Chapter 2: Case study

Version 2: Assembly code [advanced]

Let us now see that the compiler really does the right job here. By looking at the assembly code produced by the compiler, we see that the innermost loops

```
for (int ka = 0; ka < na; ++ka) {
    for (int kb = 0; kb < nb; ++kb) {
        float x = d[nab*i + ka * nb + kb];
        float y = t[nab*j + ka * nb + kb];
        float z = x + y;
        vv[kb] = std::min(vv[kb], z);
    }
}
```

get translated into this assembly code:

```
LOOP:
        vmovss      (%rax), %xmm0
        addq        $16, %rax
        addq        $16, %rdx
        vaddss      -16(%rdx), %xmm0, %xmm0
        vminss      %xmm1, %xmm0, %xmm1
        vmovss      -12(%rax), %xmm0
        vaddss      -12(%rdx), %xmm0, %xmm0
        vminss      %xmm4, %xmm0, %xmm4
        vmovss      -8(%rax), %xmm0
        vaddss      -8(%rdx), %xmm0, %xmm0
        vminss      %xmm3, %xmm0, %xmm3
        vmovss      -4(%rax), %xmm0
        vaddss      -4(%rdx), %xmm0, %xmm0
        cmpq        %rsi, %rax
        vminss      %xmm2, %xmm0, %xmm2
        jne         LOOP
```

If we ignore the operations related to the loop counters, we can see that the compiler did a very straightforward translation from C++ to machine code here. The compiler unrolled the loop that always counts for 4 rounds. It decided to keep variable `x` in register `%xmm0`, and array elements `vv[0]` to `vv[3]` in registers `%xmm1`, `%xmm4`, `%xmm3`, and `%xmm2`. We can easily read the (unrolled and slightly modified) C++ code and assembly code side by side:

Modified C++ code	Assembly code
`x = d[... + 0];`	`vmovss ..., %xmm0`
`x = t[... + 0] + x;`	`vaddss ..., %xmm0, %xmm0`
`vv[0] = std::min(vv[0], x);`	`vminss %xmm1, %xmm0, %xmm1`
`x = d[... + 1];`	`vmovss ..., %xmm0`
`x = t[... + 1] + x;`	`vaddss ..., %xmm0, %xmm0`
`vv[1] = std::min(vv[1], x);`	`vminss %xmm4, %xmm0, %xmm4`
`x = d[... + 2];`	`vmovss ..., %xmm0`
`x = t[... + 2] + x;`	`vaddss ..., %xmm0, %xmm0`
`vv[2] = std::min(vv[2], x);`	`vminss %xmm3, %xmm0, %xmm3`
`x = d[... + 3];`	`vmovss ..., %xmm0`
`x = t[... + 3] + x;`	`vaddss ..., %xmm0, %xmm0`
`vv[3] = std::min(vv[3], x);`	`vminss %xmm2, %xmm0, %xmm2`
`x`	`%xmm0`
`vv[0]`	`%xmm1`
`vv[1]`	`%xmm4`
`vv[2]`	`%xmm3`
`vv[3]`	`%xmm2`

Most importantly, we can see that the compiler did not do anything special related to instruction-level parallelism. There is no special annotation in the machine code that instructs the CPU which operations can be executed in parallel. The CPU does the job by itself, and it automatically realizes that all of these instructions can be safely executed in parallel (as soon as the other operand is available):

```
...
vminss %xmm1, ..., %xmm1
vminss %xmm4, ..., %xmm4
vminss %xmm3, ..., %xmm3
vminss %xmm2, ..., %xmm2
...
```

But what about `vaddss` and `%xmm0`?

At first it might look like the register `%xmm0` would be a bottleneck here. Every `vaddss` operation needs it, every `vaddss` operation modifies it, and finally every `vminss` needs it. Doesn't this make the code inherently sequential, and prevent the processor from executing instructions efficiently?

The short answer is: no, this is not a problem, thanks to register renaming. The names such as `%xmm0` do not really refer to physical registers in the processor, but they are just some labels that the programmer can use – a lot like the names of the local variables in C++ code, except that there is a limited number of those.

Internally, the processor is free to assign these labels into physical registers, as long as it does not change the result. Let us use fictitious names such as `%xmm0a` for the physical registers. The processor might rename the registers e.g. as follows:

Original registers	Physical registers
`vmovss ..., %xmm0`	`vmovss ..., %xmm0a`
`vaddss ..., %xmm0, %xmm0`	`vaddss ..., %xmm0a, %xmm0a`
`vminss %xmm1, %xmm0, %xmm1`	`vminss %xmm1, %xmm0a, %xmm1`
`vmovss ..., %xmm0`	`vmovss ..., %xmm0b`
`vaddss ..., %xmm0, %xmm0`	`vaddss ..., %xmm0b, %xmm0b`
`vminss %xmm4, %xmm0, %xmm4`	`vminss %xmm4, %xmm0b, %xmm4`
`vmovss ..., %xmm0`	`vmovss ..., %xmm0c`
`vaddss ..., %xmm0, %xmm0`	`vaddss ..., %xmm0c, %xmm0c`
`vminss %xmm3, %xmm0, %xmm3`	`vminss %xmm3, %xmm0c, %xmm3`
`vmovss ..., %xmm0`	`vmovss ..., %xmm0d`
`vaddss ..., %xmm0, %xmm0`	`vaddss ..., %xmm0d, %xmm0d`
`vminss %xmm2, %xmm0, %xmm2`	`vminss %xmm2, %xmm0d, %xmm2`

Clearly, the new code is equivalent to the old code. What we achieved is the same thing as if we had used separate local variables in the C++ code, instead of reusing the same variable `x`:

Original C++ code	Renamed variables
`x = d[... + 0];`	`xa = d[... + 0];`
`x = t[... + 0] + x;`	`xa = t[... + 0] + xa;`
`vv[0] = std::min(vv[0], x);`	`vv[0] = std::min(vv[0], xa);`
`x = d[... + 1];`	`xb = d[... + 1];`
`x = t[... + 1] + x;`	`xb = t[... + 1] + xb;`
`vv[1] = std::min(vv[1], x);`	`vv[1] = std::min(vv[1], xb);`
`x = d[... + 2];`	`xc = d[... + 2];`
`x = t[... + 2] + x;`	`xc = t[... + 2] + xc;`
`vv[2] = std::min(vv[2], x);`	`vv[2] = std::min(vv[2], xc);`
`x = d[... + 3];`	`xd = d[... + 3];`
`x = t[... + 3] + x;`	`xd = t[... + 3] + xd;`
`vv[3] = std::min(vv[3], x);`	`vv[3] = std::min(vv[3], xd);`

After renaming, it is much easier to see that there is lots of potential for parallelism. For example, we can execute the following memory lookup instructions in parallel:

```
vmovss ..., %xmm0a
vmovss ..., %xmm0b
vmovss ..., %xmm0c
vmovss ..., %xmm0d
```

And we can do the following additions in parallel:

```
vaddss ..., %xmm0a, %xmm0a
vaddss ..., %xmm0b, %xmm0b
vaddss ..., %xmm0c, %xmm0c
vaddss ..., %xmm0d, %xmm0d
```

And finally we can also do the following minimum operations in parallel:

```
vminss %xmm1, %xmm0a, %xmm1
vminss %xmm4, %xmm0b, %xmm4
vminss %xmm3, %xmm0c, %xmm3
vminss %xmm2, %xmm0d, %xmm2
```

This is not the whole truth. The CPU looks at the instruction stream much further than just one iteration of the loop, and does register allocation and instruction scheduling for a code segment that spans several iterations of the innermost loop. Hence, it might well happen that the CPU is executing simultaneously e.g.

- memory lookups related to iteration `ka + 2`,
- `vaddss` operations related to iteration `ka + 1`, and
- `vminss` related to iteration `ka`,

Hence, e.g., the results of `vaddss` will be conveniently available when `vminss` needs them.

Analysis

Recall that `vaddss` and `vminss` have a throughput of 2 instructions per clock cycle. A bit more precisely, there are 2 execution ports ("port 0" and "port 1") in the CPU core that are able to run `vaddss` and `vminss` operations. In each clock cycle, we can start either one `vaddss` or one `vminss` operation in port 0, and simultaneously we can also start either one `vaddss` or one `vminss` operation in port 1.

Hence if we have any sequence of `vaddss` and/or `vminss` operations, we can at best hope to complete 2 operations per clock cycle per CPU core.

And it turns out that we are now pretty close to the theoretical limits. For n = 4000, we now manage to do approx. 1.6 operations per clock cycle per CPU core, so there is not much room for improvement unless we manage to write code that does something substantially different from `vaddss` and `vminss`.

Note that the number 1.6 is the number of "useful" operations, i.e., the number of `vaddss` and `vminss` operations per clock cycles. In addition to these operations, we also do memory lookups, manipulations of the loop counters, comparisons, and conditional jumps. The grand total is approx. 3.3 machine language instructions per clock cycle per core (we can easily find these statistics with the help of the perf tool – just run the program under `perf stat` and see the output). This is a relatively high number. It is reasonable to expect the CPU cores to execute some 3–4 instructions per clock cycle, but much more than that is going to be difficult to achieve in practice. Hence, for further speedups we will need to do more useful work per instruction. This is exactly what we can achieve with vector instructions, which is our next topic.

Interactive assembly

As before, a simplified version of this code is provided on Compiler Explorer. You can experiment with this, e.g., see what happens if you change the `nb` variable. Here are some suggestions to investigate:

- What happens if you increase `nb` a bit?
- What happens if you remove the `constexpr` specifier? Also try this with less aggressive optimization, i.e., `-O2`.
- What happens if you remove the `constexpr` specifier, but move the declaration of `nb` inside `step`?
- What happens if you increase `nb` a lot, e.g., to 20?