Advanced Systems Lab

Assignment 1

Albert Cerfeda (20-980-272)

Contents

1.1 (15 pts) Get to know your machine . . . 1
1.2 (20 pts) LU Factorization . . . 2
a) . . . 2
b) . . . 2
c) . . . 2
d) . . . 2
1.3 (25 pts) Performance analysis and bounds . . . 3
a) . . . 3
b) . . . 3
c) . . . 3
d) . . . 3
1.4 (25 pts) Basic optimization . . . 3
a) . . . 3
b) . . . 4
c) . . . 4
d) . . . 4
1.5 (10 pts) ILP analysis . . . 4
a) . . . 4
b) . . . 4

28.02.2024
Eidgenössische Technische Hochschule Zürich
D-INFK Department
Switzerland

1.1 (15 pts) Get to know your machine

a) - d)

Below you can see microarchitectural parameters of the laptop used to complete this Homework. **Note**: Apple is notoriously secretive about the detailed specifications of their devices, so the table has been filled on a best-effort basis.

Processor manufacturer	TSMC
Processor name	Apple M1 Pro (2021)
Serial number	KP9H9FY0KF[1]
Microarchitecture	Firestorm/Icestorm[2]
CPU Base Frequency	3.28 GHz (performance cores), 2.06 GHz (efficiency cores)
CPU Maximum Frequency	Not available
Support for Intel Turbo Boost or similar	No

Table 1: Microarchitectural parameters

e)

```c
int fp(float a, float b) {
    return a/b;
}
```

(a) C code performing floating point division

```
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 14, 0    sdk_version 14, 2
    .globl    _fp                              ; -- Begin function fp
    .p2align    2
_fp:                                    ; @fp
    .cfi_startproc
; %bb.0:
    fdiv    s0, s0, s1
    fcvtzs    w0, s0
    ret
    .cfi_endproc
                                        ; -- End function
.subsections_via_symbols
```

(b) Compiled assembly code, using `GCC` (Apple clang version 15.0.0) and `-O3` optimization level

Figure 1: Floating point division compilation

In Figure 1b we can see the assembly code generated by the `GCC` compiler for the C code shown in Figure 1a.
Using `Apple clang`, the resulting compiled code utilizes the `FDIV` instruction for performing floating point division. While the instruction shares the same name as the one used in the x86 architecture, it is important to note that the underlying microarchitecture of the Apple M1 Pro is ARM-based, and the instruction set is `ARMv8.5-A`. Furthermore, SSE/SSE2 is not supported.

f)

The x86 floating point operations are included in the standard x86 instruction set architecture while SSE/SSE2 is an extension of the x86 instruction set. Both operate on special dedicated registers, with the difference that the x87 FPU have a higher precision, while the SIMD registers are 128 bits, allowing to perform arithmetic operations on multiple floating points in parallel.

[1] Serial number: The device serial number is indicated, since there is no apparent serial number for the processor itself provided by Apple.

[2] Firestorm/Icestorm: The microarchitecture of the Apple M1 Pro (2021) uses *Firestorm* for its performance cores and *Icestorm* for its efficiency cores [Source: notebookcheck.net]

g)

For the following exercises the Dougall Johnson's documentation about the Firestorm architecture has been used. The article mentions that Firestorm has a pipeline width of 8 instructions per cycle, and 4 SIMD ports each of which can execute both floating-point and SIMD instructions. So the maximum would be 4 floating-point operations per cycle.

h)

The article does not seem to make a difference between single and double precision, so the latency for addition is 3 and throughput is is 4, for both single-precision and double precision.

i)

j)

The latency for FP division is 10 and throughput is 1.

1.2 (20 pts) LU Factorization

a)

b)

The `compute` function performs n^2 FP divisions, n^3 FP multiplications and n^3 FP subtractions, for a total of $n^2 + 2n^3$ floating point operations. **Note:** Subtractions are reported since they are typically implemented using additions, due to 2s complement arithmetic.

c)

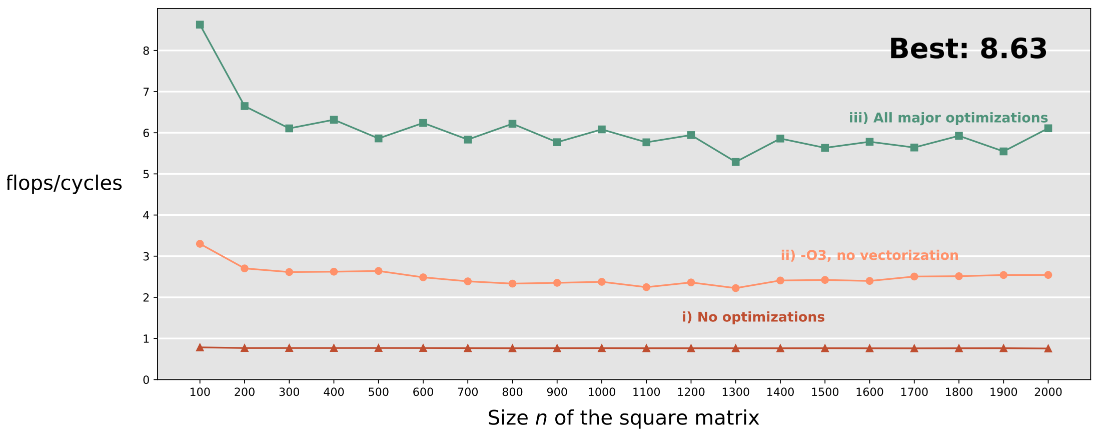

Figure 2: flops/cycle performance of the `compute` function compiled with different optimization flags

d)

1. No optimizations: The code is neither optimized nor vectorized. The performance is the worst and flat across the line.
2. Optimized, but no vectorization: The compiler performs some optimizations resulting in a performance improvement, but no vectorization is performed.

3. All major optimizations: The compiler both performs optimizations and vectories the code, resulting in the best performance. We also notice how for the matrix sizes we tested, the performance does not seem to vary greatly. The peak performance recorded is around 8.63 flops/cycles.

1.3 (25 pts) Performance analysis and bounds

a)

b)

c)

d)

1.4 (25 pts) Basic optimization

a)

```c
void compute(double *x, double *y, double *z, int n) {
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < 2; k++) {
            z[k] += x[i + 1 - k] * y[i + k];
        }
    }
}
```

```c
void compute(double *x, double *y, double *z, int n) {
    double z0 = z[0];
    double z1 = z[1];
    int i;
    for (i = 0; i < n; i+=3) {
        int next = i+1;
        int nnext = i+2;
        int nnnext = i+3;
        z0 += (x[next] * y[i]) + (x[nnext] * y[next])
            + (x[nnnext] * y[nnext]);
        z1 += (x[i] * y[next]) + (x[next] * y[nnext])
            + (x[nnext] * y[nnnext]);
    }
    for (; i < n; i++) {
        int next = i+1;
        z0 += x[next] * y[i];
        z1 += x[i] * y[next];
    }
    z[0] = z0;
    z[1] = z1;
}
```

Figure 3: Unoptimized and optimized `comp` functions

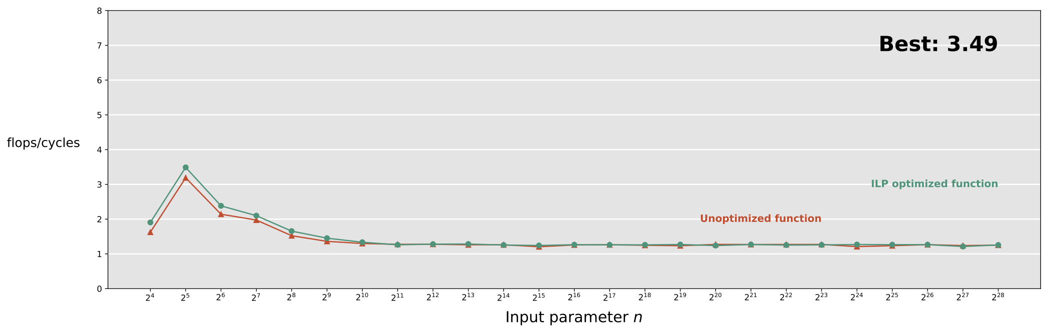

Figure 4: flops/cycle performance of the `comp` functions

b)

c)

d)

Figure 3 shows a comparison between the unoptimzied `comp` function and its optimized counterpart. In order to improve ILP I perfomed 3x loop unrolling and introduced accumulators for reducing memory accesses to the array `z`. Initially I implemented 4x loop unrolling, but the performance overhead became noticeable so I resorted to a lower 3x loop unrolling.

When running the optimized function on Code Expert, we obtain a 2.73 speedup, but as you can see from the plot in Figure 4, when compiling and benchmarking on my Macbook M1 machine, there isn't any performance improvement over the unoptimized version. This is due to the `Apple Clang` compiler and that M1 is a different architecture to the one used to run the benchmarks on Code Expert. The best performance recorded on my machine is around 3.49 flops/cycles.

1.5 (10 pts) ILP analysis

a)

b)

```c
double artcomp (double a, double b, double c, double d) {
    double r;
    r = (a*a*a) / (a*b + (c - d ));
    return r;
}
```