Advanced Systems Lab

Assignment 1

Albert Cerfeda (20-980-272)

Contents

ETH zürich

28.02.2024
Eidgenössische Technische Hochschule Zürich
D-INFK Department
Switzerland

1.0 Disclaimers and assumptions

This little subsection is here to clarify the assumptions and shortcuts I had to take for this assignment. I figured it would be better to have them all in one place, rather than scattered throughout the document.

Machine used
The machine used to complete this assignment is a 2021 Macbook Pro M1 (10 cores CPU - 16 cores GPU), which is an ARM-based machine.

Microarchitectural parameters
While the handout reccomends to use the Dougall Johnson's documentation of the Firestorm architecture for gathering information about the microarchitectural parameters, the article does not suffice in painting a complete picture of the machine, as M1 devices have both Performance cores (running Firestorm) and Efficiency cores (running Icestorm).

Frequency and plots
The `c_clock` function is unreliable on my machine. Therefore the cycles for each benchmark have been calculated by using the time passed in seconds and the frequency of the machine. Performance and Efficiency cores run at different clock speeds, so the frequency used in the calculation has been obtained by performing a weighted average of the Performance and Efficiency cores (`2.53 GHz`). This is obviously not ideal, resulting in a higher margin of error.

Compiler
The compiler used is `Apple clang version 15.0.0 (clang-1500.3.9.4)`, compiling for target `arm64-apple-darwin23.2.0`.

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)

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.

[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]

```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

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 is an 80 bit precision floating point and thus has a higher precision, while the SIMD registers are 128 bits, allowing to perform arithmetic operations on multiple floating points in parallel (either 4 signle-precisions FP or 2 double-recision FP).

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 flops/cycle.

h)

Latency: 4 cycles. Throughput: 0.25 cycles per intruction. Instruction: FMUL.

i)

j)

Latency: 10. Throughput: 1 cycle per instruction. Instruction: FDIV.

1.2 (20 pts) LU Factorization

a)

b)

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

$W(n) = n^3 - \frac{n^2}{2} - \frac{n}{2}$ and $Q(n) \geq 16n^2$. $I(n) \leq \frac{n}{16}$

c)

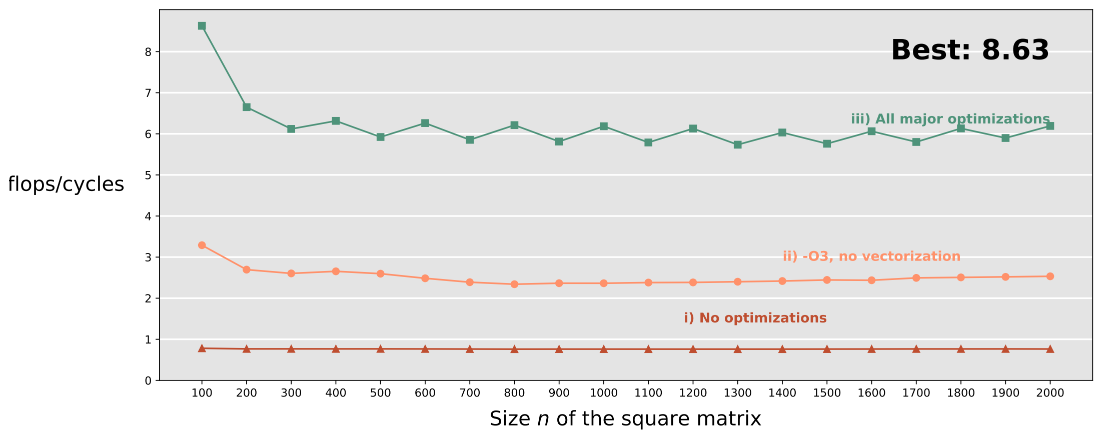


Figure 2: flops/cycle performance of the `compute` function compiled with different optimization flags

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)

$C(n) = C_{add} * N_{add} + C_{mult} * N_{mult}$

b)

$$\begin{aligned} N_{add} &= 3n, \\ N_{mult} &= 2n, \\ C(n) &= C_{add} * (3n) + C_{mult} * (2n) \end{aligned}$$

c)

i) We have a total of $3n$ floating point additions and $2n$ floating point mutiplications. All operations can be scheduled either on Port 0 or Port 1. This results in a lower bound of $2.5n$ cycles.

ii) Now that FMA instructions are enabled, we can fuse the last two operations into one single FMA instruction (i.e $(u_i*(w_i+x_i))*(u_i-x_i)+y_i)$. We therefore have now a total of n FMA instructions, $2n$ floating point additions and n floating point multiplications, and a lower bound of $2n$ cycles.

iii) In our computation we have to read at least $5n$ doubles from the arrays x, w, u, z respectively. Therefore, $r_{L3} \geq \frac{5n}{4}$ and $r_{RAM} \geq \frac{5n}{2}$.

d)

The operation intensity is $I(N) \leq \frac{5n \text{ flops}}{8(5n) \text{ bytes}} = \frac{1}{8}$ flops/bytes.

1.4 (25 pts) Basic optimization

a)

```
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];
        }
    }
}
```

```
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

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.

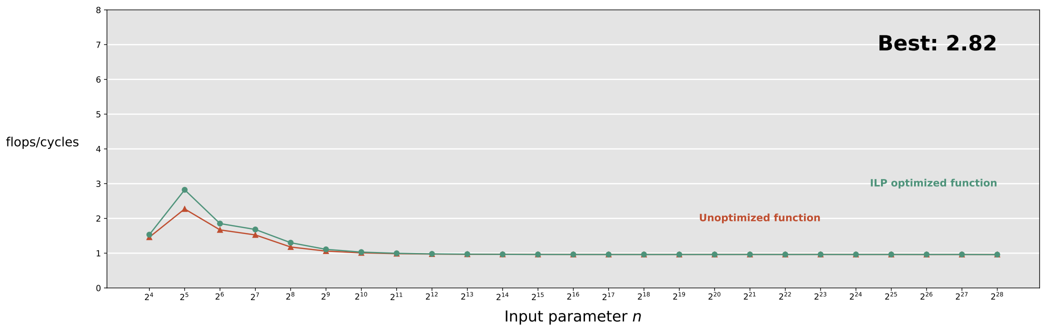


Figure 4: flops/cycle performance of the `comp` functions

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 2.82 flops/cycles.

1.5 (10 pts) ILP analysis

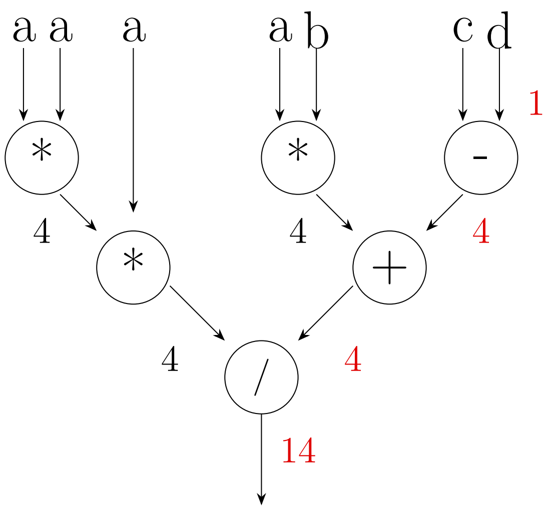
```
double artcomp (double a, double b, double c, double d) {
    double r;
    r = (a*a*a) / (a*b + (c - d ));
    return r;
}
```

a)

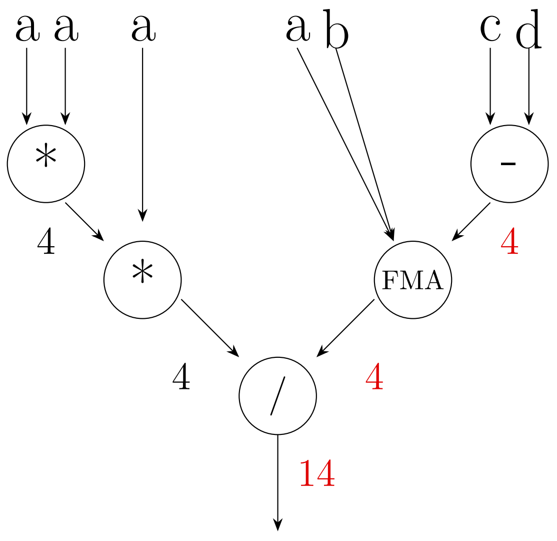
In Figure 5a we can see a runtime of at least **23 cycles**. Note that the two mults and 1 subtraction can't start at the same time since the can be only executed on Port 0 and Port 1, introducin gadditional delay.

b)

In Figure 5b we can see a runtime of at least **22 cycles**. This time around there is no additional delay as we can execute the $a * a$ and $c - d$ operation at the same time.



(a)



(b)

Figure 5: Dependency trees