 10; i++) {
    for(int j = 0; j < 10; j++) {
        for(int k = 0; k < 10; k++) {
            X(k,j) += Y(i,j);
```

I don't have the same guarantee. Since `i` does is not used to select an element in `X`, every other thread will write to that location as well. In that case, I could do

```
#pragma omp parallel for
for(int i = 0; i < 10; i++) {
    for(int j = 0; j < 10; j++) {
        for(int k = 0; k < 10; k++) {
#pragma omp critical
            X(k,j) += Y(i,j);
```

Which will probably be really slow, or I could create a private tensor, do my updates there, and then merge them into `X`.

5. Pay close attention to write sharing when you are deciding how to parallelize. Can the iterations of your parallel loop iterations write to the same place?
6. `omp simd` only works on inner loops.
7. `omp parallel for` works best on outer loops.
8. `gprof` doesn't work for multithreading, so use `wall_time()` to measure how long things take.

Regarding #7: If you've parallelize part of your program, and speedup is good but not great, use `wall_time()` to identify serial parts of your code that are slow. The problem may be that your x in Amdahl's Law is too small.

16.8.6 Useful C++ (Partly New For Lab 5)

`wall_time()` (New for Lab 5)

We've provide `wall_time()` which returns the current time as a `double`. It has microsecond accuracy. You can call it before and after part of your program, subtract the resulting values, and get a pretty good measure of execution time. There's an example in `matexp_reference.hpp`.

Controlling Compiler Optimizations

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
```

Assertions

The `assert()` macro is useful tool for debugging and to avoid silly errors.

If you say

```
assert(a > b);
```

And the expression is not true at run time, the assert with "fail" your program will crash with a somewhat useful error message.

This is a useful way to document and enforce assumptions you make in your code. For instance, I used an assert in `convolution_tiled_split()` to ensure that the tile size was > 8.

You can get access to `assert()` with

```
#include<cassert>
```

The overhead of asserts is low, but not zero. I would not put any in one of your performance-critical loops.

If you want to include asserts in performance-critical areas, you can add `-DNDEBUG` to the optimizations in `config.make`. It'll disable all the `assert()`s.

16.9 Do Your Work Here [...]

Below are the key commands you'll need to make progress on the lab.

16.10 Tools [...]

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

16.11 Final Measurement [...]

17 Recap [...]

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 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.

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.

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

18.4 The Programming Assignment

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

18.5 Lab Survey

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

In [103]:

```
1 from IPython.display import IFrame
2 IFrame('https://docs.google.com/forms/d/e/1FAIpQLScHbK7yLlixJqdYsRnpv
3
```

Out[103]:

CSE142L Lab Survey

swanson@eng.ucsd.edu Switch account

* Required

Email *

Your email

Student ID (to get credit) *

Your answer

Student ID (to confirm) *

Your answer

Which lab is this? *

Choose

How hard did you think this lab was? *

	1	2	3	4	5	
Very easy	○	○	○	○	○	Ve

How much did you learn from this lab? *

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

How interesting was this lab? *

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

How many hours did you spend on this lab? *

Your answer

Name one thing you liked about the lab *

Your answer

Name another thing you liked about the lab *

Your answer

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

Your answer