Performance on a single core

1 Mental models of performance

In the last lecture, we mentioned that we judge our parallel programming efforts by scaling experiments in which we compare the time to solve a problem on P processors to the time required to solve the same problem on a single processor. But the only way this approach can be intellectually honest is if we believe that our single-processor code is well tuned. Therefore, our goal in this lecture is to consider what constitutes "good" performance on a single processor, and how we can achieve such good performance.

When we reason about the *correctness* of a program written in C, we can use a simple, idealized model of the machine in which memory is an (effectively infinite) address space of named words and C instructions map in an "obvious" way into machine code, which then runs in the "obvious" order. When we start to reason about the *asymptotic* performance, we add costs to our simple model, usually assuming that memory opertions, arithmetic, and logical operations all cost about the same amount. This gives us information about how the time to run an algorithm scales with problem size, which is very useful stuff. You may very well have seen this sort of "big-O" performance analysis in classes on algorithm analysis.

Though our naive model of how machines execute code is useful for understanding how computational costs scale as a function of problem size, it can mis-predict the actual performance of a code. In fact, a modern machine runs many instructions simultaneously, and even the time to run the same type of instructions — a memory read, for example — can vary by orders of magnitude depending on the state of the system. To arrive at a more realistic model for predicting performance, we need to understand more about how modern machines work "under the hood". We will focus on two aspects of the machine architecture: instruction-level parallelism, and the effects of the memory hierarchy. Fortunately, we do not need to think of *all* the details of the computer architecture in order to get good performance; modern compilers are much better than human beings at generating code that makes good use of low-level architectural features of modern systems. Instead, we would like a high-level understanding of the major architectural features that is accurate enough so that we can make *high-level* decisions about algorithm and data structure design that allow us to attain good performance.

2 Single-processor parallelism

2.1 Pipeline parallelism

A single processor can execute several instructions at once using a *pipeline*. Think of doing laundry: if you have several loads to clean, you would usually simultaneously fold the first load, dry the second load, and wash the first load. The three loads are being handled in parallel, with each load at a different stage. No individual load of laundry goes any faster in this system — that is, the *latency* is unaffected by the pipelining — but the total amount of laundry per unit time (the laundry *bandwidth*) is substantially improved.

Similarly, in a simple pipelined architecture there might be five stages to executing an instruction:

1. Fetch the instruction from memory;
2. Decode the instruction (e.g. decide it's an addition);
3. Execute the instruction (e.g. actually add two numbers);
4. Read from memory, if needed;
5. Write back the results (e.g. store the sum in a register).

In a processor with this five-stage pipeline, we might have five instructions in flight simultaneously. In general, if we have a p-stage pipeline operating at maximum efficiency, then we expect to take $m + p - 1$ cycles to execute m instructions[1]. This is a pretty good improvement over the time it would take if we ran the instructions serially (mp cycles). But there's a catch: our five-stage pipeline can only handle five instructions simultaneously if those instructions are sufficiently independent. If the first instruction writes a result to a register and the second instruction uses that same register as input, then the second instruction cannot start to execute (enter stage 3) until the first instruction has finished writing (exited stage 5). In this case, there will be a "bubble" in the pipeline while the second instruction waits for the first to complete.

[1]The assumption here is that each pipeline stage takes the same amount of time. If this is not the case, the length of a "cycle" is limited by the time it takes to finish the slowest stage.

Pipeline bubbles can occur because of data dependencies or because of branches. In either case, they reduce the level of parallelism available to the processor. This might not seem like such a big deal with a five stage pipeline, but the original Pentium machines have a twenty stage pipeline — and my old desktop, a Pentium 4 Prescott had a pipeline with 31 stages. Despite branch prediction and other clever tricks for mitigating the effect of pipeline stalls, pipeline bubbles were a serious performance barrier for these machines. This is why the newer Intel processors, like the Intel Core 2 Duo on my laptop, have only 14 pipeline stages.

2.2 Beyond pipelines

In fact, the architectural picture is even more complicated that we have described. For example, consider again the Intel processor on my laptop. This machine is capable of starting four instructions *simultaneously*[2]. These instructions are broken down into more basic "micro-operations"[3] Modern "superscalar" chips aggressively schedule instructions to use many functional units at once, so the processor can simultaneously execute several instructions in the "same" pipeline stage, assuming those instructions use different functional units in the hardware. These instructions can even be executed out of order, though the hardware ensures that in the end the output of the instructions is written in serial order. The details are interesting, but they are well beyond the scope of this class.

Fortunately, we don't have to understand *all* the detals of chip architecture in order to write fast code. One of the jobs of an optimizing compiler is to schedule instructions in a way that lets the chip make the best use of these parallel features; and typically, compilers do a much better job at this sort of scheduling than human beings. The role of the human, then, is to make any available instruction-level parallelism obvious to the compiler.

[2] With two cores running at 2.5 GHz, each of which can start four instructions per cycle, my laptop has the potential to run *twenty billion* operations each second. And I use it to check my email.

[3]In essence, modern Pentium chips are hardware interpreters for the x86 instruction set — internally, they translate this into a RISC-like instruction set. There are even caches and optimizers for this micro-operations; in essence, the hardware is acting just like a Just-In-Time compiler for Java bytecodes would!

2.3 Short-vector extensions

A pipelined, superscalar architecture is good at simultaneously executing many *different* types of instructions (or at least different stages of instructions). A vector unit is a special piece of hardware that does the *same* operation to several different pieces of data. This type of parallelism is sometimes called SIMD: Single Instruction, Multiple Data. The old Cray machines were vector machines, but vector processing then waned in popularity for a time. Commodity machines started adding vector instructions about ten years ago because they are very useful for accelerating graphics and multimedia applications. The early Pentium chips had MMX (variously explained as "MultiMedia eXtensions" or "Matrix Math eXtensions," though Intel never gave an official expansion of the acronym); later Pentium chips included SSE (Streaming SIMD Extensions), and then SSE2. A Pentium with SSE2 instruction is capable of doing simultaneous arithmetic operations with 16 pairs of byte operands, 4 pairs of float operands, or 2 pairs of double operands. Some chips from IBM or from Motorola (including the G4 and G5 chips used in Macs before Apple switched to Intel) have similar SIMD instructions, called the Altivec instructions.

Modern GPUs (graphics processing units) take this sort of vector processing to a new level. We may talk about graphics processors more, later in the class.

2.4 The punchline

In principle, compilers understand the low-level architectural details of my machine much better than I do, and can do a good job of juggling around my code in order to take full use of the resources the chip provides. In practice, the compiler might need my help:

- I can set optimization flags, pragmas, etc to tell the compiler that I really do want the best performance rather than the most debuggable code, or to help the compiler understand that certain computations are independent even if it looks like they might not be.
- I can rearrange code in order to make instruction level parallelism and instruction reordering possibilities more obvious to the compiler.
- I can use special intrinsics or library routines to express what I want to do at a higher level than the language might normally permit.

- I can choose data layouts and algorithms that suit the machine.

3 Memory hierarchies and locality

Between faster clock rates and more parallelism, processor performance (as measured by benchmark codes) has doubled roughly every 18 months until recently. In contrast, memory *latency* (the time between asking for a piece of memory and when the first bit arrives) has doubled roughly every ten years. Memory *bandwidth* (the number of bytes of dat that memory delivers per unit time) has improved more quickly, though not as quickly as processor performance.

By now, fetching data from main memory costs around a hundred nanoseconds, while instructions can be started at a rate of several per nanosecond. Computation is now cheap; it's getting data that's expensive! Fortunately, in any short time period, most programs use only a small amount of data, often called the *working set*. This reuse is called *temporal locality*. Most programs also have *spatial locality*; that is, if the program reads or writes one memory word, it is likely to read or write nearby memory words around the same time.

Using locality, computer architects have a partial solution to the problem of feeding the execution stream in the face of slow memory. The solution is a *memory hierarchy* consisting of different "levels" of storage. Lower levels are fast but small, and higher levels are slow but large. Because of locality, most data a program uses can be kept in fast cache memory, with relatively few reads from the slow main memory.

3.1 Hierarchy levels

My laptop (an Intel Core 2 Duo T9300 processor) has the following levels of memory:

1. The register file, which operates at the same speed as instruction execution.

2. A 32 KB L1 data cache ("level 1" cache)[4]. While I haven't found the latency listed anywhere, I measure the time as about 4 ns.

[4]This machine also has a 32 KB L1 cache for code, called the execution trace cache. The execution trace cache actually stores "micro-operations" rather than the machine

3. A 6 MB L2 data cache, which has a latency of about 14 ns (again, according to my estimated measurements).
4. 2 GB of main memory (DRAM, or dynamic random access memory), which has a latency of about 50 ns.
5. 233 GB of disk, which has a latency of about 10 ms.

In addition, the system provides *virtual memory*, which means that there's a data structure that maps the addresses used by different programs to physical memory addresses. Part of this mapping is kept in a fast memory called a translation lookaside buffer (TLB). If a program uses address that isn't listed in the TLB, the system consults the full page table, which is stored in main memory. So on a TLB miss, getting data from memory can cost two or three ordinary memory accesses (or more).

3.2 Cache lines and spatial locality

The L1 data cache on my machine is organized into 64-byte *cache lines*. Whenever there is an L1 cache miss – that is, whenever the processor needs a piece of data and cannot find it in the L1 cache – it reads 64 bytes of data from the L2 cache, even if the processor only requested four or eight bytes. That way, when the processor needs the next four or eight bytes, which is likely to happen becuse of temporal locality, that data will already be waiting in the L1 cache. This organization also takes advantage of the relatively rapid improvements in bandwidth when compared to latency: moving an entire cache line at a time uses more bandwidth than just moving each byte as requested, but it saves the latency cost that would be required to make a separate request for every byte.

Because caches are organized this way, we can get the best performance by organizing programs to maximize spatial locality. For example, consider the following two versions of a loop to compute the centroid of $\{(x_i, y_i)\}_{i=1}^n$:

```
void centroid1(double* xy, int n, double* result)
{
    double x = 0, y = 0;
    int i;
```

instructions that are normally stored in memory — I think. At least, it stored micro-operations on my last desktop P4.

```
    for (i = 0; i < n; ++i) x += xy[2*i];
    for (i = 0; i < n; ++i) y += xy[2*i+1];
    result[0] = x/n;
    result[1] = y/n;
}

void centroid2(double* xy, int n, double* result)
{
    double x = 0, y = 0;
    int i;
    for (i = 0; i < n; ++i) {
        x += xy[2*i];
        y += xy[2*i+1];
    }
    result[0] = x/n;
    result[1] = y/n;
}
```

On a data set of five million points, I got the following timings (in nanoseconds) and effective processing rate (in MB/s) [5] on my old P4 desktop:

```
Ver 1: 49967165 (4.99672 ns/double; 1601.05 MB/s)
Ver 2: 26626221 (2.66262 ns/double; 3004.56 MB/s)
```

That is, the second version runs about twice as fast as the first version — which makes sense, because the first version loads each cache line the data set twice.

3.3 Cache eviction and temporal locality

When we read more data than can fit in a cache, we have to *evict* old data in order to make room for new data. The choice of which entry should be evicted is the *replacement policy*. A common policy is LRU: replace the entry that was *least recently used*. If a program has temporal locality — i.e., if data is likely to be used in bursts — then newer entries are more likely

[5] This is when I compiled using `gcc` with the optimization level `-O2`. At the highest level of optimization (`-O3`), the optimizer rearranged the calculation so that the sums were all done at the same time that I created the data array, which meant both `centroid` routines appeared to run about instantaneously.

to be used again soon, so evicting the oldest entry is natural. In practice, most implementations would use an approximation to LRU that requires only counters with a finite number of bits.

We can use a pure LRU replacement policy with an *associative* cache in which any piece of data can be stored in any location in the cache. However, fully associative caches cost a lot of hardware resources. An alternative that requires far less hardware is a *direct mapped* cache, in which any address is associated with a unique cache location. In a direct-mapped cache with c entries, address a goes in cache slot b if $a \equiv b \mod c$. So if we had a four-slot direct-mapped cache with entries 0, 1, 2, and 3, and we read address 200 followed by address 204, both would map to slot position 0. We would have to evict the data from address 200 to resolve the conflict.

Direct-mapped caches use less hardware than fully-associative caches; but they also restrict how well we can take advantage of temporal locality. A *set-associative* cache costs less than a fully-associative cache, but still lets us use temporal locality. In a set-associative cache, data from each address can go into one of a small number of cache slots. To extend our example above, suppose we had a 2-way set-associative cache with eight entries (and so four sets). Then if we read address 200 followed by address 204, both would go into set 0 without conflict. If we then read address 208, we would evict the data from address 200, since that data was least recently used.

Categorizing cache misses

It is useful to categorize cache misses into three basic types:

- *Compulsory* (or *cold start*) misses occur when we use a piece of data that we have not used before. These types of cache misses are basically unavoidable.
- *Capacity* misses occur when our working set is larger than the cache, so that we throw out old data that we will need again soon (and would do so even if the cache were fully associative).
- *Conflict* misses occur due to insufficient associativity in the cache.

There's not much we can do about compulsory misses, but we will see that we can often substantially improve the performance of our codes by rearranging them to minimize capacity and conflict misses.

A memory microbenchmark

The caches on my laptop are (I think)

- 32K L1 data and memory caches (per core)
  - 8-way set associative
  - 64-byte cache line
- 6 MB L2 cache (shared by both cores)
  - 16-way set associative
  - 64-byte cache line

I was not able to find information about the cache latencies.

We can see a surprising amount of the structure of the memory hierarchy using a memory microbenchmark (`membench`) that times regulare read/write memory patterns. The inner loop that we are timing looks like this:

```
for (i = 0; i < len; i += stride)
  x[i]++;
```

where `x` is a 32-bit integer. For `stride = 4`, for example, the loop goes through a buffer of length `len` and increments every fourth four-byte integer.

I ran `membench` on my laptop, and in Figure 1, I show the result for different size buffers. The picture is complicated, but we can make a few observations from what we know about the memory subsystem:

- For small strides, the combination of cache line reuse and prefetching keeps the processor pretty well fed, so that the memory overheads are relatively low. For larger strides, the cost starts to go up. When the stride is large enough, every data item we visit fits in L1 cache, and costs are again low. This explains the basic "hump" shape of the curves.
- In Figure 1, there is a "knee" at a stride of 64 bytes for the small data sets. This corresponds to where we go from a stride smaller than the cache line size to a stride longer than a cache line.[6]

[6]The knee is not all that sharp — I think automatic prefetching might be coming into play.

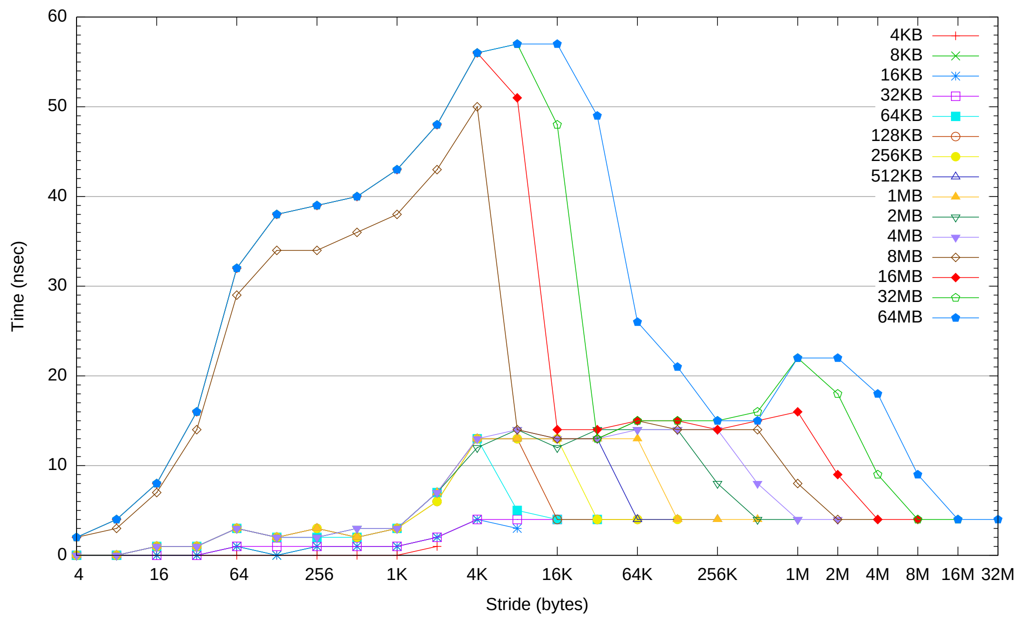

Figure 1: `membench` results on an Intel Core 2 Duo T9300.

- When the data set is smaller than the L2 cache size (6MB), the worst update time is about 14 ns. Once the data set reaches the size of the L2 cache and the program starts to miss the cache and go to main memory, average access times get much worse.
- When all the data fits in L1 cache, the update time is about 4 ns (right side). We can see that the L1 cache is 8-way set associative by looking at when we get to that 4 ns update time, because long power-of-two strides virtually guarantee conflict misses. For 64 MB of data, for example, we only get a 4 ns update when the stride is 8 MB.
- The worst case time occurs starting at a stride of about 4 KB. This is because pages are about 4 KB. Notice that we only see this worst-case behavior when the total amount of data is 8 MB — which fits, since the TLB on this machine has 256 entries.

One moral of this exercise is that even for simple programs, performance is a very complicated function of the architecture! We need to understand at least a little about the relevant details in order to write fast programs, but we would prefer not to think about such things all the time. Instead, we would like simple models that help us understand efficiency, and some common tricks based on those models that can help us design fast codes. We will introduce one such trick, blocking, next time.