5 SIMD – Single Instruction Multiple Data

According to Flynn's taxonomy [9], a Single Instruction Multiple Data (SIMD) processor exhibits data parallelism by providing instructions that operate on entire blocks of data, called *vectors*. This is in contrast with scalar instructions that operate on single data items, one at a time.

In the early 1970s, vector supercomputers were designed around this concept. Through a single instruction, they performed the same basic operation on an entire vector. Nowadays, this vector technology is mainstream and is more commonly known as "SIMD," but the underlying principles are the same. With a single instruction, multiple elements are updated concurrently.

Through the SIMD support in OpenMP, the user has an easy and portable way to express this instruction-level parallelism. SIMD is tightly integrated with the OpenMP threading model, supporting multi-level parallelism. In this chapter, the OpenMP SIMD constructs are covered in detail.

5.1 An Introduction to SIMD Parallelism

In OpenMP, vectorization is referred to as *SIMD parallelism*, or "SIMD" for short. SIMD provides data-parallelism at the instruction level: a single instruction operates upon multiple data elements concurrently. SIMD instructions use special *SIMD registers* containing multiple data elements. The width of these registers determines the *vector length*, which is the number of scalar data items that can be processed in parallel by a SIMD instruction.

Figure 5.1 illustrates the SIMD concept. Two arrays b and c are added together, and the result is stored into the array a. For the purpose of this example, the arrays have 16 elements only. The loop that implements this operation requires 32 load instructions, 16 add instructions, and 16 store instructions.[1] On a system with SIMD instructions with a vector length of four, it takes only 8 load instructions, 4 add instructions, and 4 store instructions.

The advantage is that the SIMD instructions are (almost) as fast as their scalar counterpart. In other words, the number of processor cycles needed to add 4 elements in SIMD mode is the same as the time it takes to execute one scalar add instruction. This means that, in this case, the processor time is reduced by a factor

[1] The few instructions needed to execute the loop, the "loop overhead," are not considered here.

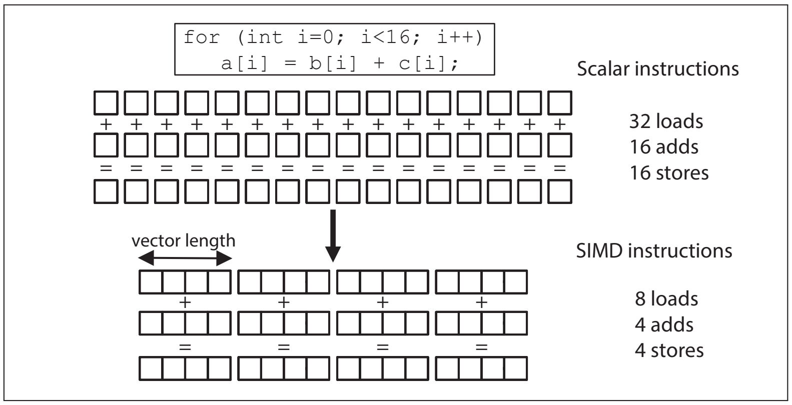


Figure 5.1: **An example of vectorization** – Vector instructions improve the performance by processing multiple data items concurrently.

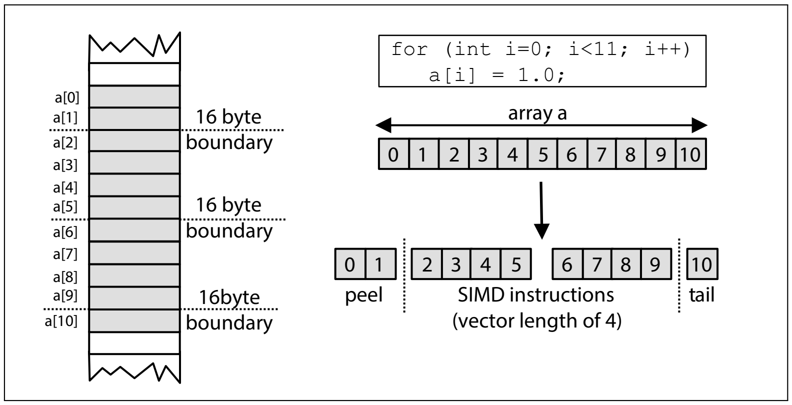
of four. This is an upper limit, however. In practice, the improvement also depends upon the time it takes to move the data.

A compiler may be able to identify which operations are suitable for optimization using SIMD instructions, but it must prove that it is safe (or legal) to do so. Some of the challenges that a compiler encounters when trying to automatically generate SIMD instructions include: imprecise dependence information, data layout and alignment, conditional execution, packing and unpacking of scalar data items into vectors, calls to functions, and loop bounds that are not always an even multiple of the vector length.

In particular, C/C++ pointer variables pose a challenge to a compiler's dependence analysis. Different pointers may be aliases for the same memory block, resulting in seemingly independent operations having an implied dependence. Compilers must start with an assumption that a dependence exists between two accesses to memory and then attempt to prove that there is not one.[2]

Once it has been determined that a loop is safe to vectorize, there are some more issues to consider. To get the full benefit from SIMD, the starting address of the vectors may need to be aligned on the correct boundary. This means that the address in memory must be a multiple of the vector length in bytes.

[2]This is explained in more detail in the Glossary.



```
for (int i=0; i<11; i++)
   a[i] = 1.0;
```

Figure 5.2: **Loop modifications needed to vectorize a loop using SIMD instructions** – To enforce alignment and ensure the remaining loop length is a multiple of the vector length, the compiler may need to peel off iterations and also treat the tail end separately.

If this is not the case, the compiler uses *loop peeling* to process the first element(s) separately, such that the first element processed with SIMD instructions starts on the correct boundary in memory. The vector is then processed in chunks that are a multiple of the vector length. If the number of loop iterations that remain after loop peeling is not a multiple of the vector length, the remaining items are processed in the *tail* part of the loop.

This is illustrated in Figure 5.2, where the array `a` with 11 elements is processed. This array is of type `float` and each element takes 4 bytes in memory. On a system with a SIMD register width of 16 bytes and with the memory-layout shown here, the first two elements need to be processed separately. The next 8 elements require two SIMD instructions. The remaining last element is handled separately.

The parts of the array processed separately are relatively inefficient. In most cases, the need to handle these situations can only be detected during program execution. In Section 5.2.5 it is shown which controls OpenMP provides to pass on more information to the compiler, which it can use to generate more efficient code.

```
#pragma omp simd [clause[[,] clause]...] new-line
      for-loops
```

```
!$omp simd [clause[[,] clause]...]
      do-loops
!$omp end simd
```

Figure 5.3: **Syntax of the simd construct in C/C++ and Fortran** – The `simd` construct is the fundamental construct to express SIMD parallelism in a loop.

5.2 SIMD loops

The OpenMP compiler may transform a loop that is marked with the `simd` construct, into a *SIMD loop*. As a result, multiple iterations of the loop may be executed concurrently by a single thread. A SIMD loop of length n, consists of the logical iterations $0, 1, \ldots, n-1$. The numbering denotes the sequential order in which loop iterations are executed.

A *SIMD chunk* refers to the set of iterations that are executed concurrently by a SIMD instruction in a single thread. Within a chunk, each iteration is executed by a *SIMD lane*, which refers to the mechanism that a SIMD instruction uses to process one data element. The number iterations in a SIMD chunk is the vector length.

5.2.1 The Simd Construct

The `simd` construct applies to one or more subsequent loops. The structure of the loops on which the `simd` construct is placed must conform to the same restrictions that are required by the `for` (`do` in Fortran) construct. The `simd` construct syntax in C/C++ and Fortran is given in Figure 5.3. The clauses that are supported by the `simd` construct are listed in Figure 5.4.

The OpenMP specification describes the execution model of the `simd` construct as follows: "When an OpenMP thread encounters a `simd` construct, the iterations of the loop associated with the construct may be executed concurrently using the SIMD lanes that are available to the thread." Intentionally, this allows for much freedom in the implementation.

The code in Figure 5.5 sums the two arrays pointed to by `b` and `c` and stores the result in the array pointed to by `a`. All three arrays are of length n. For this operation, all loop iterations are independent and may be executed concurrently.

private *(list)*
lastprivate *(list)*
reduction *(reduction-identifier : list)*
collapse *(n)*
simdlen *(length)*
safelen *(length)*
linear *(list[:linear-step])*
aligned *(list[:alignment])*

Figure 5.4: **Clauses supported by the simd construct** – The semantics of the `private`, `lastprivate`, `reduction`, and `collapse` clauses have been extended to the `simd` construct and are described in the main text. The `simdlen` clause is described in Section 5.2.2. The `safelen` clause is described in Section 5.2.3, The `linear` clause is described in Section 5.2.4. The `aligned` clause is described in Section 5.2.5.

```
void simd_loop(double *a, double *b, double *c, int n)
{
  int i;

  #pragma omp simd
  for (i=0; i<n; i++)
     a[i] = b[i] + c[i];
}
```

Figure 5.5: **Example of the simd construct** – The vectors `b` and `c` are added and the result is stored in vector `a`.

This can be exploited by threads, for instance with the `for` construct, or with a SIMD loop (or with both).

Adding the OpenMP `simd` construct instructs the compiler to generate a SIMD loop. If, in fact, the pointer variable `b` or `c` is an alias for the pointer variable `a`, adding the `simd` construct to this loop would be a user error, the resulting behavior would be undefined, and incorrect results should be expected.

In this particular case, no further clauses are necessary. Similar to the `for` construct, the loop iteration variable `i` is `private`. However, with the `simd` construct, one private instance will be created per SIMD lane. The memory pointed to by `a`, `b`, and `c` (the arrays) is shared. The compiler is free to select the appropriate vector length that is suitable for the target architecture.

```
void simd_loop_private(double *a, double *b, double *c, int n)
{
  int i;
  double t1, t2;

  #pragma omp simd private(t1, t2)
  for (i=0; i<n; i++)
  {
    t1 = func1(b[i], c[i]);
    t2 = func2(b[i], c[i]);
    a[i] = t1 + t2;
  }
}
```

Figure 5.6: **Example using private variables in a simd construct** – The two function calls return intermediate results that must be stored in private variables to avoid a data race. Each SIMD lane has its own instance of a private variable.

A `simd` construct has a data environment with shared and private variables. The `private` and `lastprivate` clauses modify the `simd` construct's data environment. In Section 1.2.3, the concept of an "execution instance" was introduced. In the context of the `simd` construct, the execution instance is a SIMD lane, and one instance of each variable is created per SIMD lane.

The code fragment in Figure 5.6 illustrates the use of the `private` clause. The two function calls to `func1()` and `func2()` return intermediate results that are stored in the variables `t1` and `t2`, respectively. Because loop iterations are executed simultaneously by SIMD lanes, both variables must be privatized in order to avoid data races. The privatization is accomplished by listing the variables `t1` and `t2` in the `private` clause. Note that in this particular example, the same effect may be achieved by declaring both variables inside the loop.

The loop iteration variable `i` is privatized by default. However, in a `simd` construct, it is treated as `lastprivate`. After the loop, the original loop iteration variable contains the final value of the private loop iteration variable that was computed in the last sequential iteration.

```c
void simd_loop_collapse(double *r, double *b, double *c,
                        int n, int m)
{
  int i, j;
  double t1;

  t1 = 0.0;
  #pragma omp simd reduction(+:t1) collapse(2)
  for (i = 0; i<n; i++)
      for (j = 0; j<m; j++)
          t1 += func1(b[i], c[j]);
  *r = t1;
}
```

Figure 5.7: **Example of the reduction and collapse clauses on the simd construct** – The two perfectly nested loops are collapsed into a single iteration space. The `reduction` clause is required to parallelize the accumulation into `t1`.

An instance of a private variable is not initialized and no assumption about its initial value can be made. Further, the lifetime of a private variable is restricted to the `simd` construct, and there is no method for accessing the value of a private variable after the region completes.

Variables that appear as list items in a `lastprivate` clause on a `simd` construct are private. As explained in Section 1.2.3, the final value of the private variable that is computed in the last sequential iteration is available after the construct in the original variable that the `lastprivate` variable corresponds to.

The `reduction` clause described in Section 2.4.3, on page 93 works similarly in the context of the `simd` construct. The `reduction` clause takes a list of variables and the reduction operator as its arguments. For each variable in the list, a private instance is used during the execution of the SIMD loop. The partial results are accumulated into the private instance. The reduction operator is applied to combine all partial results, such that the final result is returned in the original variable and made available after the SIMD loop.

The `collapse` clause described in Section 2.1.3, starting on page 43, works the same on the `simd` construct. The clause takes a positive integer number as its argument that indicates the number of loops that are collapsed into one iteration space. All the loop iteration variables in the associated collapsed loops are `lastprivate`.

```
unsigned int F(unsigned int *x, int n, unsigned int mask)
{
   #pragma omp simd simdlen(32/sizeof(unsigned int))
   for (int i=0; i<n; i++) {
      x[i] &= mask;
   } // End of simd region
}
```

Figure 5.8: **Example of the simdlen clause on the simd construct** – The `simdlen` clause is a hint to guide the compiler in the selection of a vector length for the loop.

The use of both the `reduction` and `collapse` clauses is illustrated in Figure 5.7. Through the `collapse` clause, the two loops are merged into a single loop. The `reduction` clause is used to correctly compute the variable `t1`.

The `collapse` clause should be used with caution in the context of vectorization, because it increases the complexity of the generated SIMD code. We recommend to verify that such a loop is vectorized as intended.

Many modern compilers provide a feature known as "compiler commentary," which provides information about the optimizations, or lack thereof, performed.[3] If vectorization is supported on the target processor, messages related to this may be helpful in determining how to effectively use the OpenMP `simd` constructs. We recommend checking the documentation of the compiler to determine if this feature is supported and, if so, how to enable it.

5.2.2 The Simdlen Clause

The `simdlen` clause takes a constant positive integer value as its argument. This value specifies the preferred number of iterations to be executed concurrently. It impacts the vector length used by the generated SIMD instructions.

The value in the clause is a preference. The compiler has the freedom to deviate from this choice and to choose a different length. In the absence of this clause, an implementation-defined default value is assumed for the vector length. The purpose of the `simdlen` clause is to guide the compiler. Perhaps the user has more informa-

[3]Success, or failure, of vectorization also depends on the compiler options used. Check the documentation of the compiler for relevant options to consider.

```
void dep_loop(float *a, float c, int n)
{
  for (int i=8; i<n; i++)
    a[i] = a[i-8] * c;
}
```

Figure 5.9: **Example of a loop-carried dependence** – The loop-carried dependence creates a limit on the vector length.

tion on the loop characteristics and knows that a specific length may be beneficial to performance. An incorrect choice may negatively impact the performance but will not lead to an incorrect result. Figure 5.8 is a code example that uses the `simdlen` clause on a `simd` construct.

5.2.3 The Safelen Clause

A limit on the vector length used by SIMD instructions is sometimes required. An example of this is when vectorizing a loop with loop-carried dependences. As shown in the code in Figure 5.9, an operation in a previous loop iteration must complete before an operation in the current loop iteration can execute. In this case, the read of `a[i-8]` on the current iteration i cannot execute until the write of `a[i]` on the previous $i-8$ iteration is completed.

The *loop-carried dependence distance* is the number of loop iterations between a previous and current iteration, which in this example is 8. It is a distance between iterations, specifying how far a dependency is in the iteration space. When using SIMD parallelism, the vector length must be less than or equal to the distance of the smallest loop-carried dependence in the loop.

In this particular example, a vector length of up to 8 could be used. The maximum safe distance between loop iterations can be specified through the `safelen` clause.

The `safelen` clause takes a constant positive integer value as its argument. The value for `safelen` provides an *upper limit* on the vector length. This number specifies the length in which it is safe to vectorize the loop. The vector length ultimately selected by the compiler is still implementation-defined, but it does not generate SIMD code using a vector length that exceeds the `safelen` value.

```
1 void simd_loop_safelen(double *a, double *b, double *c, int n,
2                        int offset)
3 {
4   int i;
5   #pragma omp simd safelen(16)
6   for (i=offset; i<n; i++)
7     a[i] = b[i-offset] + c[i];
8 }
```

Figure 5.10: **Example of the safelen clause on the simd construct** – The `safelen` clause ensures that a vector length of up to 16 is correct.

The code fragment in Figure 5.10 illustrates the use of the `safelen` clause. The accesses to the elements of array `b` use the variable `offset`. Unless the compiler can determine the value of this variable, it may select a vector length that generates incorrect results. In this case, the user has more knowledge of the value. The value 16 is specified in the `safelen` clause at line 4. This means that the user guarantees that the loop can be safely vectorized using a vector length of 16 or less.

If the clause is not specified, the assumed value for `safelen` is the number of loop iterations. Note the difference between the `safelen` and `simdlen` clauses. The `safelen` clause is required for correctness while the `simdlen` clause indicates a preference.

5.2.4 The Linear Clause

Scalar data elements are packed into vectors, operated on collectively as a vector by SIMD instructions, and then unpacked. When the scalar data elements are accessed in a linear fashion, it is easy to see how the vector is constructed. For example, assuming that alignment restrictions have been satisfied, a single SIMD load or store instruction may be used to read and write consecutive scalar data elements as vectors.

In situations in which the scalar data elements are arranged consecutively in memory, the accesses to the scalar data elements are said to occur with a stride of one (*unit stride*). If the scalar data elements are not arranged consecutively in memory, but instead are at a regular offset from each other, then the data elements may be accessed with a stride greater than one. In this case, the stride is equal to the offset between the scalar data elements.

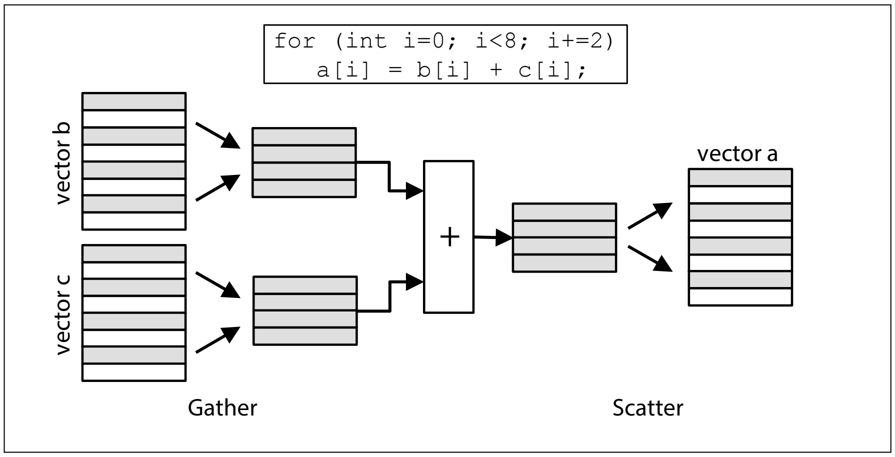


Figure 5.11: **Gather and scatter scalar data elements in and out of a vector** – Scalar data elements are gathered into vectors, operated on as a vector, and then scattered back to their destination locations.

To construct a vector, a *gather* operation reads scalar data elements from memory linearly but with a stride greater than one. Likewise, a *scatter* operation writes the scalar data elements in a vector back to memory linearly with a stride greater than one. The gather and scatter vector operations are illustrated in Figure 5.11. Some architectures have SIMD load and store instructions that support gather and scatter operations.

The best situation is when the vectors are accessed with a linear stride of one. The next best scenario is when the access pattern is linear but with a stride that is greater than one and gather and scatter instructions may be used. The worst case scenario is when no linear access pattern can be determined and the scalar data elements must be individually packed into and unpacked from vectors.

The `linear` clause is provided for the user to indicate the linear behavior of a variable in a loop. The compiler may use this information to determine the most efficient way to pack and unpack scalar data elements into vectors.

The `linear` clause is a data-sharing clause that provides a *superset* of the functionality of the `private` clause. On a loop or `simd` construct, the syntax of the clause is `linear` *(list[:linear-step])*. The `linear` clause accepts a list of variables as its argument. An optional, colon-separated, *linear-step* (stride) is supported.

```
void simd_loop_linear(double *a, double *b, double *c, int n,
                      int offset)
{
  int i, j = 0;

  #pragma omp simd linear(j:1)
  for (i=offset; i<n; i+=2)
    a[i] = b[j++] + c[i];
}
```

Figure 5.12: **Example of the linear clause on the simd construct** – The `linear` clause defines how the `j` index variable relates to the loop variable `i` that is used as an index into some of the arrays.

The `linear` clause may appear on the `simd` or `for` (`do` in Fortran) constructs. In C, a variable that appears in the clause must have an integral or pointer type. In C++, the variable must either have an integral or pointer type or be a reference to an integral or pointer type. In Fortran, the variable must have an `integer` type. If a *linear-step* expression is specified, it must be invariant during the execution of the associated loop.

The semantics for the `private` clause apply: every variable listed in the clause is private in the associated construct. For the `simd` construct, it is private to a SIMD lane. In addition, this clause asserts that a variable has a linear relationship with the iteration space of the loop to which the clause applies. The value of the private variable in each loop iteration is defined as the value of the original variable, before the construct was entered, plus the logical number of the loop iteration times the *linear-step* or 1 if no *linear-step* is given. After the loop, the original variable has the final value of the private variable from the sequentially last iteration.

A common use case of the `linear` clause is a loop, where in addition to the loop variable, additional variables are used to index into arrays. If such variables have a linear relationship with the loop variable, this clause may be used. A simple example that demonstrates the use of this clause is shown in Figure 5.12.

Arrays `a` and `c` are accessed through the loop variable `i`, but array `b` is indexed through another variable, `j`. The variable `j` has a linear relationship with the loop iteration variable `i`, which is incremented by 2 in each iteration, while `j` is incremented by 1. Variable `i` takes the values `offset, offset+2, offset+4,...`.

The sequence for `j` is `0, 1, 2, ....` By asserting the linear properties of the variable `j`, the compiler can more easily determine the access pattern for the array `b` and thus generate a SIMD instruction that reads `b` as a vector. Notice that a gather operation may be use to read the array `c` and that a scatter operation may be used to write to the array `a`.

The user may ask if the `linear` clause is necessary because a compiler can very often easily determine the linear behavior of a variable in a loop. The `linear` clause may also appear on `declare simd` directive, where it is used to specify the linear behavior of a function parameter (see Section 5.2.2). It is perhaps more useful in this capacity where the compiler may not be able to automatically determine the linear behavior of a function parameter.

5.2.5 The Aligned Clause

Data alignment is important for good performance. If a data element is not aligned on an address in memory that is a multiple of the size of the element in bytes, an extra cost is incurred when accessing the element. For example, on some architectures it may not be possible to load or store from a memory address that is not aligned to the size of the object being accessed. Instructions that do this are sometimes referred to as non-aligned loads and stores. Even if an architecture has non-aligned load and store instructions, they may execute with a higher cost to performance.

In many cases, alignment is handled by the compiler, but in certain scenarios this is impossible. As an example, consider processing a stream of data at the bit level. The program interprets this stream in terms of 32-bit integers. Depending on where in the stream the processing begins, the data may or may not be aligned.

Often, there are ways to enforce the correct alignment. For example, if two arrays are used in a loop, their relative alignment may be improved by adjusting the dimensions. There are also specific functions to enforce a certain alignment. An example is the `posix_memalign()` function from the POSIX standard. It allocates a memory block on a specific boundary. Other options are to consider vendor-specific features. We recommend checking the vendor's documentation for the details.

The alignment issues are the same when using SIMD instructions to vectorize loops. Figure 5.2 on page 223 illustrates the difference between aligned and unaligned data access in a vector loop. As explained there, in the case of a misalignment, iterations need to be "peeled off" to enforce alignment.

If vectorization is requested through the use of the OpenMP `simd` construct, but the resulting performance is poor, or the compiler commentary indicates issues in the vector code generation, then adjusting the data alignment may improve the performance.

The `aligned` clause is supported on both the `simd` construct, as well as the `declare simd` directive. This clause requires a list of variables as its argument, and an optional, colon-separated, *alignment*. The alignment must be a constant positive integer, and an implementation-defined default value is assumed if it is not given in the clause. In C, a variable that appears in the clause must have an array or pointer type. In C++, a variable that appears in the clause must have array, pointer, reference to array, or reference to pointer type. In C/C++, all the variables in the list are guaranteed to point to an object that is aligned in memory at an address that is a multiple of the number of bytes specified by the alignment. In Fortran, all the variables in the list are guaranteed to be aligned in memory at an address that is a multiple of the number bytes specified by the alignment.

If the alignment assumption is invalid and one or more variables do not have the alignment attributes specified, the behavior is implementation-dependent. It is highly recommended to avoid this type of error.

In the example code shown in Figure 5.13, the `align` clause appears on the `simd` construct at line 5. It asserts that the value of the pointer variable x is always aligned to a 16-byte boundary. Assuming that the size of the `float` data type is 4 bytes and knowing the alignment of the address in x is 16 bytes, the compiler has the information it requires to generate aligned 128-bit vector load and store instructions. It is important to note that the code is dependent on the assertion that value of `x` is always aligned to at least a 16-byte boundary.

5.2.6 The Composite Loop SIMD Construct

One of the main reasons for OpenMP to include support for SIMD parallelism is to cleanly define the interaction between vectorization through SIMD and thread-level parallelism, such as with the loop construct. Before support for SIMD was introduced in OpenMP 4.0, vendor-proprietary constructs had to be mixed with OpenMP. This not only violated portability, but there were also unwanted side-effects on performance.

```
void f_aligned(float *x, float scale, int n)
{
  int i;

  #pragma omp simd aligned(x:16)
  for (i=0; i<n; i++)
    x[i] = x[i]*scale;
}
```

Figure 5.13: **Example of the aligned clause on the simd construct** – The `aligned` clause asserts that the value of the pointer variable `x` is always aligned on a 16 byte boundary. With this information, the compiler may generate wider and more efficient vector load and store instructions.

#pragma omp for simd *[clause[[,] clause]. . .] new-line*   *for-loops*
!$omp do simd *[clause[[,] clause]. . .]*   *do-loops* **!$omp end do simd**

Figure 5.14: **Syntax of the loop simd construct in C/C++ and Fortran** – The composite construct combines thread and SIMD parallelism. Chunks of loop iterations are distributed to the threads in a team. The chunks of iterations are then executed with a SIMD loop. A clause that may appear on either the loop construct or the `simd` construct may appear on the composite clause.

In OpenMP, the `simd` construct may be combined with the loop construct, resulting in the `for simd` (`do simd` in Fortran) composite construct.[4] In the same way as the OpenMP `simd` construct, the `for simd` construct applies to the subsequent loop, and the same restrictions outlined earlier must be adhered to, in addition to possible restrictions from the `for` or `do` construct. The syntax for the `for simd` construct in C/C++ and the `do simd` construct in Fortran is given in Figure 5.14.

The composite construct addresses the following question: which is performed first, vectorization and then thread-level parallelism, or vice versa?

With the `for simd` construct, chunks of loop iterations are first distributed across the threads in a team by a method that is determined by any clauses that apply

[4]The difference between a composite and a combined construct is explained in Section 6.5.2 starting on page 283.

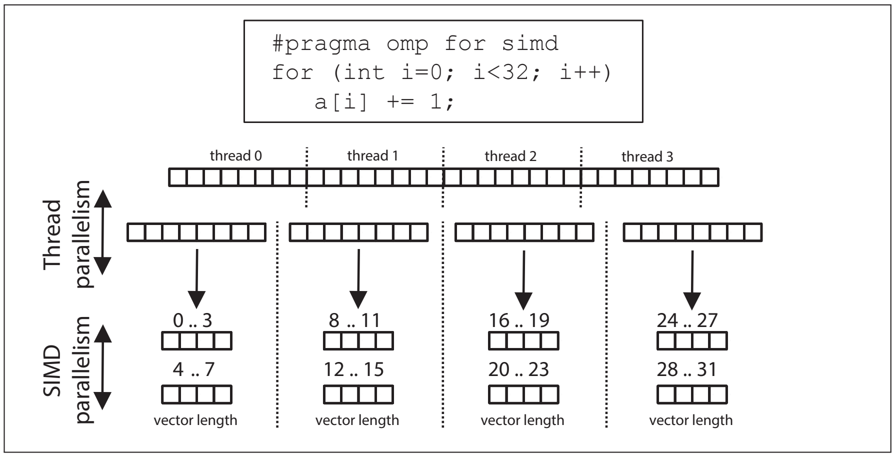


Figure 5.15: **Combining thread and SIMD parallelism** – Thread and SIMD parallelism are used to execute a loop.

to the `for` construct. Then, the chunks of loop iterations may be converted into SIMD loops in a way that is determined by any clauses that apply to the `simd` construct. The process is the same when using the `do simd` construct in Fortran. The distribution of loop iterations across threads and then SIMD loops is illustrated in Figure 5.15.

When using the composite construct, the user must be aware that both the number of threads, as well as the scheduling of the worksharing construct, may influence the efficiency of the resulting SIMD loop. Furthermore, the user must consider the effect of a private variable in the context of a composite construct. There is a private instance of the variable per SIMD lane.

Each loop has a certain amount of work associated with it. If the number of threads is increased, the amount of work performed per thread is reduced. Adding SIMD parallelism to this does not necessarily improve the efficiency. Especially if the SIMD loops that each thread works on reduce in length as the thread count increases. Most likely, some experimentation to find the optimal combination may be required.

The loop schedule of the worksharing construct affects the opportunities for vectorization. For good performance this must be taken into account. If the `static` schedule is used with a small chunk size, or if the `dynamic`, or `guided`, schedule is

```
 1 void func_1(float *a, float *b, int n)
 2 {
 3   #pragma omp for simd schedule(static, 5)
 4   for (int k=0; k<n; k++)
 5   {
 6     // do some work on a and b
 7   }
 8 }
 9
10 void func_2(float *a, float *b, int n)
11 {
12   #pragma omp for simd schedule(simd:static, 5)
13   for (int k=0; k<n; k++)
14   {
15     // do some work on a and b
16   }
17 }
```

Figure 5.16: **SIMD loops without and with the simd schedule modifier** – The `simd` schedule modifier in `func_2()` guarantees that a preferred implementation-defined vector length is respected when distributing the loop

used, the SIMD efficiency may not be optimal. The reason is that the number of iterations per thread may be too small for SIMD to be beneficial. The compiler may not be able to generate the most efficient SIMD code because of loop tails and poor memory access patterns. Ideally, the selected chunk size for the threads is a multiple of the vector length.

The code in function `func_1()` in Figure 5.16 shows an example of a problematic chunk size. Each thread gets assigned a loop with only 5 iterations. This loop is vectorized, but it is very short, and depending on the vector length of the target architecture, this may lead to inefficiencies.

To address this issue, the `simd` schedule modifier is used at line 12 in Figure 5.16. For the loop in function `func_2()`, the modifier results in an adjustment of the chunk size. If `chunk_sz` denotes the chunk size, the formula to adjust this for a given vector length is given by: $\mathit{ceiling}(\texttt{chunk\_sz}/\texttt{simd\_len}) \times \texttt{simd\_len}$, with `simd_len` the vector length of the target architecture. For example, if the vector length is eight `float` elements, the resulting chunk size is increased from 5 to 8.

```
 1 double compute_pi(int n)
 2 {
 3   const double dH = 1.0 / (double) n;
 4   double dX, dSum = 0.0;
 5
 6   #pragma omp parallel for simd private(dX) \
 7                       reduction(+:dSum) schedule(simd:static)
 8   for (int i=0; i<n; i++) {
 9     dX = dH * ((double) i + 0.5);
10     dSum += (4.0 / (1.0 + dX * dX));
11   }
12   // End parallel for simd region
13
14   return dH * dSum;
15 }
```

Figure 5.17: **Example that uses the composite for simd construct** – The numerical solution for integration has a compute-intensive loop that is parallelized and vectorized.

An example that combines many of the topics covered in this section is shown in Figure 5.17. The number π can be approximated by computing the integral $\int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$ through numerical integration. The `compute_pi()` function implements a simple numerical solution to this problem.

In Figure 5.17, the combined use of a `parallel` construct and the composite `for simd` construct is illustrated by placing the `parallel for simd` directive before the loop that spans lines $8 - 11$. Variable `dX` must be private, because it is modified in every loop iteration and the value differs for every SIMD lane. The computation of the summation at line 9 is a reduction operation that requires the `reduction` clause at line 7. This is an algorithm without load balancing issues and the `simd:static` schedule is most efficient.

5.2.7 Use of the Simd Construct with the Ordered Construct

The `ordered` clause is described in detail in Section 1.6.4. In short, the `ordered` region within a parallel loop is guaranteed to be executed in the original sequential order of the loop iterations. The code outside of this region is executed in parallel.

```
 1 extern int x;
 2 extern void global_update(int, int);
 3 #pragma omp declare simd
 4 extern int vec_work(int);
 5
 6 void F(int *a, int *b, int n)
 7 {
 8   #pragma omp simd
 9   for (int i=0; i<n; i++)
10   {
11     // vectorize this part of the loop
12     a[i] = vec_work(b[i]);
13
14     // execute this part of the loop in sequential order
15     #pragma omp ordered simd
16     {
17       global_update(x, a[i]);
18     } // End ordered simd region
19   } // End simd region
20 }
```

Figure 5.18: **An ordered region within a SIMD loop** – The `ordered` region is executed with one SIMD lane in the original sequential order of the loop iterations.

Since OpenMP 4.5, the `simd` clause is supported on the `ordered` construct. Following the semantics of the loop construct, this clause has the effect that any encountering thread uses only one SIMD lane to execute the `ordered` region in the sequential order of the loop iterations. An example that uses the `simd` clause on the `ordered` construct is shown in Figure 5.18.

On line 12, the call to the function `vec_work()` is executed with SIMD parallelism. The call to the function `global_update()` at line 17 is executed in the original sequential loop iteration order by one SIMD lane at a time. This approach is efficient if the `ordered` region contains a small portion of the code that is not critical to the overall runtime.

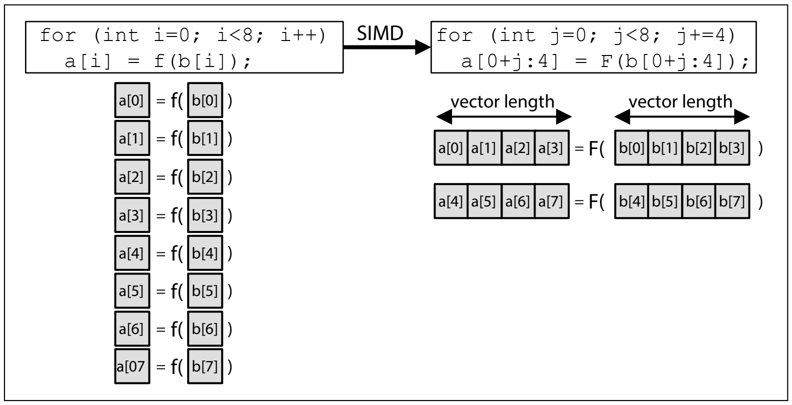


Figure 5.19: **Illustration of function calls with SIMD** – The scalar function `f()` is modified and renamed to `F()` in this example. This function supports vector input arguments and returns an entire vector.

5.3 SIMD Functions

Function calls inside a SIMD loop obstruct the generation of efficient SIMD instructions. In the worst case scenario, the call to the function must be done using scalar data elements, which will most likely negatively impact the efficiency of the generated code.

To fully exploit SIMD parallelism, a function called from within a SIMD loop must be to a SIMD equivalent version of the function. This means that the compiler must generate a special version of the function with SIMD parameters and code that uses SIMD instructions.

The concept of a SIMD function is illustrated in Figure 5.19. The compiler generates a SIMD version of the function `f()`. The function's scalar parameters are converted to vector parameters. Scalar operations in the function body are replaced with corresponding vector operations. If the function `f()` is called outside the context of a simd region, the scalar variant is used.

This `declare simd` directive and its clauses are used to tell the compiler to generate one or more SIMD versions of a function. These specialized versions of a function may then be called from SIMD loops.

#pragma omp declare simd *[clause[[,] clause]. . .] new-line* *function declaration or definitions*
!$omp declare simd*[(proc-name)] [clause[[,] clause]. . .]*

Figure 5.20: **Syntax of the declare simd directive in C/C++ and Fortran** – The `declare simd` directive is used to declare that one or more versions of a SIMD function should be generated.

simdlen *(length)*
linear *(list[:linear-step])*
aligned *(list[:alignment])*
uniform *(argument-list)*
inbranch
notinbranch

Figure 5.21: **Clauses supported by the declare simd directive** – The `simdlen`, `linear`, `aligned`, and `uniform` clauses, discussed in Section 5.3.2, declare SIMD attributes for function parameters. The `inbranch` and `notinbranch` clauses, described in Section 5.3.3, specify that a SIMD function variant is always called under a conditional branch and never called under a conditional branch.

5.3.1 The Declare Simd Directive

The `declare simd` directive is used to declare that a SIMD variant of a function may be called from a simd region. The syntax for this directive in C/C++ and Fortran is shown in Figure 5.20. The clauses that may appear on the `declare simd` directive are listed in Figure 5.21.

When the `declare simd` directive is placed before a function declaration, it asserts that a SIMD variant of the function, with characteristics that are described by the clauses on the directive, may be called from within a loop in a simd region. The user may think of this like a special type of SIMD function prototype. If the `declare simd` directive is placed in the same translation unit as the definition of the function, then a SIMD variant of the function is generated by the compiler.

The compiler may generate multiple versions of a SIMD function and select the appropriate version to invoke at a specific call-site in a `simd` construct. The user may tailor a SIMD function variant by using clauses on the directive. For example, different SIMD versions of a function may be specialized for either different SIMD instruction widths or other function parameter attributes.

```
 1 #pragma omp declare simd
 2 double my_func(double b, double c)
 3 {
 4   double r;
 5   r = b + c;
 6   return r;
 7 }
 8
 9 void simd_loop_function(double *a, double *b, double *c, int n)
10 {
11   int i;
12   #pragma omp simd
13   for (i=0; i<n; i += 2)
14   {
15     a[i] = my_func(b[i], c[i]);
16   }
17   // End simd region
18 }
```

Figure 5.22: **Example of the declare simd directive** – The function `my_func()` is called from within a `simd` construct. The `declare simd` directive declares that the compiler should generate a SIMD version of the function.

There are two restrictions. If a SIMD function has a side effect, the resulting behavior is undefined.[5] Furthermore, in C++, a function that appears under a `declare simd` directive is not allowed to throw any exceptions.

An example using the `declare simd` directive is shown in Figure 5.22. At line 1, the `declare simd` directive is used to inform the compiler that the function `my_func()` may be called from a simd region. This instructs the compiler to generate at least one additional SIMD version of the function `my_func()`. Because of the `simd` construct at line 12, SIMD instructions are used to execute the loop that spans lines 13 – 16. The call to `my_func()` at line 15 will be to a SIMD variant of the function that was declared at line 1.

[5] In the Glossary it is explained what a side effect is.

Note that in some examples in this section, the function call and the function definition are presented as if they were both in the same file (or translation unit). This is done to keep the examples simple.

Because the compiler can see the definition of the function and the place where the function is called, the user may wonder why the `declare simd` directive is needed. Depending on the compiler used, it may not be.

The `declare simd` directive is more critical when a function is defined in one file and called in another file. In that case, the compiler is instructed to generate a SIMD function variant even though it may not see the call to the function in a place where SIMD instructions are used.

5.3.2 SIMD Function Parameter Attributes

The `uniform`, `linear`, `simdlen`, and `aligned` clauses are used to specify attributes for SIMD function parameters. Except for the `simdlen` clause, the variables that appear in these clauses must be parameters of the function to which the directive applies.

When a parameter is listed in the `uniform` clause, it indicates that the parameter is passed an argument with the same value for all concurrent calls to the function in the execution of a single SIMD loop. This means that all SIMD lanes observe the same value for the parameter. If the arguments from consecutive loop iterations are passed as a vector to a SIMD variant of a function, then each element in the vector has the same value.

The `linear` clause has a different meaning when it appears on a `declare simd` directive. In this context, it is not a data-sharing clause as described in Section 5.2.4. Instead, it indicates that an argument passed to a function parameter has a linear relationship across the concurrent invocations of a function. Each SIMD lane observes the value of the argument in the first SIMD lane plus the offset of the SIMD lane from the first SIMD lane times the *linear-step*. For example, if the linear step is 2, there are 8 simd lanes, and the first simd lane observes the value 10, then the second lane observes the value 12, the third lane observes the value 14, and so on.

The example in Figure 5.23 shows the combined use of the `uniform` and the `linear` clauses on a function that is marked for vectorization with the `declare simd` directive. At line 14, the function `cosScaled` is called with the loop-invariant base address of the array `b`. The variable `c` is loop-invariant and the loop variable

```
#include <math.h>
#pragma omp declare simd uniform(ptr, scale) linear(idx:1)
double cosScaled(double *ptr, double scale, int idx)
{
  return (cos(ptr[idx]) * scale);
}

void simd_loop_uniform_linear(double *a, double *b, double c,
                              int n)
{
  int i;

  #pragma omp simd
  for (int i=0; i<n; i++) {
    a[i] = cosScaled(b, c, i);
  }
  // End simd region
}
```

Figure 5.23: **Example of the linear and uniform clauses on the declare simd directive** – The SIMD variant of the `cosScaled` function may assume that `ptr` and `scale` are loop invariant and that `idx` is incremented by 1 each time through the loop from where the function is called.

`i` is an offset into the `b` array. In the function `cosScaled`, the `ptr` parameter is a pointer and the `scale` parameter is a scalar.

The `uniform(ptr, scale)` clause informs the compiler to generate a SIMD function that assumes both of these parameters are passed arguments that are loop-invariant. The `idx` parameter appears in a `linear` clause, indicating to the compiler that the `idx` parameter is passed an argument that is incremented by 1 each time through the SIMD loop from where the function is called. By listing `ptr` in a `uniform` clause and `k` in a `linear` clause, the compiler can generate a unit stride SIMD load instruction for the write to `ptr[idx]`.[6]

Figure 5.24 is another example that uses the `uniform` and `linear` clauses to optimize the access to a multi-dimensional array within a function that is called from a SIMD loop. If the index in the first dimension is always the same, and the

[6]In [26], the vector code generated by the Intel compiler for a similar example is discussed.

```
#pragma omp declare simd uniform(x, y, d1, i, a) linear(j)
void saxpy_2d(float *x, float *y, float a, int d1, int i, int j)
{
  y[(d1*i)+j] = a*x[(d1*i)+j] + y[(d1*i)+j];
}
```

Figure 5.24: **Example using the uniform and linear clauses for multi-dimension array access** – Access in the first dimension is uniform. The loop that calls the `saxpy_2d()` function indexes linearly through the second dimension via the variable `j`.

```
#pragma omp declare simd simdlen(16)
char F(char x, char y, unsigned char mask)
{
   return (x + y) & mask;
}

void img_mask(char *img1, char *img2, int n, unsigned char *m)
{
  #pragma omp simd simdlen(16)
  for (int i=0; i<n; i++) {
    img1[i] = F(img1[i], img2[i], m[i]);
  } // End simd region
}
```

Figure 5.25: **Example of the simdlen clause on the declare simd directive** – Use the `simdlen` clauses on the `declare simd` directive to generate a SIMD variant of the function `F()` that has a vector length of 16.

outer (SIMD) loop progresses over the second dimension, then the index variable of the first dimension may appear in a `uniform` clause to assert that this variable has an invariant value.

A `simdlen` clause may appear on a `declare simd` directive and has a similar meaning as it does for the `simd` construct. The constant value that appears in the clause specifies a length for vectorized arguments. The clause is treated as a hint on the `simd` construct; on the `declare simd` directive, it is not. The compiler generates a SIMD version of the function, which expects its vector arguments to have a length specified in the clause. The function variant may be called from

```
#pragma omp declare simd linear(src,dst) \
                    aligned(src,dst:16) simdlen(32)
void copy32x8(char *dst, char *src)
{
   *dst = *src;
}

#pragma omp declare simd uniform(x,y) linear(i) \
                         aligned(x,y:64) simdlen(16)
float saxpy(float a, float *x, float *y, int i)
{
  return a * ((x[i]) + (y[i]));
}
```

Figure 5.26: **Example of the aligned clause on the declare simd directive** – Use the `aligned` clauses on the `declare simd` directive to generate SIMD variants of the functions `copy32x8()`, and `saxpy()` that expect the addresses passed in pointer arguments to have a specific byte alignment.

a SIMD loop whose corresponding `simd` construct has a `simdlen` clause with the same value. An example that does this is shown in Figure 5.25.

When a pointer variable appears in an `aligned` clause on a `declare simd` directive, it declares that the value of the pointer variable argument has the specified byte alignment. The SIMD version of the function may then use aligned vector memory accesses for the pointer variable. When the SIMD function is called from a simd region, the object pointed to by the pointer variable argument must be aligned to the specified byte boundary. This can be ensured by using the `aligned` clause on the `simd` construct that encloses the call to the SIMD function. Examples that use the `aligned` clause combined with other clauses on `declare simd` directives are shown in Figure 5.26.

The `linear` clause supports a *modifier* that provides additional capabilities for C++ and Fortran. The following description focuses on the semantics of the modifier in C++.[7] The syntax of the `linear` clause with a modifier is `linear`*(modifier(list)[:linear-step])*. The modifier may be `uval` or `ref`. The `uval`

[7]See the OpenMP 4.5 specifications for more details on using the `linear` clause with a modifier in Fortran.

```
#pragma omp declare simd linear(ref(x)) linear(uval(c))
void increment(int& x, int& c)
{ x += c; }

void Fref(int *a, int n)
{
  #pragma omp simd
  for (int i=0; i<n; i++) {
    increment(a[i], i);
  } // End simd region
}
```

Figure 5.27: **Example of the ref and uval modifiers in the linear clause** – The `ref` modifier declares that the address of `x` is linear. The `uval` modifier declares that the address of `c` is uniform, and its value is linear.

or `ref` modifier can be used only if the function parameter has a reference type. The modifier defines how the address and the value of a variable are observed across SIMD lanes as follows:

- `uval(x)`: The storage address of `x` is uniform. The value in the storage location `x` is linear.
- `ref(x)`: The storage address of `x` is linear.

A simple C++ example that uses the `ref` and `uval` modifiers in the `linear` clause on a `declare simd` directive is shown in Figure 5.27.

5.3.3 Conditional Calls to SIMD Functions

The `inbranch` and the `notinbranch` clauses declare whether or not a SIMD variant of a function is called conditionally from a simd region. There are no arguments for these clauses.

A conditional branch is a point in a program in which the execution control may be transferred to another point within the program. An example is an `if-then-else` statement. The `if` part is the branch (instruction). If the condition evaluates to `false`, execution is transferred to the `else` part.

In the presence of conditional branches, the generation of SIMD instructions using the full vector length may not be possible. Because a conditional branch

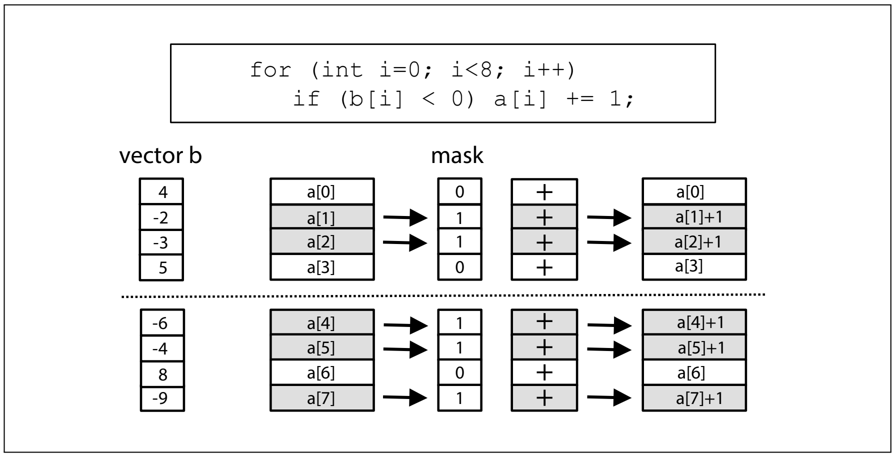


Figure 5.28: **Conditional control flow converted to masked vector instructions** – A vector mask predicate is used to enable or disable an operation on a given vector element. If the indexed mask is 1, the operation occurs. When it is 0, the operation is masked off.

creates an uncertainty regarding the number of consecutive elements available, it is more difficult for the compiler to generate SIMD instructions that use the full vector.

Depending on the target architecture, a compiler may generate *masked* vector code. Masked vector instructions use a bit vector, called "the mask," to ensure that the vector operation is applied only to those elements for which the mask bit is set to `true`. This concept is illustrated in Figure 5.28.

The `inbranch` clause asserts that the function is always called from within a conditional branch in a SIMD loop. With the `inbranch` clause present, the compiler must restructure the code to handle the possibility that a SIMD lane may not execute the code in the function. One approach to doing this is to pass the vector mask as an extra argument to the SIMD function variant.

The `notinbranch` clause may be used when the function is never called from within a conditional branch in a SIMD loop. The `notinbranch` clause enables the compiler to be more aggressive at optimizing the code in the function to use SIMD instructions. It can do this, because it does not need to consider the conditional execution of the instructions that execute in a SIMD lane.

```
 1 #pragma omp declare simd inbranch
 2 float do_mult(float x)
 3 {
 4   return (-2.0*x);
 5 }
 6
 7 #pragma omp declare simd notinbranch
 8 extern float do_pow(float);
 9
10 void simd_loop_with_branch(float *a, float *b, int n)
11 {
12   #pragma omp simd
13   for (int i=0; i<n; i++) {
14     if (a[i] < 0.0 )
15       b[i] = do_mult(a[i]);
16
17     b[i] = do_pow(b[i]);
18   } /* --- end simd region --- */
19 }
```

Figure 5.29: **Example of the inbranch and notinbranch clauses on the declare simd directive** – The `inbranch` clause tells the compiler to generate a SIMD variant of the function `do_mult()` that must be called conditionally within a SIMD loop. The `notinbranch` clause on the declaration of the `do_pow()` function tells the compiler that there is a SIMD variant of the function that must be called unconditionally within a SIMD loop.

If neither clause is specified, then the SIMD version of the function may or may not be called from within a branch, and the code the compiler generates must handle either situation.

The code in Figure 5.29 illustrates the use of the `inbranch` and `notinbranch` clauses. At line 1, the `inbranch` clause is used on the `declare simd` directive to tell the compiler to generate a SIMD variant of the function `do_mult()` that assumes it is always called conditionally. At line 15, the function `do_mult()` is called only when the condition computed at line 14 evaluates to 1 (`true`). The compiler generates a call to the SIMD variant of the function declared at line 1. If the compiler uses masking, the function is called unconditionally with an extra argument that passes the condition mask.

```
#pragma omp declare simd linear(pixel) uniform(mask) inbranch
#pragma omp declare simd linear(pixel) notinbranch
#pragma omp declare simd
extern void compute_pixel(char *pixel, char mask);
```

Figure 5.30: **Example of multiple declare simd directives for a function** – Multiple SIMD versions of the function are generated. The invocation of a specific version of the function is determined by where it is called.

At line 7, the `notinbranch` clause is used on the `declare simd` directive to tell the compiler that there is a SIMD variant of the function `do_pow()` that is optimized to assume it is never called conditionally. At line 17, the function `do_pow()` is called unconditionally. The compiler generates a call to the SIMD variant of the function declared at line 7, which does not pass a vector mask to the function.

5.3.4 Multiple Versions of a SIMD Function

As shown in Figure 5.30, multiple consecutive `declare simd` directives with different clauses may appear before the declaration of a function. When the declaration is in the same file as the definition of the function, the SIMD function variants are generated for each `declare simd` directive.

The multiple consecutive `declare simd` directives must appear on a function declaration that is visible when the function is called in a `simd` construct. This enables the compiler to select the best SIMD function variant for the function call.

5.4 Concluding Remarks

This chapter has covered how to use the OpenMP SIMD constructs to exploit SIMD instructions, which are present in several contemporary microprocessors. The performance improvements from using SIMD instructions can be substantial. In [15], the authors proposed a draft version of what became the OpenMP SIMD constructs and evaluated the performance improvement for selected benchmarks. In [14], the authors discuss vectorization for a specific architecture.

The SIMD constructs in OpenMP may be used as a stand-alone feature to vectorize loops, but in many cases, we expect the SIMD constructs to be used to further improve the performance of an application already parallelized using OpenMP threads. This is achieved by adding the `simd` constructs presented here to the time-consuming loops.

The capabilities of modern compilers to successfully vectorize code differ. Check the compiler's documentation for the relevant options to consider and to see if it supports a compiler commentary feature.

The SIMD loop body should not contain complex nested conditional branches, and the `if` clause should be avoided. When possible, align objects on byte boundaries that match an architecture's vector length. Use the `aligned` clause to indicate a variable's alignment. Use the `safelen` clause to vectorize loops with dependences, but care needs to be taken to use it the correct way. Use the `schedule(simd:static)` on the `for simd` and `do simd` constructs to balance thread and SIMD parallelism.

Utilize the `linear`, `uniform`, `simdlen`, and `aligned` clauses on the `declare simd` directive to generate specialized SIMD function variants. Add the `inbranch` and `notinbranch` clauses to the `declare simd` directive to generate SIMD versions of functions that may be called conditionally or unconditionally depending on the function call site. Be careful when using multiple `declare simd` directives for the same function. Each directive may result in another version of the function, which can increase the size of a program.

The `simd` and `for simd` (`do simd` in Fortran) constructs are combined with other OpenMP constructs. The `taskloop simd` construct is discussed in Chapter 3. The `distribute parallel for simd`, `distribute parallel do simd`, and `distribute simd` constructs are discussed in Chapter 6. This underlines the strength of OpenMP. In a consistent, portable, and integrated way, multiple levels of parallelism are supported.