Exercise session for lecture 4

In this exercise session, we will look into the memory hierarchy, by investigating array accesses. To do this, we will run the example code on Learnit and use the tracing tool perf to give us further insight.

Array accesses

We are going to run with the fairly basic example below.

File: caching.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int horizontal(u_int64_t size, u_int64_t *mem) {
  int x = 0;
  for (int rep = 0; rep < 1000; rep++) {
    for (u_int64_t i = 0; i < size; i++) {
      for (u_int64_t j = 0; j < size; j++) {
        mem[i * size + j] == 0 ? x++ : x--;
      }
    }
  }
  return x;
}

int vertical(u_int64_t size, u_int64_t *mem) {
  int x = 0;
  for (int rep = 0; rep < 1000; rep++) {
    for (u_int64_t i = 0; i < size; i++) {
      for (u_int64_t j = 0; j < size; j++) {
        mem[j * size + i] == 0 ? x++ : x--;
      }
    }
  }
  return x;
}

int main() {

  u_int64_t size = 10;
  u_int64_t max = 2000;

  printf("Testing up to array of size %lu\n", max * 2);

  clock_t start, end;
  double cpu_time_used;

```

```
40   int aux = 0;
41
42   for (u_int64_t c = size; c <= max; c += 250) {
43     printf("Testing for array of size %lu * %lu = %lu\n", c, c, c * c);
44     u_int64_t *arr = malloc(c * c * 8);
45
46     start = clock();
47     aux += horizontal(c, arr);
48     end = clock();
49     cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
50     printf("Horisontal took %f cpu milliseconds\n", cpu_time_used);
51
52     start = clock();
53     aux += vertical(c, arr);
54     end = clock();
55     cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
56     printf("Vertical took %f cpu milliseconds\n\n", cpu_time_used);
57   }
58
59   printf("%d", aux);
60 }
```

It allocates memory and then accesses that in two different patterns, horizontally and vertically, measuring the time it takes for both to finish. Those patterns are visualized below.

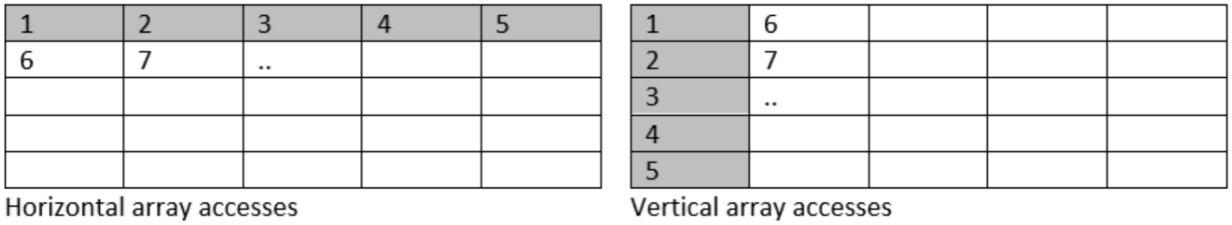

Figure 1: Horizontal vs. vertical array access

Use the provided Makefile to compile the program. This will produce multiple outputs. Run the caching-0 program and ignore the other output files for now. Determine whether horizontal or vertical array accesses are faster. Why is one faster than the other? Is it always faster?

The perf tracing tool

To better understand why either pattern is faster, we can use perf. The perf tool gives us access to certain performance counters, such as number of CPU cycles, CPU migrations, cache misses etc. To see a more extensive list of performance commands run the following command
`$ perf list`

If you have access to another Linux environment, e.g. on your laptop, try and compare what perf reports on the server and your own environment. Is there are difference?

Perf can be started by running the following:
`$ perf stat -e <events...> <command>`

More examples of how to use perf can be found here.

In order to reason about the array access patterns, you can the following command:
`$ perf stat -e cycles,instructions,L1-icache-load-misses,L1-dcache-load-misses, LLC-load-misses,cache-misses,stalled-cycles-frontend,stalled-cycles-backend, branch-misses,iTLB-load-misses,dTLB-load-misses ./horizontal-0`

To compare the patterns, the caching.c file has also been split into the two files, horizontal.c and vertical.c. To compare the performance, you can run the above perf command on horizontal-0 and vertical-0 and leave the rest of the files for now.

The given perf command gives an extensive output of performance metrics. Which metrics seems to have an impact on performance? Why is that?

Optimization

The provided Makefile compiles two versions of all the program. One is compiled with the gcc flag -O0 and one is compiled with the flag -O3. When compiling C programs, you can give an optimization flag for GCC (e.g. gcc -O0, gcc -O3) to set the degree of the compiler optimizations of the compiler in regards to optimization. -O0 will not optimize anything, while -O3 will optimize everything possible.

With perf, compare the results of running the optimized with results gathered from the unoptimized. What is the major differences?

Manual optimizations

By having the compiler optimize for us, it can become difficult to understand exactly what transformations the original program has undergone. One can naturally compile the program with the specified degree of optimization and compile it to assembly by compiling with `-fverbose-asm` and analyze the code at assembly level. This is however difficult and time consuming.

Therefore, it is often worthwhile to implement manual optimizations, such as loop unrolling[1]. Loop unrolling is a program transformation that aims to reduce the number of iterations for a loop by increasing the number of elements computed at each iteration. Consider the simple example of adding elements of a vector that are all set to 1, meaning that the sum should equal the size of the array:

[1]For more thorough information, see Bryant, R. E., David Richard, O. H., & David Richard, O. H. (2003).

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int start_size = 1000;
    long n = 1000000000;
    int64_t sum = 0;
    int *arr = NULL;
    clock_t start, end;

    for (int size = start_size; size <= n; size *= 10) {
        arr = (int*) malloc (size * sizeof(int));

        for (int i = 0; i < size; i++) {
            arr[i] = 1;
        }

        start = clock();
        for (int i = 0; i < size; i++)
        {
            sum += arr[i];
        }
        end = clock();
        double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000;

        printf("Size %d - The sum is %lld and took time %f ms\n", size, sum, time_taken);

        free(arr);
    }

    return 0;
}
```

In 15 we see that the variable i is incremented by 1 in each iteration. This means that we will perform `n` iterations in order to compute the sum. Instead, we can decrease the amount of iterations by iterating by a fixed amount n and indexing into the array. Such an example is as such:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int start_size = 1000;
    long n = 1000000000;
    int64_t sum = 0;
    int *arr = NULL;
    clock_t start, end;

    for (int size = start_size; size <= n; size *= 10) {
        arr = (int*) malloc (size * sizeof(int));

        for (int i = 0; i < size; i++) {
            arr[i] = 1;
        }

        start = clock();
        for (int i = 0; i < size; i+=2) {
            sum += arr[i];
            sum += arr[i+1];
        }
        end = clock();
        double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000;

        printf("Size %d - The sum is %lld and took time %f ms\n", size, sum, time_taken);

        free(arr);
    }

    return 0;
}
```

In reality, there is nothing holding us back from unrolling even further. The idea is to simply increment i by n and then index n times into the array. For example, we could change lines 16-20 as such:

```c
    for (int i = 0; i < n; i += 100)
    {
    sum += arr[i];
    sum += arr[i + 1];
    sum += arr[i + 2];
    sum += arr[i + 3];
    sum += arr[i + 4];
    sum += arr[i + 5];
    sum += arr[i + 6];
    .
    .
    .
    }
```

On LearnIT you will find the `Makefile` to compile all of the programs. Each program will be compiled with the compiler flags `-O0` and `-O3`, including the loop unrolling example. Try to play around with the programs to see the differences. Some of the programs may execute very quickly, so try also to reason about which `perf` metrics that could prove useful to profile for. Furthermore, try also to consider some of the drawbacks of loop unrolling. Does it always increase performance? Is it always better to just rely on the compiler and use `-O3`?