Programming Techniques for Supercomputers
Assignment – 1

Boran Gündogan – Mehmet Arif Bagci – Nedimcan Aytemur

Task1)

The peak floating point performance of one core can be calculated by following formula:

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

Where:

n_{super}^{FP} = superscalarity (instruction/cycle) → given as 2 in the question.

n_{FMA} = FMA factor → FMA is given available on the question, so it is 2.

f = clock speed (Gcycle/seconds) → given as 2(GHz) in the question.

n_{SIMD}= SIMD factor(lanes/instruction) → we should calculate explicitly according to the information on the question.

n_{SIMD} can be calculated as:

$$n_{SIMD} = \frac{Vector\ Width\ (bit)}{Width\ of\ a\ FP\ operation\ (bit)}$$

Vector width = 512-bit wide SIMD registers

Double Precision = 64 bit

$$n_{SIMD} = \frac{512\ bit}{64\ bit} = 8bit$$

Right now we can calculate the the P_{core}:

$$P_{core} = 2 \, x \, 2 \, x \, 8 \, x \, 2 = 64 \text{ GFlop/s}$$

In order to calculate the peak floating point performance of the chip, we should multiply the P_{core} with number of cores in the chip.

$$P_{chip} = P_{core} \, x \, N(number\ of\ cores)$$

$$P_{chip} = 64\ GFlop/s \, x \, 52 = 3328 \text{ GFlop/s}$$

This is equal to **3.328TFlop/s**.

Task 2)

a) The primary performance bottleneck in the loop arises from the accumulation operation $s = s + a[i] * a[i]$. Due to the data dependency on the scalar variable s, each new addition requires the result of the previous one, creating a loop-carried dependency that stalls the pipeline. As a result, the floating-point ADD instruction becomes the limiting factor. Since each iteration performs 3 floating-point operations (2 multiplications and 1 addition), and the latency of the ADD pipeline is 5 cycles, the achievable performance is:

$$Performance = \frac{FLOPs}{Cycles} = \frac{3}{5} = 0.6 \frac{FLOPs}{cycle}$$

b) In order to achieve optimal performance, Modulo Variable Expansion (MVE) must be applied with a minimum unroll factor of **5**. The reason of that is, the floating floating-point ADD pipeline has a **5-cycle latency.** In order to fully hide this latency and eliminate the loop-carried dependency on s, the loop must be unrolled by at least the depth of the ADD pipeline (5 iterations). This allows the use of **5 independent accumulator variables** (e.g., s0, s1, ..., s4) to break the dependency chain, enabling parallel execution of ADDs.

- After applying MVE, the bottleneck shifts to the **floating-point MULT throughput**.

1. **Operations per iteration**:
   - a[i] * a[i] → 1 MULT
   - s += ... → 1 ADD
   - b[i] *= t → 1 MULT

   Total: 2 MULTs + 1 ADD = **3 FLOPs/iteration**.
2. **Throughput Constraints**:
   - The CPU can issue **1 MULT per cycle** (no FMA support).
   - Each iteration requires **2 MULT operations**, which take **2 cycles** to complete (due to the 1 MULT/cycle throughput limit).
   - ADDs and LOAD/STORE operations can overlap with MULTs but do not contribute to the bottleneck.

Performance Calculation

- **FLOPs per Cycle**:
  - Total FLOPs per iteration: **3 FLOPs**
  - Cycles per iteration: **2 cycles** (limited by MULT throughput).
  - $Performance = \frac{3FLOPs}{2cycles} = 1.5\, FLOPs/cycle$

As a result, the minimum Modulo Variable Expansion (MVE) unrolling factor is 5. The hardware bottleneck is the multiplication unit, as each iteration requires two multiplications but only one multiplication port is available. Consequently, the optimal achievable performance is 1.5 FLOPs per cycle.

Programming Techniques for Supercomputers
Assignment – 1

Boran Gündogan – Mehmet Arif Bagci – Nedimcan Aytemur

Task3) For this task, a source file named main.cpp has been created.

The code is compiled using the following command:

• icpx -O3 -march=native -std=c++17 main.cpp -o benchmark.out

The reason of compiling with this command is -fno and -alias is not supported on icpx, so we used the equivalent command for this.

After compilation, a compute node on Fritz is allocated, and the experiment is executed using the commands below:

• salloc --nodes=1 --time=01:00:00

• srun --cpu-freq=2200000-2200000:performance ./benchmark.out

Within the main.cpp file, two CSV files are generated: daxpy_results.csv for the DAXPY benchmark and sch_tri_results.csv for the Schönauer Triad benchmark.

Additionally, a Python script named plot_benchmarks.py has been developed to visualize the data from the CSV files. This python file is compiled and run on local machine. The output of this script is a PNG image named benchmark_comparison.png.

Figure 1 illustrates the benchmark_comparison.png file.

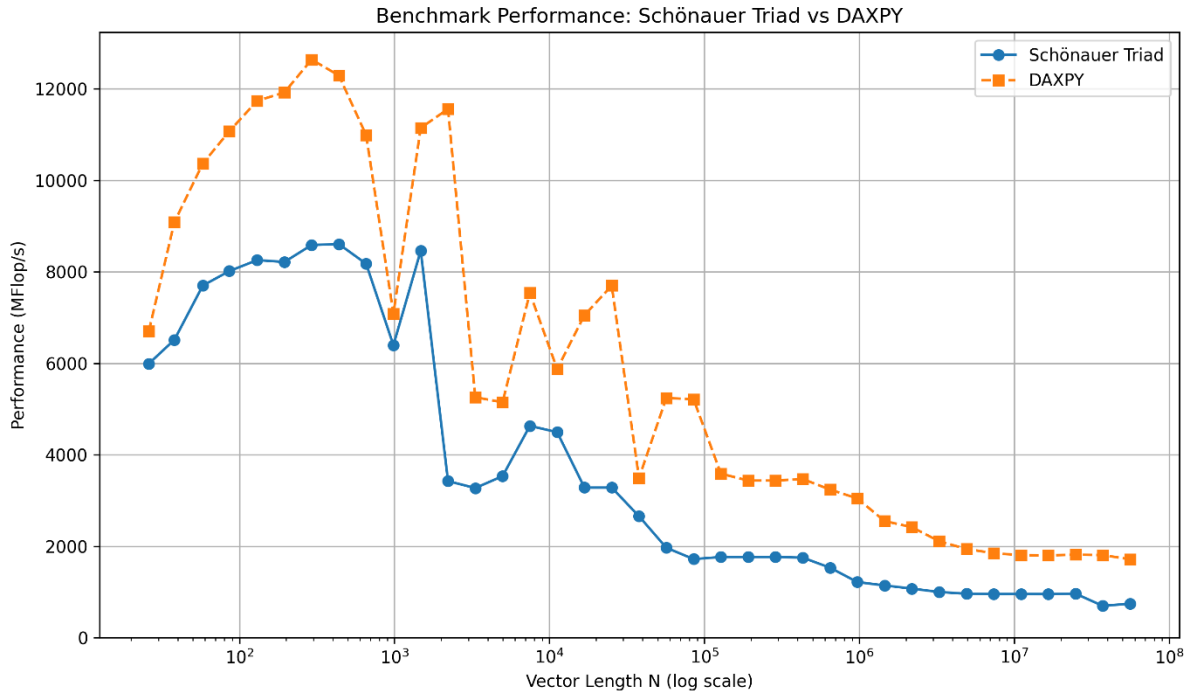

Figure 1. Benchmark Comparision for DAXPY and Schönauer Triad

As it can be seen from the graph, the **Peak performance** occurs around vector length $\mathbf{N \approx 10^2 – 10^3}$, where:

- **DAXPY** reaches ~**12,000 MFLOP/s**
- **Schönauer Triad** reaches ~**8,500 MFLOP/s**

The first sudden drop happens just after ~$\mathbf{N \approx 10^3}$ likely due to exceeding **L2/L3 cache capacity**.

A second major drop appears beyond $\mathbf{N \approx 10^4}$ where performance levels off due to **main memory bandwidth limits**.