Today

Last lecture

- Memory hierarchies
- Basic performance models

Today

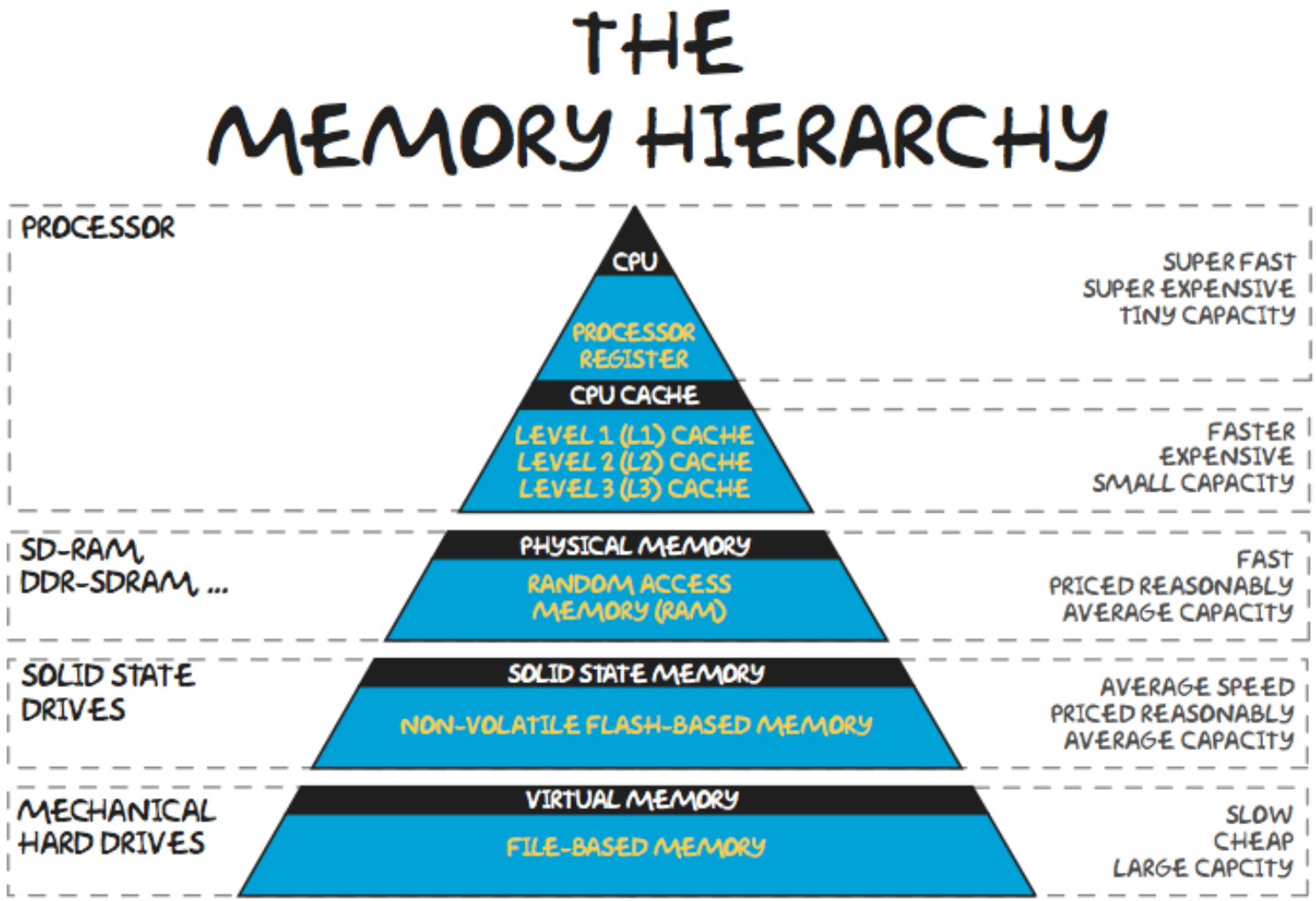
- Tools: valgrind, cachegrind
- Single core performance
- Vectorization and pipelining
- Tool: git

Announcements and upcoming

- No class next week 2/17 because of Presidents' Day
- No office hour on 2/17
- First homework is due Mon, Feb 24, 2020

Memory hierarchies

On my Mac Book Pro: 32KB L1 Cache, 256KB L2 Cache, 3MB Cache, 8GB RAM



CPU: $\mathcal{O}(1\text{ns})$, L2/L3: $\mathcal{O}(10\text{ns})$, RAM: $\mathcal{O}(100\text{ns})$, disc: $\mathcal{O}(10\text{ms})$

Memory hierarchies

Important terms:

- latency: time it takes to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)
- bandwidth: rate at which data can be read/written (for large data); in (bytes/second);
- cache-hit: required data is available in cache $\Rightarrow$ fast access
- cache-miss: required data is not in cache and must be loaded from main memory (RAM) $\Rightarrow$ slow access

Computer architecture is complicated. We need a basic performance model.

- Processor needs to be "fed" with data to work on.
- Memory access is slow; memory hierarchies help.

1. Only consider two levels in hierarchy, fast (cache) and slow (RAM) memory
2. All data is initially in slow memory
3. Simplifications:
   - Ignore that memory access and arithmetic operations can happen at the same time
   - assume time for access to fast memory is 0
4. Computational intensity: flops per slow memory access

$$q = \frac{f}{m}, \text{ where } f \ldots \#\text{flops}, m \ldots \#\text{slow memop}.$$

Computational intensity should be as large as possible.

Memory hierarchy

Example: Matrix-matrix multiply Comparison between naive and blocked optimized matrix-matrix multiplication for different matrix sizes: Different algorithms can increase the computational intensity significantly.
BLAS: Optimized Basic Linear Algebra Subprograms

- Temporal and spatial locality is key for fast performance.
  - Eliminate memory operations by saving data in fast memory and reusing them, i.e., temporal locality: Access an item that was previously accessed
  - Explore bandwidth by moving a chunk of data into the fast memory: spatial locality: Access data nearby previous accesses
- Since arithmetic is cheap compared to memory access, one can consider making extra flops if it reduces the memory access.
- In distributed-memory parallel computations, the memory hierarchy is extended to data stored on other processors, which is only available through communication over the network.

Valgrind and cachegrind

Valgrind

- memory management tool and suite for debugging, also in parallel
- profiles heap (not stack) memory access
- simulates a CPU in software
- running code with valgrind makes it slower by factor of 10-100
- Documentation: http://valgrind.org/docs/manual/

memcheck

finds leaks
inval. mem. access
uninitialize mem.
incorrect mem. frees

cachegrind

cache profiler
sources of cache misses

callgrind

extension to cachegrind
function call graph

Valgrind and cachegrind

Usage (see examples):

Run with valgrind (no recompile necessary!)

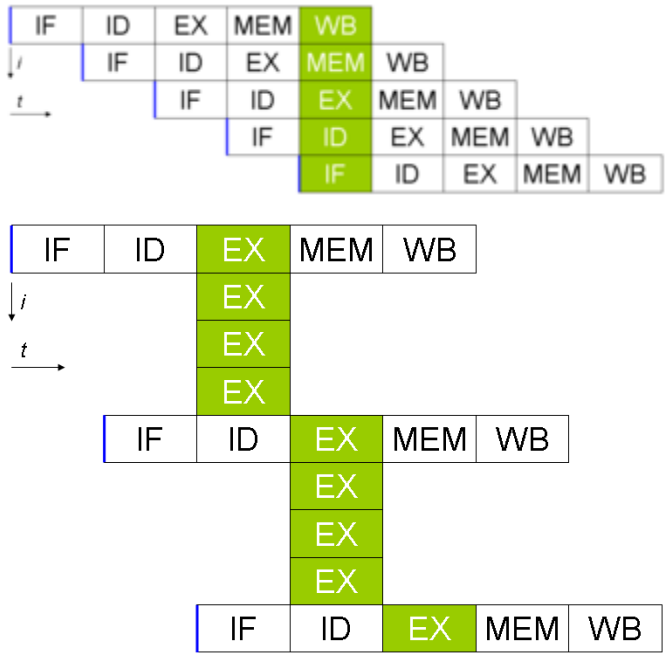
```
valgrind --tool=memcheck [options] ./a.out [args]
```

Levels of parallelism

- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle

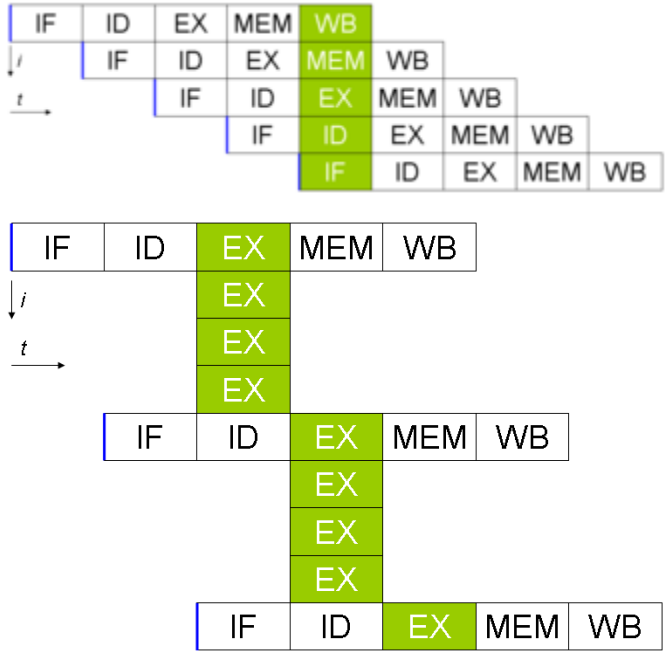
Levels of parallelism

- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle
- Vectorization: single instruction on multiple data (SIMD)



Levels of parallelism

- Parallelism by pipelining (overlapping of execution of multiple instructions); "assembly line" parallelism, Instruction-Level-Parallelism (ILP); several operators per cycle



- Vectorization: single instruction on multiple data (SIMD)

all of the above assume single sequential control flow

- process/thread level parallelism: independent processor cores, multicore processors; parallel control flow

Vectorization

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 vectorization with Intel compiler (similar statments for gcc)

```
#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 vectorization

```
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)

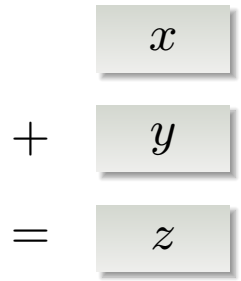
Vectorization

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

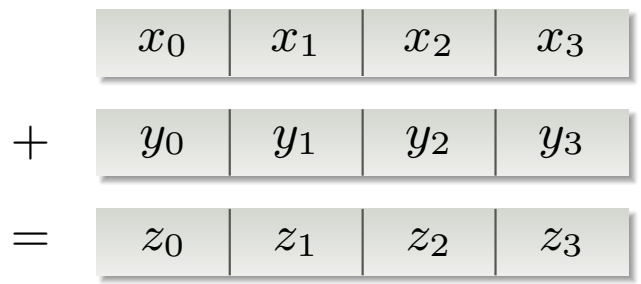
$$t_{\mathrm{load}} + t_{\mathrm{op}}$$



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

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

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



with costs

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

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

Vectorization

vectorization 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 vectorization

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-vectorization, 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
```

For gcc

- Flag "-ftree-vectorize" turns auto-vectorization on
- Flag "-mavx" and "-msse2" tells the compile what is supported
- Flag "-fopt-info" gives information about vectorization (cryptic)
- If "-O3", then automatically vectorizes
- Both compilers will only vectorize for optimisation levels -O2 or higher

DEMO (vec01.cpp)

Auto vectorization

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, vectorization may be

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

Furthermore, to achieve *maximal* vectorization 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

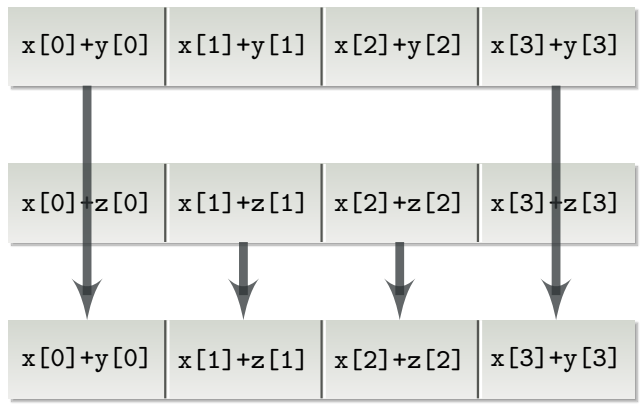
```
void f ( int n, double * x, double * y, double * z ) {
   for ( int  i = 0; i < n; ++i ) {
      if ( i % 2 == 0 )
        z[i] =       x[i]*y[i];
      else
        z[i] = z[i] + x[i]*y[i];
   }
}
```

Branches can prevent pipelining as well

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 cannot 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 vectorize 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 vectorized (depends on compiler!):

```
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 vectorized 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] );
   }
}
```

The inline functions are a C++ enhancement feature to increase the execution time of a program. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called.

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

DEMO (vec02.cpp)

Non-Unit stride

If the loop variable does not follow a unit increment, vectorization 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 vectorize, 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] );
   }
}
```

DEMO (vec01.cpp with STRIDE)

Indirect addressing

If the memory position is accessed *indirectly*, vectorization 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] ];
   }
}
```

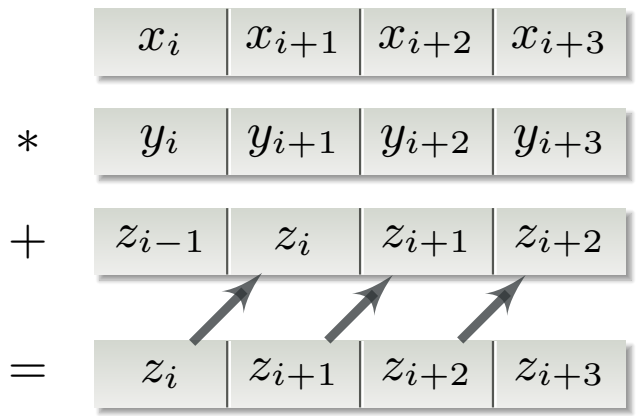
Indirect addressing is a special form of non-unit stride.

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 vectorized.

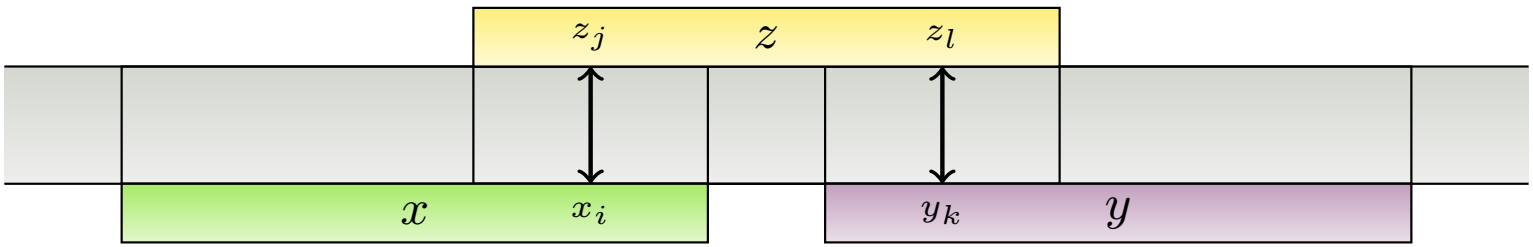
Note: Except, if dependency is not within one vector register, e.g. $z_i = z_{i-4} + \cdots$ for AVX.

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.

The compiler may either omit vectorization 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, vectorization unsafe
        ...
    }
    else {
        // no aliasing, vectorization safe
        ...
    }
}
```

Remark

For small n, the extra tests might be too expensive!

DEMO (vec03.cpp)

Aliasing

One may explicitly tell the compiler that no aliasing exists.

Pragma for specific function

```
//#pragma GCC ivdep
for(int i = 0; i < n; ++i) {
  z[i] = z[i] + x[i]*y[i];
}
```

or via command line

```
> g++ -O2 -ftree-vectorize -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

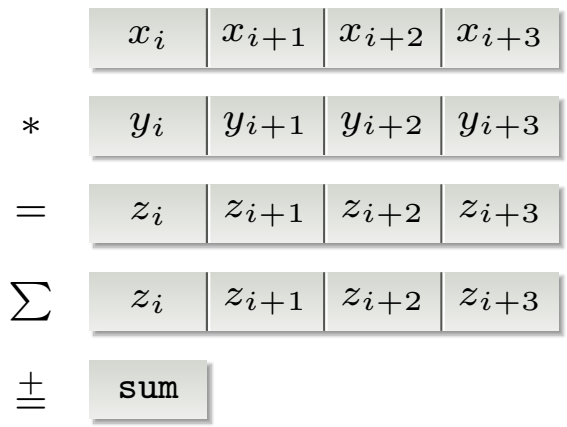
A special variant of data dependencies are *reduction* operations, e.g.

```
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 vectorized, 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 OpenMP pragma `simd reduction`:

```
double dot ( int n, double * x, double * y ) {
   double  sum = 0.0;

   #pragma omp 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 omp simd reduction (+:xsum,ysum)
   for ( int  i = 0; i < n; ++i ) {
      xsum = xsum + x[i];
      ysum = ysum + y[i];
   }
}
```

Remark

Note that this is an OpenMP statement

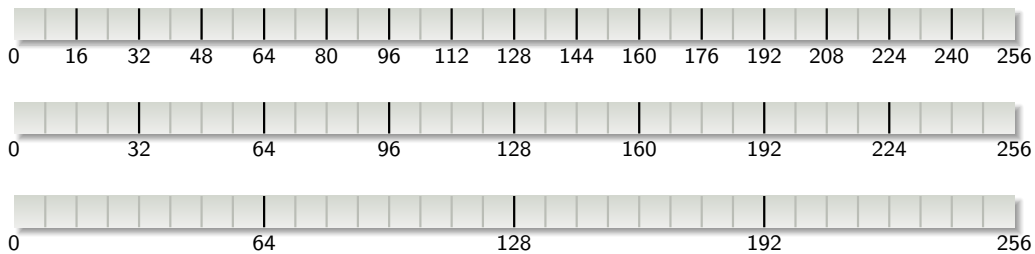
DEMO (vec04.cpp)

Memory 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.

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
  ...
}
```

Dynamic Data

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

Direct allocation of aligned data (C++11):

```
  void f ( const size_t  n ) {
    double * a = aligned_alloc(VEC_LEN*sizeof(double), n * sizeof(double));
    ...
  }
```

Vectorization reports

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

Intel Compiler

vectorization reports are activated via

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

with

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

GNU Compiler

The command line option for the GNU compiler is

```
> g++ -O2 -ftree-vectorize -fopt-info -mavx -c f.cc
```

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.
```

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.
```

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.
```

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.
```

Further examples for messages are:

Data type unsupported on given target architecture

As an example, complex data types can only be vectorized efficiently using SSE3 instructions. Therefore, if only SSE2 is supported, vectorization will be omitted.

Low trip count

The number of loops is not sufficient for vectorization.

Not inner loop

Only the innermost loop will be vectorized. Note, that the compiler may apply loop-unrolling *and change the order of nested loops!*

Manual vectorization

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`.

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;
};
```

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+B

__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)
```

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

```
__m256d _mm256_invsqrt_pd ( __m256d A );              // return (1/sqrt(A0),...,1/sqrt(A3))
__m256d _mm256_cbrt_pd    ( __m256d A );              // return (cbrt(A0),...,cbrt(A3),)
__m256d _mm256_invcbrt_pd ( __m256d A );              // return (1/cbrt(A0),...,1/cbrt(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(B
```

OpenMP SIMD

The OpenMP standard starting with 4.0 contains vectorization instructions, very similar to the corresponding instructions of the Intel compiler.

Loop vectorization

To instruct the compiler to apply vectorization, 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 vectorization.