Loops, ILP, vectorization, Stream SIMD Extensions (SSE)

Salvatore Orlando

Thanks to Mats Aspnäs @ Åbo Akademi

Optimizing single thread performance

- **Assuming that all instructions are doing useful work, how can you make the code run faster?**
  - **Some sequence of code runs faster than other sequence**
    - **Optimize for memory hierarchy**
    - **Optimize for specific architecture features such as pipelining**
  - **Often optimizations correspond to changing the execution order of the instructions.**

```
A[0][0] = 0.0;
A[1][0] = 0.0;
…
A[1000][1000] = 0.0;
```

```
A[0][0] = 0.0;
A[0][1] = 0.0;
…
A[1000][1000] = 0.0;
```

Both code initializes A, is one better than the other?

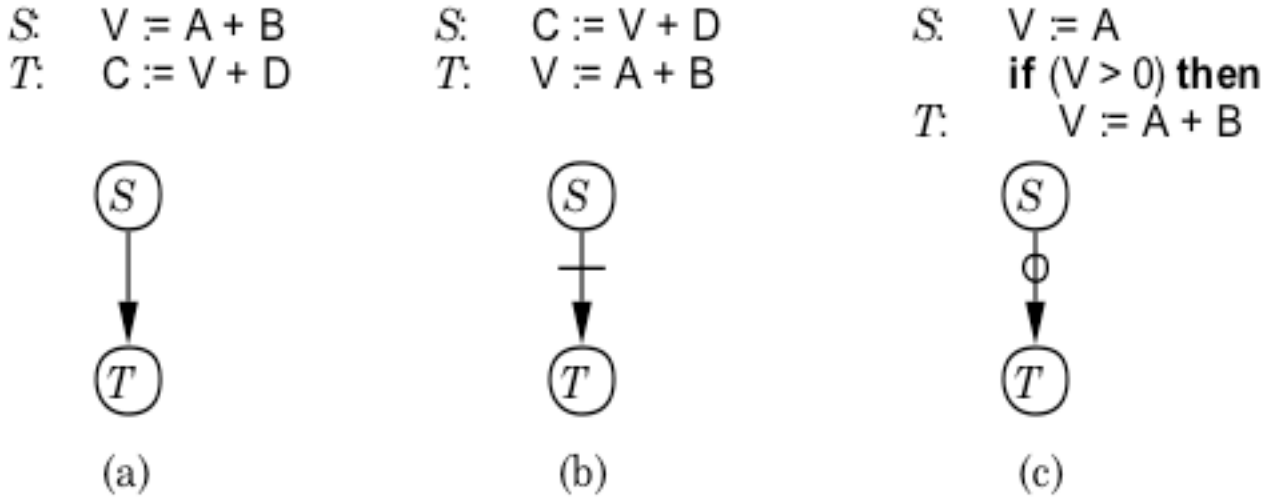
When is it safe to change order?

- When can you change the order of two instructions without changing the semantics?
  - They do not operate (read or write) on the same variables.
  - They can be only read the same variables

- This is formally captured in the concept of data dependence
  - True dependence: Write X – Read X (RAW)
  - Output dependence: Write X – Write X (WAW)
  - Anti dependence: Read X – Write X (WAR)
  - If you detect RAR, it's safe changing the order

Data dependencies

- ***Data-Flow* or *True Dependency***
  - **(a) RAW (Read After Write)**
- ***Anti Dependency***
  - **(b) WAR (Write After Read)**
- ***Output Dependency***
  - **(c) WAW (Write After Write)**



Optimizing through loop transformations

- 90% of execution time in 10% of the code
  - Mostly in loops
- Relatively easy to analyze
- Loop optimizations
  - Different ways to transform loops with the same semantics
  - Goal
    - Single-thread system: mostly optimizing for memory hierarchy.
    - Multi-thread and vector system: loop parallelization

Loop dependencies

```
do i1 = 1, n1
   doi2 = 1, n2
      .
        .
           doik = 1, nk
              <Loop body>
              ….
              S
              T
              ….
           enddo
        .
      .
   enddo
enddo
```

- **We have to look at the dependencies between instances of *S* and *T*, both statements of the body loop*:***

  i.e.: $S_{h1,h2,\ldots,hk}$
  $T_{m1,m2,\ldots,mk}$

- **If the dependencies refer to instances of *S* and *T* belonging to distinct loop iterations, we have:**

  loop-carried dependencies

$S_{1,1,1,1,1}$ $S_{1,1,1,1,2}$ … $S_{1,1,1,1,nk}$ $S_{1,1,1,2,1}$ $S_{1,1,1,2,2}$ …
$T_{1,1,1,1,1}$ $T_{1,1,1,1,2}$ $T_{1,1,1,1,nk}$ $T_{1,1,1,2,1}$ $T_{1,1,1,2,2}$

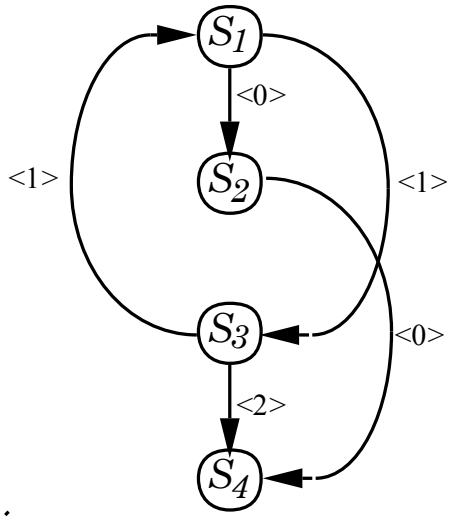
Loop-carried dependence

- **Loop-carried dependence may prevent loops from being parallelized.**
  - **Important since loops contains most parallelism in a program.**
- **Loop-carried dependence can sometimes be represented by dependence vectors**
  - **tells which iteration depends on which iteration.**
- **When one tries to change the loop execution order, the loop carried dependence needs to be honored.**

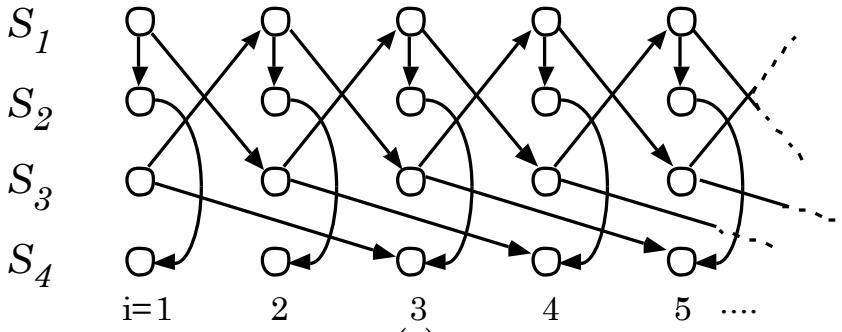
An example of loop and dependency graphs

```
do  i = 1, N
      S1:  A[i] := B[i] + C[i - 1]
      S2:  D[i] := A[i] * 2
      S3:  C[i] := A[i - 1] + C[i]
      S4:  E[i] := D[i] + C[i - 2]
enddo
```

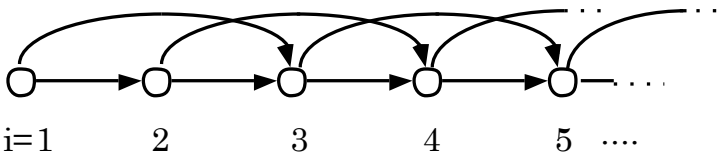
(a)



(b)



(c)



(d)

a) Loop
b) Dependency graph with vectors <i>: *loop-carried*
c) Dependency graph with spatial distribution of the iterations
d) The same graph of point c), with a node for each loop body instance

Vectorization

- **The vector SIMD machines have instructions that work in an efficient ways on *vectors***

- **A vector instruction corresponds to an *inner-loop* whose operations are independent**
  - **without *loop-carried dependencies***

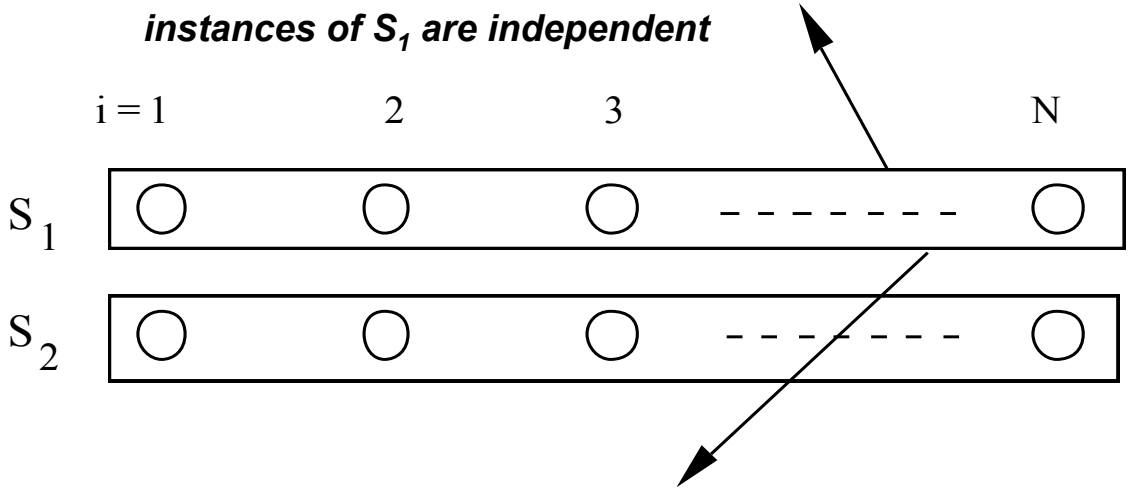
```
do i = 1, N
   A[i]  :=  B[i] * C[i]   ⇒      A[1:N] = B[1:N] * C[1:N]
enddo
```

Conditions for vectorizing

```
do i = 1, N
   S1
   S2
enddo
```

Vector instruction (provided that registers are large enough, otherwise multiple instructions):

instances of S_1 are independent



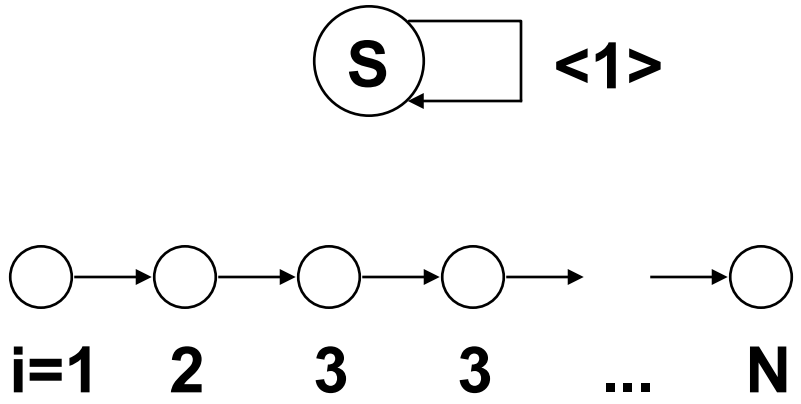
No of these operations must depend on operations in the next vector instructions

No backward edges in the dependency graph

Dependencies that prevent vectorization

- **We cannot vectorize if we have:**
  - ***statements that depend on themselves***
  - **i.e., *loop-carried dependencies* represented as cycles in the dependency graph**

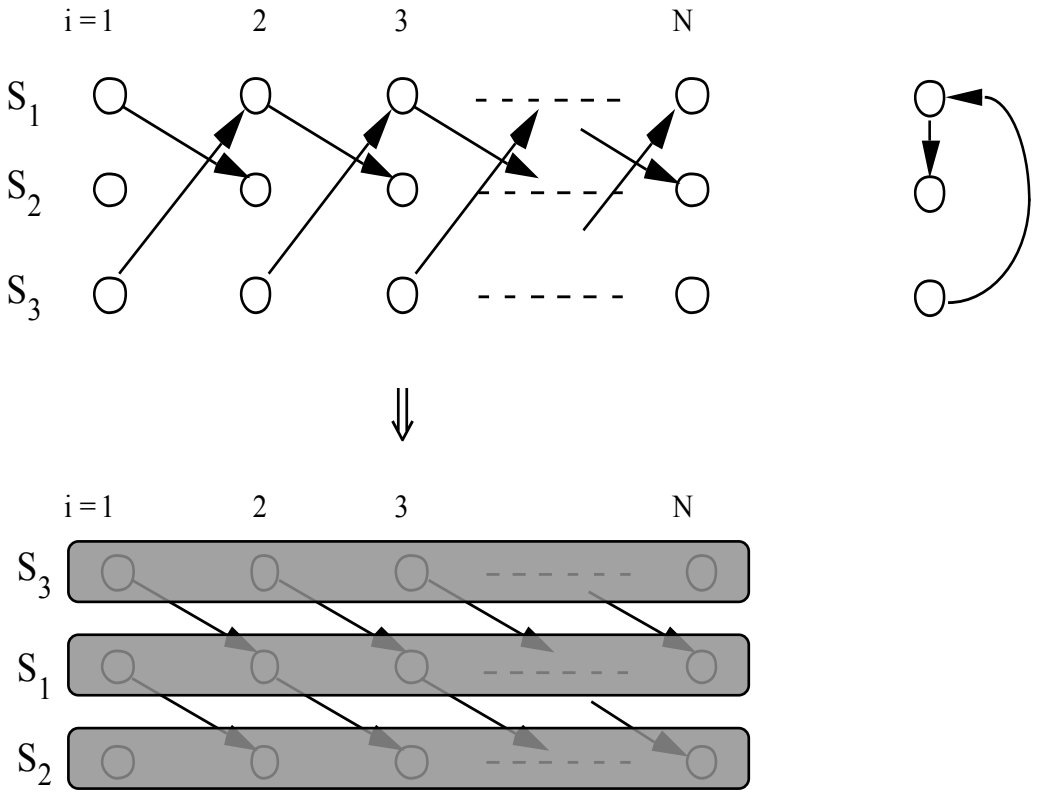
```
do i = 1,N
    S:  A[i+1]:= A[i] + const
enddo
```



Dependencies that prevent vectorization

- ***Loop-carried backward dependencies***
  - **We can remove by REORDERING**

```
doi = 1,N
    S1:   A[i+1]  := C[i] + const
    S2:   B[i]    := B[i]+ A[i]
    S3:   C[i+1]  := D[i] + const
enddo
```



Dependencies that prevent vectorization

- **Loop interchange**
  - **In the following example it is enough to exchange the loop indexes for being able to vectorize**

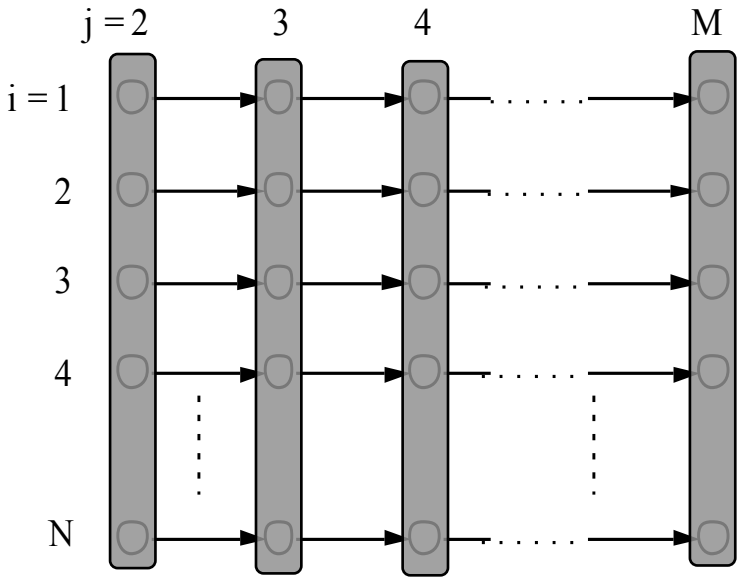
```
do i = 1, N
   do j = 2, M
      S:   A[i,j]  :=  A[i,j-1] * B[i]
   enddo
enddo
```

⇓

```
do j = 2, M
   do i = 1, N
      S:   A[i,j]  :=  A[i,j-1] * B[i]
   enddo
enddo
```

⇓

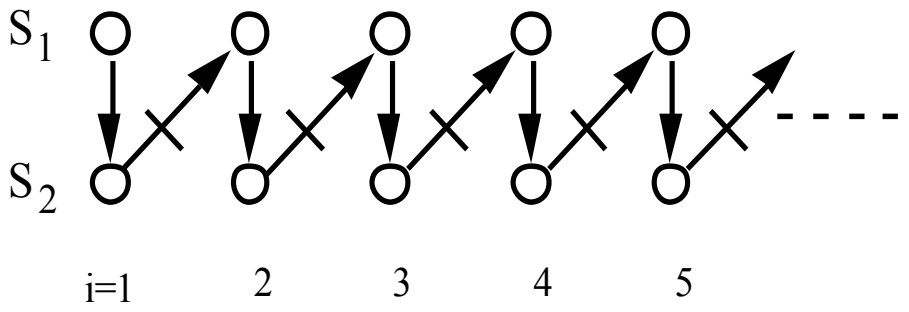
```
do  j = 2, M
      A[1:N, j]  :=  A[1:N, j-1] * B[1:N]
enddo
```



Renaming to remove dependencies

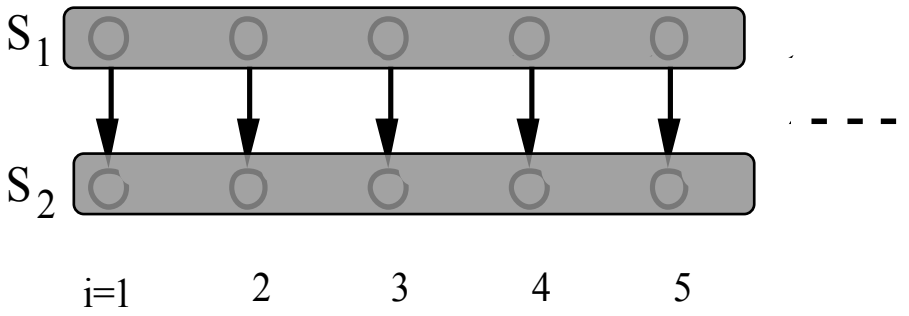
- **The dependencies different from RAW (data-flow) can be removed by introducing a new variable for each new assignment (see *single-assignment languages)***
  - ***Example of WAR*** **(R:** A[i+1] at iteration i, **W:** A[i] at iteration i+1)

```
do i = 1,N
   S1:  A[i] := B[i] * C[i]
   S2:  D[i] := A[i+1] + A[i]
enddo
```



The assignments of A[ ] **are replaced by assignments to a new temporary** T[ ]**.**
Renaming **of all the next *usages* of** A[ ] **with** T[ ]

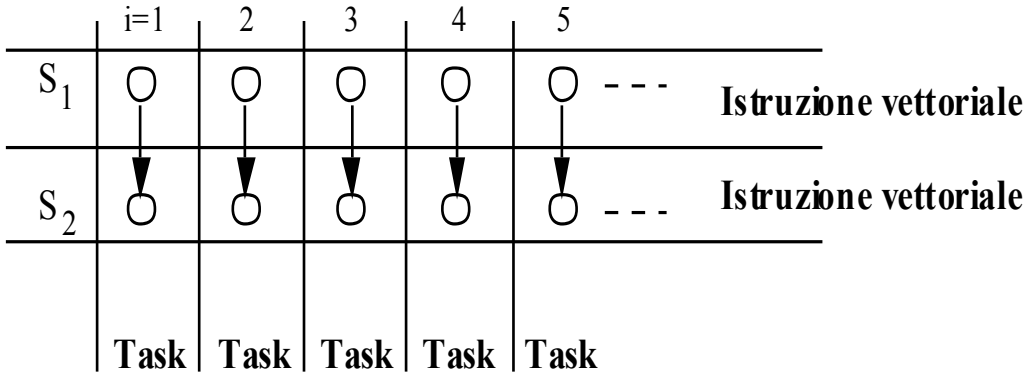
```
do i = 1,N
   S1:  T[i] := B[i] * C[i]
   S2:  D[i] := A[i+1] + T[i]
enddo
```



Loop parallelization on SM Multiprocessors

```
do i = 1,N
   S1:  T[i]  := B[i]  * C[i]
   S2:  D[i]  := D[i] + T[i]
enddo
```

```
doall i = 1,N
   S1:  T[i]  := B[i]  * C[i]
   S2:  D[i]  := D[i] + T[i]
end doall
```



To transform a loop into *doall (forall)*, where each body loop is an independent task executed by a distinct thread, no *loop-carried dependencies must exist*

Parallelization on SM Multiprocessors: *Loop alignment*

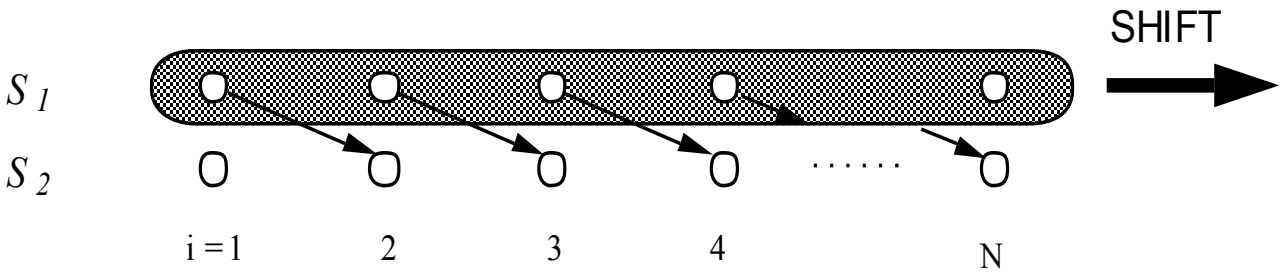
```
do i = 1,N
   S1:  A[i+1] := B[i] * C[i]
   S2:  D[i] := A[i] + 2
enddo
```

⇒

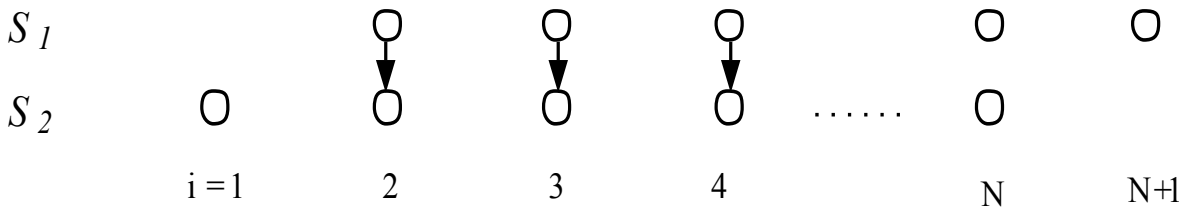
S_2: D[1] := A[1] + 2

```
doall i=2,N
   S1: A[i] := B[i-1] * C[i=1]
   S2: D[i] := A[i] + 2
```

S_1: A[N+1] := B[N] * C[N]



(a)



(b)

Loop tiling to optimize memory hierarchies

- **Replacing a single loop into two loops.**

  For(i=0; i<n; i++) … →

  ```
  for(i=0; i<n; i+=t)
     for (ii=i, ii < min(i+t,n); ii++) …
  ```

  t is called *tile size*

```
For (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; j<n; k++)
```

```
For (i=0; i<n; i+=t)
   for (ii=i; ii<min(i+t, n); ii++)
     for (j=0; j<n; j+=t)
       for (jj=j; jj < min(j+t, n); jj++)
         for (k=0; j<n; k+=t)
           for (kk = k; kk<min(k+t, n); kk++)
```

Loop tiling to optimize memory hierarchies

- When using with loop interchange, loop tiling create inner loops with smaller memory trace – great for locality.
  - Reduce the working set size and change the memory reference pattern.

```
for (i=0; i<n; i+=t)
   for (ii=i; ii<min(i+t, n); ii++)
      for (j=0; j<n; j+=t)
         for (jj=j; jj < min(j+t, n); jj++)
            for (k=0; j<n; k+=t)
               for (kk = k; kk<min(k+t, n); kk++)
```

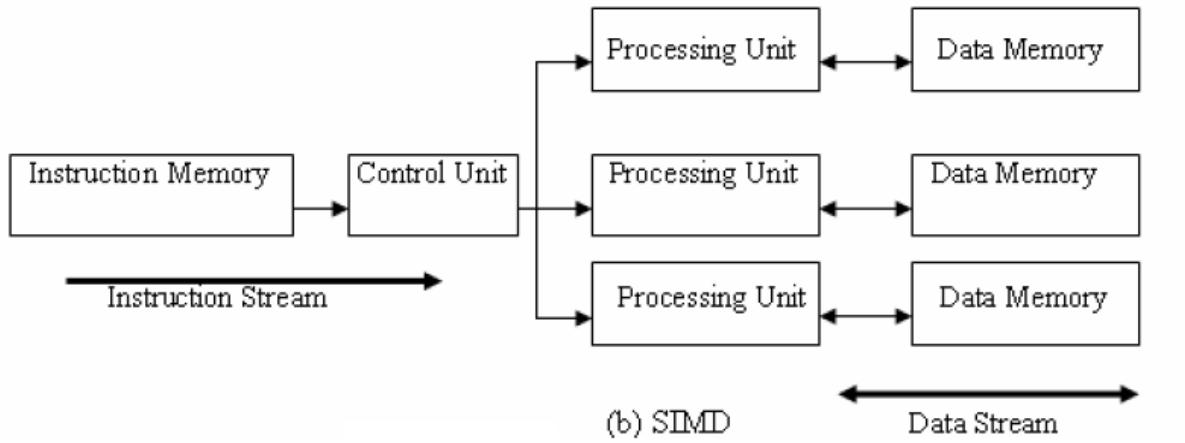
Inner loop with much smaller memory footprint

```
for (i=0; i<n; i+=t)
   for (j=0; j<n; j+=t)
      for (k=0; k<n; k+=t)
         for (ii=i; ii<min(i+t, n); ii++)
            for (jj=j; jj < min(j+t, n); jj++)
               for (kk = k; kk<min(k+t, n); kk++)
```

Streaming SIMD Extension (SSE)

SIMD architectures

- A data parallel architecture
- Applying the same instruction to many data
  - Save control logic
  - A related architecture is the vector architecture
  - SIMD and vector architectures offer high performance for vector operations.



Vector operations

- **Vector addition Z = X + Y**

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

$$\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \dots\dots \\ x_n + y_n \end{pmatrix}$$

- **Vector scaling Y = a * X**

  for(i=0; i<n; i++) y[i] = a*x[i];

$$a * \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} a * x_1 \\ a * x_2 \\ \dots\dots \\ a * x_n \end{pmatrix}$$

- **Dot product**

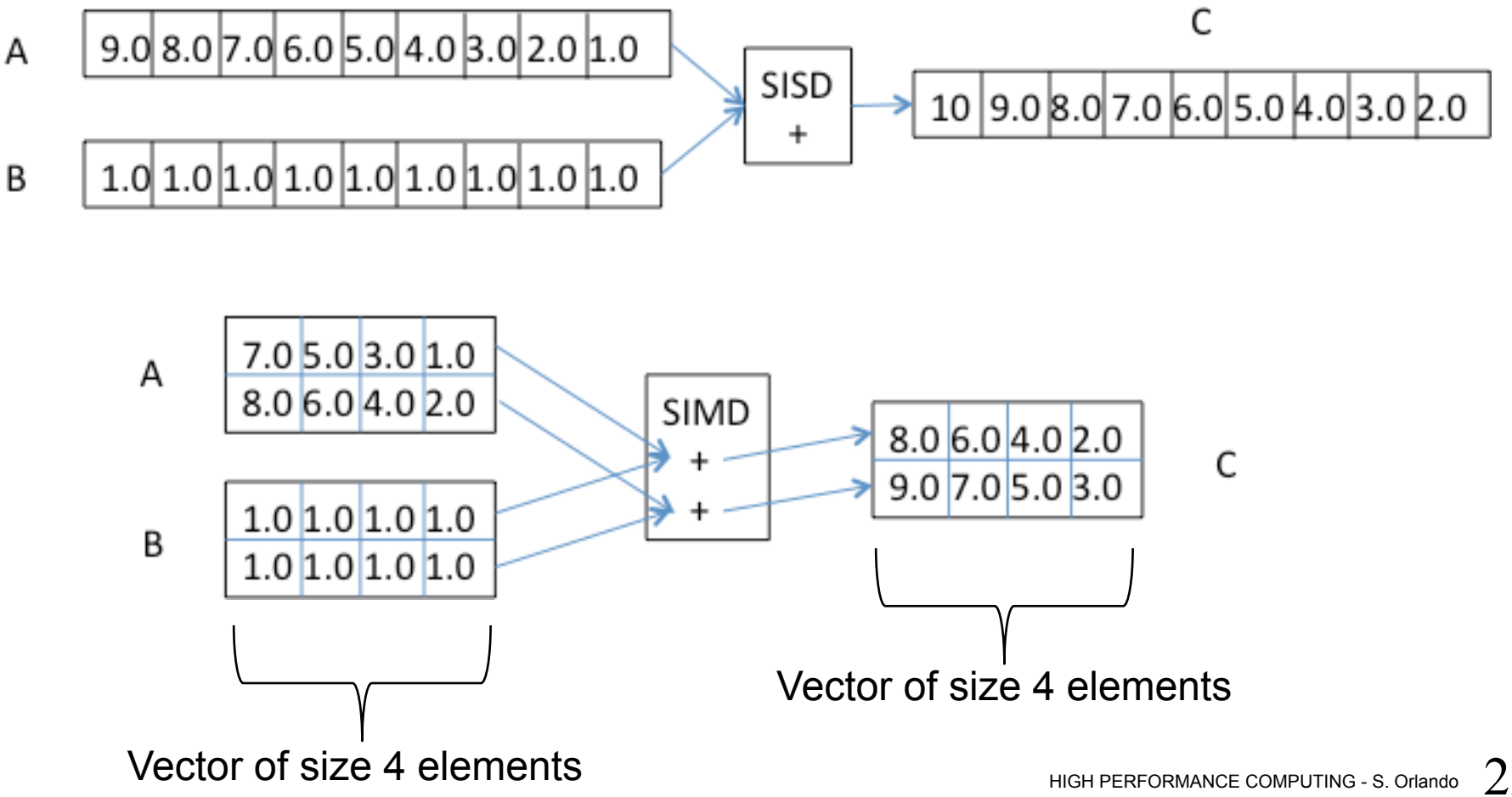
  for(i=0; i<n; i++) r += x[i]*y[i];

$$\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} \bullet \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = x_1 * y_1 + x_2 * y_2 + \dots\dots + x_n * y_n$$

SISD and SIMD vector operations

- **C = A + B**

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



x86 architecture SIMD support

- Both current AMD and Intel's x86 processors have ISA and microarchitecture support for SIMD operations.
- ISA SIMD support
  - MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX, AVX512
    - See the flag field in `/proc/cpuinfo`
    - `sysctl -a | grep features` on Mac OS X
  - SSE (Streaming SIMD extensions): a SIMD instruction set extension to the x86 architecture
    - Instructions for operating on multiple data simultaneously (vector operations).
- Micro architecture support of SSE
  - Many functional units
  - 8 128-bit vector registers, XMM0, XMM1, …, XMM7

SSE programming

- Vector registers support three data types:
  - Integer (16 bytes, 8 shorts, 4 int, 2 long long int, 1 dqword)
  - single precision floating point (4 floats)
  - double precision float point (2 doubles).

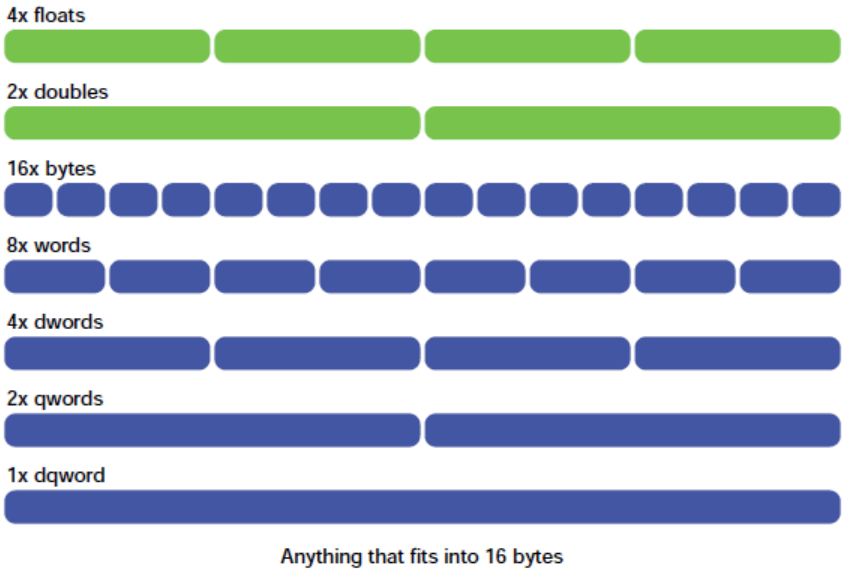


Figure 1. SSE/SSE2 data types

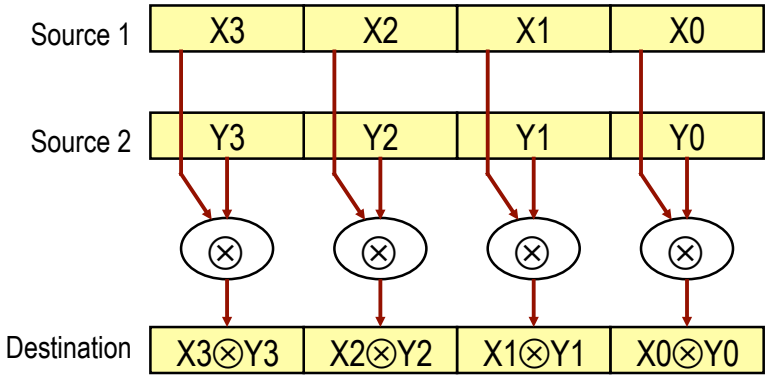
[Klimovitski 2001]

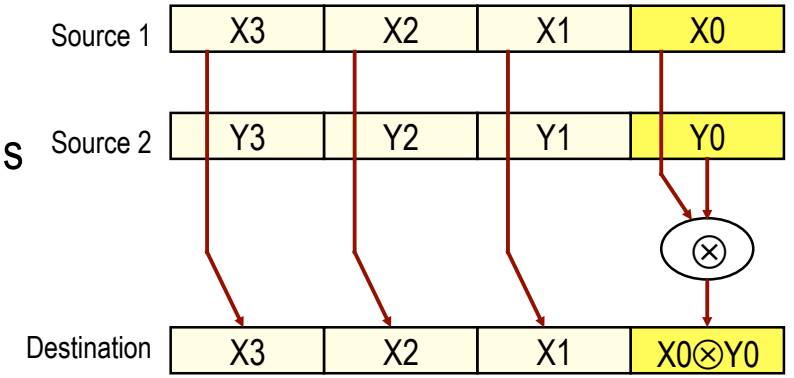
SIMD extensions

- MMX – Multimedia Extensions
  - introduced in the Pentium processor 1993
  - 8 64-bit vector registers, supports only integer operations,
- SSE – Streaming SIMD Extension
  - introduced in Pentium III 1999
  - 8 or 16 128-bit vector registers, supports single-precision floating point operations
- SSE2 – Streaming SIMD Extension 2
  - introduced in Pentium 4, 2000
  - 8 or 16 128-bit vector registers, supports double-precision floating point operations – later extensions: SSE3, SSSE3, SSE4.1, SSE4.2
- AVX – Advanced Vector Extensions
  - announced in 2008, supported in the Intel Sandy Bridge processors, and later – extends the vector registers to 16 registers of length 256 bits
- AVX2 – Advanced Vector Extensions 2
  - introduced in the Haswell microarchitecture 2013 – extends the vector registers to 16 256-bit registers
- AVX-512 are 512-bit extensions to AVX2
  - proposed by Intel in July 2013, and scheduled to be supported in 2015 with Intel's Knights Landing processor.

Packed and scalar operations

- Supports both packed and scalar operations
  - packed instructions have a suffix P
  - scalar instructions have a suffix S
- Packed vector instructions apply an operation in parallel on all values in the register
- Scalar instructions apply an operation on a single (scalar) value
- The compilers use scalar SSE/AVX instructions for floating-point operations instead of the x87 floating point unit





Programming with vector instructions

There are different ways to use vector instructions in a program

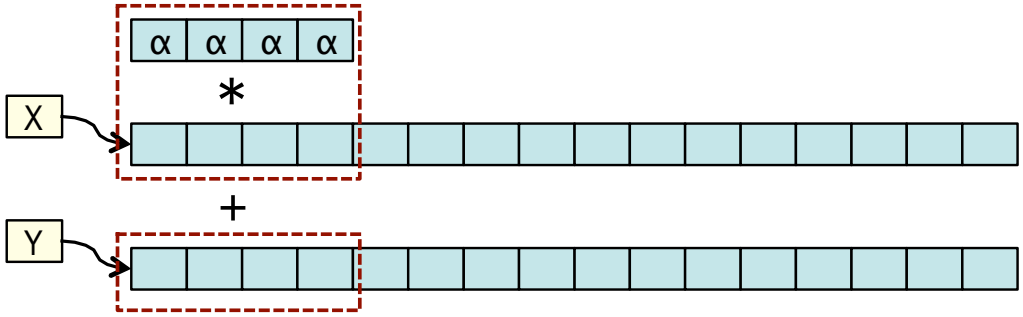
1. **Automatic vectorization by the compiler**
   - **no explicit vectorized programming is needed, but requires a vectorizing compiler**
   - **have to express the code so that the compiler can recognize possibilities for vectorization (in particular aliasing and alignment)**
   - **enable compiler vectorization in GCC with –O3 or *-ftree-vectorize***
2. **Use a vectorized library**
   - **available for instance in Intel Math Kernel Library, AMD Core Math Library, Intel compiler's vector classes, ...**
3. **Express the computation as arithmetic expressions on vector data types**
   - **declare variables of a vector data type**
   - **express computations as normal arithmetic expression on the vector variables – the compiler generates vector instructions for the arithmetic operations**
4. **Use compiler intrinsic functions for vector operations**
   - **functions that implement vector instructions in a high-level language**
   - **requires detailed knowledge of the vector instructions**
   - **one function often implements one vector assembly language instruction**

Example: SAXPY procedure (SSE vectorizations)

- SAXPY (Single-precision Alpha X Plus Y)
  - computes Y = αX+Y, where α is a scalar value and X and Y are vectors of single-precision type
  - one of the vector operation in the BLAS library (Basic Linear Algebra Subprograms)

```
#define N 1000
void saxpy(float alpha, float *X, float *Y) {
  for (int i=0; i<N; i++)
       Y[i] = alpha*X[i] + Y[i];
}
```

- The SSE vectorized code will do the computation on 4 float values at a time
  - multiplies four values of *X* with *alpha*
  - adds the results to the corresponding four values of *Y*



Using compiler intrinsic functions

- Compiler built-in functions for vector operations on packed data
  - implemented as functions which call the corresponding vector instructions
  - implemented with inline assembly code
  - allows the programmer to use C/C++ function calls and vector variables instead of inline assembly code
- Defines a separate C/C++ function for each vector instruction
  - there are also some intrinsic functions composed of several vector instructions
- Vectorized programming with intrinsic functions is very low-level
  - have to exactly specify the operations that are done on the vector values

- Operate on the vector data types *__m128*, *__m128d* and *__m128i*
- Often used for vector operations that can not be expressed as normal arithmetic operations (+, -, *, /)
  - loading and storing of vectors, shuffle operations, type conversions, masking operations, …

SAXPY with vector intrinsic functions

```
#include <emmintrin.h>
#define N 1000
void saxpy(float alpha, float *X, float *Y) {
  __m128 x_vec, y_vec, a_vec, res_vec;  /* Declare vector variables */
  a_vec = _mm_set1_ps(alpha);           /* Vector of 4 alpha values */
  for (int i=0; i<N; i+=4) {
     x_vec = _mm_loadu_ps(&X[i]);      /* Load 4 values from X */
     y_vec = _mm_loadu_ps(&Y[i]);      /* Load 4 values from Y */
     res_vec = _mm_add_ps(_mm_mul_ps(a_vec, x_vec), y_vec); /* Compute */
     _mm_storeu_ps(&Y[i], res_vec);  /* Store the result */
  }
}
```

- Declare vector variables of the appropriate type (*m128* for 4 float values)
- Load data values into the variables
- Do arithmetic operations on the vector variables by calling intrinsic functions
  - it is possible to nest calls to intrinsic functions (they normally return a vector value)
  - can of course also use normal arithmetic expressions (+, -, *) on the vector variables
- Load and store operations are unaligned (*mm_loadu_ps*)
  - there are also corresponding functions for aligned load/store: *_mm_load_ps* and *_mm_store_ps*

Vector Instructions

- Arithmetic intrinsic functions have a mnemonic name that tries to describe the operation
  - the function name starts with *_mm*
  - after that follows a name describing the operation: *add, mul, div, load, set*, …
  - the next character specifies whether the operation is on a packed vector or on a scalar vaue: P stands for Packed and S for Scalar operation
  - the last character describes the data type
    - S – single precision floating point values
    - D – double precision floating point values
- Examples:
  - *_mm_load_ps* – load packed single-precision floating-point values
    _mm_add_sd – add scalar double precision values
    _mm_rsqrt_ps – reciprocal square root of four single-precision fp values
    _mm_min_pd – minimum of the two double-precision fp values in the arguments
    _mm_set1_pd – set the two double-precision elements to some value
- See the Intel Intrinsics Guide at https://software.intel.com/sites/landingpage/IntrinsicsGuide

Vectoring conditional constructs

- The compiler will typically not be able to vectorize loops containing conditional constructs
- *Example*: conditionally assign an integer value to *A[i]* depending on some other value *B[i]*

```
// A, B, C and D are integer arrays
for (i=0; i<N; i++) {
     A[i] = (B[i] > 0) ? (C[i]) : (D[i]);
}
```

```
for (i=0; i<N; i++) {
    if (B[i] > 0)
        A[i] = C[i];
    else
        A[i] = D[i];
}
```

- This can be vectorized by first computing a Boolean mask which contains the result of the comparison: *mask[i] = (B[i] > 0)*
  - then the assignment can be expressed as

    A[i] = (C[i] && mask) || (D[i] && ¬mask)

    where the logical operations AND (&&), OR (||) and NOT (¬) are done bitwise on the values

Example

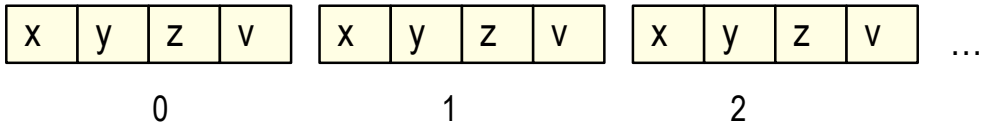
- As an example we write a vectorized version of the following (slightly modified) code

```
for (int i=0; i<N; i++) {
   A[i] = (B[i] > 0) ? (C[i]+2) : (D[i]+10);
}
```

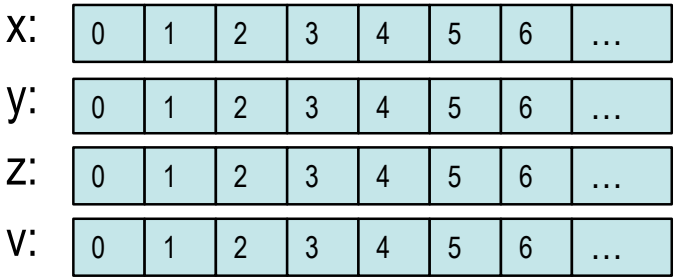
```
__m128i zero_vec = _mm_set1_epi32(0);   // Vector of four zeros
__m128i two_vec  = _mm_set1_epi32(2);   // Vector of four 2's
__m128i ten_vec  = _mm_set1_epi32(10);  // Vector of four 10's

for (int i=0; i<N; i+=4) {
    __m128i b_vec, c_vec, d_vec, mask, result;
    b_vec = _mm_load_si128((__m128i *)&B[i]);  // Load 4 elements from B
    c_vec = _mm_load_si128((__m128i *)&C[i]);  // Load 4 elements from C
    d_vec = _mm_load_si128((__m128i *)&D[i]);  // Load 4 elements from D

    c_vec = _mm_add_epi32(c_vec, two_vec);     // Add 2 to c_vec
    d_vec = _mm_add_epi32(d_vec, ten_vec);     // Add 10 to d_vec
    mask = _mm_cmpgt_epi32(b_vec, zero_vec);   // Compare b_vec to 0
    c_vec = _mm_and_si128(c_vec, mask);        // AND c_vec and mask
    d_vec = _mm_andnot_si128(mask, d_vec);     // AND d_vec with NOT(mask)
    result = _mm_or_si128(c_vec, d_vec);       // OR c_vec with d_vec
   _mm_store_si128((__m128i *)&A[i], result);  // Store result in A[i]
}
```

Arranging data for vector operations

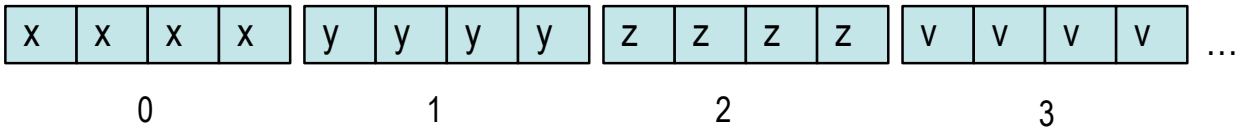
- It is important to organize data in memory so it can be accessed as vectors
  - consider a structure with four elements: x, y, z, v
- Array of structure:



- Structure of arrays:



- Hybrid structure:



- Rearranging data in memory for vector operation is called *data swizzling*

SSE instructions

- Assembly instructions
  - Data movement instructions
    - moving data in and out of vector registers
  - Arithmetic instructions
    - Arithmetic operation on multiple data (2 doubles, 4 floats, 16 bytes, etc)
  - Logical instructions
    - Logical operation on multiple data
  - Comparison instructions
    - Comparing multiple data
  - Shuffle instructions
    - move data around SIMD registers
  - Miscellaneous
    - Data conversion: between x86 and SIMD registers
    - Cache control: vector may pollute the caches
    - State management:

SSE intrinsics

- **Data alignment issue**
  - **Some intrinsics may require memory to be aligned to 16 bytes.**
    - **May not work when memory is not aligned.**
- **Writing more generic SSE routine**
  - **Use posix malloc to ensure alignment to 16 B**

- **Compile with** `gcc –march=native`

Summary

- **Loop carried dependencies, when self or backward, prevent vectorization**
- **Loop carried dependencies prevent** doall **to be exploined**
- **Contemporary CPUs have SIMD support for vector operations**
  - **SSE is its programming interface**
  - **SSE can be accessed at high level languages through intrinsic functions.**
  - **SSE Programming needs to be very careful about memory alignments**
    - **Both for correctness and for performance.**