NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Programming Techniques for Supercomputers Tutorial

Erlangen National High Performance Computing Center

Department of Computer Science

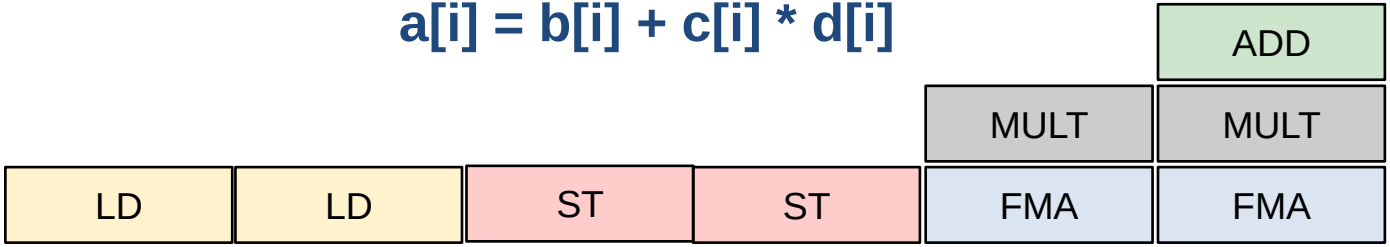
FAU Erlangen-Nürnberg

Sommersemester 2024

Assignment 3 – Task 1

Schönauer triad:

a[i] = b[i] + c[i] * d[i]



Intel Ice Lake – scalar code: 32/64-Bit instruction

Peak Floating Point (FP) Performance:

$$P_{core} = n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$$

Super-scalarity

FMA factor

SIMD factor

Task 1

Code has 3 LOAD + 1 STORE + 1 FMA + loop mechanics
Performance Limits: Bottleneck is LOAD as 3LOADS→1.5cycle
Performance: P

16 flops / 1.5 cy = 10.7 flop/cy x 2.1 Gcy/sec = 22.4 Gflop/s

Peak Performance : Pmax

```
32 flop/cy x 2.1 Gcy/sec = 67.2 Gflop/s
```

Performance ratio: P/Pmax

```
22.4Gflop/sec / 67.2 Gflop/sec = 0.33 Pmax
```

AVX2: 1.5 cy for 4 (scalar) iterations → Pmax=8 flops / 1.5 cy * 2.1 Gcy/s = **11.2 Gflop/s (or 5.33 flop/cy)**

Scalar: 2 flops / 1.5 cy (2.8 Gflop/s)

Assignment 3 – Task 2

(a) **(5 credits)** Assuming that the compiler applies no unrolling and no SIMD vectorization, calculate the maximum possible performance of the loop in Flops/cycle if the data comes from the L1 cache.

Per iteration: 2 LD, 1 ADD, 1 MULT → ADD is the bottleneck with latency 4

Pmax = 2Flop /4 cycle = 0.5 Flop/cycle

(b) **(15 credits)** Calculate the performance to expect at N=8.

Overall pipeline latency (time to first result):

L = 5+1 cy (LD) + 4 cy (MULT) + 4 cy (ADD) = 14 cy

Task 2

Performance at loop length N: Startup latency + (N-1) times the cycles in throughput mode

$$P = \frac{N}{T(N)} = \frac{N}{L + 4(N-1)} = \frac{8}{42}\, it/cy \rightarrow 0.38\, flop/cy$$

(c) **(10 credits)** Now assume that the compiler can SIMD-vectorize the loop with AVX, but does no unrolling beyond that. Also assume that adding up the four slots ("horizontal add") of an AVX register takes eight additional cycles. Calculate how this changes the predictions from (a) and (b). How would the numbers change for 8-wide SIMD units (64-byte registers)? (Horizontal add takes 12 cycles in this case)

Assuming 4-way SIMD capability, with no unrolling:
Steady-state **Pmax** achieved with 2 LD, 1 ADD, 1 MULT per SIMD iteration
→ 4 cy per SIMD iteration → speedup of 4x w.r.t. scalar code → **1.52 flop/cy**

Task 2

Performance with N=8: Consider modified overall pipeline latency of
L = (14+8) cy = 22 cy
The number of AVX (i.e., assembly) iterations is 4x smaller than N.

$$\frac{N/4}{T(N/4)} = \frac{N/4}{L + 4(N/4 - 1)} = 0.067\,AVX/cy \rightarrow 0.53\,Flop/cy$$

8-wide SIMD units → L = (14+12) cy = 26 cy, and that is it because the work is only 8 iterations → 16 flops / 26 cy = **0.61 flop/cy**
Note: We assume that the compiler generates a proper loop with an ADD instruction. If the compiler knows the loop length, it will probably do the horizontal add right after the MULT, leading to 22 cy)

Task 2

(d) **(15 credits)** Now assume that compiler performs a 2-way modulo unrolling of the loop, on top of AVX. Calculate the maximum expected performance with AVX if the data comes from the L1 cache. Would 4-way unrolling change anything? Why (not)?

2-way MVE on top of 4-way SIMD:
→ ADD limitation at 4 cy per 2 AVX iterations (2 out of 4 ADD stages can be filled)
→ However, 2 iterations need 4 loads → LOAD throughput limit also at 4 cy per 2 AVX iterations
→ The bottleneck is now ADD and LOAD → P = 16 flops/4 cy = 4 flop/cy
Additional MVE unrolling would change nothing because we are already at the LOAD limit

Task 2

(e) Comparing the non-unrolled code from (a) with the AVX + 2-way MVE variant from (d), how many instructions (as a function of N) must be executed by the core to traverse the whole loop? Assume that the "loop mechanics" (handling the iteration count) always needs 3 instructions (add, compare, branch).

Variant from (a): 2LOAD + 1ADD + 1MUL + 3 with N iterations

4 + 3 instructions per assembly iteration → 7 N instructions overall

Variant from (d):

4LOAD + 2ADD + 2MUL + 3 instructions per assembly iteration → 11 N_{asm}

Only N/8 assembly iterations → (2*4+3)*N/8+1 instructions overall, i.e., 1.375 N +1

→ More than a factor of 4 fewer instructions because of less overhead from loop mechanics

Assignment 3 – Task 3

Write a benchmark code for the double-precision "vector update" kernel and modify it so that only each Mth element is used. Fix the clock frequency. What happens if you increase M even further?

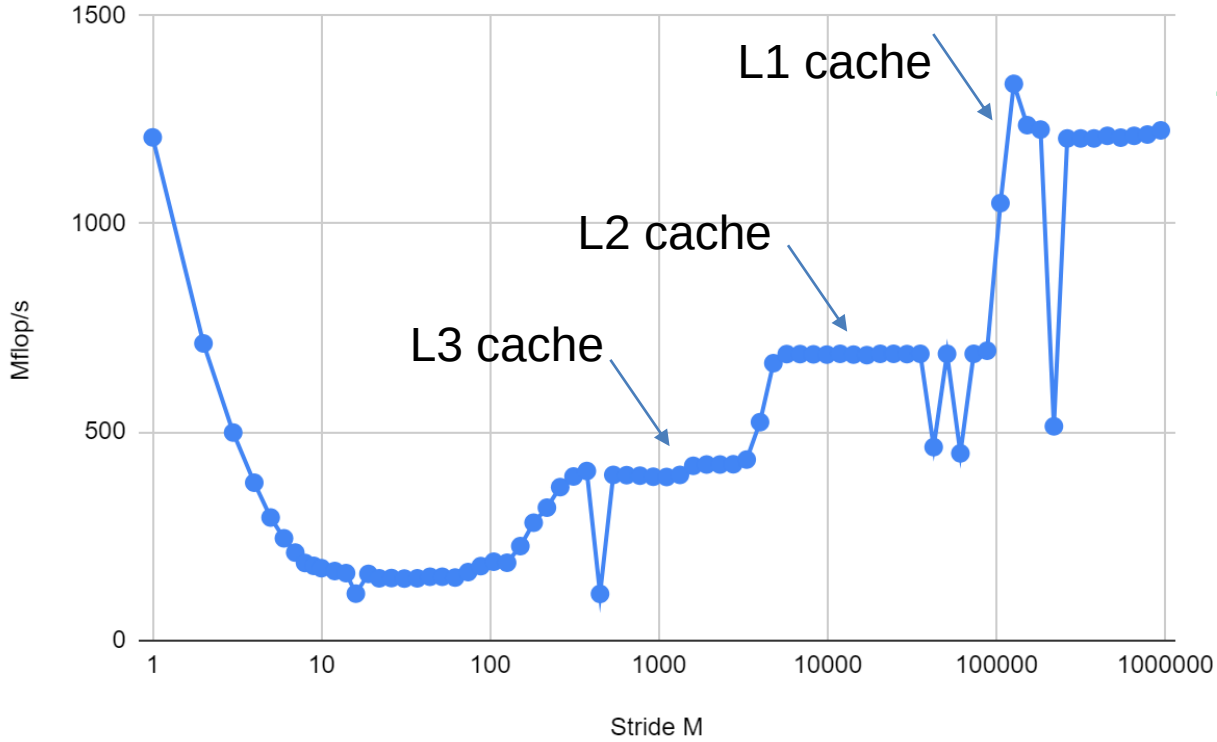
for(i=0; i<N; i+=M) a[i] = s * a[i];

Explain the change in performance with growing M.

Choose strides of $M=8*1.2^n$, with n a positive integer and $M<=10^6$. Explain why this behavior is expected.

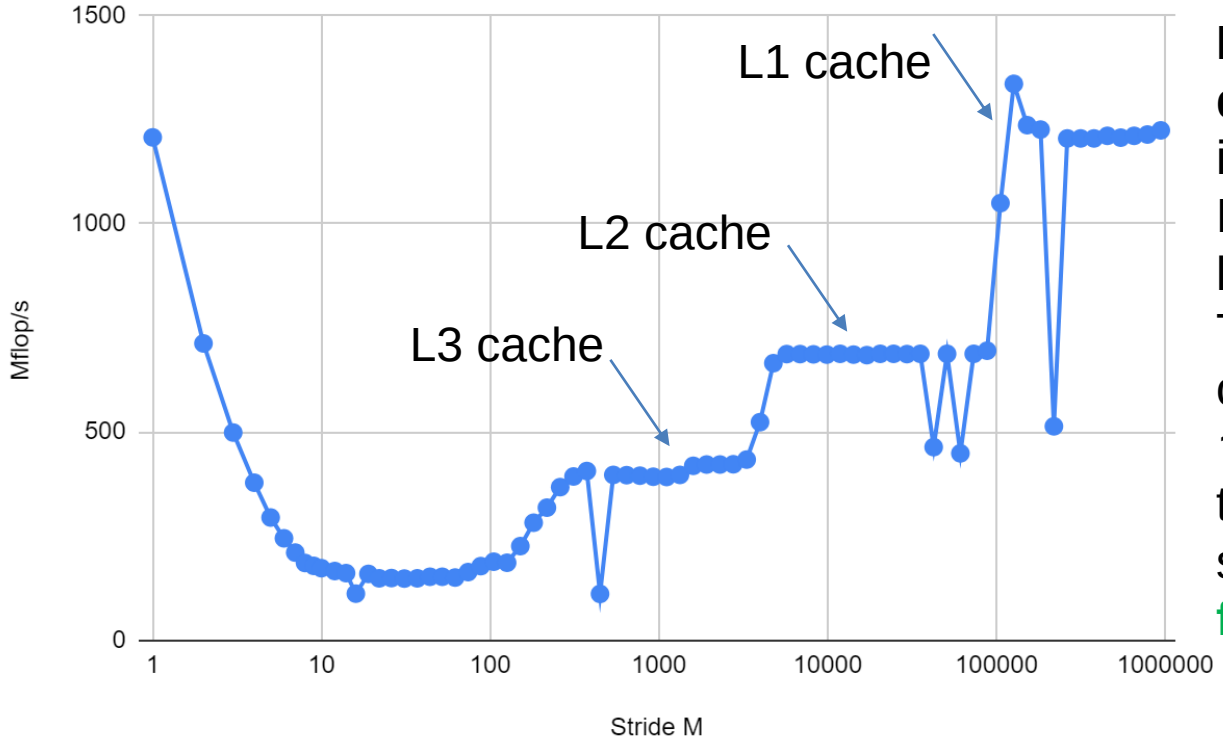
```c
void vector_update(double* a, double s, int N, int M) {
    #pragma omp parallel for
    for (int i = 0; i < N; i += M) {
        a[i] = s * a[i];
    }
}

int main() {
    int N = 100000000; // Size of the vector (10^8)
    double s = 2.0; // Scalar multiplier
    double* a = (double*)malloc(N * sizeof(double));
    for (int i = 0; i < N; ++i) {
        a[i] = 1.0;
    }

    for (int n = 0; ; ++n) {
        int M = (int)(8 * pow(1.2, n));
        if (M > 1000000) break;

        clock_t start = clock();
        vector_update(a, s, N, M);
        clock_t end = clock();
        double elapsed = (double)(end - start) / CLOCKS_PER_SEC;

        printf("Time taken for vector update with M = %d: %f seconds\n", M, elapsed);
    }

    free(a);
    return 0;
}
```

Benchmark code



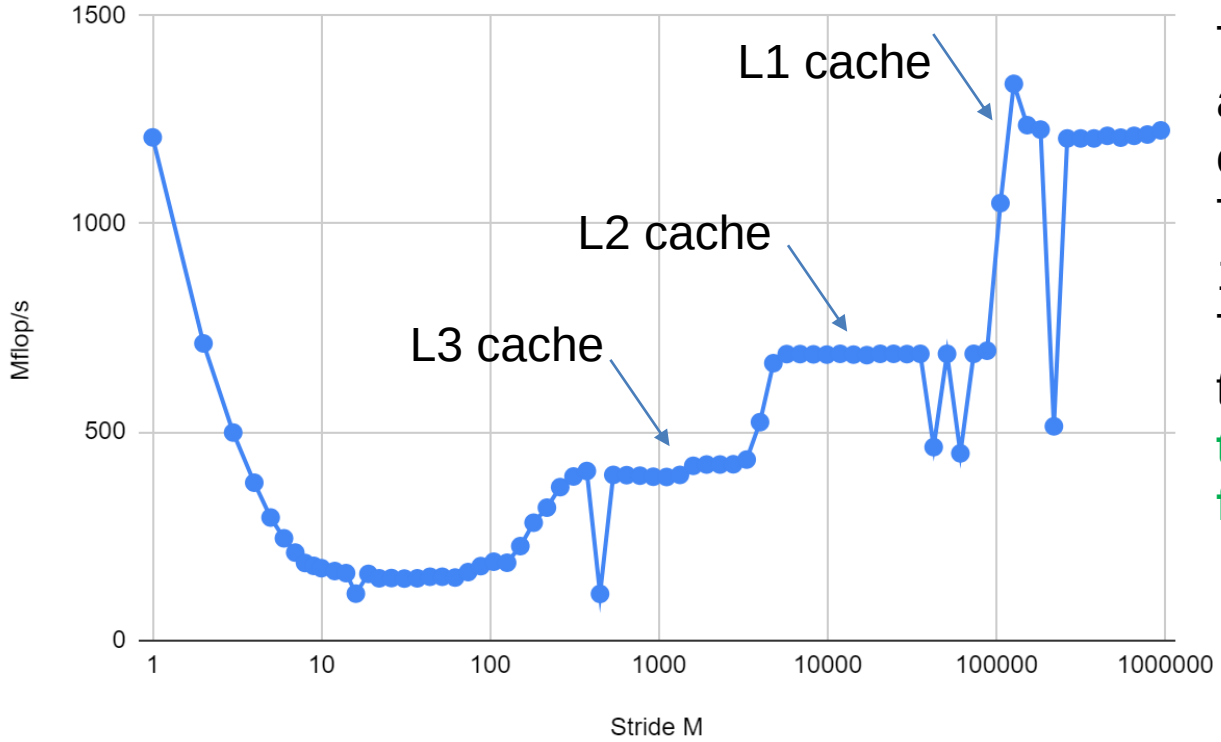
Performance drops by a factor of about 1/M initially because only every Mth element is used but all the cache lines still have to be loaded Beyond M=8 there is no strong change any more at first.

Benchmark code



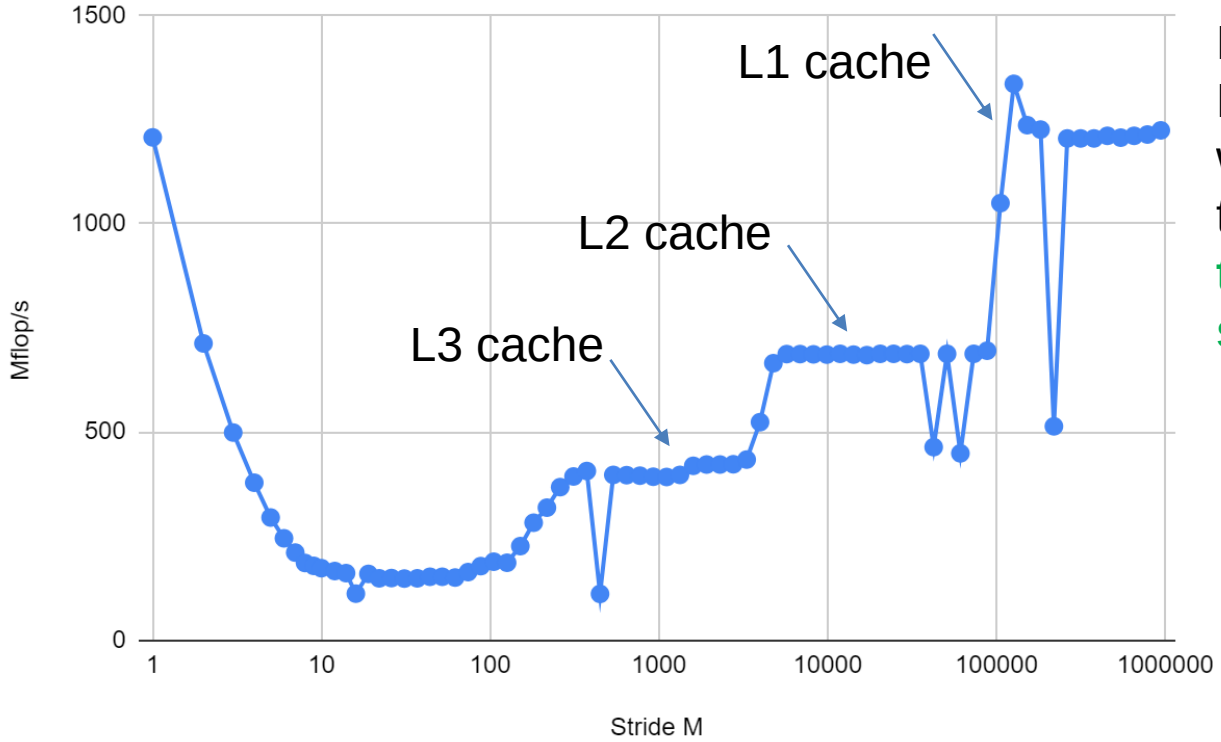
Behavior for larger M (blue dots): Starting at M=300 there is a **rise** in performance. At M=300, only every 19th CL is loaded (adjacent CL prefetch!). This is a working set of 10^8 (elements) x 8 bytes / 19 ≈ 42 MB. The L3 cache of the Ice Lake CPU is 54 MiB, so the **complete working set fits into the L3** at this point.

Benchmark code



The **second sharp rise** is at about M=4000. Here, only every 500th CL is loaded. The working set is 8 x 10^8 bytes / 500 ≈ 1.6 MB. The L2 size is 1.25 MiB, so this is near the point where **the complete working set fits into the L2 cache**

Benchmark code



Finally, the last sharp rise at M=100000 indicates a working set of 64 kB, while the L1 size is 48 KiB, hence **this is where the working set fits into the L1 cache**

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Do let me know if you have Questions.

Thank you.