Hardware Aware Scientifc Computing (HASC) - Exercise 01

Manuel Trageser
Justin Sostmann

Exercise 1 *Pointer Chasing*

a)

- What happens in the main experiment?

The main experiment runs a loop `s` times, which is the current stride. Inside that we run a while loop where we access the array at the current index `i`, starting at `i=0` and then set the index `i` to the value at that position. This is repeated until the index is `0` again.
Essentially, we follow pointers in the array, starting at `0` and ending at `0` again, but with a stride of `s`.

- What is the purpose of the empty experiment and why do we measure its time?

The purpose of the empty experiment is to measure the overhead of the for/while loops themselves, such that we can subtract it from the main experiment and get the actual cache access times.

- Why do we include the loop `for (int k=0; k<s; k++)`?

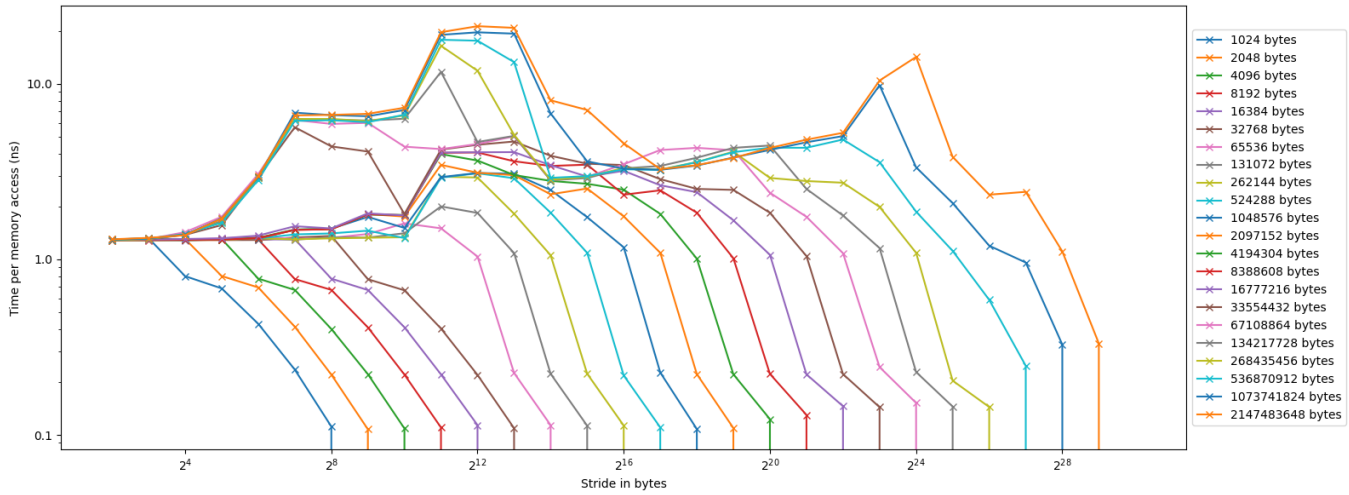
The higher our stride `s` the less array elements we access, so we run the experiment `s` times to overall access the same amount of elements for each stride.

b)

√

c)

`-std=c++17 -O1 -funroll-loops -fargument-noalias`



d)

```
Model name:            AMD Ryzen 5 5600X 6-Core Processor
  Base Clock: 3.7GHz
  Boost Clock: 4.6GHz
  Thread(s) per core:  2
  Core(s) per socket:  6
  Socket(s):           1
Caches (sum of all):
  L1d:                   192 KiB (6 instances)  32 KiB per core
  L1i:                   192 KiB (6 instances)  32 KiB per core
  L2:                    3 MiB (6 instances)  512 KiB per core
  L3:                    32 MiB (1 instance)
```

Values below 1:

We have less cache misses than accesses, because we access the same elements multiple times, which are then cached.

Stride 1:

Only every 16th access is a cache miss, because the cache line size is 64 bytes and we access 4 bytes per element, so we access 16 elements per cache line. Cache prefetching loads the next cache line into the cache before we access it, so we have no additional cache misses. -> Constant access times, independant of array size

???:

Why does the latency increase for strides above 64bytes (cache line size)? At a stride of 64 bytes we should have the maximum amount of cache misses/loads. The higher the stride the less elements are loaded in total, which should result more cached elements -> less cache misses -> lower latency, but this obviously is not the case.

Exercise 2 *Peak Performance*

a)

Transferred Memory:
Local AIDA64 Extreme Memeory Read Benchmark

```
=> 42765 MB/s
Memory: 1471.2 MHz
CPU Clock: 4560.8 MHz
```

Online results Source: `50972 MB/s`

Operation Throughput:
Online results:
408GFLOP/s Geekbench
891.9GFLOP/s AIDA64

b)

Theoretical peak performance:

$$FLOP/s = \text{cores} * \frac{\text{cycles}}{\text{second}} * \frac{\text{FLOPs}}{\text{cycle}}$$

6 (number cores) * 4.6 GHz (clock frequency turbo) * 32 (AVX2 & FMA (256-bit)) = 883.2 GFLOP/s Singlecore: 147.2 GFLOP/s

Source

Theoretical peak memory bandwidth:

$$GB/s = \text{channels} * \text{memory frequency} * \text{number of bytes of width}$$

2 channels
DDR4 3000MHz -> memory frequency = 1500MHz
64 bit width
$2 * (1500MHz * 2) * 8bytes = 48000MB/s = 48GB/s$

Exercise 3 *Vectorized Numerical Integration*

a)

See midpoint.cc for the implementation.

b)

See midpoint.cc for the implementation.

d)

For function 2 our vectorized version achieves a speedup of approximately 2x the speed of the normal version (depending on N). We did not get a speedup for function 1. Vectorizing this function makes it slower in our case (roughly 2-3 times slower).

e)

FLOPs per function:

```
float func1(float x) {
    return powf(x, 3)           // 2 FLOPs
           -                    // 1 FLOP
           (2 * powf(x, 2))     // 2 FLOPs
           +                    // 1 FLOP
           (3 * x) - 1;         // 2 FLOPs
    // 8 FLOPs total
}
```

```
float func2(float x) {
    float sum = 0;
    for (int i = 0; i <= 15; ++i) {   // 16 FLOPs because of sum addition
        sum += powf(x, i);           // 105 FLOPs for powf
    }
    return sum;
    // 121 FLOPs total
}
```

```
float midpoint(int a, int b) const {
        float const step_size =
            (float((b - a)) / _n);  // 2 FLOPs
        float sum = 0;
        auto x_i = float(a);
        for (int i = 0; i < _n; ++i) {
            float const midpoint = x_i + (step_size / 2); // 2 FLOPs
            sum += _func(midpoint); // 1 FLOP
            x_i += step_size; // 1 FLOP
        } // N * 4 FLOPs
        return step_size * sum; // 1 FLOP
        // 3 + N * (func+4) FLOPs total
    }
```

N = 8

Function	FLOPs	Time (ns)	GFLOPs/s
1	99	53.2311	1.859
1 vectorized	99	143.02	0.692
2	1003	569.706	1.760
2 vectorized	1003	301.002	3.332

N = 64

Function	FLOPs	Time (ns)	GFLOPs/s
1	771	402.032	1.917
1 vectorized	771	1132.22	0.680
2	8003	4444.24	1.800
2 vectorized	8003	2465.64	3.245