Vectorisation

Introduction

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

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

Auto Vectorisation

```
#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

```
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)

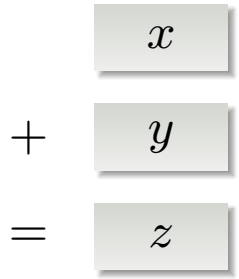
Introduction

Standard processing mode of a processor is *scalar*:

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

with $\mathrm{op}_i : \mathbb{R}^i \to \mathbb{R}$ and costs

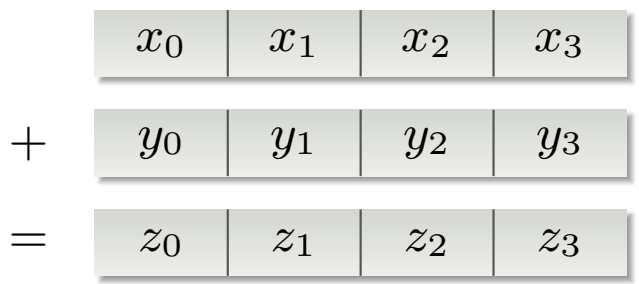
$$t_{\mathrm{load}} + t_{\mathrm{op}}$$



Here, t_{load} denotes the time to load the data from memory.

With *vectorisation* we have $\mathrm{op}_i : \mathbb{R}^{i \cdot n} \to \mathbb{R}^n$, e.g.

$$\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix} := \mathrm{op} \left(\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \right)$$



with costs

$$n \cdot t_{\mathrm{load}} + 1 \cdot t_{\mathrm{op}}$$

Vectorisation is a realisation of an SIMD parallel architecture with algorithms employing the data parallel model.

Introduction

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 : 2×2 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

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

will be automatically converted into

```
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

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

GNU Compiler

```
> 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.

Countability

The loop count must be known *before* entering the loop, i.e.

- no data dependent loop exit

```
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

```
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];
   }
}
```

Branching

Avoid branches in control flow, i.e.

- no `switch` statements

```
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

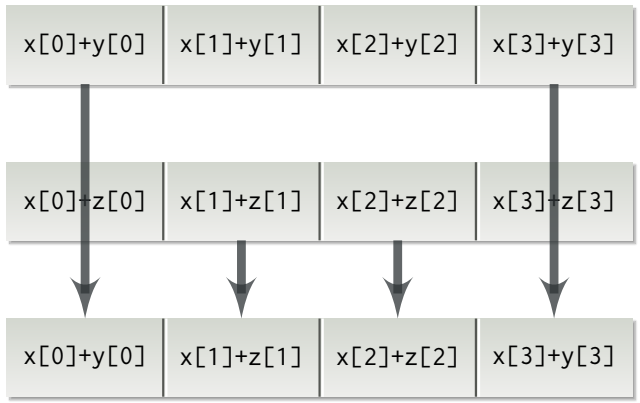
```
void f ( int n, double * x, double * y, double * z ) {
   for ( int  i = 0; i < n; ++i ) {
      if ( i % 2 == 0 )
        z[i] =        x[i]*y[i];
      else
        z[i] = z[i] + x[i]*y[i];
   }
}
```

Branching

Masking

For `if` statements the compiler may still generated 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.

```
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:

```
for ( size_t i = 0; i < n; ++i ) {
  if ( x[i] != 0 )
    x[i] = y[i] / x[i]; // division by zero
}
```

Function Calls

Avoid function calls within loop:

```
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:

```
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] );
   }
}
```

Function Calls

Remark

Most of the mathematical functions will especially benefit from using vectorised code, as their computation is very expensive, e.g. trigonometic functions.

Also allowed are *inlined* functions:

```
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] );
   }
}
```

Remark

Locally defined functions (in same source file) are automatically considered as inline functions, even without the `inline` *keyword.*

Function Calls

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:

```
__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.

```
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*.

```
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.

```
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:

```
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:

```
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

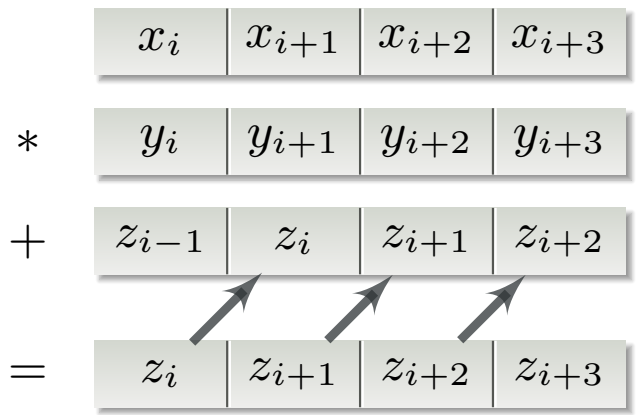
Indirect addressing is a special form of non-unit stride.

Data Dependency

Consider

```
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:



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.

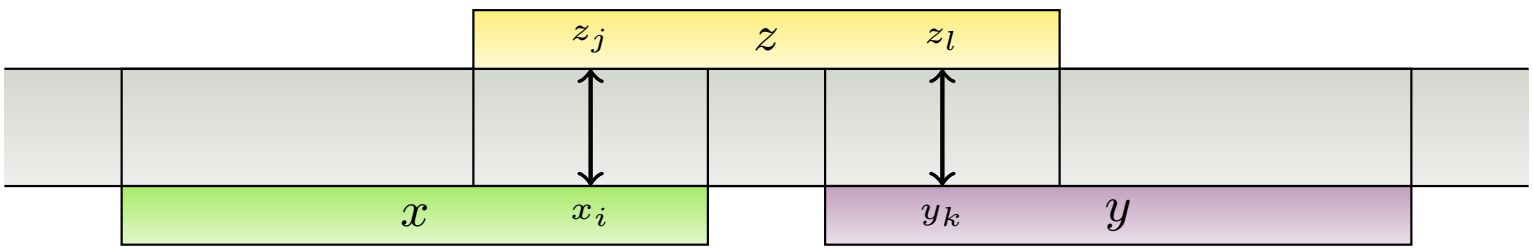
Aliasing

Aliasing is a generalisation of the previous data dependency example.

Consider

```
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

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

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.

Aliasing

The compiler may either omit vectorisation at all or insert code for testing whether aliasing exists, e.g.

```
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!

Aliasing

Alternativly, 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

```
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
```

Aliasing

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

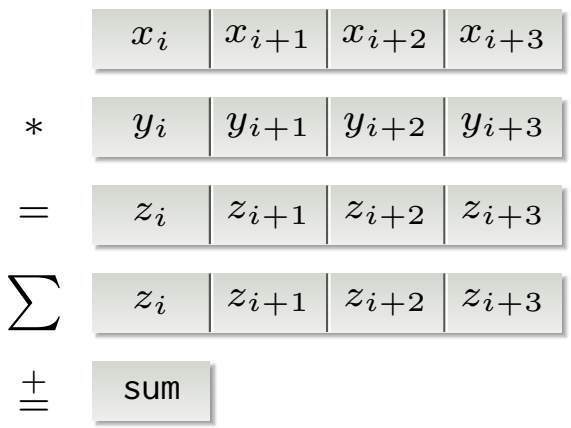
Note, that pragmas only apply locally, whereas command line options apply globally to a module. Hence, make sure that aliasing rules really apply to all functions and variables in a module when using command line options.

Reduction

A special variant of data dependencies are *reduction* operations, e.g.

```
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



For reduction operation, only elementary datatypes are supported, e.g. `int`, `float` or `double`. Compound datatypes, e.g. `struct` or `class`, are not allowed.

Reduction

In case, the reduction operation is *not* detected, one may help the compiler using the pragma `simd reduction`:

```
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:

```
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

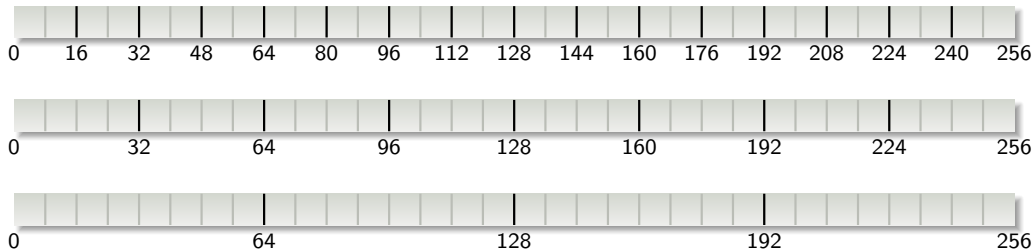
This pragma is only supported by the Intel Compiler.

Alignment

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:

```
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.

```
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.

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

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

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

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):

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

- Padding:

```
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.

Vectorisation Pragmas

The set of pragmas for helping the Intel compiler to vectorise source code contains several instructions:

`#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:

```
#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

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

with

$n = 0$	no diagnostic information
1	report vectorised loops only (default)
2	additionally report non-vectorised loops
3	additionally report prohibiting data dependence information
4	report non-vectorised only loops
5	additionally report prohibiting data dependence information
6	like level 3 and 5 with additional details

Vectorisation Reports

GNU Compiler

The command line option for the GNU compiler is

```
> 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

Vectorisation Reports

Example output for the above mentioned cases:

Countability

```
1 void f ( int n, double * x, double * y, double * z ) {
2    for ( int  i = 0; i < n; ++i ) {
3       if ( z[i] == 0.0 )
4         break;
5       else
6         z[i] = z[i] + x[i]*y[i];
7    }
8 }
```

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

```
1 void f ( int n, double * x, double * y, double * z ) {
2    for ( int  i = 0; i < n; ++i ) {
3       if (( z[i] == 0.0 ) && ( i < n-1 ))
4         ++i;
5       z[i] = z[i] + x[i]*y[i];
6    }
7 }
```

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

Vectorisation Reports

Branching

```
1  void f ( int n, double * x, double * y, double * z ) {
2     for ( int  i = 0; i < n; ++i ) {
3        switch ( i % 3 ) {
4          case 0 : z[i] =       x[i]*y[i]; break;
5          case 1 : z[i] = z[i] - x[i]*y[i]; break;
6          case 2 : z[i] = z[i] + x[i]*y[i]; break;
7        }
8     }
9  }
```

```
> 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.
branching1.cc(4): (col. 18) remark: vector dependence:
                            assumed ANTI dependence between y line 4 and z line 6.
```

Function Calls

```
1  double g ( double z, double x, double y );   // external function
2
3  void f ( int n, double * x, double * y, double * z ) {
4     for ( int  i = 0; i < n; ++i ) {
5         z[i] = g( z[i], x[i], y[i] );
6     }
7  }
```

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

Vectorisation Reports

Indirect Addressing

```
1 void f ( int n, double * x, double * y, double * z, int * idx ) {
2    for ( int  i = 0; i < n; ++i ) {
3       z[i] = z[i] + x[i] * y[ idx[i] ];
4    }
5 }
```

```
> icpc -vec-report5 -xsse2 -c indirect.cc
indirect.cc(4): (col. 4) remark: loop was not vectorized:
                         vectorization possible but seems inefficient.
```

Data Dependency

```
1 void f ( int n, double * x, double * y, double * z ) {
2    for ( int  i = 1; i < n; ++i )
3       z[i] = z[i-1] + x[i]*y[i];
4 }
```

```
> 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

```
1 void f ( int n, double * x, double * y, double * z ) {
2    for ( int  i = 0; i < n; ++i )
3       z[i] = z[i] + x[i]*y[i];
4 }
```

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

Vectorisation Reports

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:

```
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

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.

Data Structure Layout

The first optimisation approach is to decouple mass and pos for all particles:

```
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

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:

SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
1.58x	1.59x	1.68x

Data Structure Layout

The second approach is to *fully* decouple all data elements:

```
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:

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

Speedup compared to

	SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
second algorithm:	1.06x	1.18x	2.65x
first algorithm:	1.67x	1.88x	4.45x

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

Data Structure Layout

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:

```
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:

```
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:

SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
1.00x	1.13x	2.75x

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:

$$\begin{aligned} \frac{dx_i(t)}{dt} &= v_i(t) \\ \frac{dv_i(t)}{dt} &= G \sum_{j \neq i} \frac{m_j (x_j(t) - x_i(t))}{|x_j(t) - x_i(t)|^3} \end{aligned}$$

Example: N-Body Problem

Applying Euler's scheme to this system with a timestep Δt we end up with

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

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

```
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

```
struct particle_t {
  double     mass;
  vector3_t  position;
  vector3_t  velocity;
};
```

Example: N-Body Problem

Finally, the particle system together with a single timestep is implemented as

```
1  struct particle_system_t {
2    std::vector< particle_t >  particles;
3
4    // compute single timestep using Euler scheme
5    void timestep ( const double  dt ) {
6      for ( size_t  i = 0; i < particles.size(); ++i ) {
7        const vector3_t  pos_i( particles[i].position );
8        vector3_t        force;
9
10       // compute new velocity
11       for ( size_t  j = 0; j < particles.size(); ++j ) {
12         if ( i != j ) {
13           const vector3_t  pos_j( particles[j].position );
14           const vector3_t  dist( pos_j - pos_i );
15
16           // Σ_{i≠j}(m_j(x_j(t) − x_i(t)))/(|x_j(t) − x_i(t)|^3)
17           force.add( particles[j].mass / dist.norm_cubed(), dist );
18         }
19       }
20
21       // v_i(t + Δt) = v_i(t) + Δt · Σ ···
22       particles[i].velocity.add( dt, force );
23     }
24
25     // update position
26     for ( auto &  p : particles )
27       p.position.add( dt, p.velocity );  // x_i(t + Δt) = x_i(t) + Δt · v_i(t + Δt)
28   }
29 };
```

Remark

All values are scaled such that $G = 1$.

Example: N-Body Problem

The implementation follows the AOS approach. The computationally expensive loop is the inner loop of the `timestep` function.

Compiling the source code reveals some problems:

```
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.
```

The first report corresponds to the outer loop:

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

The inner loop starting at line 11 has a potential *division by zero*:

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

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

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

Example: N-Body Problem

The main problem is the inner loop. Two possible solutions are:

1. Enforce vectorisation even though division by zero occurs:

```
#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:

```
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:

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

Both changes will result in a speedup of

SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
1.92x	2.05x	2.03x

Example: N-Body Problem

The previous implementation uses the AOS approach, whereas an SOA based code would instead look like (changes marked red):

```
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:

	SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
SOA vs AOS	1.03x	1.01x	3.18x

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:

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

This yields an additional (and absolute) speedup of

	SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
SOA aligned vs SOA	1.01x	1.01x	1.06x
SOA aligned vs AOS	1.04x	1.02x	3.36x

None of the imlementations will benefit significantly from alignment.

Example: N-Body Problem

Implementing the same algorithm without any compound datatypes, and thereby having a C source, additional reduction operations may explicitly be applied:

```
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:

	SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
SOA in C vs SOA in C++	1.00x	1.00x	1.00x

Example: N-Body Problem

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

	SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
AOS pragma vs. AOS	1.92x	2.05x	2.03x
SOA vs AOS pragma	1.03x	1.01x	3.18x
SOA aligned vs SOA	1.01x	1.01x	1.06x
SOA in C vs SOA	1.00x	1.00x	1.00x
optimal vs base	2.00x	2.09x	6.84x

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

	SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
AOS float vs AOS double	2.41x	2.48x	1.53x
AOS pragma vs AOS	1.17x	2.20x	2.76x
SOA vs AOS pragma	1.18x	1.55x	3.08x
SOA aligned vs SOA	0.96x	0.98x	1.10x
SOA in C vs SOA	1.04x	0.95x	0.99x
optimal vs base	1.38x	3.41x	9.35x

Manual Vectorisation

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

- Learning a new language.
- Syntax is more error prone.
- Manual allocation of vector registers.
- ...

Hence, this is suggested only in *extreme* cases and if you know what your are doing!

```
..B1.6:
        lea       16(%r8), %eax
        cmpq      %rax, %rcx
        jl        ..B1.19
..B1.7:
        movl      %edi, %eax
        subl      %r8d, %eax
        andl      $15, %eax
        subl      %eax, %edi
        xorl      %eax, %eax
        testq     %r8, %r8
        jbe       ..B1.11

..B1.8:
        vmovsd    .L_2il0floatpacket.3(%rip), %xmm0

..B1.9:
        vmulsd    (%rdx,%rax,8), %xmm0, %xmm1
        vaddsd    (%rsi,%rax,8), %xmm1, %xmm2
        vmovsd    %xmm2, (%rsi,%rax,8)
        incq      %rax
        cmpq      %r8, %rax
        jb        ..B1.9

..B1.11:
        vmovupd   .L_2il0floatpacket.4(%rip), %ymm0
        movslq    %edi, %rax

..B1.12:
        vmovupd   (%rdx,%r8,8), %xmm1
        vinsertf128 $1, 16(%rdx,%r8,8), %ymm1, %ymm2
        vmulpd    %ymm2, %ymm0, %ymm3
        vaddpd    (%rsi,%r8,8), %ymm3, %ymm4
        vmovupd   %ymm4, (%rsi,%r8,8)
```

Compiler Intrinsics

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`.

Compiler Intrinsics

Vector Types

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

SSE2	`__m128d` (2x double)	`__m128` (4x float)
AVX	`__m256d` (4x double)	`__m256` (8x float)
MIC	`__m512d` (8x double)	`__m512` (16x float)

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

```
__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:

```
__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

```
__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 (sqrt(A0),...,sqrt(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)
```

Compiler Intrinsics

The Intel compiler also provides intrinsics for high level functions, e.g.:

```
__m256d _mm256_invsqrt_pd ( __m256d A );             // return (1/√A0, ..., 1/√A3)
__m256d _mm256_cbrt_pd    ( __m256d A );             // return ( ∛A0, ..., ∛A3, )
__m256d _mm256_invcbrt_pd ( __m256d A );             // return (1/∛A0, ..., 1/∛A3, )
__m256d _mm256_pow_pd     ( __m256d A, __m256d B );  // return (A0^B0, ..., A3^B3)

__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:

```
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:

```
typedef  __m128d  complex_t;
```

The fundamental arithmetic operations are then implemented using SSE2 functions:

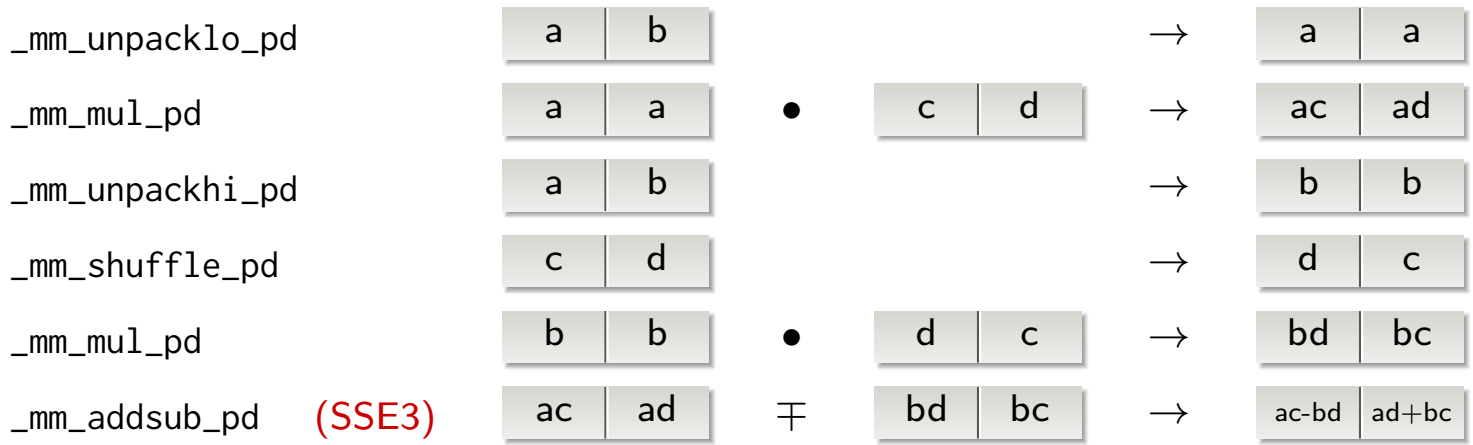
```
complex_t operator + ( const complex_t  z0, const complex_t  z1 ) {
  return _mm_add_pd( z0, z1 );
}

complex_t operator - ( const complex_t  z0, const complex_t  z1 ) {
  return _mm_sub_pd( z0, z1 );
}
```

Complex Numbers

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:



The final function looks as

```
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.

Complex Numbers

Similarly, the complex division is implemented:

```
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

```
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%.

Complex Numbers

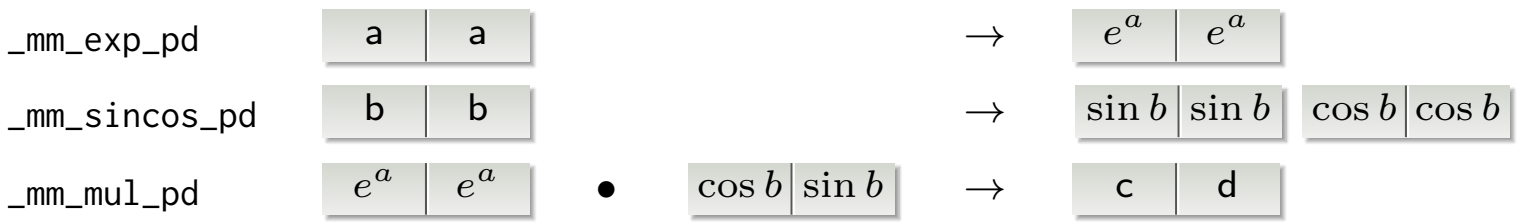
Using the high level intrinsics of the Intel compiler, more complex functions may be implemented.

As an example, consider the complex exponential function:

$$e^{a+bi} = (e^a \cos b) + (e^a \sin b) \cdot i$$

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

A vectorised version consists of the steps



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.

Complex Numbers

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

```
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.

OpenMP 4

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:

```
#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:

```
#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.

Cilk

The *Cilk* C/C++ language extension contains a special array notation:

```
double  a[n];

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

Working with arrays simplifies to single statements:

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

```
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.

Literature

"*A Guide to Auto-vectorization with Intel C++ Compilers*".
http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers.

"*Intel 64 and IA-32 Architectures Software Developer Manuals*".
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

"*Intel C++ Compiler XE 13.1 User and Reference Guide*".
http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm.

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