Vectorisation
Let \( x, y, z \in \mathbb{K}^n \) with \( n \) large. Compute

\[
z_i := e^{z_i + x_i \cdot y_i}, \quad \forall i \leq n
\]

Standard implementation

```cpp
for ( size_t i = 0; i < n; ++i )
    z[i] = std::exp( z[i] + x[i]*y[i] );
```

Auto Vectorisation

```cpp
#pragma simd
for ( size_t i = 0; i < n; ++i )
    z[i] = std::exp( z[i] + x[i]*y[i] );
```

Speedup: **2.79x** (\( \mathbb{K} = \mathbb{R}, \) Xeon E5-2640), **5.01x** (\( \mathbb{K} = \mathbb{C}, \) XeonPhi 5110P)

Manual Vectorisation

```cpp
for ( size_t i = 0; i < n; i += 4 ) {
    const __m256d vx = _mm256_load_pd( & x[i] );
    const __m256d vy = _mm256_load_pd( & y[i] );
    __m256d vz = _mm256_load_pd( & z[i] );
    vz = _mm256_exp_pd( _mm256_add_pd( vz, _mm256_mul_pd( vx, vy ) ) );
    _mm256_store_pd( & z[i], vz );
}
```

Speedup: **3.20x** (\( \mathbb{K} = \mathbb{R}, \) Xeon E5-2640), **6.93x** (\( \mathbb{K} = \mathbb{C}, \) XeonPhi 5110P)
Standard processing mode of a processor is *scalar*:

\[ z := \text{op}_1(x) \quad \text{or} \quad z := \text{op}_2(x, y) \quad \text{or} \quad \cdots \]

with \( \text{op}_i : \mathbb{R}^i \rightarrow \mathbb{R} \) and costs

\[ t_{\text{load}} + t_{\text{op}} = z \]

Here, \( t_{\text{load}} \) denotes the time to load the data from memory.

With *vectorisation* we have \( \text{op}_i : \mathbb{R}^{i\cdot n} \rightarrow \mathbb{R}^n \), e.g.

\[
\begin{pmatrix}
    z_1 \\
    z_2 \\
    \vdots \\
    z_n
\end{pmatrix}
:= \text{op}
\begin{pmatrix}
    x_1 & y_1 \\
    x_2 & y_2 \\
    \vdots & \vdots \\
    x_n & y_n
\end{pmatrix}
\]

with costs

\[ n \cdot t_{\text{load}} + 1 \cdot t_{\text{op}} \]

Vectorisation is a realisation of an SIMD parallel architecture with algorithms employing the data parallel model.
Vectorisation needs hardware support, e.g. special processors.

- **Vector Processors**: special processors specifically designed around vector operations (Cray, NEC),

- **Support in x86 CPUs**:
  - **MMX**: only for integer operations
  - **SSE2**: two double prec. numbers per operation
  - **AVX**: four double prec. numbers per operation

- **Support in POWER/PowerPC CPUs**:
  - **AltiVec**: no double precision numbers,
  - **VSX**: $2 \times 2$ double precision numbers (two pipelines)

- **Support in Accelerator Cards**:
  - **Intel MIC**: eight double prec. numbers per operation
Auto-Vectorisation

Most C/C++ compilers will *automatically* use vector instructions for handling suitable data, e.g. the loop

```c
for ( int i = 0; i < n; ++i )
    z[i] = z[i] + x[i]*y[i];
```

will be automatically converted into

```c
for ( int i = 0; i < n; ++i )
    z[i:i+3] = z[i:i+3] + x[i:i+3]*y[i:i+3];
```

on a vector CPU with four entries per register.

To explicitly activate this auto-vectorisation, different compiler flags are used:

**Intel Compiler**

```bash
> icpc -O2 -msse2 -vec -c f.cc
> icpc -O2 -mavx -vec -c f.cc
> icpc -O2 -mmic -vec -c f.cc
```

**GNU Compiler**

```bash
> g++ -O2 -ftree-vectorize -msse2 -c f.cc
> g++ -O2 -ftree-vectorize -mavx -c f.cc
```

**Remark**

*Both compilers will only vectorise for optimisation levels -O2 or higher.*
Auto-Vectorisation

Remark

When handling floating point code, most compilers will default to vector instructions, even if code is not vectorized.

Auto-vectorization depends on several conditions concerning

- control flow,
- called functions,
- data access and data dependencies.

If these are not fulfilled, vectorisation may be

- discarded by the compiler or
- may result in sub optimal performance.

Furthermore, to achieve maximal vectorisation efficiency, the data layout has to be optimised for vector operations.
The loop count must be known \textit{before} entering the loop, i.e.

- no data dependent loop exit

```c
void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; ++i ) {
        if ( z[i] == 0.0 )
            break;
        else
            z[i] = z[i] + x[i]*y[i];
    }
}
```

- no change of loop variable in loop body

```c
void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; ++i ) {
        if (( z[i] == 0.0 ) && ( i < n-1 ))
            ++i;
        z[i] = z[i] + x[i]*y[i];
    }
}
```
Avoid branches in control flow, i.e.

- no switch statements

```c
void f ( int n, double * x, double * y, double * z ) {
  for ( int i = 0; i < n; ++i ) {
    switch ( i % 3 ) {
      case 0 : z[i] = x[i]*y[i]; break;
      case 1 : z[i] = z[i] - x[i]*y[i]; break;
      case 2 : z[i] = z[i] + x[i]*y[i]; break;
    }
  }
}
```

- no if statements

```c
void f ( int n, double * x, double * y, double * z ) {
  for ( int i = 0; i < n; ++i ) {
    if ( i % 2 == 0 )
      z[i] = x[i]*y[i];
    else
      z[i] = z[i] + x[i]*y[i];
  }
}
```
Masking

For if statements the compiler may still generate vector code, but data assignment is applied only for those cases, for which the if (or else) condition holds. This is implemented using *masked* versions of vector instructions.

```c
for ( size_t i = 0; i < n; ++i ) {
    if ( x[i] >= 0 )
        x[i] = x[i] + y[i];
    else
        x[i] = x[i] + z[i];
}
```

Masking can not be used, if the computation may lead to an exception:

```c
for ( size_t i = 0; i < n; ++i ) {
    if ( x[i] != 0 )
        x[i] = y[i] / x[i]; // division by zero
}
```
Avoid function calls within loop:

```c
double g ( double z, double x, double y ); // external function

void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; ++i ) {
        z[i] = g( z[i], x[i], y[i] );
    }
}
```

Here, due to missing knowledge about \( g \), the compiler will not vectorise the loop.

One exception to this rule are standard math functions, e.g.

```
sqrt  exp  exp2  log  log2  pow
abs  max  min  round  trunc  ceil
cos  cosh  sin  sinh  tan  tanh
acos  acosh  asin  asinh  atan  atanh
```

Hence, the following code will be vectorised:

```c
void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; ++i ) {
        z[i] = sin( z[i] + x[i] * y[i] );
    }
}
```
Most of the mathematical functions will especially benefit from using vectorised code, as their computation is very expensive, e.g. trigonometric functions.

Also allowed are \textit{inlined} functions:

\begin{verbatim}
inline double g ( double z, double x, double y ) {
    return z + x*y;
}

void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; ++i ) {
        z[i] = g( z[i], x[i], y[i] );
    }
}
\end{verbatim}

Locally defined functions (in same source file) are automatically considered as inline functions, even without the \texttt{inline} keyword.
Elemental Functions

The Intel compiler has a feature to build vector versions of functions. For this, the function arguments may only be elementary datatypes of the same type, e.g. int, float, double:

```c
__attribute__((vector))
double g ( double z, double x, double y )
{
    return z + x*y;
}
```

In contrast to inlined functions, g may be in a different module, but now exists in different versions, e.g. for SSE2 and AVX.

```c
double g ( double z, double x, double y ); // standard version
double g_sse2 ( double z[2], double x[2], double y[2] ); // SSE2 version
double g_avx ( double z[4], double x[4], double y[4] ); // AVX version
```

The above function attribute keyword is specific to the operating system:

**Linux:** __attribute__((vector))

**Windows:** __declspec(vector)
Non-Unit Stride

If the loop variable does not follow a unit increment, vectorisation may be inefficient, since each variable has to be loaded (and stored) separately.

```c
void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; i += 3 ) {
        z[i] = z[i] + x[i]*y[i];
    }
}
```

The only exception are loop strides of a power of 2.

```c
void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; i += 2 ) {
        z[i] = z[i] + x[i]*y[i];
    }
}
```

Otherwise, it is usually only worth to vectorise, if the work per variable is expensive:

```c
void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; i += 3 ) {
        z[i] = sqrt( z[i] + x[i]*y[i] );
    }
}
```
Indirect Addressing

If the memory position is accessed *indirectly*, vectorisation may also be omitted:

```c
void f ( int n, double * x, double * y, double * z, int * idx ) {
    for ( int i = 0; i < n; ++i ) {
        z[i] = z[i] + x[i] * y[ idx[i] ];
    }
}
```

**Remark**

*Indirect addressing is a special form of non-unit stride.*
Consider

```c
void f ( int n, double * x, double * y, double * z ){
    for ( int i = 1; i < n; ++i )
        z[i] = z[i-1] + x[i]*y[i];
}
```

For each loop, \( n \) elements have to be loaded for a single vector instruction before executing the instruction:

\[
\begin{array}{cccc}
  x_i & x_{i+1} & x_{i+2} & x_{i+3} \\
  * & y_i & y_{i+1} & y_{i+2} & y_{i+3} \\
  + & z_{i-1} & z_i & z_{i+1} & z_{i+2} \\
  = & z_i & z_{i+1} & z_{i+2} & z_{i+3}
\end{array}
\]

Hence, the vector instruction works on old, not yet updated data for \( z_i, z_{i+1} \) and \( z_{i+2} \).

Such form of data dependency can not be vectorised.
Aliasing is a generalisation of the previous data dependency example.

Consider

```c
void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; ++i )
        z[i] = z[i] + x[i]*y[i];
}
```

Here, $x$, $y$ and $z$ may share the same memory block:

And for some $i, j, k, l$

$$\text{addr}(x_i) = \text{addr}(z_j) \quad \text{and} \quad \text{addr}(y_k) = \text{addr}(z_l)$$

Hence, writing to $z$ changes $x$ and $y$.

Using vector instructions, the contents of $x$ and $y$ might have been loaded before changing the content of the arrays, leading to computational errors.
The compiler may either omit vectorisation at all or insert code for testing whether aliasing exists, e.g.

```c
void f ( int n, double * x, double * y, double * z ) {
    if ( x < z && x + n > z && ... ) {
        // aliasing, vectorisation unsafe
        ...
    } else {
        // no aliasing, vectorisation safe
        ...
    }
}
```

**Remark**

*For small \( n \), the extra tests might be too expensive!*
Alternatively, one may explicitly tell the compiler that no aliasing exists using special *pragmas* or command line options.

### Intel Compiler

The pragma for a specific loop is

```c
void f ( int n, double * x, double * y, double * z ) {
  #pragma ivdep
  for ( int i = 0; i < n; ++i )
    z[i] = z[i] + x[i]*y[i];
}
```

For function arguments, the command line options is

```
> icpc -xavx -vec -fargument-noalias -c f.cc
```

And for all data accesses:

```
> icpc -xavx -vec -fno-alias -c f.cc
```
GNU Compiler

The GNU compiler has no (simple) way to specify per-loop-aliasing. For all data accesses per module, the command line options is

```
> g++ -O -ftree-vectorize -mavx -fstrict-aliasing -c f.cc
```

Remark

Note, that pragmas only apply locally, whereas command line options apply globally to a module. Hence, make sure that aliasing rules really apply to all functions and variables in a module when using command line options.
A special variant of data dependencies are reduction operations, e.g.

```c
double dot ( int n, double * x, double * y ) {
    double sum = 0.0;
    for ( int i = 0; i < n; ++i )
        sum = sum + x[i]*y[i];
    return sum;
}
```

Such data dependency patterns are normally detected by the compiler and correctly vectorised, e.g. first compute the vector operation and the combine the individual results

\[
\begin{array}{cccc}
  x_i & x_{i+1} & x_{i+2} & x_{i+3} \\
  \times & y_i & y_{i+1} & y_{i+2} & y_{i+3} \\
  = & z_i & z_{i+1} & z_{i+2} & z_{i+3} \\
  \sum & z_i & z_{i+1} & z_{i+2} & z_{i+3} \\
  \pm & \text{sum}
\end{array}
\]

For reduction operation, only elementary datatypes are supported, e.g. int, float or double. Compound datatypes, e.g. struct or class, are not allowed.
In case, the reduction operation is not detected, one may help the compiler using the pragma `simd reduction`:

```c
double dot ( int n, double * x, double * y ) {
    double sum = 0.0;
    #pragma simd reduction (:+sum)
    for ( int i = 0; i < n; ++i )
        sum = sum + x[i]*y[i];
    return sum;
}
```

Here, (:+sum) indicates the reduction operation and the reduction variable. This may be extended to multiple reductions of the same type:

```c
void f ( int n, double * x, double * y ) {
    double xsum = 0.0;
    double ysum = 0.0;
    #pragma simd reduction (:+xsum, ysum)
    for ( int i = 0; i < n; ++i )
        xsum = xsum + x[i];
        ysum = ysum + y[i];
}
```

Remark

_This pragma is only supported by the Intel Compiler._
Before the vector unit of the CPU can run, the data has to be fetched from memory.

For this, the standard load instructions demand a specific alignment of the memory address of the data, i.e. the address must be a multiple of

- 16 bytes for SSE2,
- 32 bytes for AVX,
- 64 bytes for MIC.

SSE2 and AVX also provide load instructions for unaligned data, although, they are less efficient.

For simple loops, unaligned data may also be handled in two steps:

```c
double * p = new double[n];
size_t i = 0;

for ( ; i < aligned(p); ++i ) p[i] = ...; // use unaligned load
for ( ; i < n; ++i ) p[i] = ...; // use aligned load
```

For a small $n$, this may result in always using the sub optimal, unaligned load instructions.
Alignment

If more data is involved in the computation, the previous solution may not work.

```c
void f ( size_t n, double * p1, double * p2 ) {
    for ( size_t i = 0; i < n; ++i )
        p1[i] = p1[i] * p2[i];
}
```

If p1 and p2 have a different alignment, this will also hold for the memory position of any entry of both arrays. Hence, the loop can not be split into an unaligned and aligned part.

**Static Data**

If data is statically allocated, e.g.

```c
void f ( const size_t n ) {
    double a[100];
    double b[n];
    ...
}
```

the alignment may be explicitly defined using the type attribute `aligned(·):`

```c
void f ( const size_t n ) {
    __attribute__((aligned(32))) double a[100]; // 32 byte alignment
    __attribute__((aligned(64))) double b[n]; // 64 byte alignment
    ...
}
```
Dynamic Data

For dynamically allocated data this type attribute can not be used, since the underlying malloc works without any type informations.

Two alternative strategies are possible:

- Direct allocation of aligned data (within new operator):

  ```c
  void f ( const size_t n ) {
      double * a = memalign( sizeof(double) * n, 32 );
      double * b = new( memalign( sizeof(double) * n, 64 ) ) double[n];
      ...
  }
  ```

- Padding:

  ```c
  template <typename T>
  T * align_ptr ( T * ptr, const size_t align ) {
      return reinterpret_cast< T * >( (size_t(ptr) + ( align - (size_t(ptr) % align))) );
  }

  void f ( const size_t n ) {
      double * mem_a = new double[ n + 8 ]; // allocate padded memory
      double * a = align_ptr( mem_a, 64 ); // use only aligned part
      ...
  }
  ```

For memory in STL containers, a user defined `allocator` has to be provided, implementing one of the above techniques.
The set of pragmas for helping the Intel compiler to vectorise source code contains several instructions:

```c
#pragma vector
```

Instruct the compiler to vectorise the following loop, ignoring the internal cost model. The vectorised code must be valid, i.e. not contain illegal instructions.

Several arguments may further specify vectorisation strategies:

```c
#pragma vector {always|aligned|unaligned}
```

- **always**: Enforce vectorisation even if it leads to inefficient code or to exceptions during runtime.
- **aligned**: Use aligned load and store vector instructions for all data in the loop. If the data is actually unaligned, the resulting code will be illegal.
- **unaligned**: Use unaligned load and store vector instructions for all data in the loop.
Vectorisation Pragmas

#pragma novector
Instruct compiler to *not* vectorise the following loop.

#pragma ivdep
Instruct compiler to assume no data dependencies in the following loop.

#pragma simd
Enforces vectorisation of the following loop, even if the resulting code is illegal.

#pragma simd {reduction (op:var1(),var2)}

reduction: Apply reduction operation op to the variables var1, var2, ... More than one reduction operation is allowed, but the set of variables must be disjoint.
Vectorisation Reports

Compilers may generate *vectorisation reports* to let the programmer know, whether a specific loop was vectorised or not. Furthermore, such reports may give a reason, why no vectorisation was performed.

**Intel Compiler**

Vectorisation reports are activated via

```bash
> icpc -O2 -vec -vec-report[n] -xavx -c f.cc
```

with

<table>
<thead>
<tr>
<th>$n$</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no diagnostic information</td>
</tr>
<tr>
<td>1</td>
<td>report vectorised loops only (default)</td>
</tr>
<tr>
<td>2</td>
<td>additionally report non-vectorised loops</td>
</tr>
<tr>
<td>3</td>
<td>additionally report prohibiting data dependence information</td>
</tr>
<tr>
<td>4</td>
<td>report non-vectorised only loops</td>
</tr>
<tr>
<td>5</td>
<td>additionally report prohibiting data dependence information</td>
</tr>
<tr>
<td>6</td>
<td>like level 3 and 5 with additional details</td>
</tr>
</tbody>
</table>
GNU Compiler

The command line option for the GNU compiler is

```bash
> g++ -O2 -ftree-vectorize -ftree-vectorizer-verbose[=n] -mavx -c f.cc
```

with

\[ n = 0 \] no diagnostic information

\[ 1 \] report vectorized loops only

\[ 2 \] additionally report unvectorised “well-formed” loops and reason

\[ 3 \] additionally report alignment information for “well-formed” loops

\[ 4 \] like level 3 plus report for non-well-formed inner-loops

\[ 5 \] like level 3 plus report for all loops

\[ 6 \] print all vectoriser dump information

Kriemann, »Introduction to Parallel Programming« 28
Example output for the above mentioned cases:

Countability

```c
void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; ++i ) {
        if ( z[i] == 0.0 )
            break;
        else
            z[i] = z[i] + x[i]*y[i];
    }
}
```

> icpc -vec-report5 -xavx -c countability1.cc
countability1.cc(3): (col. 4) remark: loop was not vectorized:
  nonstandard loop is not a vectorization candidate.

```c
void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; ++i ) {
        if ( ( z[i] == 0.0 ) && ( i < n-1 ) )
            ++i;
        z[i] = z[i] + x[i]*y[i];
    }
}
```

> icpc -vec-report5 -xavx -c countability2.cc
countability2.cc(2): (col. 1) remark: routine skipped:
  no vectorization candidates.
Vectorisation Reports

Branching

```c
void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; ++i ) {
        switch ( i % 3 ) {
            case 0 : z[i] = x[i]*y[i]; break;
            case 1 : z[i] = z[i] - x[i]*y[i]; break;
            case 2 : z[i] = z[i] + x[i]*y[i]; break;
        }
    }
}
```

```
> icpc -vec-report5 -xavx -c branching1.cc
branching1.cc(2): (col. 4) remark: loop was not vectorized: existence of vector dependence.
branching1.cc(6): (col. 18) remark: vector dependence: assumed FLOW dependence between z line 6 and y line 4.
```

Function Calls

```c
double g ( double z, double x, double y ); // external function

void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; ++i ) {
        z[i] = g( z[i], x[i], y[i] );
    }
}
```

```
> icpc -vec-report5 -xavx -c funccall1.cc
funccall1.cc(5): (col. 54) remark: routine skipped: no vectorization candidates.
```
## Indirect Addressing

```c
void f ( int n, double * x, double * y, double * z, int * idx ) {
    for ( int i = 0; i < n; ++i ) {
        z[i] = z[i] + x[i] * y[ idx[i] ];
    }
}
```

> icpc -vec-report5 -xsse2 -c indirect.cc

indirect.cc(4): (col. 4) remark: loop was not vectorized:
vectorization possible but seems inefficient.

## Data Dependency

```c
void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 1; i < n; ++i )
        z[i] = z[i-1] + x[i]*y[i];
}
```

> icpc -vec-report5 -xavx -c datadep1.cc

datadep1.cc(2): (col. 4) remark: loop was not vectorized:
existence of vector dependence.

datadep1.cc(3): (col. 7) remark: vector dependence:
assumed ANTI dependence between y line 3 and z line 3.

datadep1.cc(3): (col. 7) remark: vector dependence:
assumed FLOW dependence between z line 3 and y line 3.
Vectorisation Reports

Aliasing

```c
void f ( int n, double * x, double * y, double * z ) {
    for ( int i = 0; i < n; ++i )
        z[i] = z[i] + x[i]*y[i];
}
```

`> icpc -vec-report5 -xavx -c alias0.cc
alias0.cc(2): (col. 4) remark: loop skipped: multiversioned.`
Further examples for messages are:

**Condition may protect exception**
*If the evaluation for a masked vectorisation may cause an exception, e.g. illegal memory access or division by zero, vectorisation will not be applied.*

**Data type unsupported on given target architecture**
*As an example, complex data types can only be vectorised efficiently using SSE3 instructions. Therefore, if only SSE2 is supported, vectorisation will be omitted.*

**Low trip count**
*The number of loops is not sufficient for vectorisation.*

**Not inner loop**
*Only the innermost loop will be vectorised. Note, that the compiler may apply loop-unrolling and change the order of nested loops!*
Data Structure Layout

The layout of data structures may have a large impact on the efficiency of vectorisation.

Consider:

```c
struct vector_t {
   double x, y, z;
};

struct particle_t {
   double mass;
   vector_t pos;
};

void f ( int n, particle_t * p, vector_t & t ) {
   for ( int i = 0; i < n; ++i ) {
      p[i].pos.x += t.x;
      p[i].pos.y += t.y;
      p[i].pos.z += t.z;
   }
}
```

The memory layout of an `particle_t` array is

```
... mass x y z mass x y z mass x y z mass x y z ...
```

In function `f`, each fourth element (`mass`) is unused during the computations, leading to *gaps* in the memory stream. Hence, only three values can be loaded simultaneously.
The first optimisation approach is to decouple mass and pos for all particles:

```c
struct vector_t { double x, y, z; }
struct particles_t {
    double * mass; // array of masses
    vector_t * pos; // array of positions
};
void f ( int n, particles_t & p, vector_t & t ) {
    for ( int i = 0; i < n; ++i ) {
        p.pos[i].x += t.x;
        p.pos[i].y += t.y;
        p.pos[i].z += t.z;
    }
}
```

This yields the modified data layout

|   | mass | mass | mass | mass | x | y | z | x | y | z | x | y | z |   |
|---|------|------|------|------|---|---|---|---|---|---|---|---|---|

without any gaps in the memory stream of the function f. But it is still inefficient for loading data into vector registers for *individual operations*, e.g. for x alone.

Speedup compared to first algorithm:

<table>
<thead>
<tr>
<th></th>
<th>SSE2</th>
<th>AVX</th>
<th>MIC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Xeon X5650)</td>
<td>(Xeon E5-2640)</td>
<td>(XeonPhi 5110P)</td>
</tr>
<tr>
<td></td>
<td>1.58x</td>
<td>1.59x</td>
<td>1.68x</td>
</tr>
</tbody>
</table>
The second approach is to *fully* decouple all data elements:

```c
struct particles_t {
    double * mass; // array of masses
    double * x;    // individual arrays for
    double * y;    // all coordinates
    double * z;
};

void f ( int n, particles_t & p, vector_t & t ) {
    for ( int i = 0; i < n; ++i ) {
        p.x[i] += t.x;
        p.y[i] += t.y;
        p.z[i] += t.z;
    }
}
```

leading to separate memory blocks for each value type:

```
... mass mass mass mass x x x x x y y y y z z z z ...
```

Data can now be loaded directly into vector registers for each sub-operation.

<table>
<thead>
<tr>
<th>Speedup compared to</th>
<th>SSE2 (Xeon X5650)</th>
<th>AVX (Xeon E5-2640)</th>
<th>MIC (XeonPhi 5110P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>second algorithm:</td>
<td>1.06x</td>
<td>1.18x</td>
<td>2.65x</td>
</tr>
<tr>
<td>first algorithm:</td>
<td>1.67x</td>
<td>1.88x</td>
<td>4.45x</td>
</tr>
</tbody>
</table>
Data Structure Layout

Array-of-Structures
The data structures used in the initial algorithm follow the *Array-of-Structures* principle (*AOS*), which

- is good for computations affecting data within a single item (good data locality),
- but has bad data locality for computations affecting all items, e.g. via vectorisation,
- has good code structure, e.g. all data for single item packed together (*Object Oriented* approach),

Structure-of-Arrays
The data structures of the last algorithm follow the *Structure-of-Arrays* principle (*SOA*), which

- is good for computations affecting all items,
- but has bad data locality for computations affecting data within a single item
As an example for a computation combining all local data the euclidean norm is computed per item. Here, data locality per item is of higher importance.

Using AOS:

```cpp
void norm2 ( int n, particle_t * p, double * norms ) {
    for ( int i = 0; i < n; ++i ) {
        norms[i] = std::sqrt( p[i].pos.x*p[i].pos.x + p[i].pos.y*p[i].pos.y + p[i].pos.z*p[i].pos.z );
    }
}
```

Using SOA:

```cpp
void norm2 ( int n, particle_t * p, double * norms ) {
    for ( int i = 0; i < n; ++i ) {
        norms[i] = std::sqrt( p.x[i]*p.x[i] + p.y[i]*p.y[i] + p.z[i]*p.z[i] );
    }
}
```

**Speedup SOA vs AOS:**

<table>
<thead>
<tr>
<th></th>
<th>SSE2 (Xeon X5650)</th>
<th>AVX (Xeon E5-2640)</th>
<th>MIC (XeonPhi 5110P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1.00x</td>
<td>1.13x</td>
<td>2.75x</td>
</tr>
</tbody>
</table>

The performance increase is much less than before.

Due to little data per item, it is still better to use vectorisation w.r.t. to all items, e.g. to treat $n$ for loops simultaneously, than to use vectorisation within the loop body alone.
Example: N-Body Problem

Given are \( n \) particles with

- masses: \( m_i \),
- initial positions: \( x_i(0) := x_i, x_i \neq x_j \) for \( i \neq j \), and
- initial velocities: \( v_i(0) := v_i \),

We are looking for the positions \( x_i(t) \) of all particles for \( t > 0 \).

The force \( F_i(t) \) of all particles on a single particle \( i \) is given by

\[
F_i(t) = m_i \frac{d^2 x_i(t)}{dt^2} = m_i G \sum_{j \neq i} \frac{m_j (x_j(t) - x_i(t))}{|x_j(t) - x_i(t)|^3}
\]

which leads us to a system of first-order differential equations:

\[
\begin{align*}
\frac{dx_i(t)}{dt} &= v_i(t) \\
\frac{dv_i(t)}{dt} &= G \sum_{j \neq i} \frac{m_j (x_j(t) - x_i(t))}{|x_j(t) - x_i(t)|^3}
\end{align*}
\]
Example: N-Body Problem

Applying Euler’s scheme to this system with a timestep $\Delta t$ we end up with

\[
\begin{align*}
v_i(t + \Delta t) &= v_i(t) + \Delta t \cdot G \sum_{j \neq i} \frac{m_j (x_j(t) - x_i(t))}{|x_j(t) - x_i(t)|^3} \\
x_i(t + \Delta t) &= x_i(t) + \Delta t \cdot v_i(t + \Delta t)
\end{align*}
\]

For a typical C++ implementation of this scheme, some additional data structures are introduced. First a vector in $\mathbb{R}^3$:

```cpp
struct vector3_t {
    double x, y, z;
    void add (const double f, // add f*v to (x,y,z)
               const vector3_t & v);
    double norm_cubed (); // return |v|^3
    vector3_t operator - (const vector3_t & v); // return v1-v2
};
```

A single particle is represented as

```cpp
struct particle_t {
    double mass;
    vector3_t position;
    vector3_t velocity;
};
```
Finally, the particle system together with a single timestep is implemented as

```cpp
struct particle_system_t {
  std::vector< particle_t > particles;

  // compute single timestep using Euler scheme
  void timestep ( const double dt ) {
    for ( size_t i = 0; i < particles.size(); ++i ) {
      const vector3_t pos_i( particles[i].position );
      vector3_t force;

      // compute new velocity
      for ( size_t j = 0; j < particles.size(); ++j ) {
        if ( i != j ) {
          const vector3_t pos_j( particles[j].position );
          const vector3_t dist( pos_j - pos_i );

          // $\sum_{i \neq j} \frac{m_j (x_j(t) - x_i(t))}{|x_j(t) - x_i(t)|^3}$
          force.add( particles[j].mass / dist.norm_cubed(), dist );
        }
      }

      // $v_i(t + \Delta t) = v_i(t) + \Delta t \cdot \sum \ldots$
      particles[i].velocity.add( dt, force );
    }

    // update position
    for ( auto & p : particles )
      p.position.add( dt, p.velocity );  // $x_i(t + \Delta t) = x_i(t) + \Delta t \cdot v_i(t + \Delta t)$
  }
};
```

**Remark**

*All values are scaled such that $G = 1$.***
The implementation follows the AOS approach. The computationally expensive loop is the inner loop of the \texttt{timestep} function.

Compiling the source code reveals some problems:

\begin{verbatim}
icpc -vec-report2 -xHOST -vec -O3 -c nbody.cc
ody.cc(6):  loop was not vectorized: unsupported loop structure.
ody.cc(11): loop was not vectorized: condition may protect exception.
ody.cc(26): loop was not vectorized: existence of vector dependence.
\end{verbatim}

The first report corresponds to the outer loop:

\begin{verbatim}
void timestep ( const double dt ) {
  for ( size_t i = 0; i < particles.size(); ++i ) {
    ...
  }
}\end{verbatim}

The inner loop starting at line 11 has a potential \textit{division by zero}:

\begin{verbatim}
for ( size_t j = 0; j < particles.size(); ++j ) {
  if ( i != j ) {
    ...
    force.add( particles[j].mass / dist.norm_cubed(), dist );
  }
}\end{verbatim}

Finally, the compiler assumes a data dependency in the update loop:

\begin{verbatim}
for ( auto & p : particles )
  p.position.add( dt, p.velocity );
\end{verbatim}
The main problem is the inner loop. Two possible solutions are:

1. Enforce vectorisation even though division by zero occurs:

```c
#pragma vector always
for ( size_t j = 0; j < particles.size(); ++j ) {
    if ( i != j ) {
        ...
    }
}
```

2. Change the code to obtain a valid masked version:

```c
for ( size_t j = 0; j < particles.size(); ++j ) {
    const vector3_t dist( particles[j].position - pos_i );
    const real_t d = (i == j ? 0.0 : 1.0 / dist.norm_cubed());
    force.add( particles[j].mass * d, dist ); // now always legal
}
```

As no data dependency exists in the update loop, a `#pragma ivdep` will help:

```c
#pragma ivdep
for ( auto & p : particles )
    p.position.add( dt, p.velocity );
```

Both changes will result in a speedup of

<table>
<thead>
<tr>
<th></th>
<th>SSE2</th>
<th>AVX</th>
<th>MIC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Xeon X5650)</td>
<td>(Xeon E5-2640)</td>
<td>(XeonPhi 5110P)</td>
</tr>
<tr>
<td></td>
<td>1.92x</td>
<td>2.05x</td>
<td>2.03x</td>
</tr>
</tbody>
</table>
The previous implementation uses the AOS approach, whereas an SOA based code would instead look like (changes marked red):

```cpp
struct particle_system_t {
    size_t n_particles; // for convenience ;-)  
    std::vector< double > mass; // masses   
    std::vector< double > pos_x, pos_y, pos_z; // positions  
    std::vector< double > vel_x, vel_y, vel_z; // velocities  

    void timestep ( const double dt ) {
        for ( size_t i = 0; i < n_particles; ++i ) {
            const vector3_t pos_i( pos_x[i], pos_y[i], pos_z[i] );  
            vector3_t force;  

            for ( size_t j = 0; j < n_particles; ++j ) {
                if ( i != j ) {  
                    const vector3_t pos_j( pos_x[j], pos_y[j], pos_z[j] );  
                    const vector3_t dist( pos_j - pos_i );  

                    force.add( mass[j] / dist.norm_cubed(), dist );  
                }  
            }  

            vel_x[i] += dt * force.x; vel_y[i] += dt * force.y; vel_z[i] += dt * force.z;
        }

        for ( size_t i = 0; i < n_particles; ++i ) {
            pos_x[i] += dt * vel_x[i]; pos_y[i] += dt * vel_y[i]; pos_z[i] += dt * vel_z[i];
        }
    }
}
```

Although SOA is used, the inner block may still be implemented as before, preserving most of the OO design.
Example: N-Body Problem

Executing both algorithms, the difference of the relatime runtime is:

<table>
<thead>
<tr>
<th></th>
<th>SSE2 (Xeon X5650)</th>
<th>AVX (Xeon E5-2640)</th>
<th>MIC (XeonPhi 5110P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA vs AOS</td>
<td>1.03x</td>
<td>1.01x</td>
<td>3.18x</td>
</tr>
</tbody>
</table>

For SSE2 and AVX, the vector registers are large enough to provide efficient local vectorisation. Only the MIC architecture benefits from the SOA approach.

Furthermore, for alignment can be applied by using padding:

```cpp
std::vector< double > mass_mem( n+8 );
double * mass = align_ptr( n );
```

This yields an additional (and absolute) speedup of

<table>
<thead>
<tr>
<th></th>
<th>SSE2 (Xeon X5650)</th>
<th>AVX (Xeon E5-2640)</th>
<th>MIC (XeonPhi 5110P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA aligned vs SOA</td>
<td>1.01x</td>
<td>1.01x</td>
<td>1.06x</td>
</tr>
<tr>
<td>SOA aligned vs AOS</td>
<td>1.04x</td>
<td>1.02x</td>
<td>3.36x</td>
</tr>
</tbody>
</table>

None of the implementations will benefit significantly from alignment.
Implementing the same algorithm without any compound datatypes, and thereby having a C source, additional reduction operations may explicitly be applied:

```c
for ( size_t i = 0; i < n_particles; ++i ) {
    const double xi = particles.pos_x[i];
    const double yi = particles.pos_y[i];
    const double zi = particles.pos_z[i];
    double fx = 0, fy = 0, fz = 0;

#pragma simd reduction (+:fx,fy,fz)
for ( size_t j = 0; j < n_particles; ++j ) {
    if ( i != j ) {
        const double dist_x = particles.pos_x[j] - xi;
        const double dist_y = particles.pos_y[j] - yi;
        const double dist_z = particles.pos_z[j] - zi;

        const double norm_sq = dist_x*dist_x + dist_y*dist_y + dist_z*dist_z;
        const double norm_cu = particles.mass[j] / ( norm_sq * sqrt( norm_sq ) );

        fx += dist_x * norm_cu; fy += dist_y * norm_cu; fz += dist_z * norm_cu;
    }
}
particles.vel_x[i] += dt * fx; particles.vel_y[i] += dt * fy; particles.vel_z[i] += dt * fz;
```

Unfortunately, no additional speedup can be gained by this:

<table>
<thead>
<tr>
<th></th>
<th>SSE2 (Xeon X5650)</th>
<th>AVX (Xeon E5-2640)</th>
<th>MIC (XeonPhi 5110P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA in C vs SOA in C++</td>
<td>1.00x</td>
<td>1.00x</td>
<td>1.00x</td>
</tr>
</tbody>
</table>
Example: N-Body Problem

The following table contains all previous results and the optimal speedup achieved:

<table>
<thead>
<tr>
<th></th>
<th>SSE2</th>
<th>AVX</th>
<th>MIC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Xeon X5650)</td>
<td>(Xeon E5-2640)</td>
<td>(XeonPhi 5110P)</td>
</tr>
<tr>
<td>AOS pragma vs. AOS</td>
<td>1.92x</td>
<td>2.05x</td>
<td>2.03x</td>
</tr>
<tr>
<td>SOA vs AOS pragma</td>
<td>1.03x</td>
<td>1.01x</td>
<td>3.18x</td>
</tr>
<tr>
<td>SOA aligned vs SOA</td>
<td>1.01x</td>
<td>1.01x</td>
<td>1.06x</td>
</tr>
<tr>
<td>SOA in C vs SOA</td>
<td>1.00x</td>
<td>1.00x</td>
<td>1.00x</td>
</tr>
<tr>
<td>optimal vs base</td>
<td>2.00x</td>
<td>2.09x</td>
<td>6.84x</td>
</tr>
</tbody>
</table>

The simulation may also be run with single precision arithmetic and hence, twice the entries per vector register.

<table>
<thead>
<tr>
<th></th>
<th>SSE2</th>
<th>AVX</th>
<th>MIC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Xeon X5650)</td>
<td>(Xeon E5-2640)</td>
<td>(XeonPhi 5110P)</td>
</tr>
<tr>
<td>AOS float vs AOS double</td>
<td>2.41x</td>
<td>2.48x</td>
<td>1.53x</td>
</tr>
<tr>
<td>AOS pragma vs AOS</td>
<td>1.17x</td>
<td>2.20x</td>
<td>2.76x</td>
</tr>
<tr>
<td>SOA vs AOS pragma</td>
<td>1.18x</td>
<td>1.55x</td>
<td>3.08x</td>
</tr>
<tr>
<td>SOA aligned vs SOA</td>
<td>0.96x</td>
<td>0.98x</td>
<td>1.10x</td>
</tr>
<tr>
<td>SOA in C vs SOA</td>
<td>1.04x</td>
<td>0.95x</td>
<td>0.99x</td>
</tr>
<tr>
<td>optimal vs base</td>
<td>1.38x</td>
<td>3.41x</td>
<td>9.35x</td>
</tr>
</tbody>
</table>
Instead of letting the compiler do (almost) all the work, one may also access the vector processing functions directly and thereby, optimise the code even further.

**Assembler**

The direct approach would use *assembler* instructions, which has several disadvantages (of which some are subjective):

- Learning a new language.
- Syntax is more error prone.
- ...

Hence, this is suggested only in *extreme* cases and if you know what you are doing!
Fortunately, most compilers provide a direct way to access vector operations via compiler intrinsics.

*Compiler intrinsics are functions, not implemented in a software library, but intrinsic to the compiler. Calls to intrinsic functions are always inlined and may even be more optimised than other functions because of the intrinsic knowledge about the functions by the compiler.*

Furthermore, suitable datatypes for vector operations are provided.

To make the intrinsic functions and datatypes available in your program, a special header file has to be included:

```
#include <immintrin.h>
```

**Remark**

The set of compiler intrinsics is different between the Intel and the GNU compiler. While the latter only provides intrinsics for actual CPU instructions, the Intel compiler also provides higher level functions, e.g. `sin` or `exp`. 
Vector Types

The vector types are defined according to the bit length of the vector registers:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE2</td>
<td>__m128d (2x double)</td>
<td>__m128 (4x float)</td>
</tr>
<tr>
<td>AVX</td>
<td>__m256d (4x double)</td>
<td>__m256 (8x float)</td>
</tr>
<tr>
<td>MIC</td>
<td>__m512d (8x double)</td>
<td>__m512 (16x float)</td>
</tr>
</tbody>
</table>

One may use these types like all other C++ types:

```cpp
__m256d x;
const __m128d y = { 1.0, 2.0 }; // array initialisation

struct simd_vec {
  __m256d x, y, z;
};
```

It is even possible to cast pointers to vector types to standard pointers:

```cpp
__m128d x = { 1.0, 2.0 }; // array initialisation
double * p = reinterpret_cast< double * >( & x );

std::cout << p[0] << " " << p[1] << std::endl;
```
Compiler Intrinsics

Vector Functions

Functions for vector types are named after the CPU instruction and the elementary type, e.g. single or double precision. In addition, a prefix indicates the length of the vector register. For SSE2 types (__m128, __m128d), the pattern is

- _mm_op_ps  for single precision functions
- _mm_op_pd  for double precision functions

For AVX and MIC function, the vector length in bits is part of the name, e.g.

- _mm256_op_pd  and  _mm512_op_pd

Examples for AVX functions are

```c
__m256d _mm256_set1_pd ( double f ); // return (f,f,f,f)
__m256d _mm256_set_pd ( double f0, ..., double f3 ); // return (f0,f1,f2,f3)
__m256d _mm256_add_pd ( __m256d A, __m256d B ); // return (A0+B0,....,A3+B3)
__m256d _mm256_sub_pd ( __m256d A, __m256d B ); // return (A0-B0,....,A3-B3)
__m256d _mm256_mul_pd ( __m256d A, __m256d B ); // return (A0*B0,....,A3*B3)
__m256d _mm256_div_pd ( __m256d A, __m256d B ); // return (A0/B0,....,A3/B3)
__m256d _mm256_addsub_pd ( __m256d A, __m256d B ); // return (A0-B0,A1+B1,A2-B2,A3+B3)
__m256d _mm256_sqrt_pd ( __m256d A ); // return (A0,...,A3)
__m256d _mm256_load_pd ( double const * P ); // return P[0..3] (aligned)
__m256d _mm256_loadu_pd ( double const * P ); // return P[0..3] (un-aligned)
void _mm256_store_pd ( double const * P, __m256d A ); // P[0..3] = A (aligned)
void _mm256_storeu_pd ( double const * P, __m256d A ); // P[0..3] = A (un-aligned)
```
The Intel compiler also provides intrinsics for high level functions, e.g.:

```c
__m256d _mm256_invsqrt_pd ( __m256d A ); // return (1/√A₀, ..., 1/√A₃)
__m256d _mm256_cbrt_pd ( __m256d A ); // return (∛A₀, ..., ∛A₃)
__m256d _mm256_invcbbrt_pd ( __m256d A ); // return (1/∛A₀, ..., 1/∛A₃)
__m256d _mm256_pow_pd ( __m256d A, __m256d B ); // return (A₀ᴮ₀, ..., A₃ᴮ₃)
__m256d _mm256_exp_pd ( __m256d A ); // return (exp(A₀), ..., exp(A₃))
__m256d _mm256_log_pd ( __m256d A ); // return (log(A₀), ..., log(A₃))
__m256d _mm256_exp2_pd ( __m256d A );
__m256d _mm256_sin_pd ( __m256d A ); // return (sin(A₀), ..., sin(A₃))
__m256d _mm256_sind_pd ( __m256d A ); // same, but in degrees
__m256d _mm256_sinh_pd ( __m256d A );
__m256d _mm256_asin_pd ( __m256d A );
__m256d _mm256_asinh_pd ( __m256d A );
__m256d _mm256_sincos_pd ( __m256d * A, __m256d B ); // return sin(B₀..B₃), (A₀..A₃)=cos(B₀..B₃)
```

**Remark**

*Computing the sine and cosine of a number is based on polynomial interpolation with different coefficients but the same degree for both values. Therefore, the same algorithm with different data is used, providing several optimisation possibilities if both values are computed together.*
Complex Numbers

The default data type for complex numbers uses a struct storing real and imaginary part:

```c
struct complex_t {
    double re, im;
}
```

This structure is identical to an SSE2 register, also holding two double entries. Therefore, the complex type may also be stored as:

```c
typedef __m128d complex_t;
```

The fundamental arithmetic operations are then implemented using SSE2 functions:

```c
complex_t operator + ( const complex_t z0, const complex_t z1 ) {
    return _mm_add_pd( z0, z1 );
}

complex_t operator - ( const complex_t z0, const complex_t z1 ) {
    return _mm_sub_pd( z0, z1 );
}
```
The complex multiplication of $a + bi$ and $c + di$, i.e.

$$(a + bi) \cdot (c + di) = (ac - bd) + i(ad + bc)$$

is decomposed into the following steps:

- \_mm\_unpacklo\_pd \hspace{1cm} a \quad b \quad \rightarrow \quad a \quad a
- \_mm\_mul\_pd \hspace{1cm} a \quad a \quad \cdot \quad c \quad d \quad \rightarrow \quad ac \quad ad
- \_mm\_unpackhi\_pd \hspace{1cm} a \quad b \quad \rightarrow \quad b \quad b
- \_mm\_shuffle\_pd \hspace{1cm} c \quad d \quad \rightarrow \quad d \quad c
- \_mm\_mul\_pd \hspace{1cm} b \quad b \quad \cdot \quad d \quad c \quad \rightarrow \quad bd \quad bc
- \_mm\_addsub\_pd \hspace{1cm} (SSE3) \hspace{1cm} ac \quad ad \quad \mp \quad bd \quad bc \quad \rightarrow \quad ac-bd \quad ad+bc

The final function looks as

```c
complex_t operator * ( const complex_t z0, const complex_t z1 ) {
    const __m128d t0 = _mm_mul_pd( _mm_unpacklo_pd( z0, z0 ), z1 );
    const __m128d t1 = _mm_mul_pd( _mm_unpackhi_pd( z0, z0 ), _mm_shuffle_pd( z1, z1, 1 ) );
    return _mm_addsub_pd( t0, t1 );
}
```

**Remark**

With SSE2-only functions, complex arithmetics can not be vectorised efficiently.
Similarly, the complex division is implemented:

```c
complex_t operator / ( const complex_t z0, const complex_t z1 ) {
    const __m128d t0 = _mm_mul_pd( _mm_shuffle_pd( z0, z0, 1 ), _mm_unpacklo_pd( z1, z1 ) );
    const __m128d t1 = _mm_mul_pd( z0, _mm_unpackhi_pd( z1, z1 ) );
    const __m128d t2 = _mm_addsub_pd( t0, t1 );
    const __m128d t3 = _mm_hadd_pd( _mm_mul_pd( z1, z1 ) );
    return _mm_div_pd( _mm_shuffle_pd( t2, t2, 1 ), t3 );
}
```

For the loop

```c
for ( size_t i = 0; i < n; ++i )
    z1 = z1 + ( z2 * z3 ) + ( z2 / z1 );
```

the above functions yield a speedup of 1.56x compared to the standard C++ complex class. This corresponds to an efficiency of almost 80%.
Using the high level intrinsics of the Intel compiler, more complex functions may be implemented.

As an example, consider the complex exponential function:

\[ e^{a+b\cdot i} = (e^a \cos b) + (e^a \sin b) \cdot i \]

The computation involves the real exponential, sine and cosine functions.

A vectorised version consists of the steps

\[
\begin{align*}
\text{_mm_exp_pd} & : a \rightarrow e^a \\
\text{_mm_sincos_pd} & : b \rightarrow \sin b \quad \cos b \\
\text{_mm_mul_pd} & : e^a \cdot \cos b \quad \sin b \rightarrow c \quad d
\end{align*}
\]

In the first two steps, the same values are computed twice. Only for the last step vectorisation takes place. Unfortunately, a multiplication is much faster than the other operations. Hence, the complex exponential can not benefit from using vectorisation.
The situation changes, if more than one evaluation of the complex exponential is needed. For two complex numbers \( z_0 \) and \( z_1 \), the vector implementation looks as

```c
void exp2 ( complex_t * z ) {
    const __m128d exp_re = _mm_exp_pd( _mm_unpacklo_pd( z[0], z[1] ) );
    __m128d sin_im, cos_im;
    sin_im = _mm_sincos_pd( & cos_im, _mm_unpackhi_pd( z[0], z[1] ) ) ;
    sin_im = _mm_mul_pd( exp_re, sin_im );
    cos_im = _mm_mul_pd( exp_re, cos_im );
    z[0] = _mm_unpacklo_pd( cos_im, sin_im );
    z[1] = _mm_unpackhi_pd( cos_im, sin_im );
}
```

The speedup compared to the (auto vectorised) standard C++ complex data type is 1.26x.

Similarly, the AVX and the MIC version for four and eight complex numbers is implemented. However, the MIC does not provide an efficient sincos function. Instead _mm512_sin_pd and _mm512_cos_pd should be used directly.
The next OpenMP standard will contain vectorisation instructions, very similar to the corresponding instructions of the Intel compiler.

Loop Vectorisation

To instruct the compiler to apply vectorisation, the new pragma `simd` is introduced:

```c
#pragma omp simd
for ( size_t i = 0; i < n; ++i )
    z[i] = std::exp( z[i] + x[i]*y[i] );
```

The `simd` pragma will also support reductions and alignment.

SIMD Functions

To declare vector versions of user defined functions, the pragma `declare simd` can be used:

```c
#pragma omp declare simd
double f ( double x, double y )
{
    return sin(x) * cos(y);
}
```

OpenMP also allows the function to be part of an outer branch, e.g. for masked vectorisation.
The *Cilk* C/C++ language extension contains a special array notation:

```c
double a[n];
da[:  ] = 2;  // apply RHS to whole array
da[5:10] = 4;  // apply RHS to indices [5,5+10)
da[5:10:2] = 6;  // apply RHS to indices 5,7,9,11,13
```

Working with arrays simplifies to single statements:

**C++**

```c
for ( size_t i = 0; i < n; ++i )
    x[i] = x[i] + y[i]*z[i];
for ( size_t i = 0; i < n; ++i )
    x[i] = exp(y[i]);
for ( size_t i = 0; i < n; ++i ) {
    if ( x[i] != 0 )
        x[i] = y[i] / x[i];
}
```

**Cilk**

```c
x[: ] = x[: ] + y[: ]*z[: ];
x[: ] = exp( y[: ] );
if ( mask[: ] )
    x[: ] = y[: ] / x[: ];
```

Due to the semantics of the array notation, such statements are perfect candidates for auto vectorisation.

**Remark**

*Cilk is currently only available with the Intel compiler.*
“A Guide to Auto-vectorization with Intel C++ Compilers”.  

“Intel 64 and IA-32 Architectures Software Developer Manuals”.  

“Intel C++ Compiler XE 13.1 User and Reference Guide”.  

Vladimirov, A. and V. Karpusenko. “Test-driving Intel Xeon Phi coprocessors with a basic N-body simulation”.  
  http://research.colfaxinternational.com/.