Chapter 2: Case study

Version 1: Assembly code [advanced]

Recall how we identified the relevant part of the assembly code of version 0. Let us now look at the code generated for version 1. The innermost loop is really simple this time, with only 6 instructions:

```
LOOP:
        vmovss      (%rsi,%rax,4), %xmm0
        vaddss      (%rdx,%rax,4), %xmm0, %xmm0
        addq        $1, %rax
        cmpl        %eax, %ebx
        vminss      %xmm1, %xmm0, %xmm1
        jg          LOOP
```

Interpretation

Here are the most interesting instructions, side by side with a slightly modified version of the source code. Register `%xmm0` plays the role of variable `x` and register `%xmm1` plays the role of variable `v`:

Modified C++ code	Assembly code
`x = d[n*i + k];`	`vmovss (%rsi,%rax,4), %xmm0`
`x = t[n*j + k] + x;`	`vaddss (%rdx,%rax,4), %xmm0, %xmm0`
`v = std::min(v, x);`	`vminss %xmm1, %xmm0, %xmm1`

Let us see if we can understand the results of the benchmarks based on what we see here.

Latencies and throughputs

We have been able to identify the key instructions that do the actual relevant arithmetic operations in this code: `vaddss` and `vminss`. Now we can look up the latency and throughput of these instructions e.g. in Agner Fog's Instruction tables.

In the tables, the relevant section for the processor that we use is Intel Skylake. Instructions `vaddss` and `vminss` are not listed separately, but we can find `addss` and `minss` in the table. Particularly interesting are the following rows (abbreviated here):

Instruction	Latency	Reciprocal throughput
ADDSS v,v,v	4	0.5
MINSS v,v,v	4	0.5

Basically, addition and minimum have the same latency, assuming all operands are already in the registers. (This is not the case in our example for the addition instruction, but let us put this detail aside for now.)

A reciprocal throughput of 0.5 clock cycles per operations is the same as a throughput of 2 operations per clock cycle. If there were no other bottlenecks in our code — all data was readily available in the CPU registers, and all operations were independent — we would expect to complete 2 useful operations per clock cycle. This is far from the number 0.50 that we got from our benchmarks.

Dependency chain

It turns out that in our code there is a dependency chain that makes the code latency-limited. If we only focus on the instructions that manipulate the `%xmm1` register, we will see that the CPU will execute the following sequence of operations:

```
...
vminss %xmm1, %xmm0, %xmm1
...
vminss %xmm1, %xmm0, %xmm1
...
vminss %xmm1, %xmm0, %xmm1
...
```

Each `vminss` operation depends on the previous `vminss` operation. Each operation uses the current value of register `%xmm1` as input, and stores the result in register `%xmm1`. Hence, we must complete the previous operation first in order to know what is the new value of `%xmm1` before we can start the next operation.

The same dependency chain is visible also in the original C++ code. If we only focus on the manipulations of variable `v`, we will see the following sequence of instructions:

```
...
v = std::min(v, z);
...
v = std::min(v, z);
...
v = std::min(v, z);
...
```

Each `std::min` operation depends on the result of the previous `std::min` operation.

In essence, in each iteration of the innermost loop we will have to wait for at least the latency of the `vminss` operation, which is 4 clock cycles. Put otherwise, in 4 clock cycles we can only perform 2 useful operations: one addition and one minimum. Even if there were no other bottlenecks, we would expect from this code the performance of at most 0.5 useful operations per clock cycle. And this is almost exactly the same number as what we got in our benchmarks!

In particular, now we know that getting data from the main memory to the CPU is not anymore a bottleneck. We will need to focus next on how to eliminate the sequential dependency chain in the "min" operations.

Interactive assembly

We provide a slightly modified version on Compiler Explorer, which can be found here. To reduce the amount of boilerplate code generated, this version does not contain the OpenMP directives, and only provides the part of `step` that implements the main loop, skipping over the generation of the transposed matrix.