# Chapter 2: Case study



## Version 2: Instruction-level parallelism

In the previous version, we have an inherently sequential chain of operations in the innermost loop. We accumulate the minimum in variable `v` by a sequence of `min` operations. There is no way to start the second operation before we know the result of the first operation; there is no room for parallelism here:

```
...
v = std::min(v, z0);
v = std::min(v, z1);
v = std::min(v, z2);
v = std::min(v, z3);
v = std::min(v, z4);
...
```

## Independent operations

There is a simple way to reorganize the operations so that we have more room for parallelism. Instead of accumulating one minimum, we could accumulate two minimums, and at the very end combine them:

```
...
v0 = std::min(v0, z0);
v1 = std::min(v1, z1);
v0 = std::min(v0, z2);
v1 = std::min(v1, z3);
v0 = std::min(v0, z4);
...
v = std::min(v0, v1);
```

The result will be clearly the same, but we are calculating the operations in a different order. In essence, we split the work in two independent parts, calculating the minimum of odd elements and the minimum of even elements, and finally combining the results. If we calculate the odd minimum `v0` and even minimum `v1` in an interleaved manner, as shown above, we will have more opportunities for parallelism. For example, the 1st and 2nd operation could be calculated simultaneously in parallel (or they could be executed in a pipelined fashion in the same execution unit). Once these results are available, the 3rd and 4th operation could be calculated simultaneously in parallel, etc. We could potentially obtain a speedup of a factor of 2 here, and naturally the same idea could be extended to calculating e.g. 4 minimums in an interleaved fashion.

## Instruction-level parallelism is automatic

Now that we know how to reorganize calculations so that there is potential for parallelism, we will need to know how to realize the potential. For example, if we have these two operations in the C++ code, how do we tell the computer that the operations can be safely executed in parallel?

```
v0 = std::min(v0, z0);
v1 = std::min(v1, z1);
```

The delightful answer is that it happens completely automatically, there is nothing we need to do (and nothing we can do)!

The magic takes place inside the CPU. The compiler just produces two machine language instructions, without any special annotation that indicates whether these instructions can be executed in parallel. The CPU will then automatically figure out which of the instructions can be executed in parallel.

A bit more precisely, the CPU will look at the instruction stream up to some distance in the future. If there are branches, it will do branch prediction to produce a sequential stream of instructions. Then it will see which of the instructions are ready for execution. For example, if it sees a future instruction X that only uses registers A and B, and there are no instructions before it that touch those registers, and none of the instructions that are currently in the pipeline modify those registers, either, then it is safe to start to execute X as soon as there is an execution unit that is available.

All of this happens in the hardware, all the time, fully automatically. The only thing that the programmer needs to do is to make sure there are sufficiently many independent instructions always available for execution.

## Implementation

Now we are ready to implement the idea. In the previous version, the key arithmetic operations were "+" and "min". From the perspective of the instruction throughput, we could execute in total 2 such instructions per clock cycle, but the latency limited us to 0.5 instructions per clock cycle. There is hence potential for 4-fold speedups, by simply making sure we can execute 4 independent operations in parallel.

To implement this, we will calculate 4 minimums, in an interleaved fashion, and finally combine these into one value.

This would be really easy if `n` was a multiple of 4. Then we could simply do `n/4` iterations, each time processing `4` elements. However, to handle the general case, we will need to do something else.

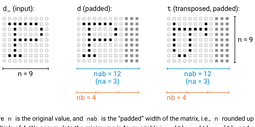
From the performance perspective, we want to keep the innermost loop as simple as possible. We do not want to have e.g. any comparisons there. Hence, we have got basically two options left:

1. Preprocess the data so that we add some padding elements so that the width of the array is a multiple of 4.
2. Do `n/4` iterations to handle the first `(n/4)*4` elements, and add a separate loop that takes care of the remaining `n % 4` elements.

As we are doing some preprocessing anyway to compute the transpose, the first option is convenient in our case. The full implementation (with appropriate OpenMP directives) might look like this:

```
constexpr float infty = std::numeric_limits<float>::infinity();

void step(float* r, const float* d,, int n) {
    constexpr int nb = 4;
    int na = (n + nb - 1) / nb;
    int nab = na*nb;

    // input data, padded
    std::vector<float> d(n*nab, infty);
    // input data, transposed, padded
    std::vector<float> t(n*nab, infty);

    #pragma omp parallel for
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            d[nab*j + i] = d_[n*j + i];
            t[nab*i + j] = d_[n*i + j];
        }
    }

    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            // vv[0] = result for k = 0, 4, 8, ...
            // vv[1] = result for k = 1, 5, 9, ...
            // vv[2] = result for k = 2, 6, 10, ...
            // vv[3] = result for k = 3, 7, 11, ...
            float vv[nb];
            for (int kb = 0; kb < nb; ++kb) {
                vv[kb] = infty;
            }
            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);
                }
            }
            // v = result for k = 0, 1, 2, ...
            float v = infty;
            for (int kb = 0; kb < nb; ++kb) {
                v = std::min(vv[kb], v);
            }
            r[n*i + j] = v;
        }
    }
}
```

Here is an illustration of what the data layout looks like if we have e.g. `n = 9`:




Here `n` is the original value, and `nab` is the "padded" width of the matrix, i.e., `n` rounded up to the next multiple of 4. We accumulate the minimums in four variables, `vv[0]`, `vv[1]`, `vv[2]` , and `vv[3]` , and combine them into one value `v`. The order of the operations is different, but the end result is identical to what we computed previously. (Note that it is exactly identical, even though we use floating point operations!)

## Notes

The code may look at first somewhat inefficient: Now there is a new loop

```
for (int kb = 0; kb < nb; ++kb) { ... }
```

that was added inside the innermost loop; doesn't this add some overhead? Also, we are using an array `vv` instead of 4 individual local variables; isn't that expensive?

As we will see next, this is fine. The compiler (with the optimization flag `-O3` ) does the right job. First, it notices that `nb` is a compile-time constant. Therefore, the loops `for (int kb = 0; ...)` always run for exactly 4 times, and it can be unrolled. In essence, the compiler will turn e.g. this code fragment:

```
for (int kb = 0; kb < nb; ++kb) {
    vv[kb] = infty;
}
```

... into this code fragment:

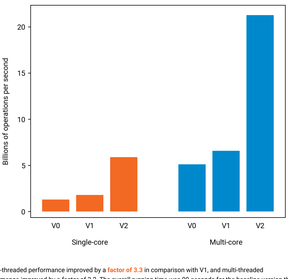
```
vv[0] = infty;
vv[1] = infty;
vv[2] = infty;
vv[3] = infty;
```

And after unrolling, all references to array `vv` use constant indexes, so the compiler is clever enough to realize that it can replace the array elements `vv[0]` , `vv[1]` , `vv[2]` , and `vv[3]` with individual variables `vv0` , `vv1` , `vv2` , and `vv3` .

There is nothing wrong with using 4 individual variables. But with an array and the loop, it is much easier to see what happens if we replace the magic number 4 by, e.g., 2 or 8. However, if we rely on the compiler to be clever in critical parts, it is a very good idea to check the assembly code produced by the compiler to make sure there is nothing silly there.

## Results

In this application, instruction-level parallelism helps a lot:




Single-threaded performance improved by a factor of 3.3 in comparison with V1, and multi-threaded performance improved by a factor of 3.2. The overall running time was 99 seconds for the baseline version that we started with, and it is now only 6 seconds for the version that uses both multi-threading and instruction-level parallelism.

With one thread, we are now doing 1.6 useful operations per clock cycle. Now we can see that each CPU core is indeed a superscalar processor: in each clock cycle, we can launch more than one instruction.