Vectorization, dependencies and outer loop vectorization: if you can't beat them, join them

March 13, 2022Computational Performance, Low Level Performance4 Replies

*We at **Johnny's Software Lab LLC** are experts in performance. If performance is in any way concern in your software project, feel free to contact us.*

As I already mentioned in earlier posts, vectorization is the holy grail of software optimizations: if your hot loop is efficiently vectorized, it is pretty much running at fastest possible speed. So, it is definitely a goal worth pursuing, under two assumptions: (1) that your code has a hardware-friendly memory access pattern[1] and (2) that there are no loop-carried dependencies[2].

However, the world is not ideal and there are loops with dependencies that we want to run faster. The question is if they can benefit from vectorization? The answer is not simple and in this post I will try to answer it.

Table Of Contents

A short introduction to vectorization

In vectorization framework, instead of executing an instruction on a single piece of data, the CPU is executing an instruction on several pieces of data at once. We call this several pieces of data *a vector of data*. In modern architectures, vector typically has a fixed size: e.g. 8 floats or 4 doubles, etc.

To illustrate vectorization, have a look at the following loop:

```
for (int i = 0; i < n; i++) {
   c[i] = a[i] + b[i];
}
```

When written like this, a developer would expect the following pseudoassembly:

```
for (int i = 0; i < n; i++) {
   reg0 = load(a + i);
   reg1 = load(b + i);
   reg2 = reg0 + reg1;
   store(c + i, reg2);
}
```

This code loads two values from the memory belonging to arrays `a` and `b` and stores them to registers `reg0` and `reg1` (lines 2 and 3). It then adds those to values together (line 4) and stores it back to the memory (line 5).

Under vectorization framework, instead loading one by one piece of data, the compiler could utilize CPU vector instruction to process four by four pieces of data. Thus, the pseudoassembly could look like this:

```
for (int i = 0; i < n; i+=4) {
   reg0 = load<4>(a + i);
   reg1 = load<4>(b + i);
   reg2 = reg0 + reg1;
   store<4>(c + i, reg2);
}
```

The code loads 4 values at once and stores those four values to register `reg0` (line 2). Similarly, it performs 4 addition (line 4) and stores 4 results back to the memory (line 5).

The problem with dependencies

Loop-carried dependencies make vectorization impossible. Take for example the following loop:

```
for (int i = 1; i < n; i++) {
    b[i] = b[i - 1] + a[i];
}
```

In this loop, there is a loop-carried dependency between values `b[i]` and `b[i - 1]`. To calculate the value for `b[i]`, we need the value of `b[i - 1]` which was calculated in the previous iteration. And for `b[i - 1]` we need the value of `b[i - 2]`. What this mean is that this loop can only be run serially, from 0 to N.

This loop is impossible to vectorize. Remember that the vectorization loads 4 values simultaneously, so at the time the CPU loads the values b[i], b[i + 1], b[i + 2] and b[i + 3] needed to calculate `b[i + 1]`, `b[i + 2]`, `b[i + 3]` and `b[i + 4]`, only value of `b[i]` is calculated and others are not. So the results would be incorrect.

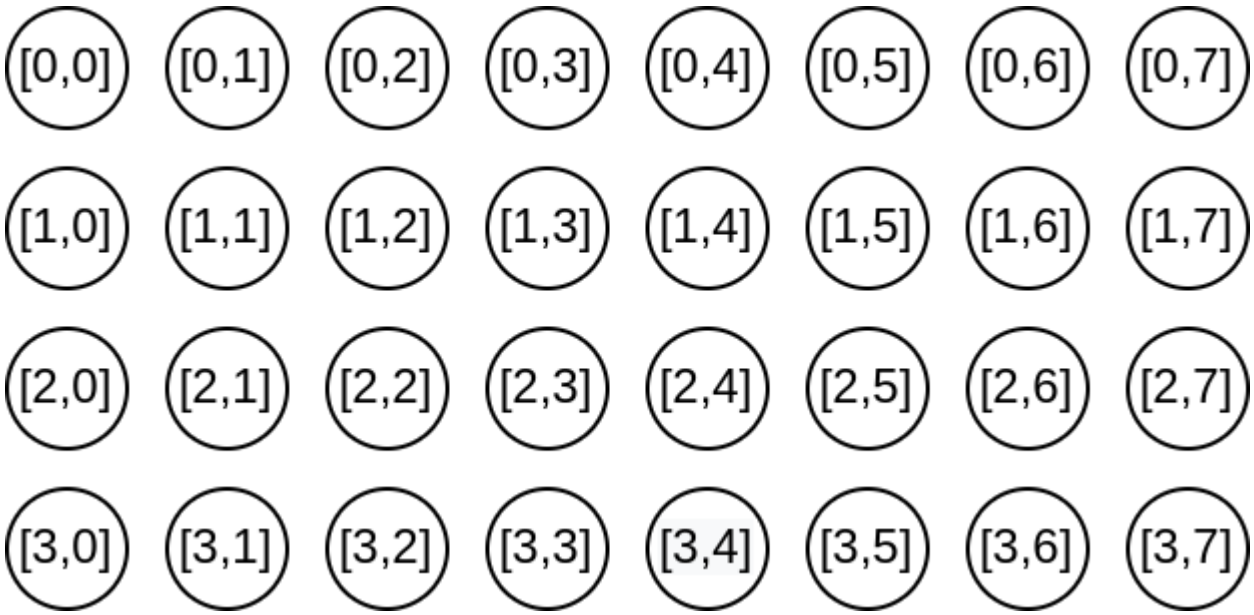
In this example, vectorization is impossible and there is nothing we can do about it. However, there are many cases of loop-carried dependencies where vectorization is possible, as you will see in this post.

Difficulty easy: loop without loop-carried dependecies

Let's investigate again the loop which adds together two matrices:

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
       c[i][j] = a[i][j] + b[i][j];
   }
}
```

Loop-carried dependencies are typically represented using dependency graphs, which in our case can look like this:



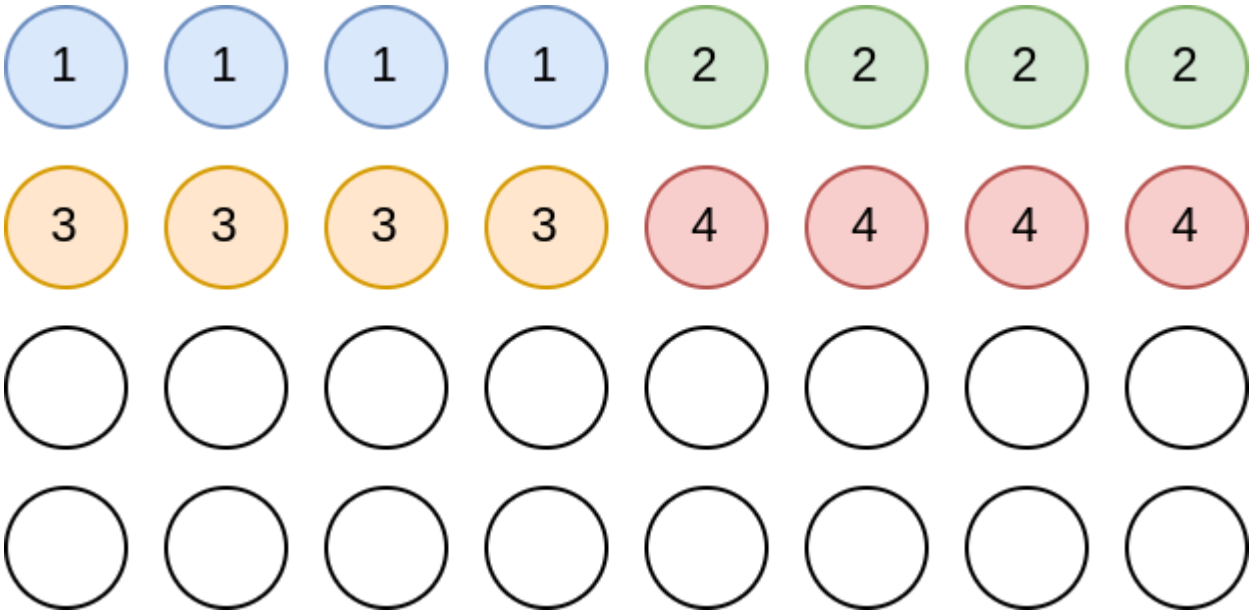
Dependency graph for two nested loops without loop-carried dependencies

A dependency graph represents how iterations of the loop depend on one another. Each iteration is represented with a ball. In this case, there are two iterator values `i` and `j`, and each iteration is represented with two indexes, e.g. `[2, 3]` corresponds to `i = 2` and `j = 3`.

The dependencies are represented with arrows going from one ball to another. In this case, there are no dependencies and thus there are no arrows. In later examples you will see graphs with dependencies.

Because of the way we wrote the loop, the iterations will get executed in this order: `[0, 0]`, `[0, 1]`, `[0, 2]`, ..., `[0,7]`, `[1, 0]`, `[1, 1]`, etc. But since the loop iterations are completely independent, it can get executed in any order, e.g. `[0,5]`, `[1, 6]`, `[3, 8]`, etc.

The compiler can automatically vectorize this loop, and execute it row-by-row. The compiler picks row-by-row execution because of the way the loop is written. Running through a matrix row-by-row is also the most efficient for the memory subsystem. If the CPU executes four iterations simultaneously, the order of execution will look like this:



How is the loop actually executed

This code is efficiently vectorizable since all architectures support loading and storing consecutive memory addresses. Also, compilers are good at automatically vectorizing such code.

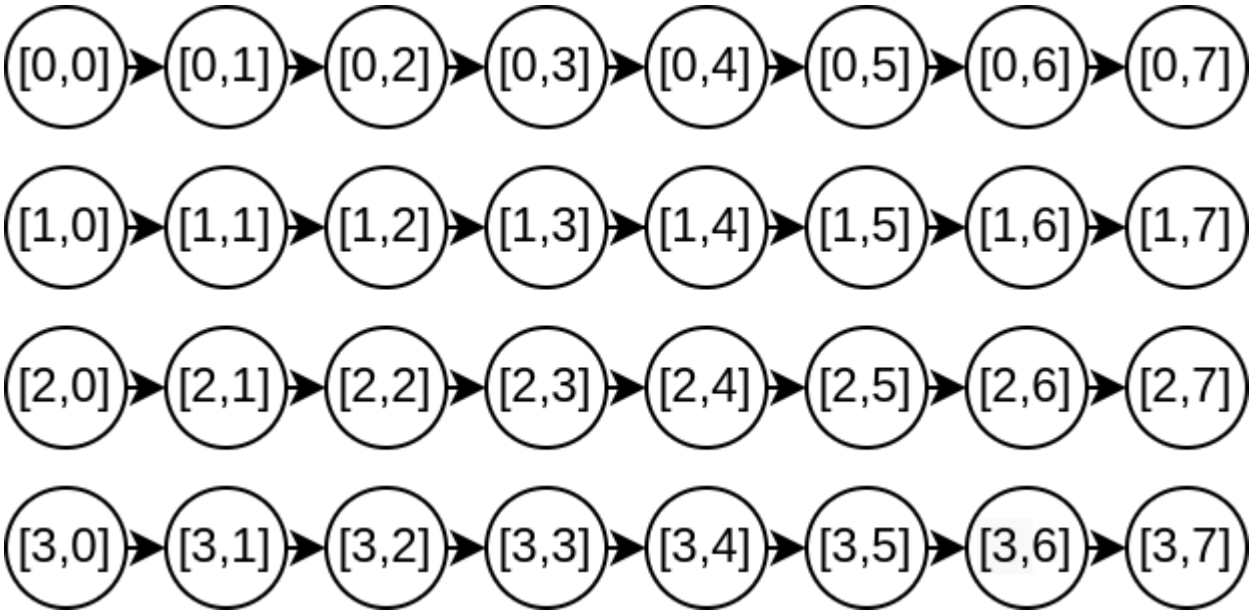
Difficulty medium: loop-carried dependencies inside a single row

Let's consider a different loop, but this time the loop has loop-carried dependencies:

```
for (int i = 0; i < N; i++) {
    out[i][0] = in[i][0];
    for (int j = 1; j < N; j++) {
        out[i][j] = out[i][j - 1] + in[i][j];
    }
}
```

So, if you look the innermost loop, you will see a loop-carried dependency. To calculate `out[i][j]`, the code needs the value `out[i][j - 1]` which was calculated in the previous iteration, etc. At first glance, the loop seems unvectorizable.

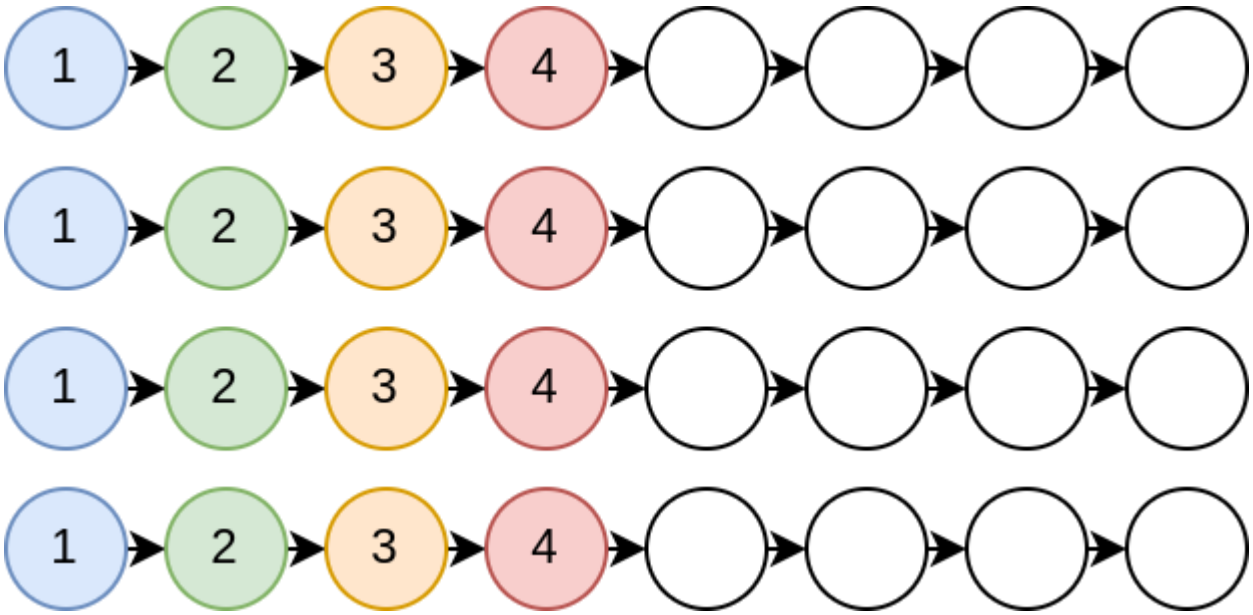
Let's draw the dependency graph for this loop:



Dependency graph for the loop with loop-carried dependencies inside a single row

The most important information one can deduct from this graph is that, although there are loop carried dependencies inside a single row, there are no loop carried dependencies between the rows. For example, row `0` and row `1` can be calculated completely independently.

Is it possible to improve the speed of such loop through vectorization? Yes, it is. But to do it, instead of executing rows one by one, we run the calculations for four rows simultaneously. The ordering would look like this:



How the loop is executed when vectorized

The first vectorized iteration runs scalar iterations [0, 0], [1, 0], [2, 0] and [3, 0]. The second vectorized iteration runs scalar iterations [0,1], [1,1], [2,1] and [3,1], etc. As you can see, inside a vectorized iteration, the iterator `j` has a single value and the iterator `i` has several values.

This approach is called *outer loop vectorization*, because, in contrast to the previous example, the outer loop is running several instances of the inner loop at once. A good explanation of outer loop vectorization used to speed up spare matrix multiplication can be found on the Intel's website.

There are two limitations to outer loop vectorization. First, the target architecture must support data gather and data scatter instructions in order for the vectorization to be efficient (read more about gather and scatter here). Unfortunately, gather and scatter instructions are not ubiquitous. On X86-64 platform, only gathers are available in commodity hardware (which typically supports AVX2 vector extensions). Scatters are available only with AVX512 vector extensions, and these CPUs are rarely seen on desktops. As far as ARM is concerned, NEON vector extensions found in phones and embedded system support neither gathers nor scatters. Gathers and scatters can be emulated, but this makes vectorization less efficient.

The second problem is that the compilers typically do not vectorize these loops automatically. In theory it should be possible with `#pragma omp declare simd` directive, but in practice it doesn't work. The result is that you will need to use compiler intrinsic to do the vectorization, and this process can be tedious.

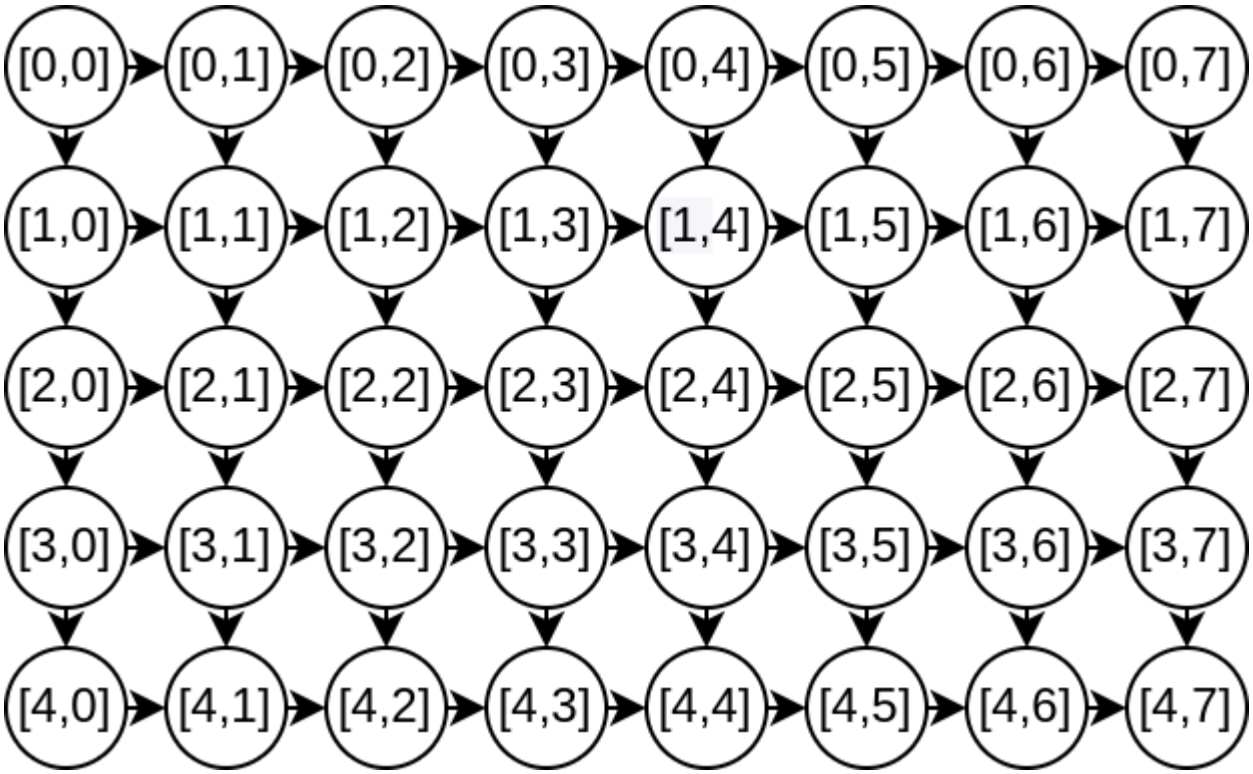
We compared the original scalar version and the vectorized version written using AVX2 compiler intrinsics. We emulated scatters, since they were not available. The runtime of the scalar version is 0.44 s, the vectorized version took 0.23 s. The vectorization definitely pays off.

Difficulty hard: loop-carried dependencies between rows and columns

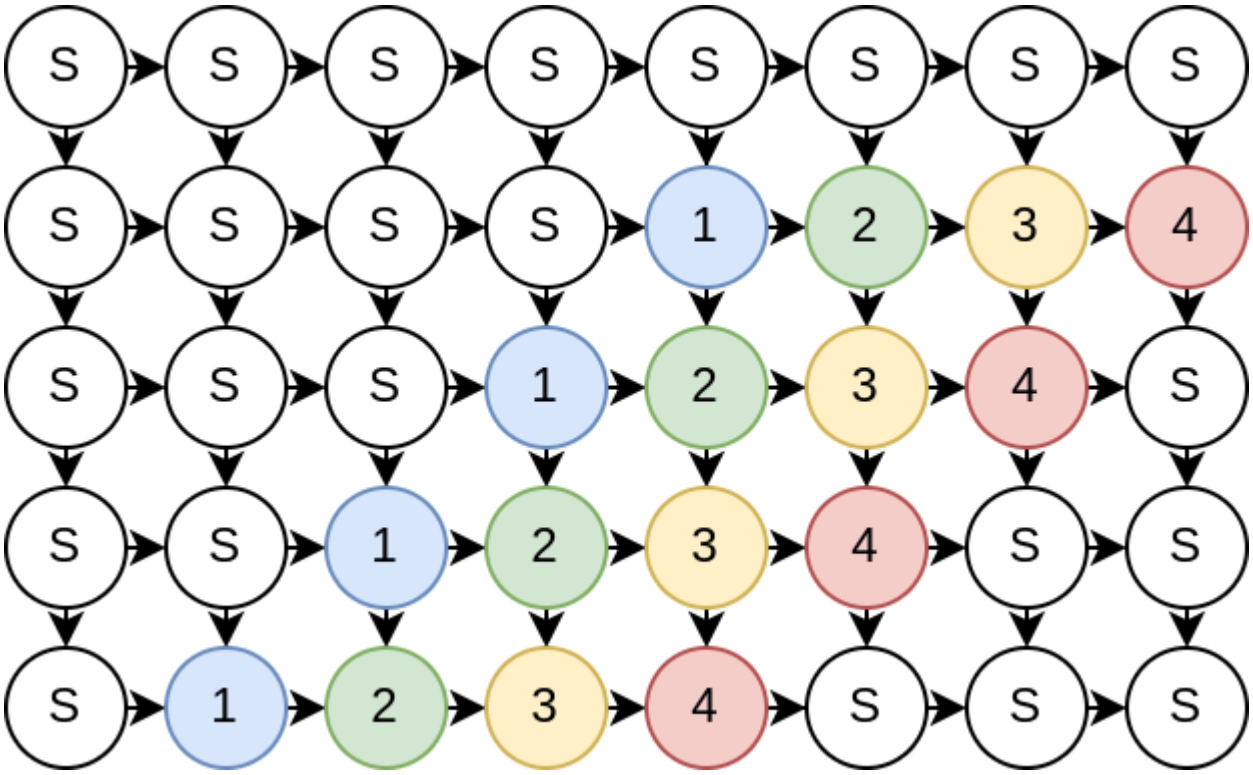
Let's increase the difficulty. We have the following loop[3]:

```
for (int i = 1; i < N; i++) {
    for (int j = 1; j < N; j++) {
        float min = std::min(res[i - 1][j], res[i][j - 1]);
        res[i][j] = min + A[i][j];
    }
}
```

In this case, we have two dependencies: the iteration `[i, j]` depends on iteration `[i, j - 1]` (previous column), but also on iteration `[i - 1, j]` (previous row). The dependency graph looks like this:



At first look, the graph suggests that vectorization is not possible. Everything depends on everything else. But this is not the case. We can vectorize the loop if we run it in the following order:



How the loop is actually executed when vectorized

We can execute iterations `[1, 4]`, `[2, 3]`, `[3, 2]` and `[4, 1]` simultaneously. `[1, 4]` depends on `[0, 4]` and `[1, 3]` ad they are already calculated. `[2, 3]` depends on `[1, 3]` and `[2, 2]` and these are already calculated, etc.

The nodes marked with S must be calculated using scalar code. The colored nodes can be executed using vector code. If the matrix is large enough, we should see speed improvements.

The problems with this approach are the same as the previous one. The compilers cannot automatically vectorize such codes, so they must be written using compiler intrinsics. In addition, gather and scatter instructions are needed as well.

We compared the original scalar version and the vectorized version written using AVX2 compiler intrinsics. We emulated scatters, since they were not available. The runtime of the scalar version is 0.85 s, the vectorized version took 0.45 s. The vectorization again pays off.

Where else can we apply outer loop vectorization?

As we've seen, one of the main reason to use outer loop vectorization is when inner loop vectorization is impossible (due to dependencies). Another reason when to use it is when inner loop vectorization is inefficient. Take the following example:

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
       a[i][j] = b[j][i] * c[index[j]];
   }
}
```

In this example, the inner loop vectorization is possible, but it doesn't pay off. This is because the memory access pattern for variables `b[j][i]` and `c[index[j]]` is inefficient. Every time the value of `j` changes by one, access to `b[j][i]` moves `n` elements in memory. For the array `c[index[j]]`, every time `j` changes by one, the CPU is accessing random memory address.

With outer loop vectorization, we run four or eight instances of the inner loop simultaneously. For example, in a single vectorized iteration, the CPU will be executing scalar iterations [i,j] for values of [i,j] between [0,0] to [7,0]. When doing it like this, the access pattern for `b[j][i]` becomes more efficient (because `j` is constant); the same applies to the access pattern of `c[index[j]]` (again because `j` is constant).

Another reason for outer loop vectorization is when the inner loop has a low trip count. Consider the following example:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

In the case where the loop trip count for the inner loop is 2 or 3 (`m = 2` or `m = 3`), the compiler will emit vectorized code but it will never run because the vector length is four. In this case, outer loop

vectorization can run four instances of the inner loop simultaneously. This is because the inner loop iterator `j` increases by one between vectorized iterations.

Why does outer loop vectorization matter?

In our examples, outer loop vectorization was limited to matrices. But its usefulness is much greater: it can be used to improve the access speed to hash maps, trees, linked lists and other data structures. Instead of performing an operation on one by one piece of data, the same operation is performed on several pieces of data at once.

For example, when performing a look up in a hash map for a collection of keys, instead of calculating the hash value one by one for each key, the vectorized code can calculate hash values for four or eight keys simultaneously and perform four or eight lookups at once. Or, instead of calculating `strlen` for one string, the user can run four calculations in parallel using outer loop vectorization.

Even though these solutions can result in speed improvements, and when the good memory access pattern is present with substantial speed improvements, in practice these solutions are rarely seen for a few reasons:

- People don't know about them.
- Most compilers do not support outer loop vectorization so writing this code is possible only using compiler intrinsics. This code is much more difficult to understand and maintain.
- Gather and scatter instruction are not present on many architectures.
- Gather and scatter instructions come with limitations. You cannot address any random address in memory, all the memory the instruction is accessing needs to be inside the same region (whose size is between 4 GB and 32 GB). This requires more complicated memory layouts or memory pools. Also, it doesn't scale if the amount of data to be processed cannot fit the memory block.

Final Words

Although dependencies are limiting factor in vectorization, not all dependencies are created equal. Outer loop vectorization can really make the CPU busy doing useful work in case dependencies are blocking vectorization. Too bad the compilers don't support this readily 😳

An extension of outer loop vectorization is applying the same operation to a collection of data: hash map lookups on a collection of keys, calculating string length for a collection of strings, etc. This is a powerful technique to improve the speed of such operations, although it comes with many limitations we talked about.