Chapter 2: Case study

Version 3: Assembly code [advanced]

We claimed that the compiler would generate machine code that uses vector registers and vector instructions if we write C++ code that uses vector types. Let us now verify this. The innermost loop in our code was:

```
for (int ka = 0; ka < na; ++ka) {
    float8_t x = vd[na*i + ka];
    float8_t y = vt[na*j + ka];
    float8_t z = x + y;
    vv = min8(vv, z);
}
```

Recall that here also `vd` and `vt` were pointers to `float8_t`, and `vv` was a local variable of type `float8_t`. The assembly code generated from this code is as follows:

```
LOOP:
            vmovaps     (%rcx,%rax), %ymm0
            vaddps      (%rdx,%rax), %ymm0, %ymm0
            addq        $32, %rax
            cmpq        %rsi, %rax
            vminps      %ymm0, %ymm1, %ymm1
            jne         LOOP
```

The compiler is indeed using 256-bit vector registers (`%ymm`), as well as vector instructions that do elementwise operations for 8 floats (`...ps`). Here “ps” refers to **p**acked **s**ingle-precision floating point numbers.

Comparison with V1

It is instructive to compare this with what we had in **version 1**. There the innermost loop was:

```
for (int k = 0; k < n; ++k) {
    float x = d[n*i + k];
    float y = t[n*j + k];
    float z = x + y;
    v = std::min(v, z);
}
```

And the assembly code looked like this:

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

The key differences are these:

- V1 uses scalar instructions (`...ss`), while V3 uses vector instructions (`...ps`).
- V1 uses 128-bit registers (`%xmm`), while V3 uses 256-bit registers (`%ymm`).

A slightly confusing detail here is that `%xmm` registers are also vector registers and they could hold 4 floats! Moreover, e.g. `%xmm0` is actually the same thing as the lower half of `%ymm0`. However, all instructions that we see here manipulate only the first element of each vector; the old version was not making any use of the vector capabilities of the CPU.

Interactive assembly

Here is a minimal version of the code to experiment with.

- What happens if you change the target architecture to something that does not support `AVX`, e.g., by switching to `-march=x86-64`?