Exercise 06

VU Performance-oriented Computing, Summer Semester 2024

Calvin Hoy

2024-04-30

(A) MMUL Tiling

The primary computation loop is this:

```
// conduct multiplication
for (int i = 0; i < N; i++) {
  for (int j = 0; j < K; j++) {
    TYPE sum = 0;
    for (int k = 0; k < M; k++) {
      sum += A[i][k] * B[k][j];
    }
    C[i][j] = sum;
  }
}
```

We can see that the indices `i` and `k` are both used for the leftmost matrix index, having the greatest reuse distance. `j` meanwhile, is only used for the rightmost index, meaning the data it indexes lie consecutive in memory. I therefore chose to tile the `i` and `k` loops as follows (see also `mmul_tiled/mmul.c`):

```
// conduct multiplication
for (int i0 = 0; i0 < N; i0 += TI) {
  int imax = i0 + TI;
  for (int k0 = 0; k0 < M; k0 += TK) {
    int kmax = k0 + TK;
    for (int i = i0; i < imax; i++) {
      for (int j = 0; j < K; j++) {
        TYPE sum = 0;

        for (int k = k0; k < kmax; k++) {
          sum += A[i][k] * B[k][j];
        }

        if (k0 == 0)
          C[i][j] = sum;
        else
          C[i][j] += sum;
      }
    }
  }
}
```

As we do not want to lose intermediate sums for `k` between iterations of `i` and `j`, we need to conditionally either initialize `C[i][j]` with the intermediate sum, or add to it. Alternatively, one could initialize the result matrix `C` to 0 and omit the conditional assignment/summation. I did not conduct any tests to see whether this is faster.

There are two tiling parameters we can adjust: TI and TK.

Performance evaluation

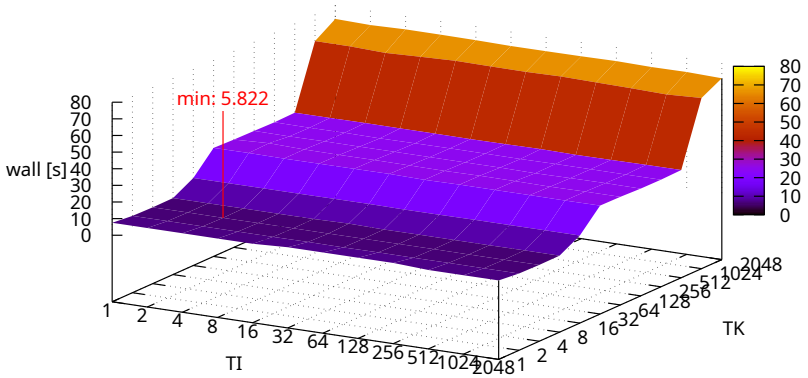

Figure 1: Effect of TI and TK on `mmul` wall time

The minimum runtime is 5.822 seconds with TI=4, TK=4. The worst runtime is 70.092 seconds with TI=4, TK=2048. The effectively untiled version of the code – TI=2048 and TK=2048 – has a runtime of 69.044 seconds, so tiling improves the runtime by a factor of 11.86.

TK=4 seem to be optimal in general, while TI does not have much of an impact at all, as seen in fig. 1.

(B) Cache investigation

This approach is informed by several Stackoverflow posts which I forgot to note down.

The basic idea is this: Create a workload that is not bound by memory bandwidth but IS bound by memory latency, and is not parallelisable. A linked list is perfect for this, as in order to find a particular node, we must first find all its predecessors. The linked list does not have to contain any data, only a pointer to its successor node, so the memory used per node is minimal

1. Create a linked list spanning a memory block of a given size.
2. Record the current time (in nanoseconds).
3. Follow the linked list for a given number of iterations.
4. Record the time again and calculate the difference.
5. Divide the difference by the number of indirections to get the (average) memory latency.

There are several problems to be solved in order to get a good estimation of cache/memory access latency:

Background noise

Our benchmark runs under a multitasking operating system, so it will periodically be switched out for other programs. To get as accurate a measurement as possible, we perform a large number of list accesses and take multiple samples per block size, averaging over both.

Loop overheads

We follow the linked list in a loop. This carries certain overheads, namely incrementing the iteration variable and checking if the loop condition still holds. To minimise this overhead and ensure that what we measure is as close as possible to raw access latency, we manually unroll this loop a large number of times.

Prefetching

Most modern cache architectures perform some sort of memory prefetching. This may be as simple as loading the next block/next few blocks of memory in case they are required, it may adjust to common memory access patterns such as strided access, or even more complex ones.

To hopefully counteract any sort of prefetching, we connect the linked list in a pseudo-random fashion by shuffling their storage locations within the memory block.

Cache line length

Most modern CPUs have a cache line length of 64 bytes, while each linked list node only needs to store a single pointer. In the case of x86_64, a pointer takes up 8 bytes, meaning that as many as 8 nodes could fit in a single cache line.

Thus, when randomly shuffling the list contents, some yet-unvisited nodes may be present in a still-loaded cacheline, allowing them to be accessed faster than otherwise. Since this would impact the accuracy of our measurements, we pad each node to ensure that it occupies an entire cache line.

(C) Cache benchmark

See `latency/latency.c` for an implementation of the basic idea and mitigation strategies mentioned above. There are, additionally, some other important implementation details:

- Besides the manual loop unrolling mentioned above, I instruct GCC to additionally perform more aggressive loop unrolling than it otherwise would by means of `#pragma GCC optimize("unroll-loops")`.
- To ensure that GCC will under no circumstances optimise out the test as a consequence of (correctly) recognising that the result of following the list is ultimately unused, I employ `__attribute__((used))` on the linked list pointer used for benchmarking,
- Finally, to allow performing the same number of list indirections for all block sizes, I set the linked list's start node to be the final node's successor.
- Between each power-of-2 step, I measure 64 equally spaced block sizes in order to produce a more complete graph.

See fig. 2 for the results produced by `latency.c` on LCC3, compiled with `gcc -O3 -march=native`. Note the steep increases around 32k and 12M, which line up with the 32k per-core L1 data cache and 12MiB shared L3 cache. The measurements around 256k (corresponding to the size of per-core L2 cache) seem somewhat anomalous, rising not quite as steeply and also earlier than I would expect.

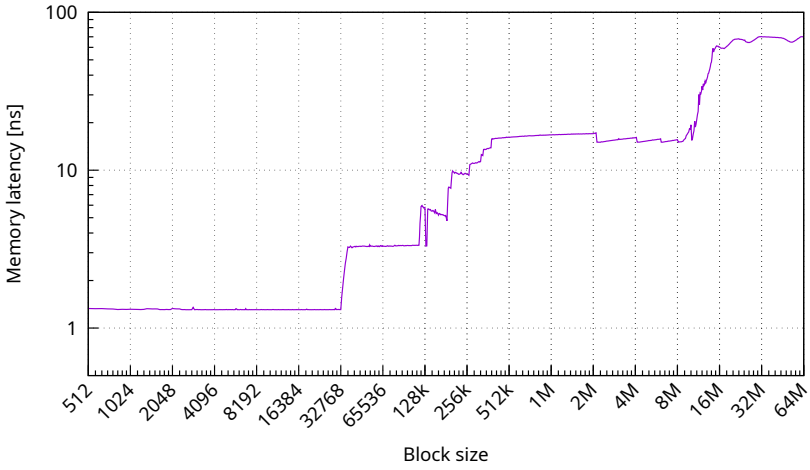

Figure 2: Memory latency vs block size on LCC3 (Xeon X5650)