Assembly Language and Microcomputer Interface

Vectorization

Ming Cai

cm@zju.edu.cn

College of Computer Science and Technology
Zhejiang University

Performance Optimization in Software Development

- Efficient data structure and algorithm
- Numerical method: Precision control
- Optimization technique
  - Parallelism: Multi-threading, Vectorization
  - Memory access: Cache Locality, Data Layout
  - Communication: Offload

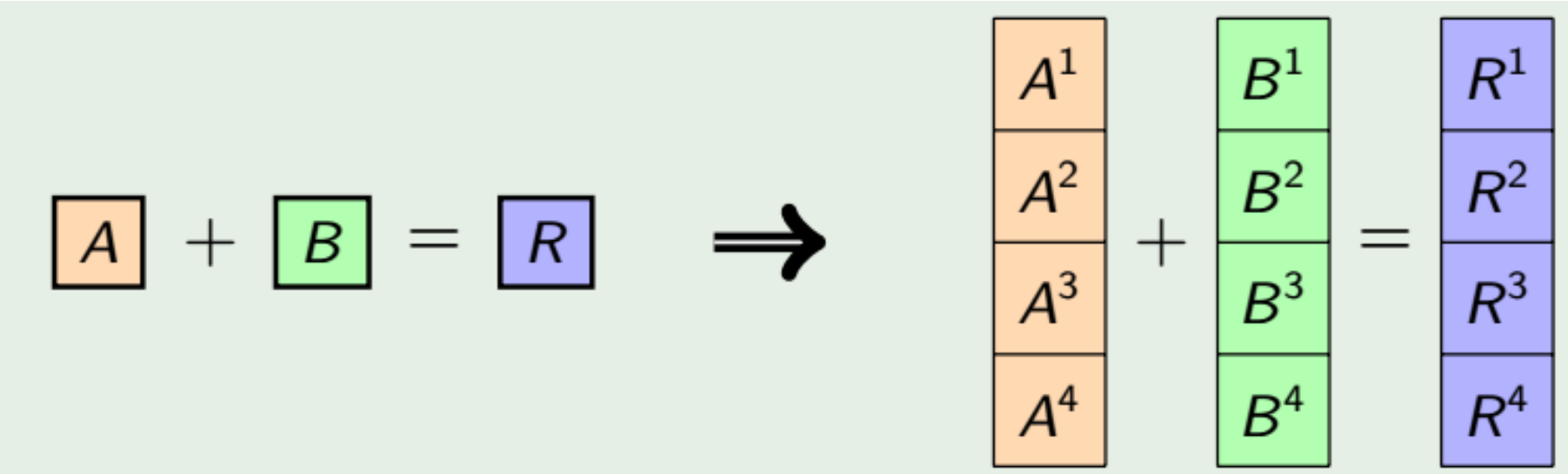
Outline

- Basic Concept
- Vectorization via SIMD
- Auto-vectorization
- Introduction to Intel Intrinsics
- Examples with Intrinsics
- Overview of ARM SVE
- Data Dependence Analysis
- Data Dependence Testing
- Transformations for Vectorization

1. Vector Instruction

- Vector instruction are an essential functionality of modern processors. These instructions are one of the forms of SIMD (Single Instruction Multiple Data) parallelism.
- Vector instructions enable faster computing in cases where a single stream of instructions inside a process or thread may be applied to multiple data elements.

What is Vectorization?

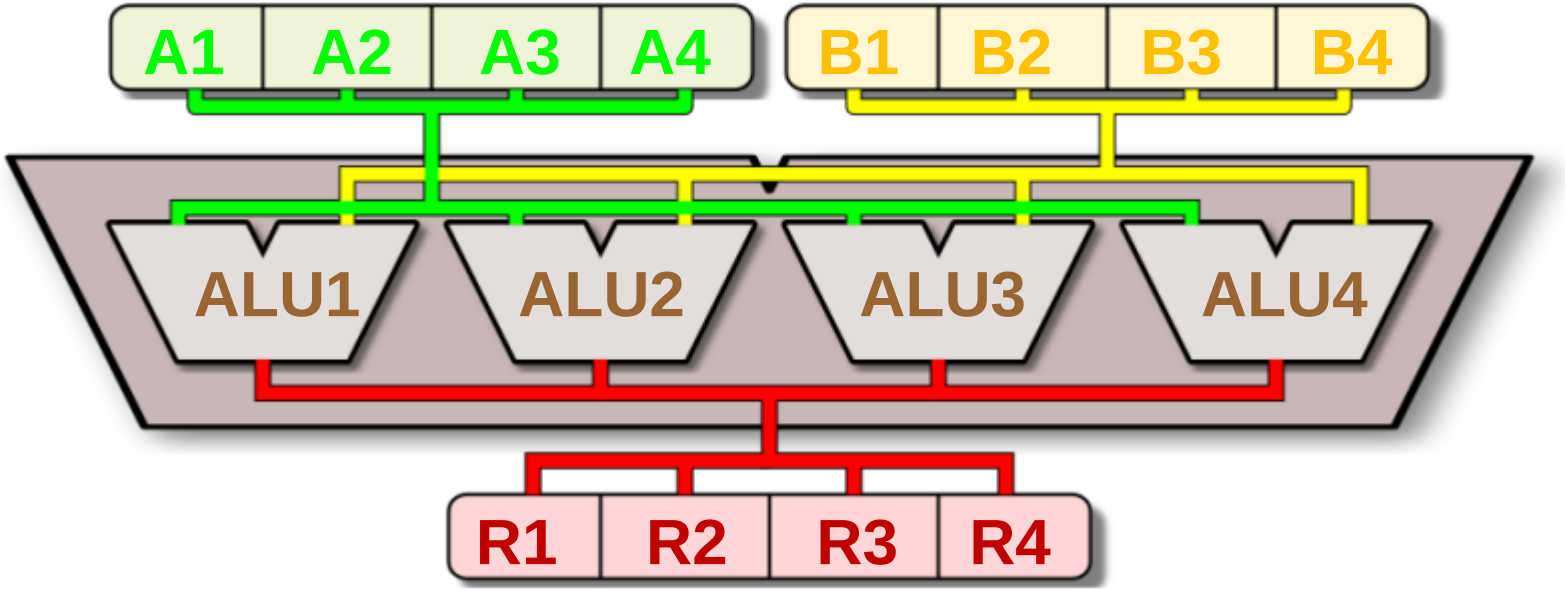


- Goal: parallelize computations over vectors to boost HPC and AI applications at low-level
- SIMD: Single Instruction Multiple Data (e.g., x86 SSE/AVX/AVX2, ARM NEON/SVE)

Architecture Support for Vectorization

- Architectures provide vector units to compute multiple elements at once.

R[1:4] = A[1:4] + B[1:4]



A Simple Example

- Think of vectorization in terms of loop unrolling

non-vectorized code

```
for (i=0; i<N;i++) {
        a[i]=b[i]+c[i];
}
```

```
for (i=0; i<N;i+=4) {
        a[i+0]=b[i+0]+c[i+0];
        a[i+1]=b[i+1]+c[i+1];
        a[i+2]=b[i+2]+c[i+2];
        a[i+3]=b[i+3]+c[i+3];
}
```

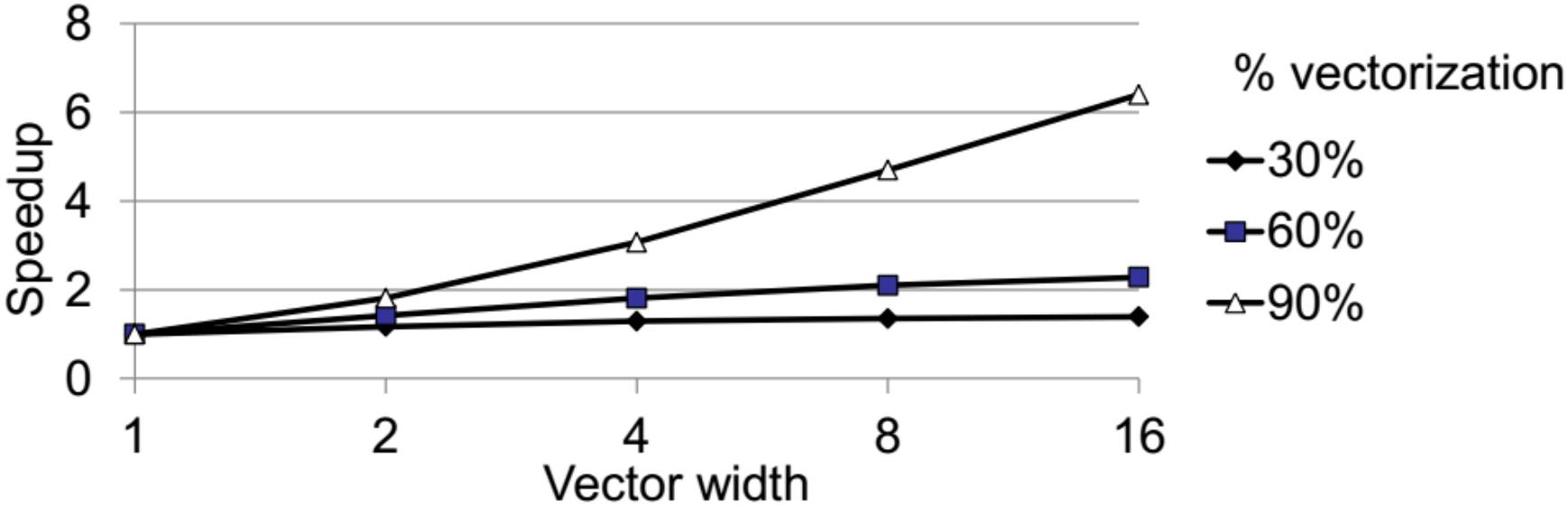
loop unrolling

vectorized code

```
Load b(i..i+3)
Load c(i..i+3)
Operate b+c->a
Store a
```

Promises of Vectorization

- The computation speedup (compared to no vector) is proportional to vector width.
- Serial portions of code and memory bandwidth are limiting factors.
  - Theoretical speedup is only a ceiling.



Vectorization versus Parallelism

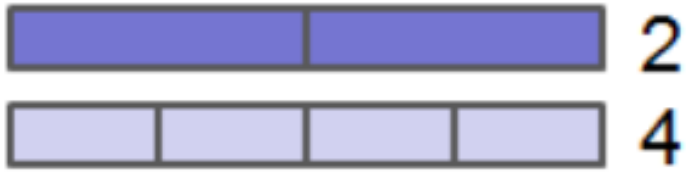
- Vectorization is a special case of instruction level parallelization.
- Parallel computing refers to four types of parallelization (bit-level, instruction-level, data-level, task-level).

- Vectorization is a single thread technique for fine grained parallelism.
- Parallelization supports both a single thread and a multi-thread technique for coarse grained parallelism.

Lane and Slot

- Multiple identical execution units in a vector register are called "lanes" or "slots".

- **For example,**
  - A 128-bit XMM register has four dword-sized lanes (lane 0-3), or two qword-sized lanes (lane 0-1).
  - ARM's SVE works on individual lanes under control of a predicate register.



128-bit XMM (x86)

	1	2	3	4
+	5	5	5	5
pred	1	0	1	0
=	6	2	8	4

SVE (ARM)

Vectorization Factor

- The vectorization factor (VF) denotes the maximum number of elements that fit into one vector, e.g., VF = 8 for single precision floating point numbers and a vector width of 256 bit.

- Specifically, the VF determines how many instructions pack together from different iterations of the loop.

Vectorization Factor

Vectorization Factor

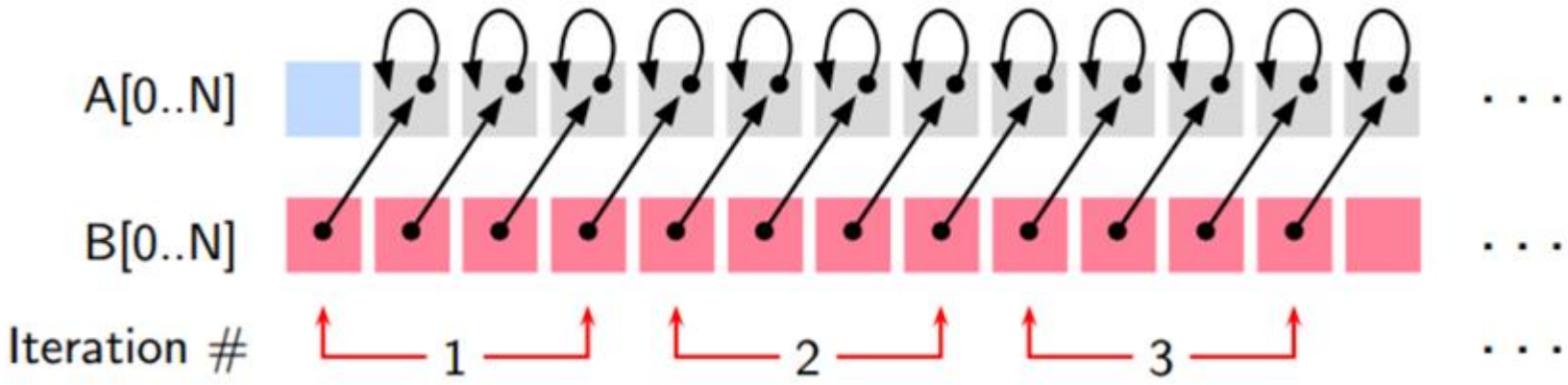
Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
  A[i:i+3] = A[i:i+3] + B[i-1:i+2];
```

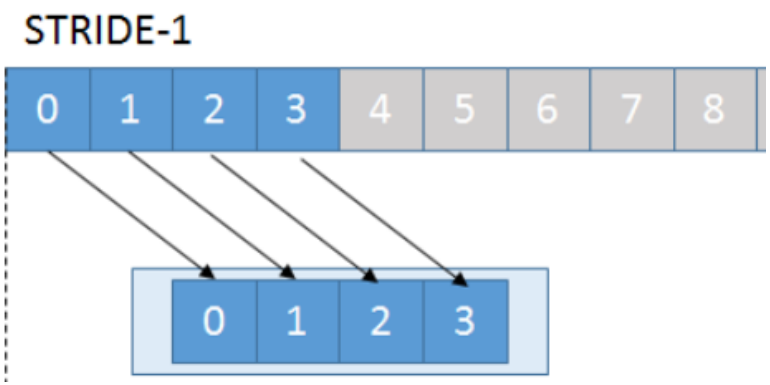
movdqa xmm0, xmmword ptr [rax]



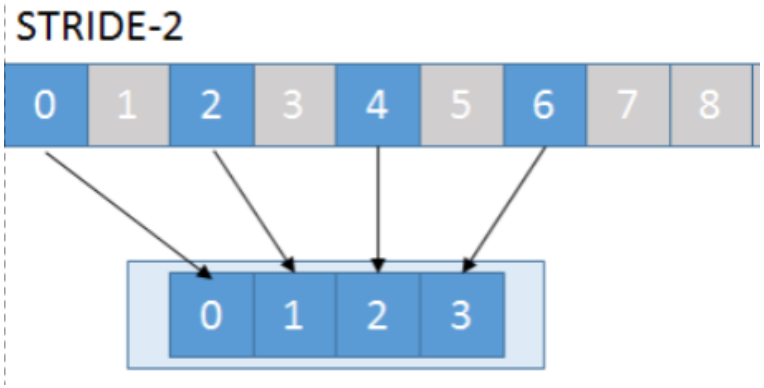
128-bit operand

Stride

- The distance between memory locations that separates the elements to be gathered into a single register is called the stride. A stride of one unit is called a unit-stride. This is equivalent to sequential memory access.
- A vector processor can handle strides greater than one, called non-unit strides.



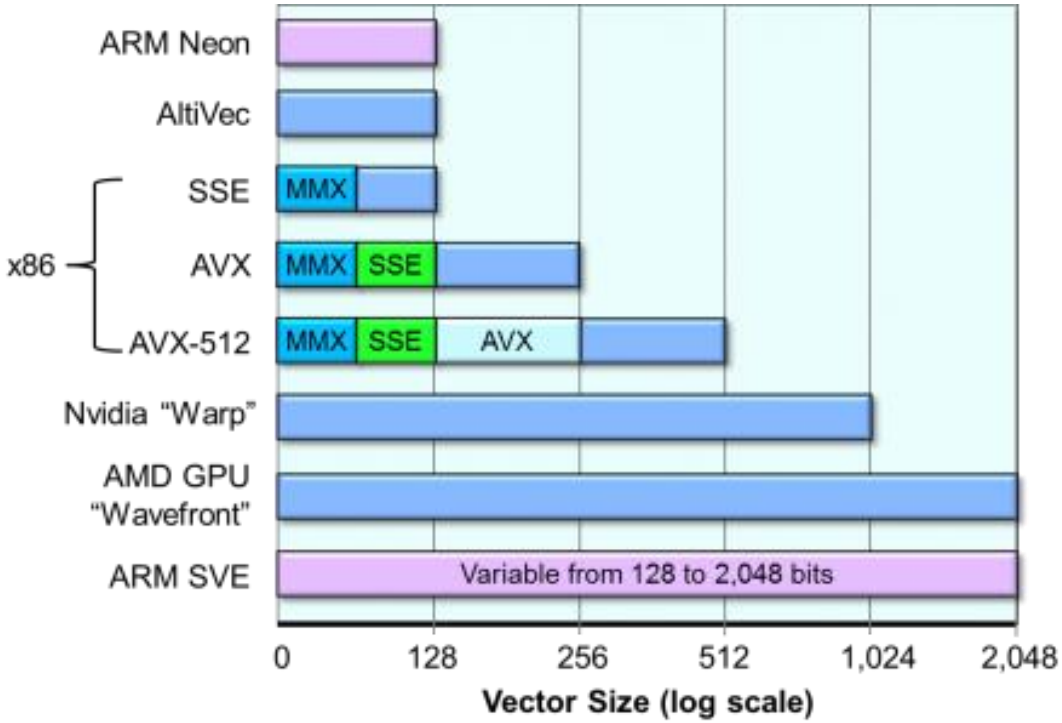
unit-stride



non-unit stride

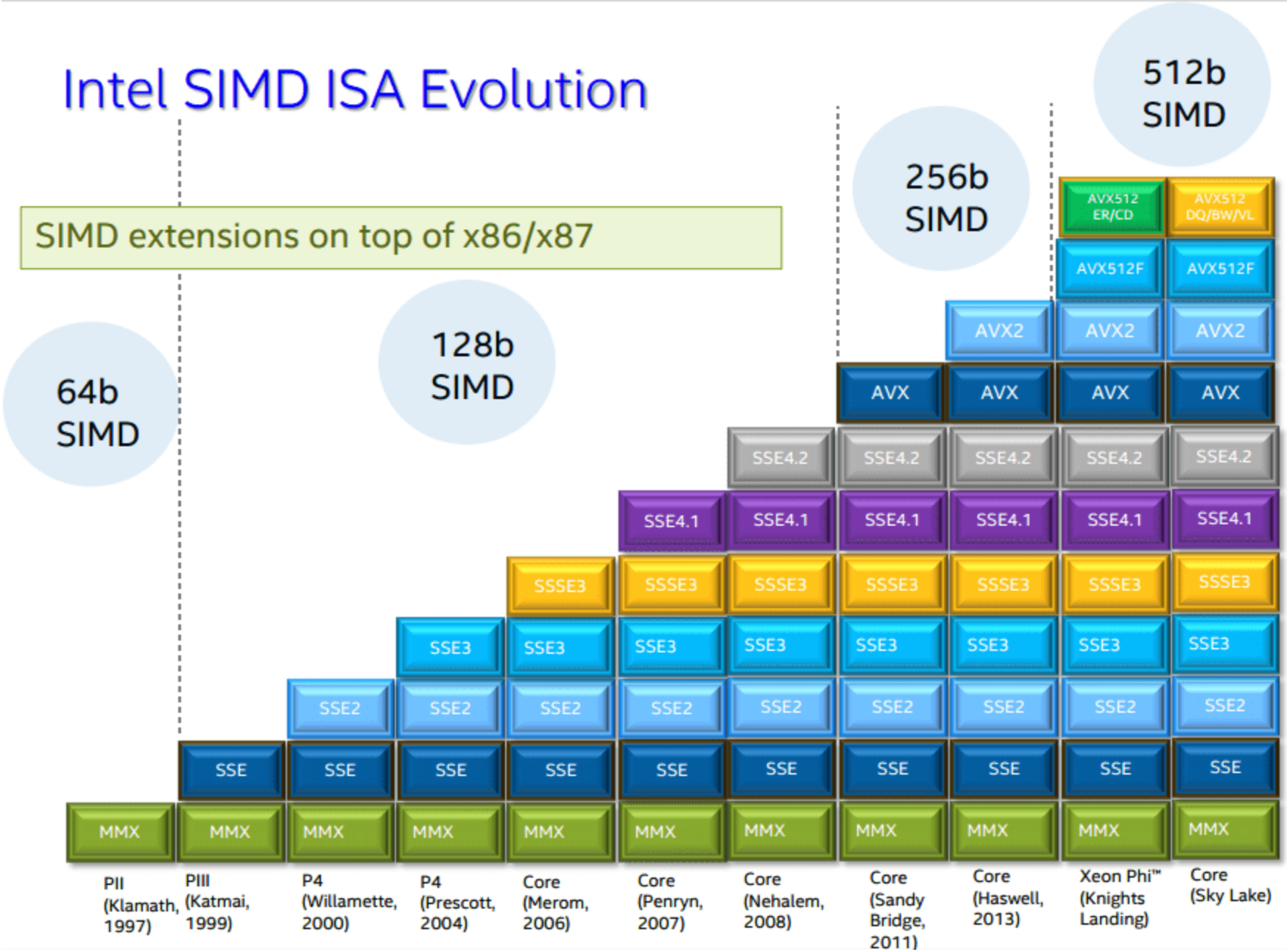
2. Overview of Vector Instructions

- x86: MMX, SSE, AVX, AVX2, AVX-512
  - 8 64-bit registers (MMX) to 32 512-bit registers (AVX-512)
- ARM: NEON, SVE, SVE2
  - 32 128-bit registers (NEON) to 32 128-2048-bit in SVE

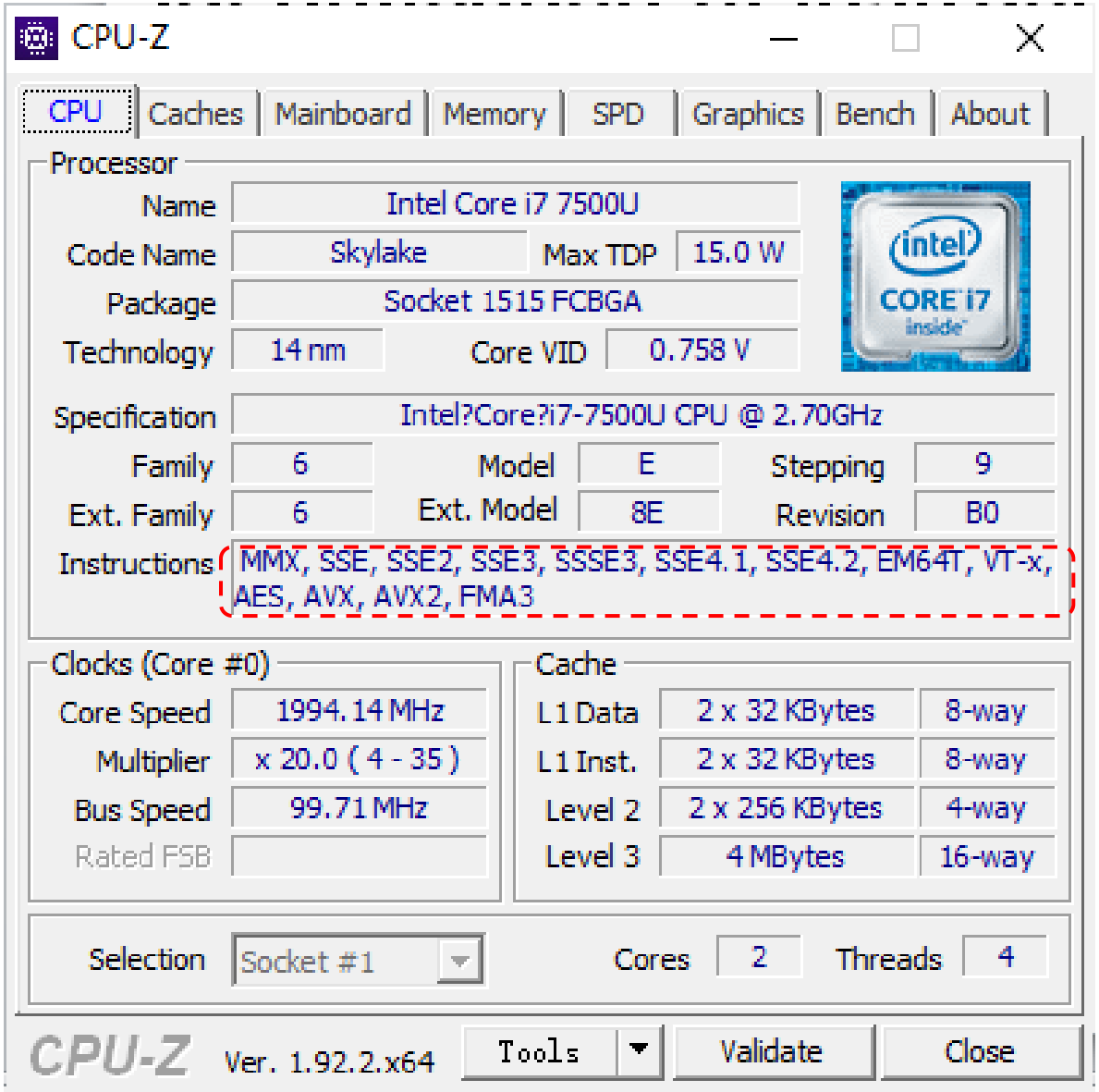


Intel SIMD ISA Evolution

SIMD extensions on top of x86/x87



Identification of Instruction Set



How to access the SIMD units?

- **Four choices**
  - C/C++ code and a vectorizing compiler

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

  - SIMD intrinsics

```
__m128i _mm_add_epi16 (__m128i a, __m128i b)
__m128i _mm_add_epi32 (__m128i a, __m128i b)
__m128i _mm_add_epi64 (__m128i a, __m128i b)
__m128i _mm_add_epi8 (__m128i a, __m128i b)
__m128d _mm_add_pd (__m128d a, __m128d b)
__m256d _mm256_add_pd (__m256d a, __m256d b)
__m64 _mm_add_pi16 (__m64 a, __m64 b)
__m64 _mm_add_pi32 (__m64 a, __m64 b)
```

  - vector library (e.g., xSIMD, VCL)

xsimd

C++ wrappers for SIMD intrinsics

  - assembly language

```
movaps    a(,%rdx,4), %xmm0
addps     b(,%rdx,4), %xmm0
movaps    %xmm0, c(,%rdx,4)
addq      $4, %rdx
```

3. Auto-vectorization

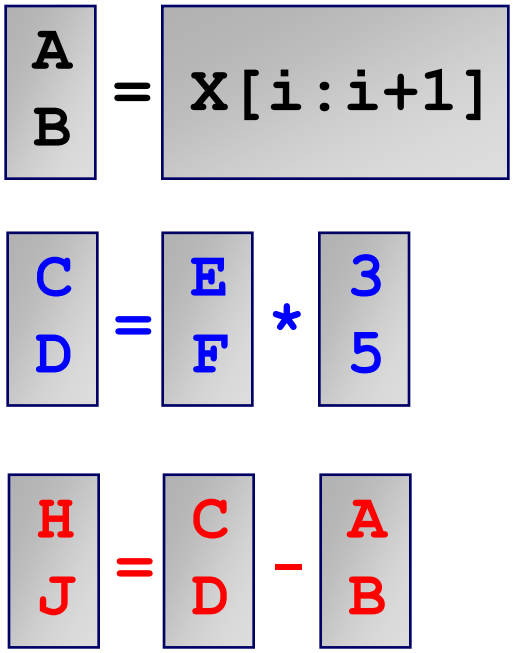
- **Two types of Auto-vectorizations**
  - SLP-based vectorization (SLP)
  - Loop-based vectorization (LV)

  Note: SLP stands for Superword-Level Parallelism or Straight-Line code Parallelism

- **Major compilers (e.g. GCC and LLVM) implement both algorithms and the two algorithms are complementary.**
- **A common configuration is to run the SLP pass after the loop-vectorization pass.**

- **SLP-based vectorizations scan the code for repeated sequences of isomorphic scalar instructions, aiming at replacing each one of them for their vector counterpart.**

```
A = X[i+0]
B = X[i+1]
C = E * 3
D = F * 5
H = C - A
J = D - B
```



Note: Latest studies focus on converting non isomorphic instruction sequences into isomorphic ones through transformation.

- **Loop-based vectorizations operate on loops and perform widening of each instruction in the loop.**

```
for (i=0; i<100; i+=1)
   A[i+0] = A[i+0] + B[i+0]
```

loop unrolling

```
for (i=0; i<100; i+=4)
   A[i+0] = A[i+0] + B[i+0]
   A[i+1] = A[i+1] + B[i+1]
   A[i+2] = A[i+2] + B[i+2]
   A[i+3] = A[i+3] + B[i+3]
```

```
for (i=0; i<100; i+=4)
   A[i:i+3] = B[i:i+3] + C[i:i+3]
```

SLP Compared to Loop Vectorization

```
for (i=0; i<N; i+=4)
    A[i]   = B[i]
    A[i+1] = B[i+1]
    A[i+2] = B[i+2]
    A[i+3] = B[i+3]
```

SLP

SLP-based vectorization with VF =4

```
for (i=0; i<N; i+=4)
    A[i:i+3] = B[i:i+3]
```

LV

Loop-based vectorization with VF =4

```
for (i=0; i<N; i+=16)
  A[i,  i+4,i+8, i+12] = B[i,  i+4,i+8, i+12]
  A[i+1,i+5,i+9, i+13] = B[i+1,i+5,i+9, i+13]
  A[i+2,i+6,i+10,i+14] = B[i+2,i+6,i+10,i+14]
  A[i+3,i+7,i+11,i+15] = B[i+3,i+7,i+11,i+15]
```

- **Main issues with Auto-vectorization**
  - Aliasing, alignment, data dependences, branching, ...
  - In general lack of knowledge of the compiler
- **Ways to solve them**
  - Compiler directives, ternary operator
  - Proper data structures (e.g., Structure of Arrays)
- **Still worth trying Auto-vectorization**
  - It’s (almost) a free lunch!
  - 100% portable code
  - No dependences

Compiler Flags

- **Optimize options**
  - For GCC, clang : -O3 or -O2 -ftree-vectorize
  - For ICC : -O2 or -O3. Use -no-vec to disable it
- **Specific architecture**
  - For avx2: -mavx2 on GCC/clang, -axAVX2 -xAVX2 on ICC
  - For avx512 on GCC/clang : -march=skylake-avx512
  - For avx512 on ICC: -xCORE-AVX512
  - For optimal vectorization depending on CPU: -march=native on GCC/clang, -xHOST on icc

Visualizing the Auto-vectorization

COMPILER EXPLORER Add... More Sponsors intel PC-lint Share Policies

intro-sum.c x86-64 gcc 11.2 (C, Editor #1, Compiler #1)

Save/Load Add new... Vim C x86-64 gcc 11.2 -O3 Output... Filter... Libraries Add new... Add tool...

```c
int Sum1ToN(int n){
 int sum = 0;
 for (int i = 0; i < n; i++){
   sum +=i;
}
 return sum;
}

int Sum(int n){
 int sum = 0,a[n],b[n];

 for (int i = 0; i < n; i++){
   sum += a[i];
   sum += b[i];
}
 return sum;
}
```

```asm
        shr     edi, 2
        sal     rdi, 4
.L15:
        movdqu  xmm2, XMMWORD PTR [rsi+rax]
        movdqu  xmm3, XMMWORD PTR [rcx+rax]
        add     rax, 16
        paddd   xmm0, xmm2
        paddd   xmm0, xmm3
        cmp     rax, rdi
        jne     .L15
        movdqa  xmm1, xmm0
        mov     edi, edx
        psrldq  xmm1, 8
        and     edi, -4
        paddd   xmm0, xmm1
        movdqa  xmm1, xmm0
        psrldq  xmm1, 4
        paddd   xmm0, xmm1
        movd    eax, xmm0
        test    dl, 3
        je      .L12
.L14:
        movsx   r8, edi
        sal     r8, 2
        add     rsi, r8
        add     rcx, r8
```

https://www.godbolt.org

Auto-vectorization Requirements (1/2)

- The loop count must be known at entry to the loop at runtime.
- Branching in the loop inhibits vectorization.
- Data Dependences in the loop could prevent vectorization.
- Non-unit stride access hampers vectorization efficiency.

Auto-vectorization Requirements (2/2)

- Only the innermost loop is eligible for vectorization.
- A function call or I/O inside a loop prohibits vectorization.
  - Intrinsic math functions (e.g., cos, sin, etc.) are allowed because such they are usually vectorized.
  - An inline function can be vectorized because there will be no more function call.

When Auto-vectorization Fails

- When Auto-vectorization fails, the programmer may need to do:
  - Add compiler directives
  - Transform the code
  - Program using vector intrinsics

Compiler Directives (1/3)

- Compiler vectorizes many loops, but many more can be vectorized if the appropriate directives are used to inform the compiler.

Directives for ICC	Semantics
#pragma ivdep	ignore assumed data dependences
#pragma vector always	ignore efficiency heuristics
#pragma SIMD	give notice to enforce vectorization
#pragma novector	disable vectorization
__restrict__	assert exclusive access through pointer
__attribute__ aligned(n)	request memory alignment
memalign(boundary,size)	malloc aligned memory
__assume_aligned(ptr, n)	assert memory alignment

Compiler Directives (2/3)

- When the compiler does not vectorize automatically due to dependences, programmers can inform the compiler that it is safe to vectorize. For example,

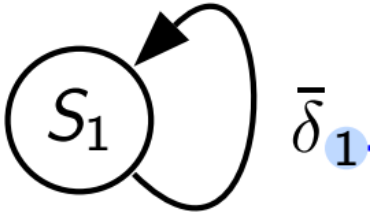
```
for (int i=0; i<LEN; i++)
S1:    a[i]=a[i+k]+b[i];
```

- This loop can be vectorized when k < -3 and k ≥ 0.

k =1

a[0]=a[1]+b[0]
a[1]=a[2]+b[1]
a[2]=a[3]+b[2]

Can be vectorized

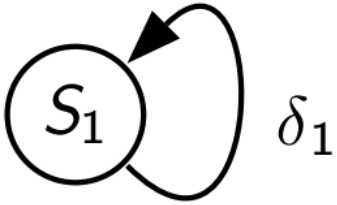


Write After Read

k =-1

a[1]=a[0]+b[1]
a[2]=a[1]+b[2]
a[3]=a[2]+b[3]

Cannot be vectorized



Read After Write

Compiler Directives (3/3)

- This loop can be vectorized when k < -3 and k ≥ 0.
- Programmer knows that k ≥ 0.
- How can the programmer tell the compiler that k ≥ 0 ?
- Intel ICC provides the #pragma ivdep to tell the compiler that it is safe to ignore unknown dependences.

```
#pragma ivdep

for (int i=0; i<LEN-k; i++)
S1:     a[i]=a[i+k]+b[i];
```

However, if the loop is vectorized when -3 < k < 0, wrong results will be obtained.

Pointer Aliasing (1/5)

- Two seemingly different pointers may point to storage locations in the same array (aliasing).
- Data dependences can arise when performing loop-based computations using pointers, as the pointers may potentially point to overlapping regions in memory.
- For vectorization to take place, the compiler needs to prove that no data dependences (overlaps) are possible. This is difficult to do when using pointers.

Pointer Aliasing (2/5)

- Consider the following example. Is it safe to vectorize the for loop?

```
void compute(double *a, double *b, double *c)
{
        for (i=1; i<N; i++) {
            a[i] = b[i] + c[i];
        }
}
```

Pointer Aliasing (3/5)

- If we invoked it as follows compute(p, p-1, q):

```
void compute(double *a, double *b, double *c)
{
        for (i=1; i<N; i++) {
            p[i] = p[i-1] + q[i];
        }
}
```

Read After Write dependence

Cannot be vectorized

- Given VF = 2, how about compute(p, p-2, q)?

p[2] = p[0] + q[2]

p[3] = p[1] + q[3]

VF = 2

dependence distances 2 ≥ VF, no dependence!

p[4] = p[2] + q[4]

Can be vectorized

Pointer Aliasing (4/5)

```
void compute(double *a, double *b, double *c)  {
        for (i=1; i<N; i++)
                a[i] = b[i] + c[i];
}
```

- b=a-1 or c=a-1 ✗
- b=a-2 or c=a-2 ✓

VF = 2

rax = c	mov rax, rdx
rdx = b+1	lea rdx, [rsi+8]
rcx = a	mov rcx, rdi
a – (b+1)	cmp rdi, rdx
a == (b+1)	je .L2
rdx = c+1	lea rdx, [rax+8]
a – (c+1)	cmp rdi, rdx
a == (c+1)	je .L2

b=a-1

c=a-1

```
.L2:
        movsd   xmm0, QWORD PTR [rsi]
        addsd   xmm0, QWORD PTR [rax]
        movsd   QWORD PTR [rcx], xmm0
        movsd   xmm0, QWORD PTR [rsi+8]
        addsd   xmm0, QWORD PTR [rax+8]
        movsd   QWORD PTR [rcx+8], xmm0
```

serial execution

```
movupd   xmm1, XMMWORD PTR [rax]
movupd   xmm0, XMMWORD PTR [rsi]
addpd    xmm0, xmm1
```

vectorized execution

Note: x86-64 Linux, the first six function arguments are passed in registers RDI , RSI , RDX , RCX , R8 and R9.

Pointer Aliasing (5/5)

- We can add directives to guide the compiler.
- C99 introduced "restrict" keyword to inform the compiler that no other pointer will access the same memory addresses (no aliasing).

```
void compute(double * restrict a, double * restrict b,
                    double * restrict c)    {
        for (i=1; i<N; i++)
            a[i] = b[i] + c[i];
}
```

↓

```
compute:
        movupd  xmm0, XMMWORD PTR [rsi]
        movupd  xmm1, XMMWORD PTR [rdx]
        movupd  xmm2, XMMWORD PTR [rdx+16]
        movupd  xmm3, XMMWORD PTR [rdx+32]
        addpd   xmm0, xmm1
```

Data Alignment (1/2)

- Vector loads/stores load/store 128 or 256 consecutive bits to a vector register.
- Data addresses need to be 16-byte or 32-byte aligned to be loaded/stored.
- In many cases, the compiler cannot statically know the alignment of the address in a pointer.
- The compiler assumes that the base address is 16-byte aligned and adds a run-time checks.
  - if the runtime check is false, then it uses another code (which may be scalar)

Data Alignment (2/2)

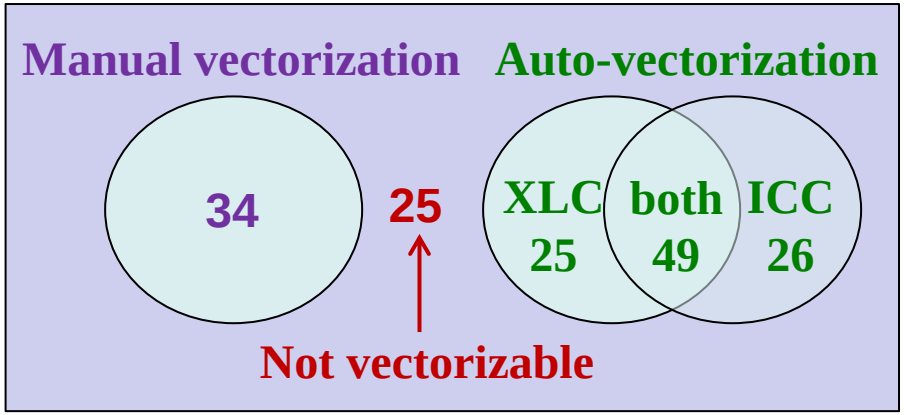
- Manual 16-byte or 32-byte alignment can be achieved by forcing the base address to be a multiple of 16 or 32, e.g.,
  - `__attribute__ ((aligned(16))) float b[N];`
  - `float* a = (float*) memalign(16,N*sizeof(float));`
- When the pointer is passed to a function, the compiler should be aware of where the aligned address of the array starts.

```
void func1(float *a, float *b, float *c) {
        __assume_aligned(a, 16);
        __assume_aligned(b, 16);
        __assume_aligned(c, 16);
        for int (i=0; i<LEN; i++)
            a[i] = b[i] + c[i];
}
```

How well do compilers vectorize ?

- **Compilers**: IBM AIX XLC, Intel ICC, GNU GCC
- **Total loops:** 159

	XLC	ICC	GCC
# of total loops	159		
# of vectorized	74	75	32
# of not vectorized	85	84	127
average speedup	1.73	1.85	1.30



- By adding manual vectorization, the average speedup was 3.78 (versus 1.73 obtained by XLC)

Auto-vectorization is still not enough

- **It is not fully mature**
  - Still very touchy despite improvements
  - Only able to vectorize loops (or almost)
  - Hardly able to handle branching via masks
  - No abstract knowledge of the application
- **It will probably never be good enough**
  - As it cannot know as much as the developer
  - Especially concerning input data such as
    - average number of tracks reconstructed
    - average energy in that data sample
  - Efficient vectorizable code requires an efficient data layout; this must be done manually.

So we need to vectorize by hand from time to time

Prerequisites for Vectorization

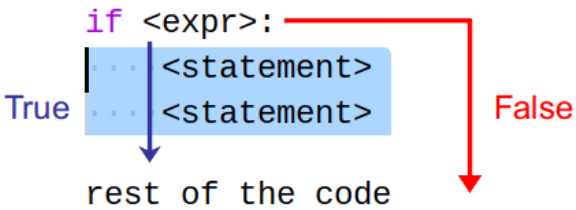
- Three constraints for vectorization
  - sequences of isomorphic scalar instructions (Feasibility)
  - no data dependences within the program to prevent vectorization (Validity/Correctness)
  - memory layout or alignment constraint (e.g., contiguous memory accesses) (Efficiency)

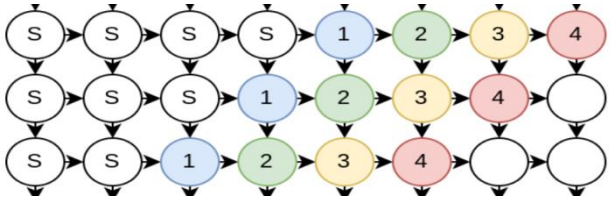
```
float a[n], b[n], c[n];

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

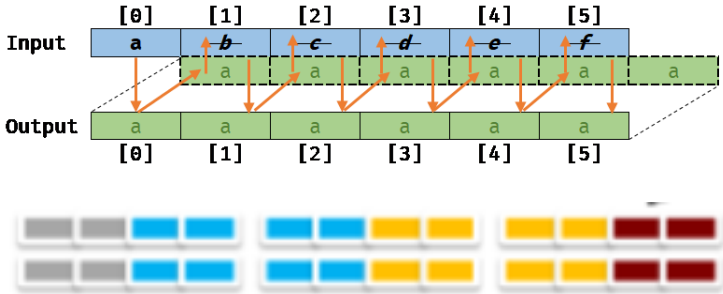
```
__m128   va, vb, vc;
for (i = 0; i < n; i+=4) {
        va = _mm_load_ps(&a[i]);
        vb = _mm_load_ps(&b[i]);
        vc = _mm_mul_ps(va,vb);
        _mm_store_ps(&c[i], vc);
}
```

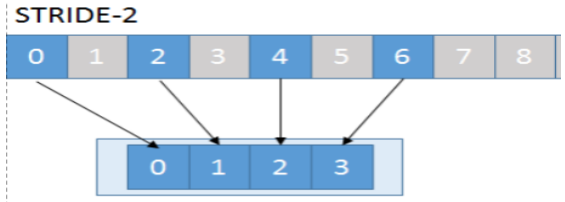
Challenge for Vectorization

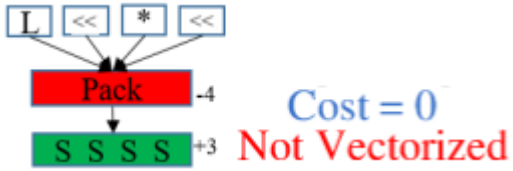
- Conditional Statements
- Data Dependences
- Pointer Aliasing
- Data Alignment
- Non-unit Strides
- Profitability Analysis











4. An Overview of Intel Intrinsics

- Principles
  a) Intrinsics are functions that the compiler replaces with the proper assembly instructions.
  b) It hides nasty assembly code but maps 1 to 1 to SIMD assembly instructions.
  c) Using intrinsics means that we are essentially writing assembler code directly in the program.
- Pros
  – Easy to use
  – Full power of SIMD can be achieved
- Cons
  – Very verbose, very low level
  – Processor specific

Intel Intrinsics Data Types

Name	16 bytes	32 bytes	64 bytes
Integers	__m128i	__m256i	__m512i
16-bit float	__m128h	__m256h	__m512h
32-bit float	__m128	__m256	__m512
64-bit float	__m128d	__m256d	__m512d

- Data Types Usage Guidelines
  a) Use data types only with the respective intrinsics.
  b) Use data types only on either side of an assignment, as a return value, or as a parameter. You cannot use it with other arithmetic expressions (e.g., +, -, etc).
  c) Use data types as objects in aggregates, such as unions, to access the byte elements and structures.

Intel Intrinsics Naming Conventions

- Naming convention:

mm<S><mask><op>_<suffix> (data_type param1, ...)

where

a) <S>: empty for SSE, 256 for AVX2 and 512 for AVX512
b) <mask>: empty or mask or maskz (AVX512 only)
c) <op>: the operator (e.g., add, mul, ...)
d) <data_type>: describes the data in the vector

- Example:

a) _mm256_mul_ps: Multiply packed single-precision
b) _mm512_maskz_add_pd: Add packed double-precision using zeromask

Intel Intrinsics Guide: https://www.intel.com/ content/www/us/ en/docs/ intrinsics-guide/index.html

Subgroups of Intrinsics (1/2)

- Initialization instructions
  - e.g., _mm_setzero_ps, _mm256_set_epi8
- Data movement instructions
  - e.g., _mm256_load_pd, _mm_maskstore_pd
- Arithmetic instructions
  - e.g., _mm256_add_ps, _mm256_mul_pd
- Fused multiply and add (FMA) instructions
  - e.g., _mm_fmadd_ps, _mm_fnmsub_ps
- Logical and shift instructions
  - e.g., _mm_or_pd, _mm_andnot_pd, _mm_srli_epi16

Subgroups of Intrinsics (2/2)

- Comparison instructions
  - e.g., _mm_cmp_pd, _mm_comigt_ss
- Permute instructions
  - e.g., _mm256_permute_pd, _mm256_permute4x64_pd
- Pack and unpack instructions
  - e.g., _mm_unpackhi_ps, _mm_unpacklo_epi32
- Shuffle instructions
  - e.g., _mm_shuffle_ps, _mm_blend_ps
- Conversion instructions
  - e.g., _mm_cvtepi32_ps, _mm256_cvtpd_epi32

Steps for Programming with Intrinsics

1. Chose the instruction set (e.g., SSE/AVX/...)
2. Declare the corresponding header

for SSE
#include <xmmintrin.h>

.....

for AVX
#include <immintrin.h>

3. Create streams and SIMD data structures
4. Call intrinsic functions

intrinsics: _mm_add_ps (__m128 a, __m128 b)

instructions: addps xmm, xmm

_mm256_sqrt_pd (__m256d a)

vsqrtpd ymm, ymm

5. Examples with Intrinsics

A. Vector Dot Product

B. Matrix Transpose

Vector Dot Product——Scalar Code

Vector Dot Product $\begin{bmatrix}2\\7\\1\end{bmatrix} \cdot \begin{bmatrix}8\\2\\8\end{bmatrix} = 2 \cdot 8 + 7 \cdot 2 + 1 \cdot 8 = 38$

```
float dotProduct (float* p1, float* p2, int count )
{
        float result = 0;
        float* const  p1End = p1 + count;
        for( ; p1 < p1End; p1++, p2++ )
                result += p1[ 0 ] * p2[ 0 ];
        return result;
}
```

https://github.com/Const-me/SimdIntroArticle/blob/master/DotProduct/scalar.cpp

Vector Dot Product——Vectorized Version

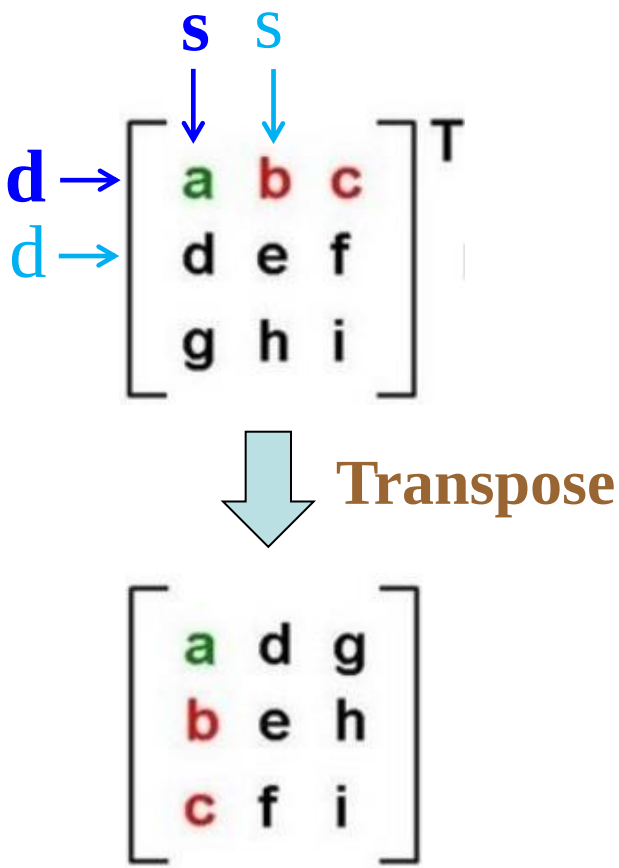
using SSE Intrinsics

```
float dotProduct( float* p1, float* p2, int count ) {
        assert( 0 == count % 4 );
        __m128 acc = _mm_setzero_ps();          //Clear dot product with  zero
        float* const p1End = p1 + count;
        for( ; p1 < p1End; p1 += 4, p2 += 4 )        {
            // Load 2 vectors, 4 floats / each
            const __m128 a = _mm_loadu_ps( p1 );     //Load 128-bits vector
            const __m128 b = _mm_loadu_ps( p2 );     //Load 128-bits vector
            // Compute dot product of them. The 0xFF constant means "use all 4
source lanes, and broadcast the result into all 4 lanes of the destination".
            const __m128 dp = _mm_dp_ps( a, b, 0xFF );
            acc = _mm_add_ps( acc, dp );        //Add the dot product
        }
        return _mm_cvtss_f32( acc );          //Convert __m128  into 32-bit floats
}
```

https://github.com/Const-me/SimdIntroArticle/blob/master/DotProduct/dpps.cpp

Matrix Transpose——Scalar Code

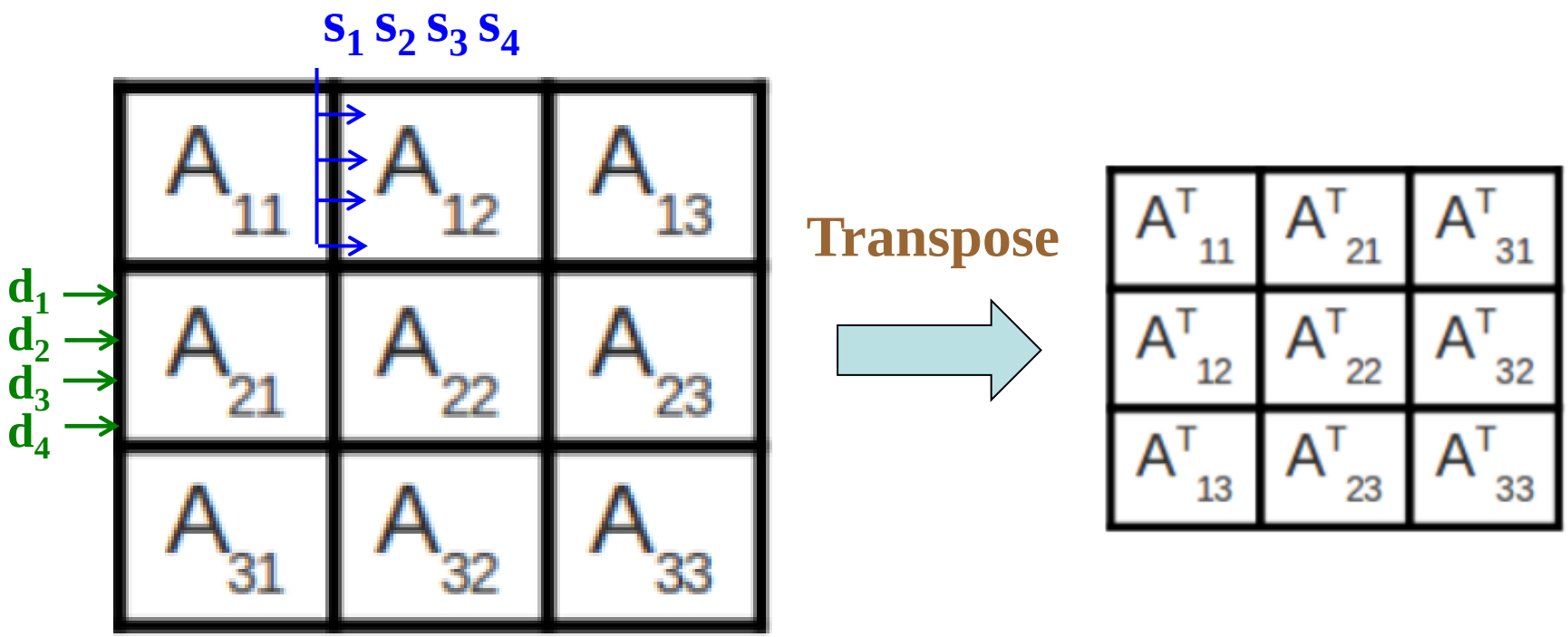
```
void transpose(double *x, double *y, int n)
{
    double *s=x,*d=y;
    for(int i=0;i<n;i++)      {
        d=y+i;
        for(int j=0;j<n;j++)  {
            *d=*(s++);
             d+=n;
        }
    }
}
```



https://www.cxyzjd.com/article/artorias123/90513600

Matrix Transpose——Vectorized Version

- We consider 2D (4 × 4) blocking to perform the transposition one submatrix at a time.



https://www.cxyzjd.com/article/artorias123/90513600

Matrix Transpose——Vectorized Version

a_0	a_1	a_2	a_3
b_0	b_1	b_2	b_3
c_0	c_1	c_2	c_3
d_0	d_1	d_2	d_3

Transpose

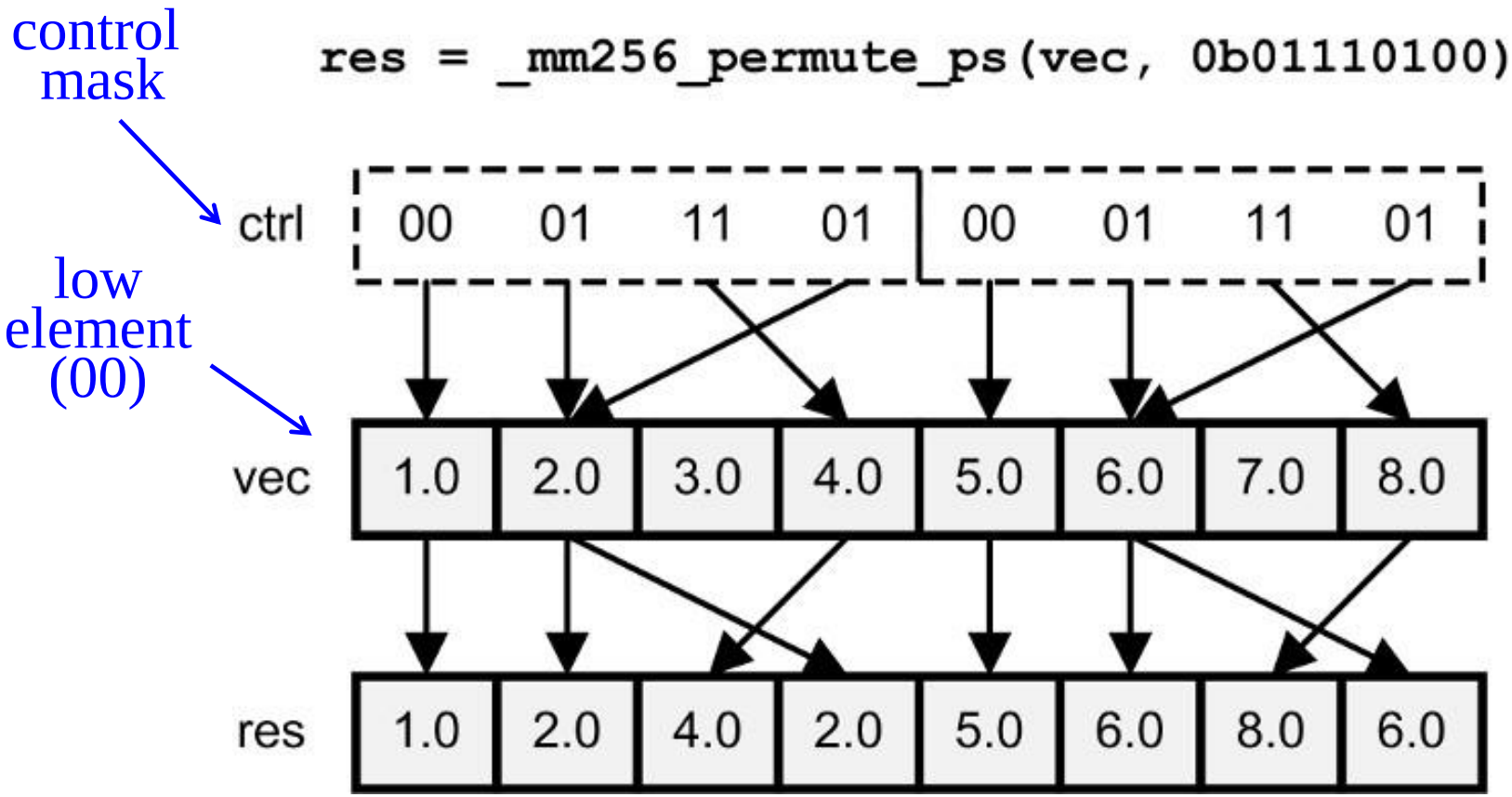
a_0	b_0	c_0	d_0
a_1	b_1	c_1	d_1
a_2	b_2	c_2	d_2
a_3	b_3	c_3	d_3

- Transpose of a 4×4 matrix (AVX)
  - Permute instruction: _mm256_permute4x64_pd
  - Shuffle instruction: _mm256_blend_pd
  - Unpack instructions: _mm256_unpacklo_pd, _mm256_unpackhi_pd

Matrix Transpose——Vectorized Version

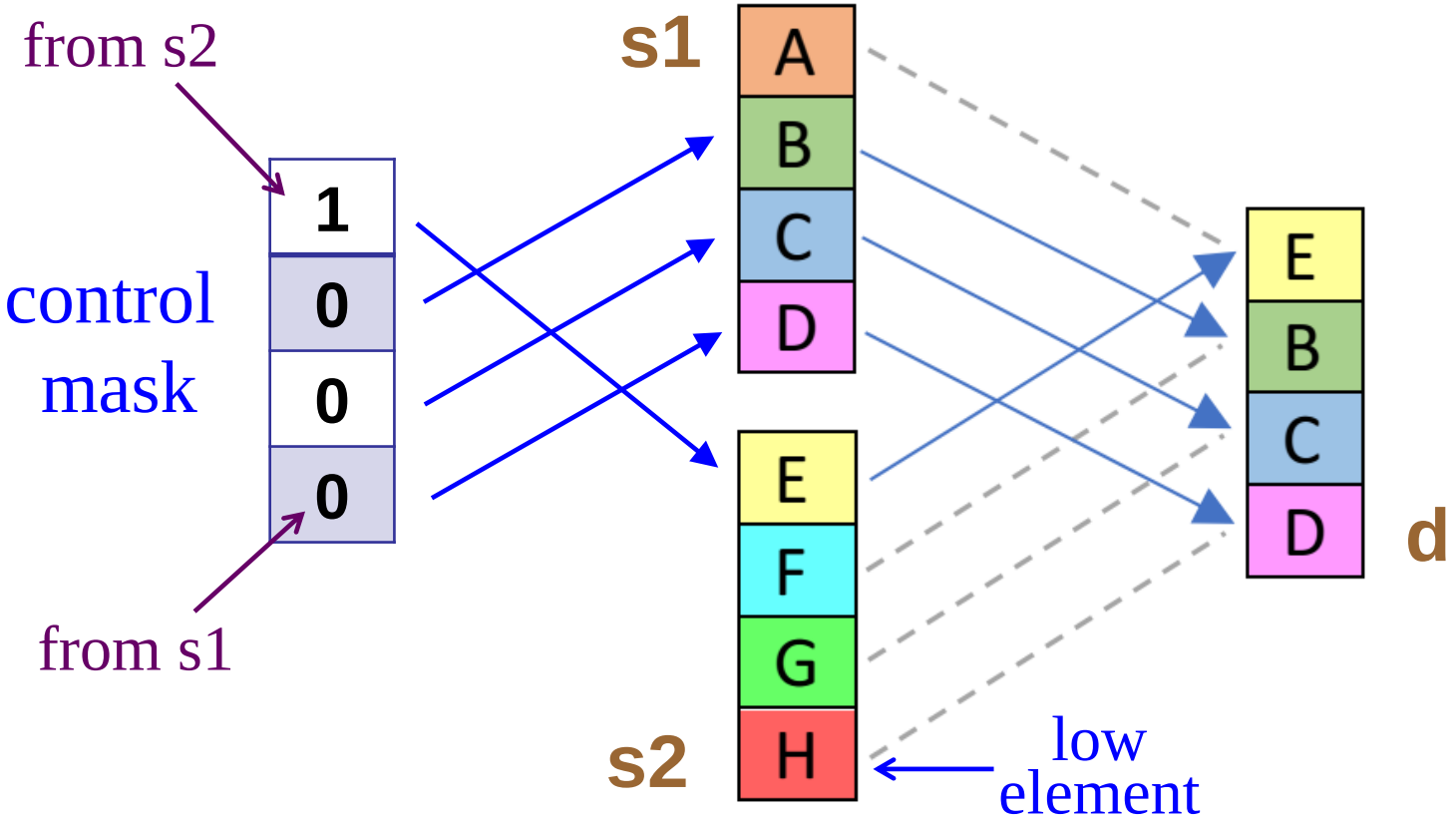
- Permute instructions provide intrinsics that return a vector containing the rearranged elements of a vector.

```
res = _mm256_permute_ps(vec, 0b01110100)
```



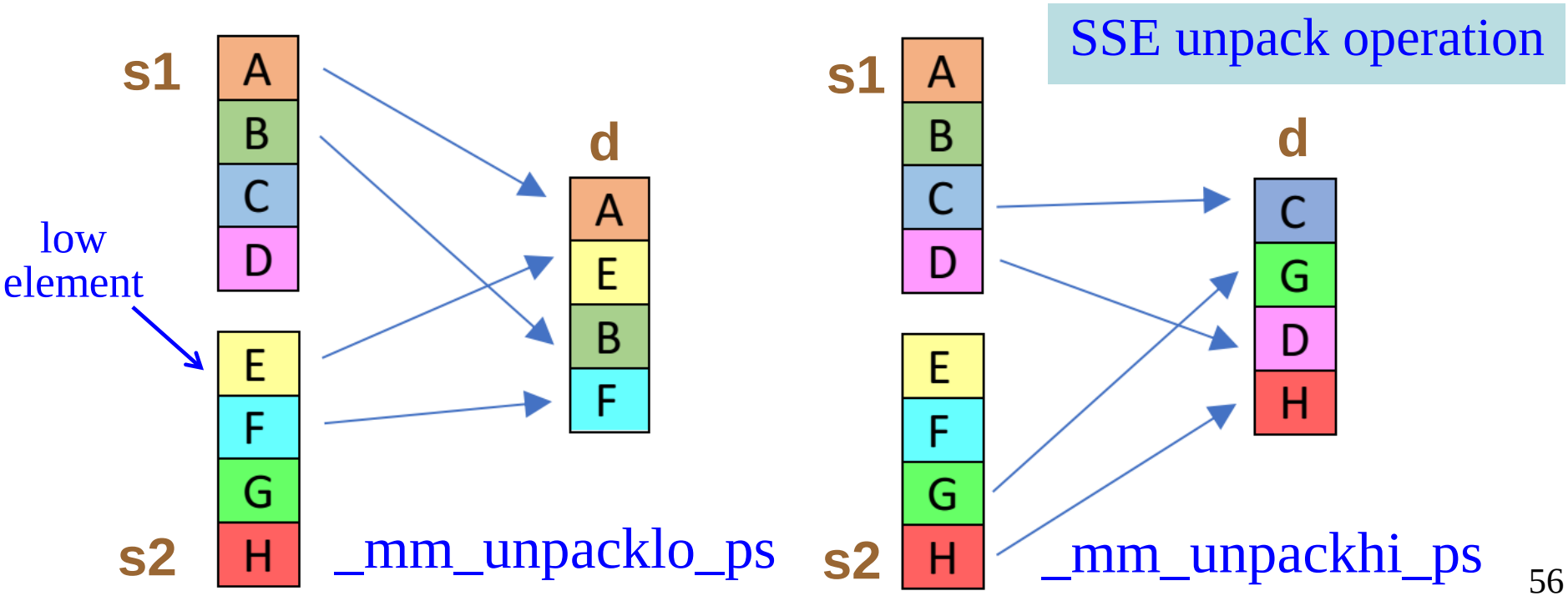
Matrix Transpose——Vectorized Version

- Shuffle intrinsics select elements from two input vectors and place them in the output vector.
- For example, d = _mm_blend_ps(s1,s2, 1000B)



Matrix Transpose——Vectorized Version

- Pack and unpack instructions support conversion between packed and unpacked data.
- For example, d =_mm_unpacklo_ps(s1,s2) unpacks and interleaves the low-order data elements of the source vectors (s1, s2) and stores the results in the destination vector d.



Matrix Transpose——Vectorized Version

- 256-bit operations in AVX are generally performed in two halves of 128-bit lanes. Most of the 256-bit AVX instructions are defined as in-lane:
  - the destination elements in each lane are calculated using source elements only from the same lane.
- E.g., dot product of packed 32-bit float

128-bit SSE Dot Product Operation

_m128 _mm_dp_ps (_m128 a, _m128 b, const int imm8)

- dst[127:0] := DP(a[127:0], b[127:0], imm8[7:0])

256-bit AVX Dot Product Operation

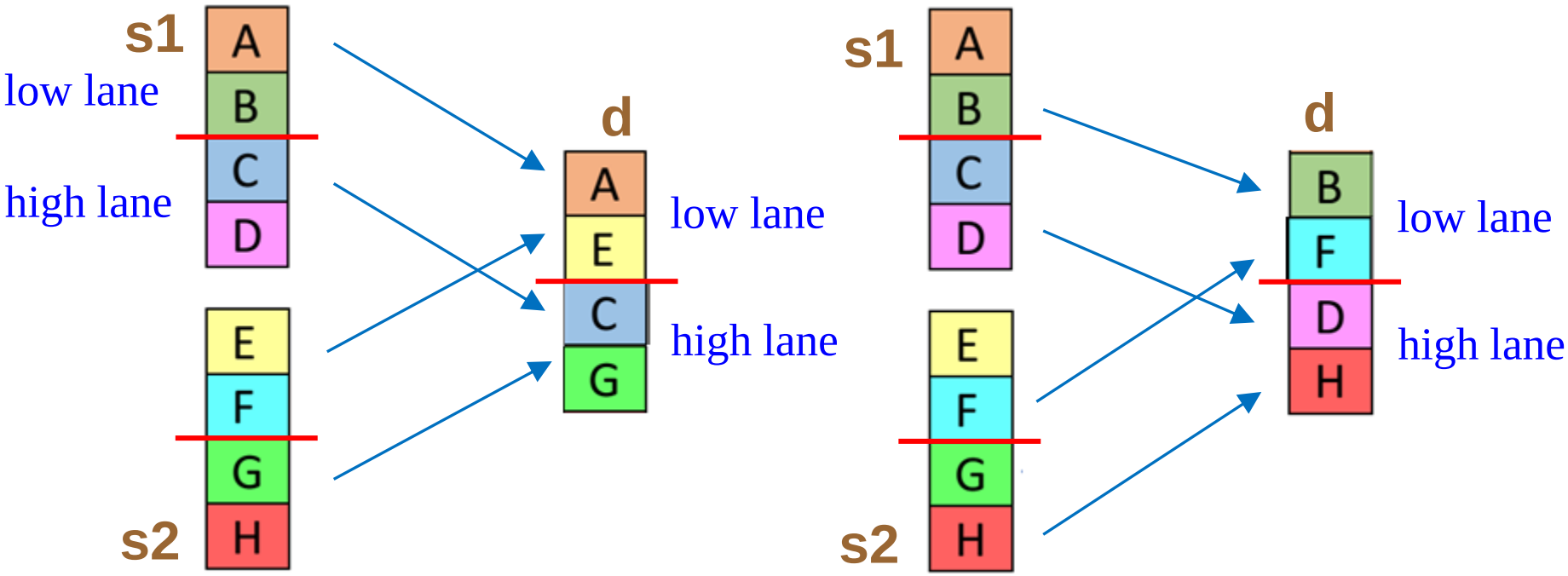
_m256 _mm256_dp_ps (_m256 a, _m256 b, const int imm8)

- dst[127:0] := DP(a[127:0], b[127:0], imm8[7:0])
- dst[255:128] := DP(a[255:128], b[255:128], imm8[7:0])

Matrix Transpose——Vectorized Version

- There are only a few cross-lane instructions (e.g., _mm256_permute4x64_pd).

AVX unpack operation



_mm256_unpacklo_pd

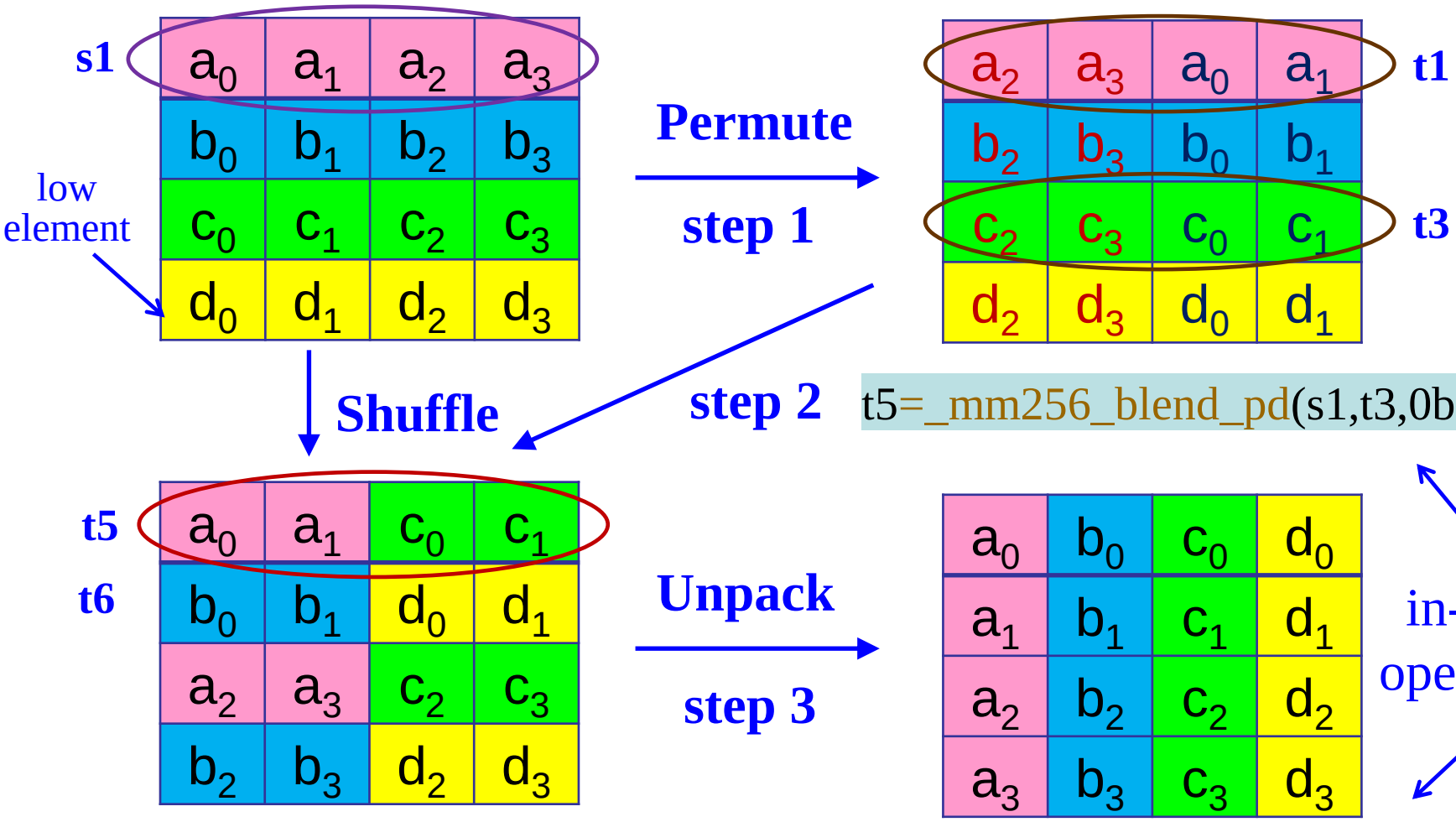
_mm256_unpackhi_ps

Matrix Transpose——Vectorized Version

- Transpose of a 4×4 matrix

cross-lane operation

```
t1=_mm256_permute4x64_pd(s1,0b01001110);
```




```
t5=_mm256_blend_pd(s1,t3,0b1100);
```

in-lane operation

```
_mm256_unpacklo_pd(t5,t6)
```

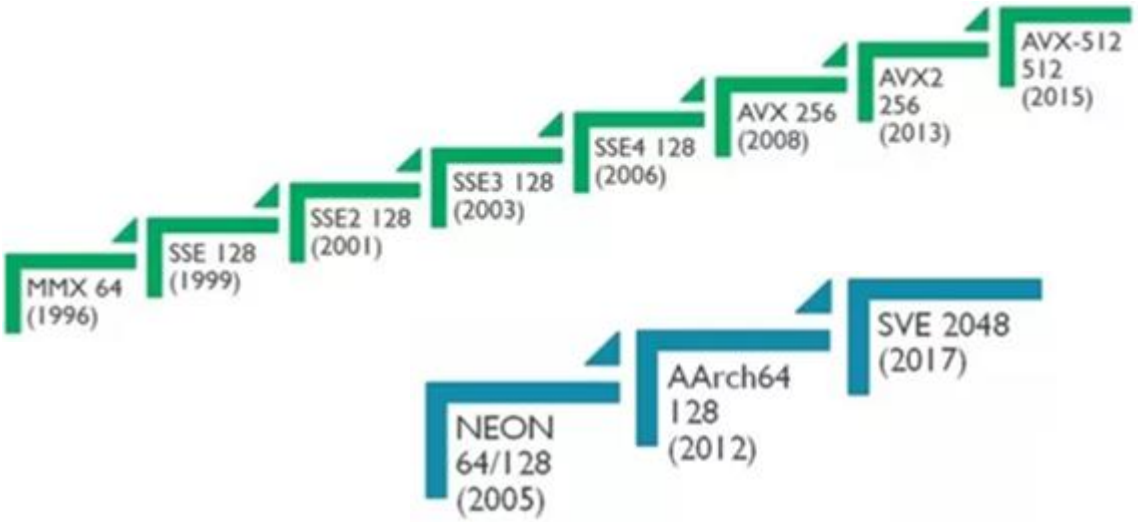
Matrix Transpose——Vectorized Version

using AVX Intrinsics

```
void tran_kernel4x4(const __m256d s1,const __m256d s2,const __m256d s3,const __m256d s4,double *d1,double *d2,double *d3,double *d4)
{
        __m256d t1,t2,t3,t4,t5,t6,t7,t8;
        t1=_mm256_permute4x64_pd(s1,0b01001110);        //a2,a3,a0,a1
        t2=_mm256_permute4x64_pd(s2,0b01001110);        //b2,b3,b0,b1
        t3=_mm256_permute4x64_pd(s3,0b01001110);        //c2,c3,c0,c1
        t4=_mm256_permute4x64_pd(s4,0b01001110);        //d2,d3,d0,d1
        t5=_mm256_blend_pd(s1,t3,0b1100);               //a0,a1,c0,c1
        t6=_mm256_blend_pd(s2,t4,0b1100);               //b0,b1,d0,d1
        t7=_mm256_blend_pd(t1,s3,0b1100);               //a2,a3,c2,c3
        t8=_mm256_blend_pd(t2,s4,0b1100);               //b2,b3,d2,d3
        _mm256_store_pd(d1,_mm256_unpacklo_pd(t5,t6));  //a0,b0,c0,d0
        _mm256_store_pd(d2,_mm256_unpackhi_pd(t5,t6));  //a1,b1,c1,d1
        _mm256_store_pd(d3,_mm256_unpacklo_pd(t7,t8));  //a2,b2,c2,d2
        _mm256_store_pd(d4,_mm256_unpackhi_pd(t7,t8));  //a3,b3,c3,d3
}
```

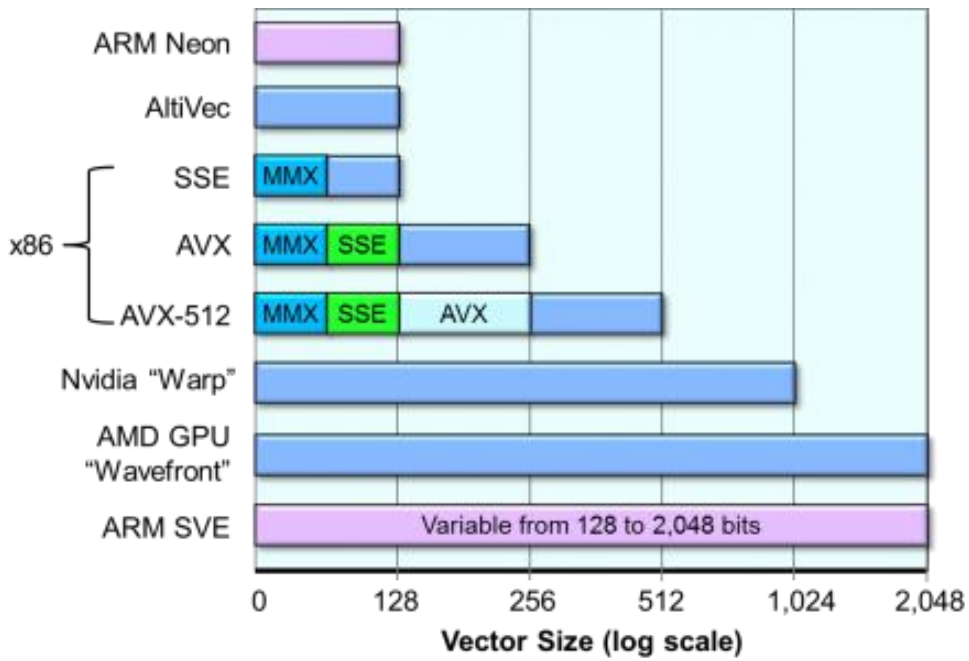
https://www.cxyzjd.com/article/artorias123/90513600

6. Overview of ARM SVE



Intel SIMD

ARM SIMD



ARM NEON

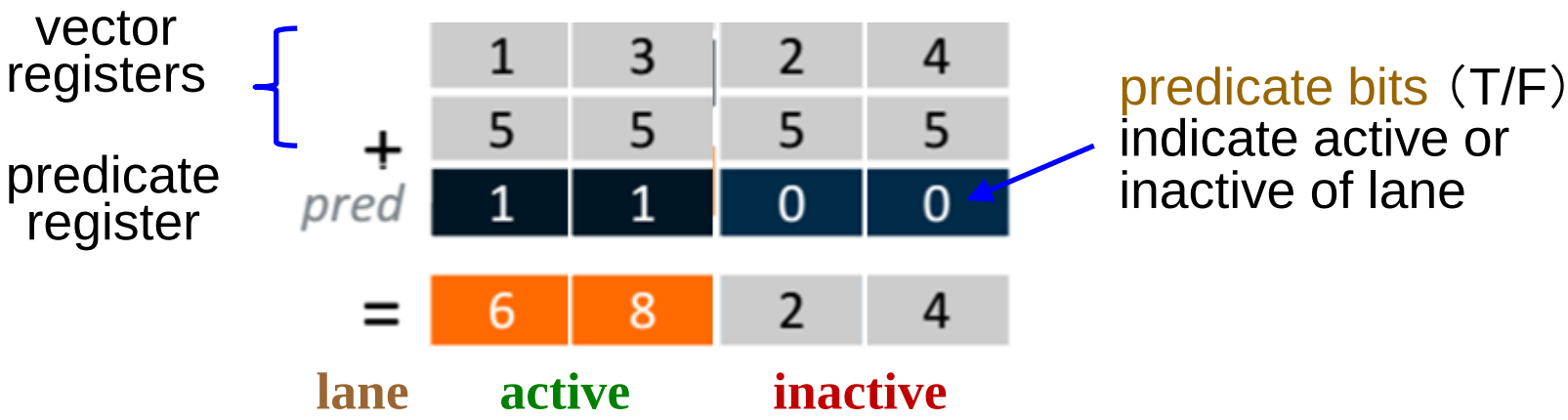
X86

(MMX/SSE/AVX)

ARM SVE

ARM SVE (Scalable Vector Extension)

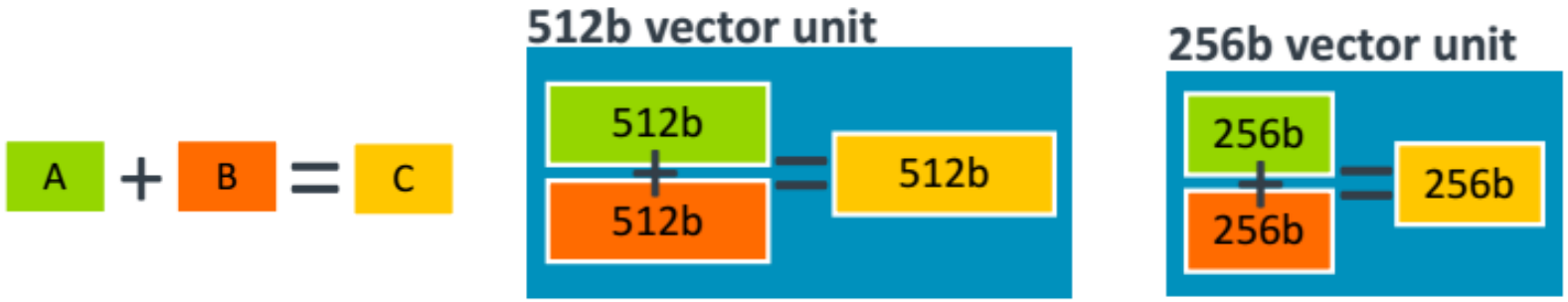
- SVE is vector length agnostic (VLA)
  - Vector length (VL) can be any multiple of 128 bits between 128 and 2048 bits.
  - SVE operates on individual lanes of vector controlled by a governing predicate register.



Per-lane predication

SVE Features

- Unlike traditional SIMD architectures, which define a fixed size for vector registers, SVE only specifies a maximum size, which permit program to adapt automatically to the current vector length at runtime.
- The program can scale dynamically to the new vector length without rewriting or recompiling.
- These features require a new programming style, called Vector Length Agnostic (VLA) programming.



The exact same binary code runs on hardware with different vector lengths

Predicate-Driven Loop Control

- For example, how do we compute data which has ten chunks of 4-bytes (totally 40-byte)?
- Intel 64 (scalar)
  - Ten iterations over a 4-byte register
- SSE (128-bit vector)
  - Two iterations over a 16-byte register + two iterations of a drain loop over a 4-byte register
- SVE (128-bit VLA)
  - Three iterations over a 16-byte VLA register with an adjustable predicate

4-byte/iteration

16-byte or 4-byte/iteration

16-byte/iteration

Daxpy (Scalar Version)

- A daxpy function (aX + Y) in C

```
void daxpy(double *x, double *y, double a, int N)
{
        int  i;

        for (i = 0; i < N; i++)
                y[i] = a*x[i] + y[i];
}
```

Scalar Code

Daxpy (Vectorized Version)

```
void daxpy(double *x, double *y, double a, int N)
{
    int i;                                  i+3 ≤ N-1
    for (i = 0; i <= N - 4; i += 4)         // vector loop
        y[i:i+3] = a*x[i:i+3] + y[i:i+3];

    for (; i < N; ++i)                      // loop tail
        y[i] = a*x[i] + y[i];
}
```

Vectorized Version-256 bit

Daxpy (SVE Version)

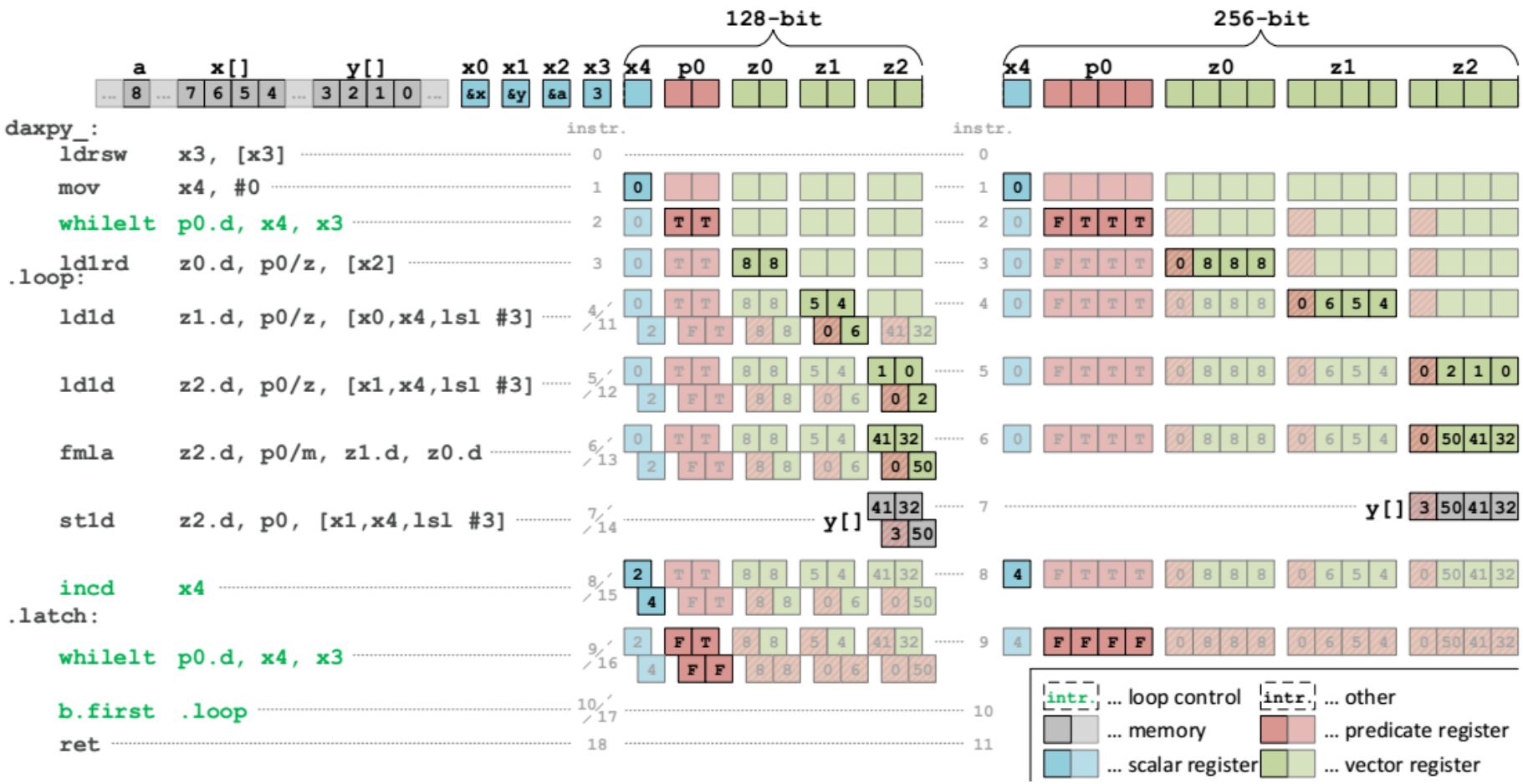
x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &N, x4 = 'i'

```
daxpy_:
  ldrsw   x3, [x3]              // x3=*n
  mov     x4, #0                // x4=i=0
  whilelt p0.d, x4, x3          // p0=while(i++<n)
  ld1rd   z0.d, p0/z, [x2]      // p0:z0=bcast(*a)
.loop:
  ld1d z1.d, p0/z, [x0, x4, lsl #3] // p0:z1=x[i]
  ld1d z2.d, p0/z, [x1, x4, lsl #3] // p0:z2=y[i]
  fmla z2.d, p0/m, z1.d, z0.d       // p0?z2+=x[i]*a
  st1d z2.d, p0, [x1, x4, lsl #3]   // p0?y[i]=z2
  incd x4                           // i+=(VL/64)
.latch:
  whilelt p0.d, x4, x3          // p0=while(i++<n)
  b.first .loop                 // more to do?
  ret
```

Vectorized Version (ARM/SVE)

Vector Length Agnostic Programming

- An example of daxpy loop (aX + Y) with vector lengths of 128-bit and 256-bit, and n = 3



The ARM scalable vector extension. IEEE micro, 2017

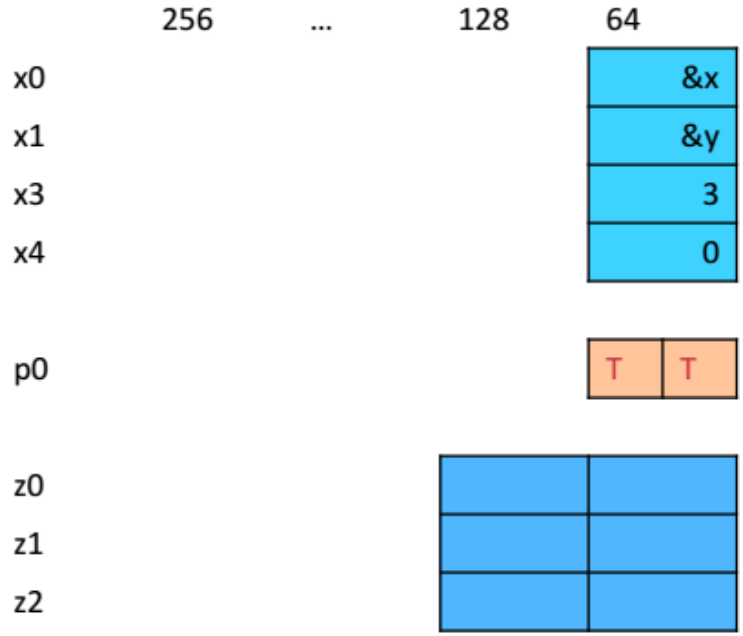
Daxpy (SVE-128 bit) (1/5)

x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &N, x4 = 'i' (a = 2, N = 3)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

build the loop predicate p0
p0.s[idx] = (x4 + idx) < x3
p0.s[0:1] = TRUE

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```



Vectorized Version (ARM/SVE-128 bit)

Daxpy (SVE-128 bit) (2/5)

x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &N, x4 = 'i' (a = 2, N = 3)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

	256	...	128	64
x0				&x
x1				&y
x3				3
x4				0
p0				T T
z0			2.0	2.0
z1			1.0	0.0
z2			0.0	0.0

Vectorized Version (ARM/SVE-128 bit)

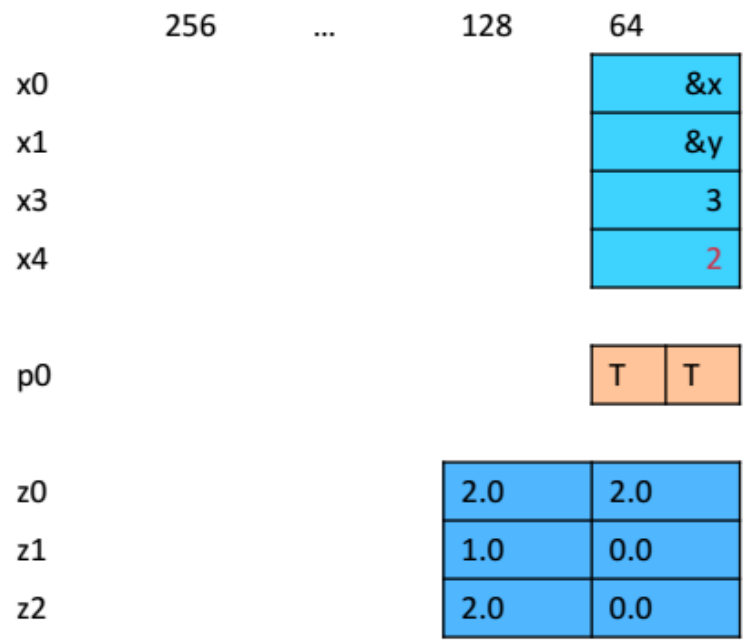
Daxpy (SVE-128 bit) (3/5)

x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &N, x4 = 'i' (a = 2, N = 3)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

x4 = x4 + VL/64
= x4 + 128/64 = x4 + 2



Vectorized Version (ARM/SVE-128 bit)

Daxpy (SVE-128 bit) (4/5)

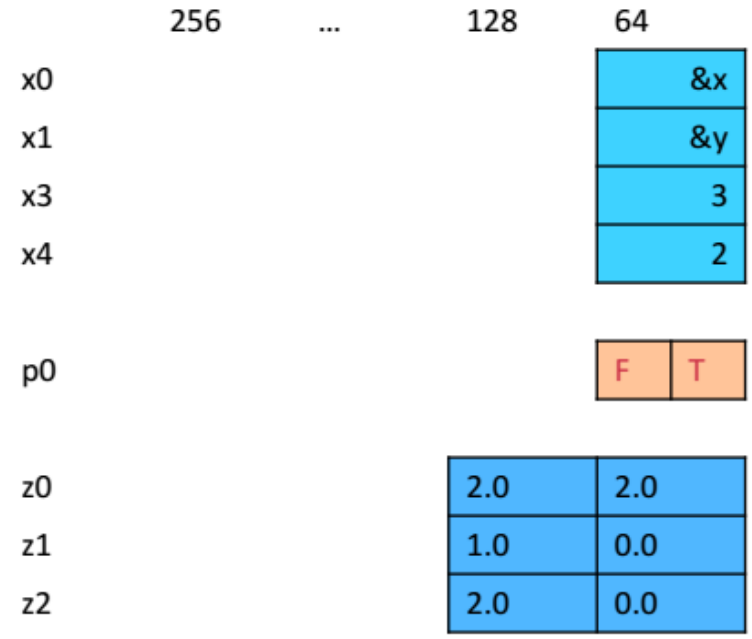
x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &N, x4 = 'i' (a = 2, N = 3)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

p0.s[idx] = (2 + idx) < x3

p0.s[0] = T, p0.s[1] = F



Vectorized Version (ARM/SVE-128 bit)

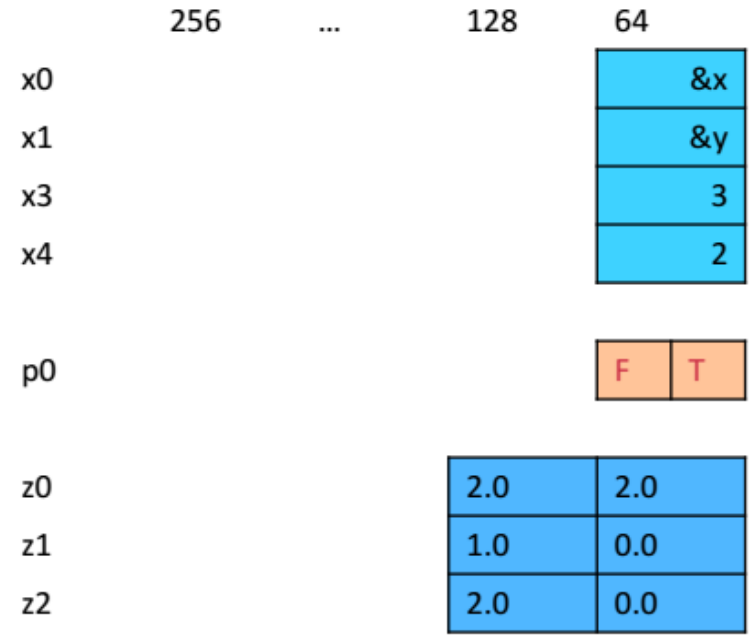
Daxpy (SVE-128 bit) (5/5)

x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &N, x4 = 'i' (a = 2, N = 3)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

if first element of p0 is true (p0.s[0] = T), then goto .loop



Vectorized Version (ARM/SVE-128 bit)

Daxpy (SVE-256 bit) (1/4)

x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &N, x4 = 'i' (a = 2, N = 3)

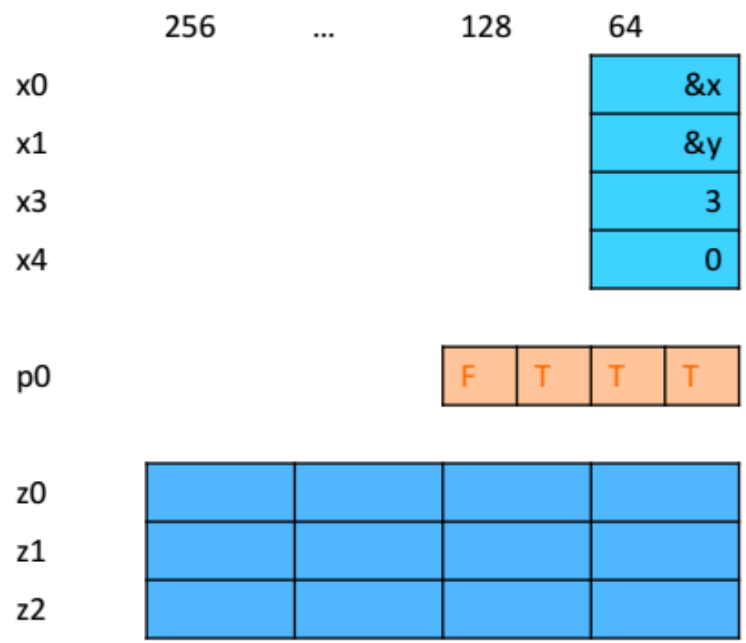
```
daxpy_:
    ldrsw    x3, [x3]
    mov      x4, #0
    whilelt  p0.d, x4, x3
    ld1rd    z0.d, p0/z, [x2]
.loop:
    ld1d     z1.d, p0/z, [x0,x4,lsl #3]
    ld1d     z2.d, p0/z, [x1,x4,lsl #3]
    fmla     z2.d, p0/m, z1.d, z0.d
    st1d     z2.d, p0, [x1,x4,lsl #3]
    incd     x4
.latch:
    whilelt  p0.d, x4, x3
    b.first  .loop
    ret
```

build the loop predicate p0

p0.s[idx] = (x4 + idx) < x3

p0.s[0:2] = T, p0.s[3] = F

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0



Vectorized Version (ARM/SVE-256 bit)

Daxpy (SVE-256 bit) (2/4)

x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &N, x4 = 'i' (a = 2, N = 3)

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	0	0	0

	256	...	128	64
x0				&x
x1				&y
x3				3
x4				0

p0			F	T	T	T

z0	0.0	2.0	2.0	2.0
z1	0.0	2.0	1.0	0.0
z2	0.0	0.0	0.0	0.0

Vectorized Version (ARM/SVE-256 bit)

Daxpy (SVE-256 bit) (3/4)

x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &N, x4 = 'i' (a = 2, N = 3)

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

x4 = x4 + VL/64
= x4 + 256/64 = x4 + 4

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	4	2	0

256 ... 128 64

x0	&x
x1	&y
x3	3
x4	4

p0	F	T	T	T

z0	0.0	2.0	2.0	2.0
z1	0.0	2.0	1.0	0.0
z2	0.0	4.0	2.0	0.0

Vectorized Version (ARM/SVE-256 bit)

Daxpy (SVE-256 bit) (4/4)

x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &N, x4 = 'i' (a = 2, N = 3)

Arrays	3	2	1	0
x[]	3	2	1	0
y[]	0	4	2	0

```
daxpy_:
    ldrsw   x3, [x3]
    mov     x4, #0
    whilelt p0.d, x4, x3
    ld1rd   z0.d, p0/z, [x2]
.loop:
    ld1d    z1.d, p0/z, [x0,x4,lsl #3]
    ld1d    z2.d, p0/z, [x1,x4,lsl #3]
    fmla    z2.d, p0/m, z1.d, z0.d
    st1d    z2.d, p0, [x1,x4,lsl #3]
    incd    x4
.latch:
    whilelt p0.d, x4, x3
    b.first .loop
    ret
```

p0.s[idx] = (4 + idx) < x3

p0.s[0:3] = F

	256	...	128	64
x0				&x
x1				&y
x3				3
x4				4
p0			F	F	F	F
z0	0.0	2.0	2.0	2.0
z1	0.0	2.0	1.0	0.0
z2	0.0	4.0	2.0	0.0

Vectorized Version (ARM/SVE-256 bit)

7. Data Dependence Analysis

- Vectorization changes the order of computation compared to sequential case.
- Now consider an example:

```
for (i=1; i≤n; i++)  {
S1:   a[i] = b[i] + c[i];
S2:   e[i] = a[i+1] * d[i];
}
```

- **serial execution:**
  S1(1), S2(1), S1(2), S2(2), ..., S1(N), S2(N)

VF = 4 validity ?

- **vectorized execution:**
  S1(1:4), S2(1:4), ..., S1(N-3:N), S2(N-3:N)

- Need to consider independence of operations – depends on vector width.

- Data dependence analysis
  - is at the core of current strategies for the automatic detection of implicit vectorization in programs.
  - determines whether it is safe to reorder or parallelize statements for vectorization.
- Consequently, the compiler must perform dependence analysis to prove that vectorization will produce correct result.

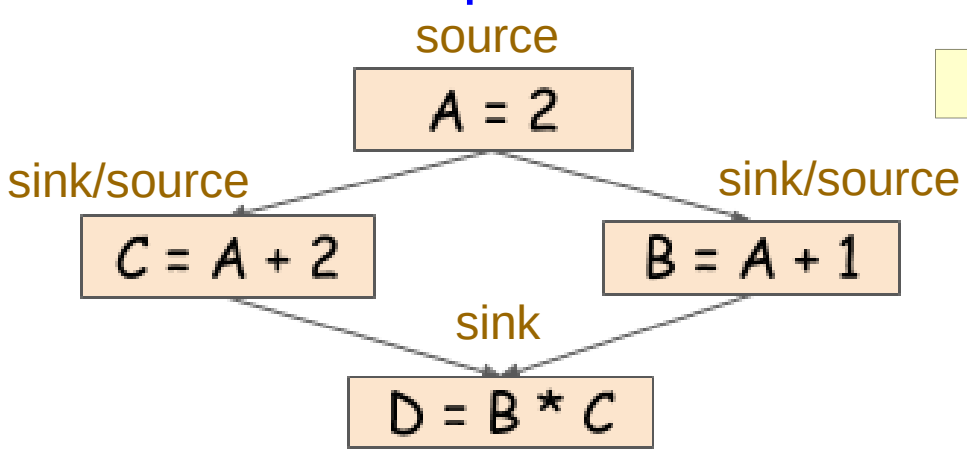
Definition of Data Dependence

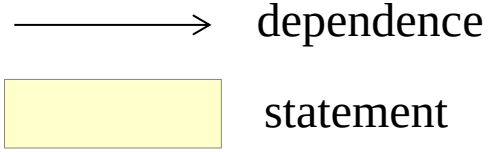
- A statement S is said to be data dependent on statement T if
  - T (source) executes before S (sink) in the original sequential or scalar program.
  - S and T access the same data item.
  - At least one of the accesses is a write.
  - A dependence is an edge from source to sink.

code

```
A = 2
B = A + 1
C = A + 2
D = B * C
```

data dependence

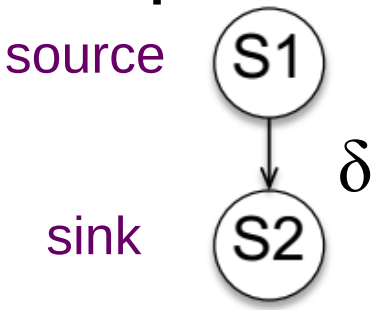




Types of Dependences (1/2)

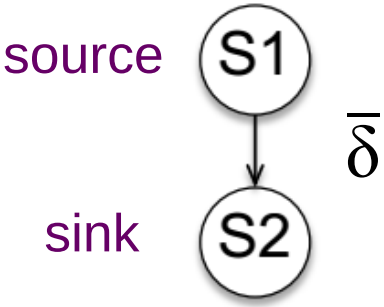
- **Read After Write** (Flow/true dependence)

S1: X = A+B
S2: C= X+A



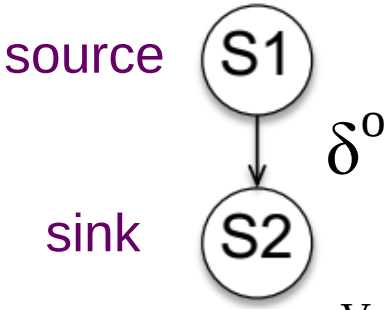
- **Write After Read** (Anti-dependence)

S1: A = X + B
S2: X = C + D



- **Write After Write** (Output dependence)

S1: X = A+B
S2: X = C + D



Types of Dependences (2/2)

Access in S_i	Access in S_j	Dependence	Notation
Write m	Read m	Flow (true) dependence	$S_i \ \delta \ S_j$
Read m	Write m	Anti (Pseudo) dependence	$S_i \ \bar{\delta} \ S_j$
Write m	Write m	Output (Pseudo) dependence	$S_i \ \delta^o \ S_j$
Read m	Read m	Input dependence, does not matter	

- True dependence cannot be eliminated.
- Pseudo dependences may be eliminated by some transformations.

No Dependences Exist

- Vectorization : Yes
- Parallelization: Yes

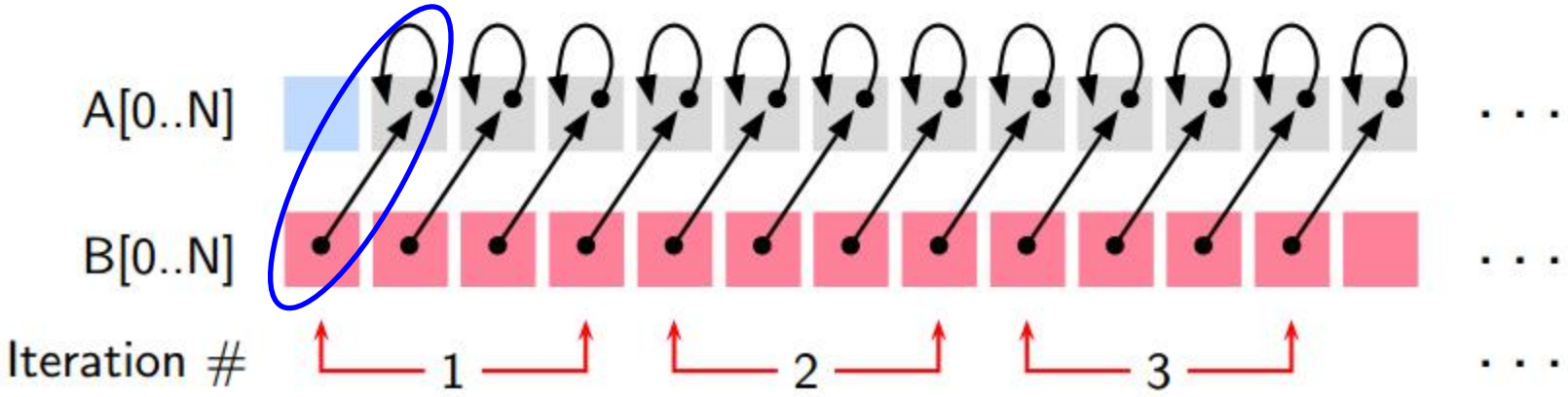
Vectorization Factor

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
  A[i:i+3] = A[i:i+3] + B[i-1:i+2];
```



Dependence: Write After Read (1/3)

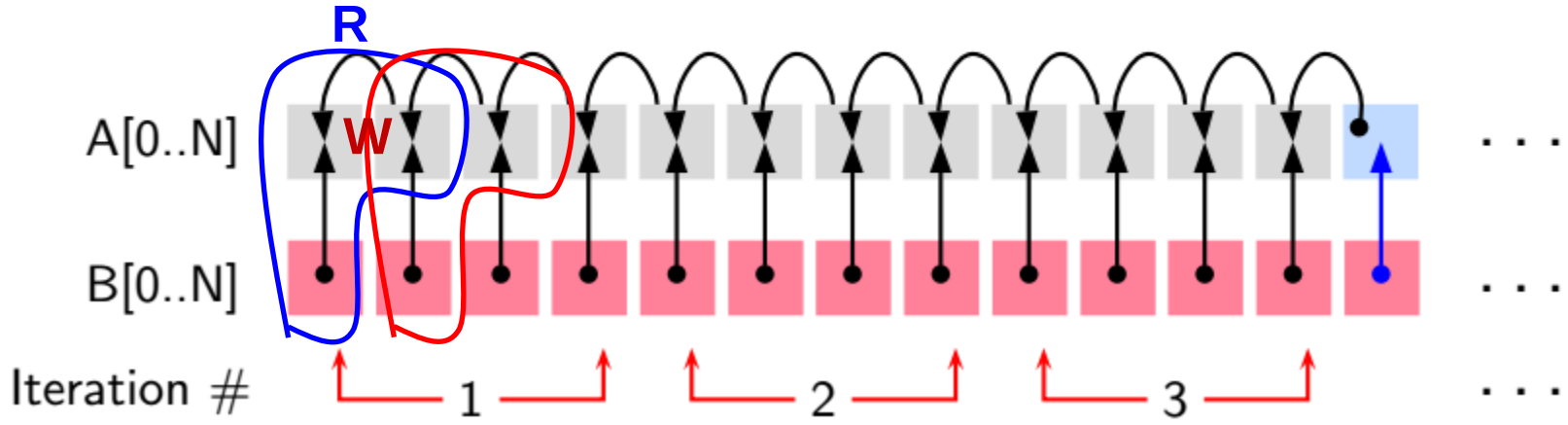
- Vectorization : Yes
- Parallelization: No

https://www.godbolt.org/z/rsYxhhY1T

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

- Vector instruction is synchronized: All reads before writes in a given instruction
- Read-writes across multiple instructions executing in parallel may not be synchronized



Dependence: Write After Read (2/3)

- Now consider an example:
  - A = {0,1,2,3,4}
  - B = {5,6,7,8,9}

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Applying each operation sequentially:

a[0] = a[1] + b[0] → a[0] = 1 + 5 → a[0] = 6

a[1] = a[2] + b[1] → a[1] = 2 + 6 → a[1] = 8

a[2] = a[3] + b[2] → a[2] = 3 + 7 → a[2] = 10

a[3] = a[4] + b[3] → a[3] = 4 + 8 → a[3] = 12

a = {6, 8, 10, 12 , 4}

Dependence: Write After Read (3/3)

- Now try vector operations:
  - A = {0,1,2,3,4}
  - B = {5,6,7,8,9}

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Applying vector operations, i={1,2,3,4}:

a[i+1] = {1,2,3,4} (load)

b[i] = {5,6,7,8} (load)

{1,2,3,4} + {5,6,7,8} = {6, 8, 10, 12} (operate)

a[i] = {6, 8, 10, 12} (store)

a = {6, 8, 10, 12, 4} = {6, 8, 10, 12, 4} Vectorizable

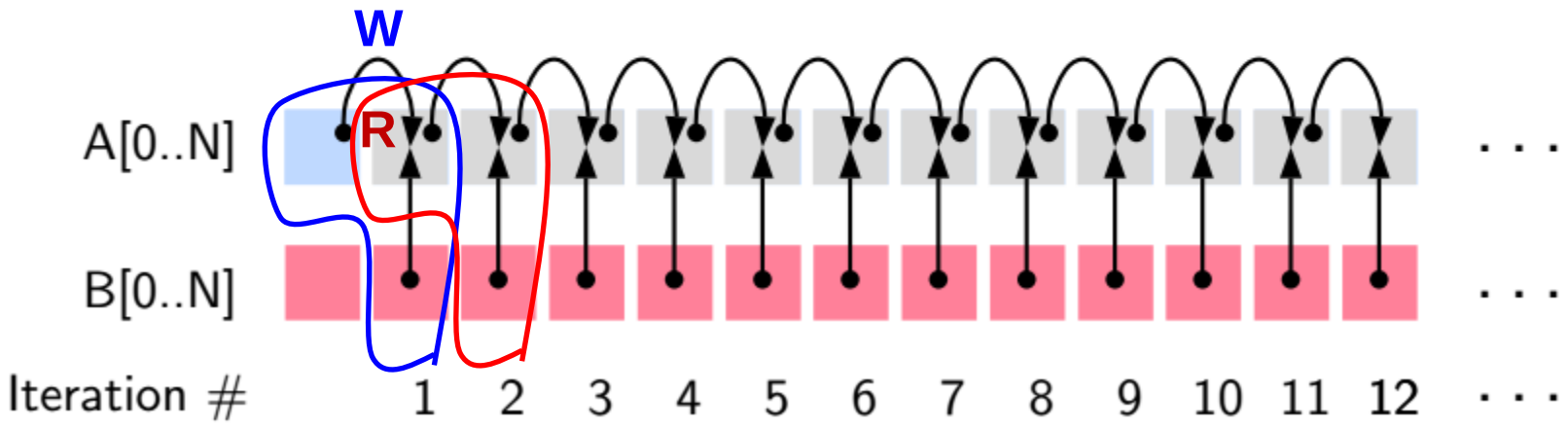
Dependence: Read After Write (1/3)

- Vectorization : No
- Parallelization: No

https://www.godbolt.org/z/Gohz5ee7o

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location



Dependence: Read After Write (2/3)

- Now consider an example:
  - A = {0,1,2,3,4}
  - B = {5,6,7,8,9}

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Applying each operation sequentially:

a[1] = a[0] + b[1] → a[1] = 0 + 6 → a[1] = 6

a[2] = a[1] + b[2] → a[2] = 6 + 7 → a[2] = 13

a[3] = a[2] + b[3] → a[3] = 13 + 8 → a[3] = 21

a[4] = a[3] + b[4] → a[4] = 21 + 9 → a[4] = 30

a = {0, 6, 13, 21, 30}

Dependence: Read After Write (3/3)

- Now try vector operations:
  - A = {0,1,2,3,4}
  - B = {5,6,7,8,9}

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Applying vector operations, i={0,1,2,3}:

A[i] = {0,1,2,3} (load)

B[i+1] = {6,7,8,9} (load)

{0,1,2,3} + {6,7,8,9} = {6, 8, 10, 12} (operate)

A[i] = {6, 8, 10, 12} (store)

A = {0, 6, 8, 10, 12} ≠ {0, 6, 13, 21, 30}

Not Vectorizable

Dependence: Write After Write

- Vectorization : No
- Parallelization: No

```
int  A[N], B[N], i
for (i=0; i<N; i++)
      A[0] = A[i] + B[i];
```

Multiple loop iterations alter the value of a single variable.

- If an identical address exists for any two (or more) store operations, the result to be stored is indeterminate.
- For this reason, “write after write” is non-vectorizable.

Dependence: Read After Read

- Vectorization : Yes
- Parallelization: Yes

```
int  A[N], B[N],  C[N],  i;
for  (i=0;  i<N;  i++)
    A[i] = B[i%2] + C[i];
```

There is not really a dependence here.

- There is not really a dependence in “read after read”. The loop could be vectorized.
- However, the compiler might find it tricky to use vector instructions.

Brief Summary

- If parallelization is possible then vectorization is trivially possible.
- Vectorization does not imply parallelization.
- For simple loop vectorization, we can obtain:
  - Read After Write
    - Not vectorizable
    - Not parallelizable
  - Read After Read
    - Vectorizable
    - Parallelizable
  - Write After Read
    - Vectorizable
    - Not parallelizable
  - Write After Write
    - Not Vectorizable
    - Not parallelizable

Are they always correct ?

Revisiting the Data Dependences

Access in S_i	Access in S_j	Dependence	Notation
Write m	Read m	Flow (true) dependence	$S_i \ \delta \ S_j$
Read m	Write m	Anti (Pseudo) dependence	$S_i \ \bar{\delta} \ S_j$
Write m	Write m	Output (Pseudo) dependence	$S_i \ \delta^o \ S_j$
Read m	Read m	Input dependence, does not matter	

- True dependence cannot be eliminated.
- Pseudo dependences may be eliminated by some transformations.

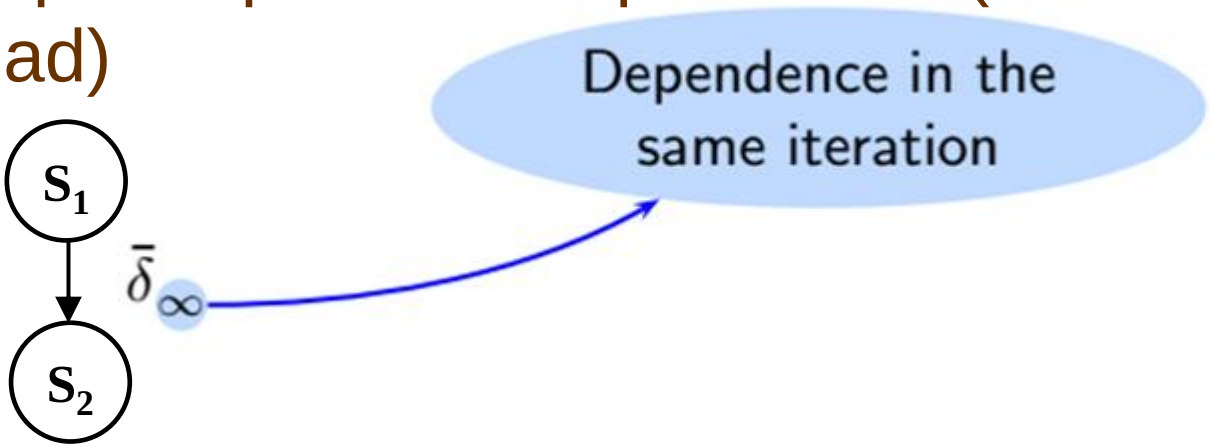
Loop-independent and Loop-carried Dependences

- If in a loop statement S_2 depends on S_1, then there are two possible ways of dependence:
- Loop-independent dependence
  - S_1 and S_2 execute on the same iteration. Namely, dependence exists within an iteration.
  - If the loop is removed, the dependence still exists.
- Loop-carried (or cross-iteration) dependence
  - Dependence exists across iterations. Namely, source and sink happen on different iterations.
  - If the loop is removed, the dependence no longer exists.

Loop-independent Dependence

```
int   A[N], B[N], C[N], i;
for  (i=1; i<N; i++)   {
S1:      C[i] = A[i] + B[i];
S2:      A[i] = A[i] + B[i];
}
```

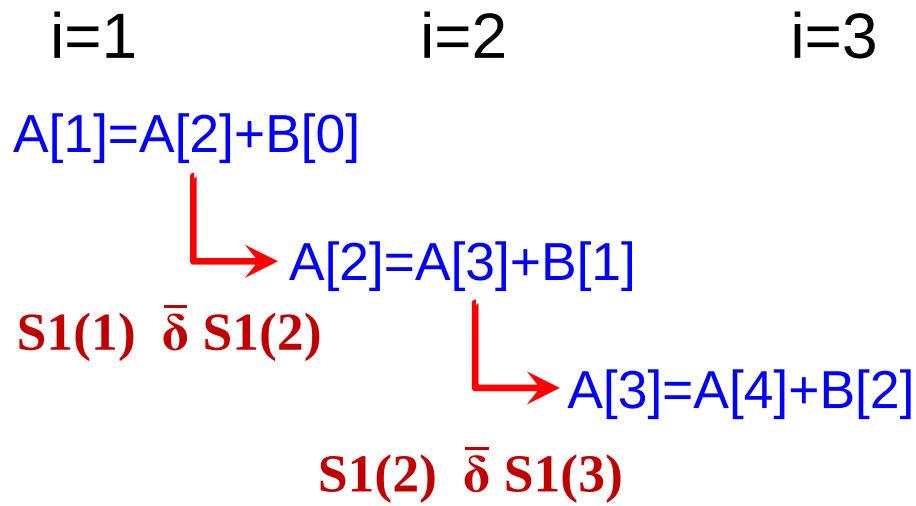
- Dependence graph
  - Loop-independent dependence (Write After Read)



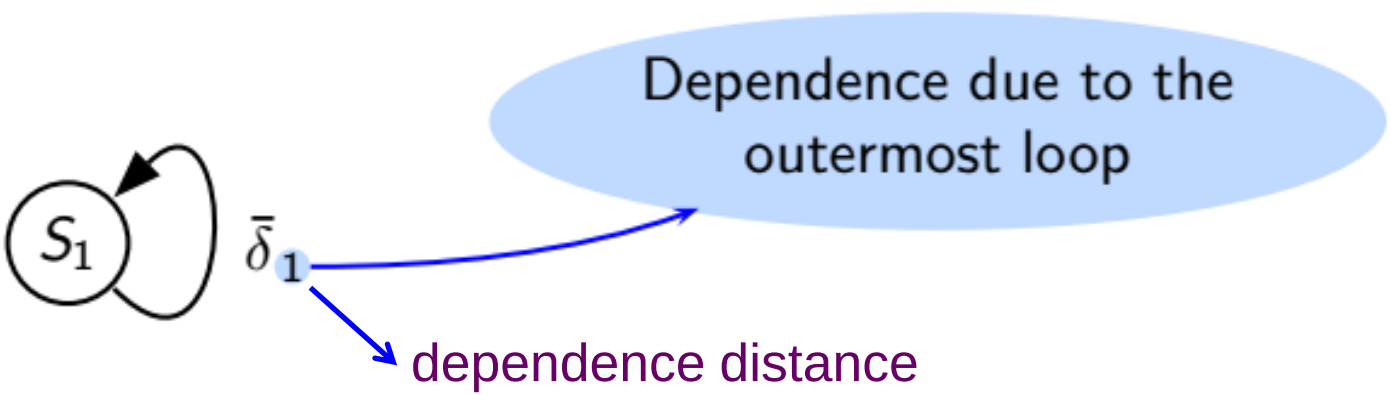
- No Loop-carried dependence

Loop-carried Dependence (1/2)

```
int   A[N], B[N], i;
for  (i=1; i<N; i++)
S1:   A[i] = A[i+1] + B[i-1];
```



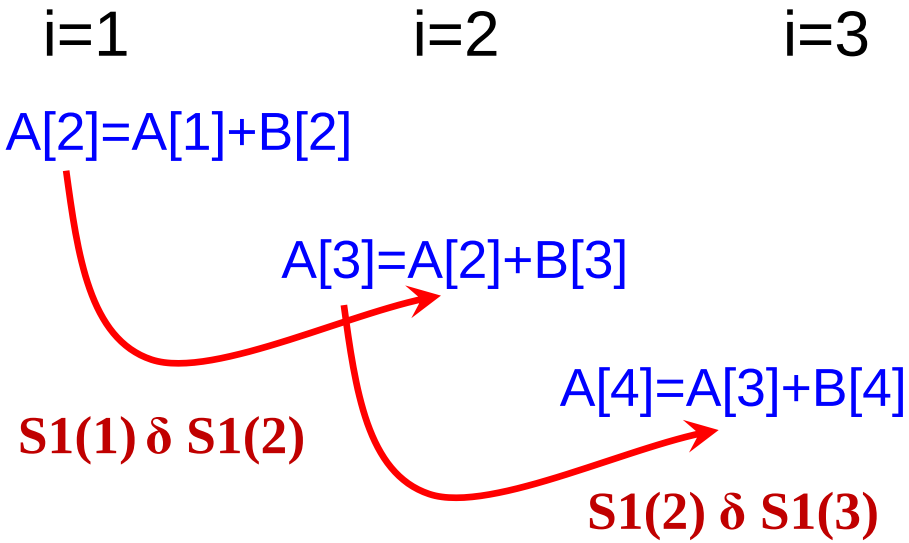
- Dependence graph
  - Loop-carried dependence (Write After Read)



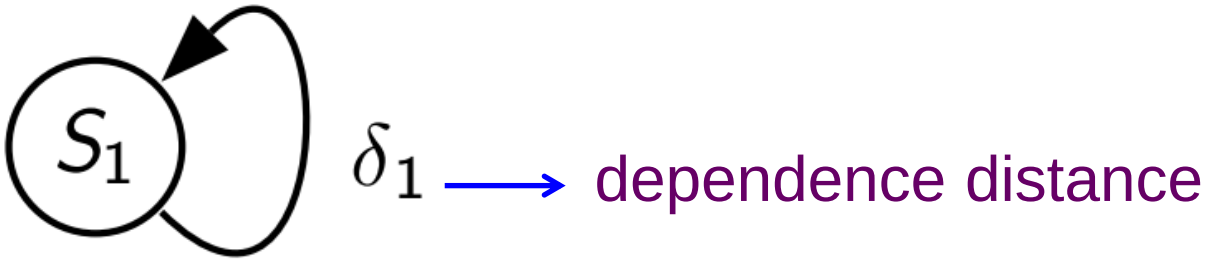
- No Loop-independent dependence

Loop-carried Dependence (2/2)

```
int   A[N], B[N], i;
for  (i=1; i<N; i++)
S1:    A[i+1] = A[i] + B[i+1];
```



- Dependence graph
  - Loop-carried dependence (Read After Write)



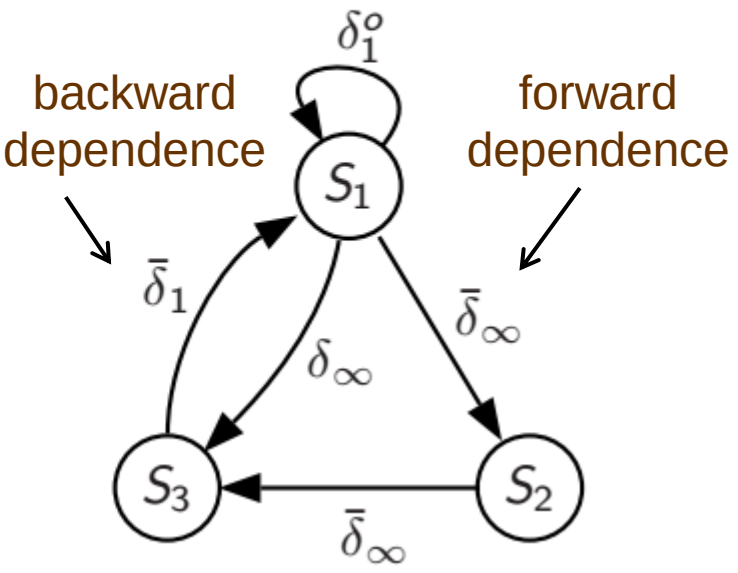
- No Loop-independent dependence

A Complicated Example

Program to swap arrays

```
for (i=0; i<N; i++)
{
    T = A[i];          /* S1 */
    A[i] = B[i];       /* S2 */
    B[i] = T;          /* S3 */
}
```

Dependence Graph

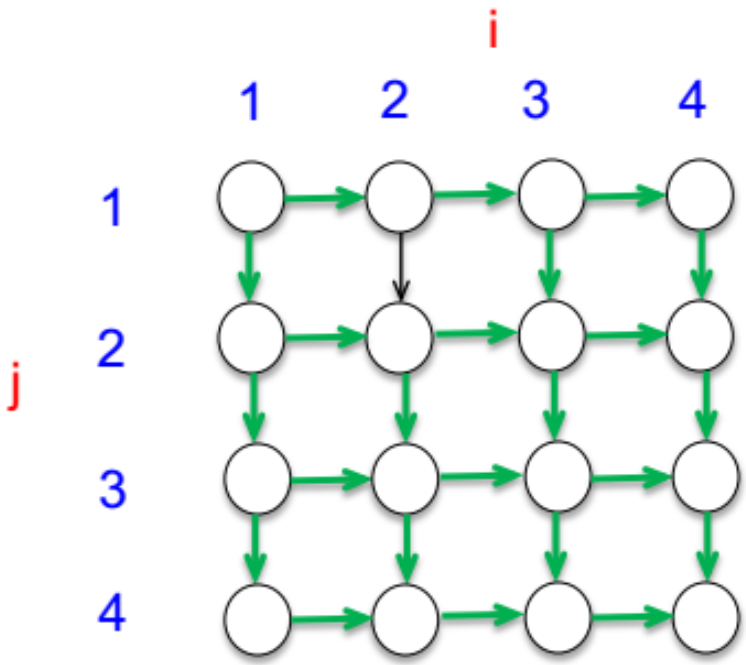


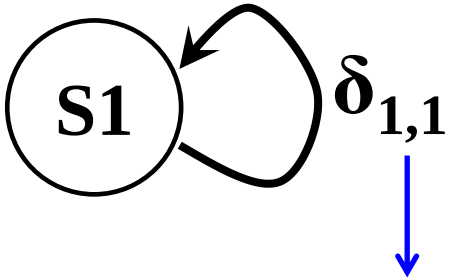
- Loop-independent dependence
  - Write After Read: $S_1 \rightarrow S_2$, $S_2 \rightarrow S_3$
  - Read After Write: $S_1 \rightarrow S_3$
- Loop-carried dependence
  - Write After Write: $S_1 \rightarrow S_1$
  - Write After Read: $S_3 \rightarrow S_1$

inconsistent with the order of execution (backward dependence)

Dependences in Nested Loops

```
for (i=1; i<n; i++)
    for (j=1; j<n; j++)
S1:     a[i][j]=a[i][j-1] + a[i-1][j];
```





dependence distance (i,j)

Validity Condition for Vectorization

- For the given program, consider its Data Dependence Graph (DDG)
  - If the DDG does not have a cycle (acyclic), vectorization is possible.
  - If the DDG has a cycle, vectorization may or may not be possible depending on the nature of cycle (e.g., dependence type, dependence direction, dependence distance, vectorization factor (VF)).
- The validity condition for vectorization is still an open problem.

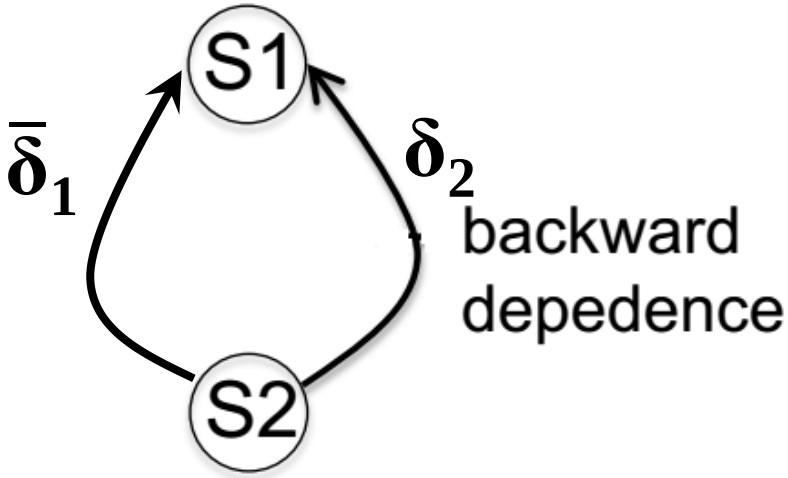
Acyclic Dependences and Vectorization

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: B[i+2] = A[i+1];
}
```

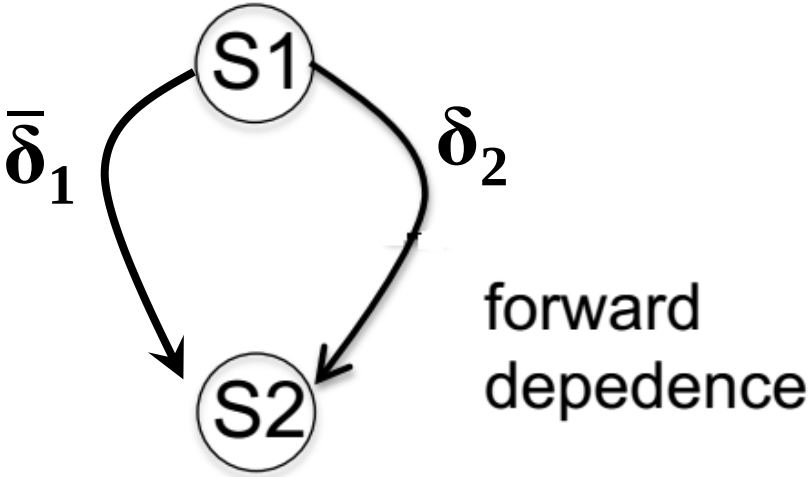
→

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i+1];
    S2: A[i] = B[i];
}
```

Reordering Statements



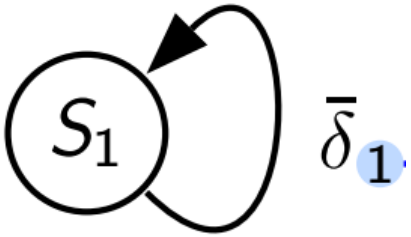
Not Vectorizable



Vectorizable

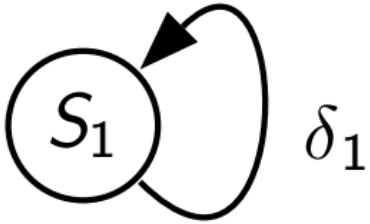
Cyclic Dependences and Vectorization (1/2)

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```



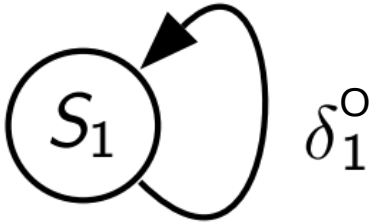
Vectorizable

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```



Not Vectorizable

```
int   A[N], B[N], i
for   (i=0; i<N; i++)
      A[0] = A[i] + B[i] ;
```



Not Vectorizable

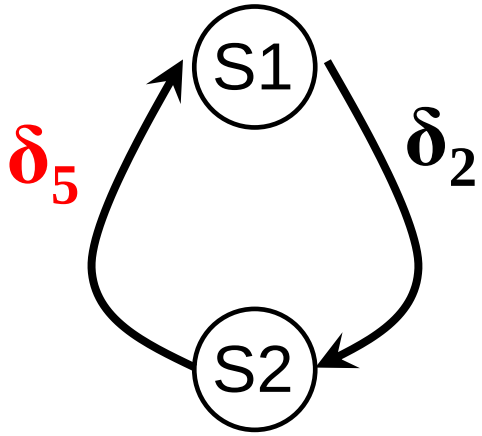
Cyclic Dependences and Vectorization (2/2)

Cyclic True Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i];
    S2: A[i+5] = B[i];
}
```

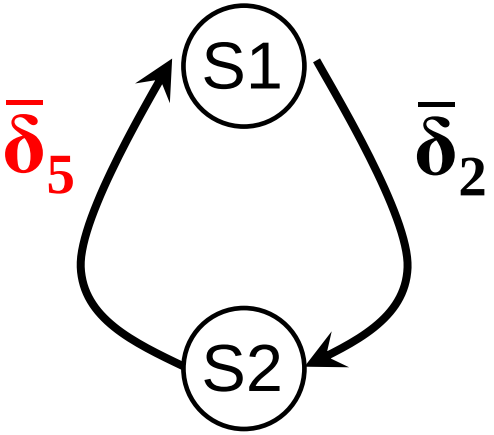
Cyclic Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i] = A[i+2];
    S2: A[i] = B[i+5];
}
```



if VF = 4

Both vectorizable!



- Though DDG has a cycle, one of the dependence distances are larger than VF (5 > 4). The vectorization is feasible.

8. Data Dependence Testing

- Data dependence testing is the heart of auto parallelization and vectorization.
- If a loop is data dependence-free between any two iterations then it can be safely executed in parallel.
- In science/engineering applications, loop parallelism is most important.

Classification for simplification: Kennedy approach

- Test for each subscript in turn, if any subscript has no dependence - then no solution.

```
for (int i=1; i<10; i++)
    for (int j=1; j<20; j++)
        a[i+1, j, 3]=a[i, j+1, 5] + 50;
```

no dependences exist !

  - Subscript in 1st dim contains dependence
  - Subscript in 2nd dim contains dependence
  - Subscript in 3rd dim contains no dependence

R. Allen and K Kennedy. Optimizing compilers for modern architectures: A dependence based approach. 2001.

Exact versus Inexact Tests

- There are three possible answers that any dependence test can give:
  1. No dependence - can prove that no dependence exists.
  2. Dependence - can prove that a dependence exists.
  3. Not sure - could neither prove nor disprove dependences. To be safe, the compiler must assume a dependence in this case. This is the conservative assumption for dependence testing, necessary to guarantee correct execution.

We call a dependence test exact if it only reports answers 1 or 2. Otherwise, it is inexact.

Delta Test (1/3)

```
for (int i=1; i<10; i++)
    a[i+1] = a[i] + 8;
```

Read After Write (RAW) dependence

- Assume a[i+1] is the source and a[i] is the sink.
  - forming an equality: $i_0 + 1 = i_0 + \Delta i \quad (\Delta i \geq 0)$
  - solving it gives us: $\Delta i = 1$ RAW dependence
- Assume a[i] is the source and a[i+1] is the sink.
  - forming an equality: $i_0 + 1 + \Delta i = i_0 \quad (\Delta i \geq 0)$
  - solving it gives us: $\Delta i = -1 < 0$

no WAR dependence

Delta Test (2/3)

```
for (int i=1; i<10; i++)
    a[2i+1] = a[2i] + 8;
```

no dependence

- Assume source - a[2i+1] and sink - a[2i]
  - forming an equality: $2\,i_0 + 1 = 2\,(i_0 + \Delta i) \quad (\Delta i \geq 0)$
  - solving it gives us: $\Delta i = 1/2$ no dependence
- Assume source - a[2i] and sink - a[2i+1].
  - forming an equality: $2\,(i_0 + \Delta i) + 1 = 2\,i_0 \quad (\Delta i \geq 0)$
  - solving it gives us: $\Delta i = -1/2 < 0$ no dependence

Delta Test (3/3)

- The delta test forms the dependence equations and computes the dependence distance between the two accesses. However, this may not always be possible!

```
for (int i=1; i<50; i++)
    for (int j=1; j<50; j++)
      a[2i, 3j-3] = a[4j+1, 6i] + 8;
```

We cannot compute dependence distance (Δi, Δj) easily.

- Assume source - a[2i, 3j-3] and sink - a[4j+1, 6i].
  - forming equalities and inequalities:

$$\begin{cases} 2i_0 = 4 \times (j_0 + \Delta j) + 1 \\ 3j_0 - 3 = 6 \times (i_0 + \Delta i) \\ \Delta i, \Delta j \geq 0 \\ 1 \leq i, j < 50 \end{cases}$$

In this case we have to solve Diophantine equations.

GCD Test (1/2)

- The Diophantine equation

  $a_1 \times i_1 + a_2 \times i_2 + \ldots + a_n \times i_n = c$

  exists solutions if and only if $\gcd(a_1, a_2, \ldots, a_n)$ evenly divides c.

- Examples:
  - $15 \times i + 6 \times j - 9 \times k = 12$ has solutions (gcd=3)
  - $12 \times i + 7 \times j + 3 \times k = 3$ has solutions (gcd=1)
  - $9 \times i + 3 \times j + 6 \times k = 5$ has no solutions (gcd=3)

GCD Test (2/2)

```
for (int i=1; i<50; i++)
    for (int j=1; j<50; j++)
      a[2i, 3j-3] = a[4j+1, 6i] + 8;
```

no dependences exist !

- GCD test for the example:
  - Subscript in 1st dim: $2 \times i = 4 \times j + 1$

    $2 \times i - 4 \times j = 1$ has no solutions (gcd=2)
  - Subscript in 2nd dim: $3 \times j - 3 = 6 \times i$

    $3 \times j - 6 \times i = 3$ has solutions (gcd=3)
- Limitations of GCD test:
  - it does not consider any bound information of loop index variables, e.g., $1 \le i, j < 50$
  - the $\gcd(a_1,\ldots,a_n)$ is often 1

Banerjee Test (1/2)

- Basic idea (Extreme Value Test)
  - if the total subscript range accessed by *ref1* does not overlap with the range accessed by *ref2*, then *ref1* and *ref2* are independent.
- For example: $A[a \times i_a + c_a] = A[b \times i_b + c_b]$
  - dependence equation

1) $a\, i_a + c_a = b\, i_b + c_b$
2) $h(i_a, i_b) = a\, i_a - b\, i_b + c_a - c_b = 0$

  - according to intermediate value theorem (IVT)

1) if $\min(h) \le 0 \le \max(h)$ then dependence exists
2) else no dependences exist

Banerjee Test (2/2)

```
for (int i=1; i<= 100; i++)
    a[2*i+3] = a[i+7];
```

- Banerjee test for the example:
  - We have $2i_a + 3 = i_b + 7$
  - $h(i_a, i_b) = 2i_a - i_b - 4$ and $1 \le i_a, i_b \le 100$
  - $\max(h) = (2 * 100 - 1 - 4) = 195$
  - $\min(h) = (2 * 1 - 100 - 4) = -102$

$\min(h) = -102 \le 0 \le \max(h) = 195$

Hence dependence may exist.

Summary

- The problem of finding dependence has been proven to be equivalent to the problem of finding solutions to a system of Diophantine equations, which is NP-complete.
- Most methods consider only linear subscript expressions.
- Three dependence tests
  - Delta test: an exact test
  - GCD test: an inexact test, not accurate
  - Banerjee test: an inexact test, widely used test

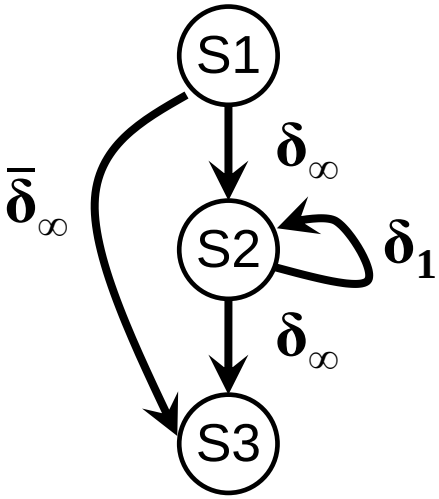
9. Transformations for Vectorization

- When the dependence graph has a cycle, vectorization may be achieved by:
  - Loop distribution
  - Strip mining
  - Reordering statements
  - Node splitting
  - Scalar expansion
  - Loop peeling
  - Freezing loops
  - Loop interchanging
  - Removing dependences

Loop Distribution (1/4)

- Loop distribution (also called loop fission or loop splitting) divides loop control over different statements in the loop body to vectorize the code.

```
for (i=1; i<n; i++)  {
S1:      b[i] = b[i] + c[i];
S2:      a[i] = a[i-1] + b[i];
S3:      c[i] = a[i] + 1;
}
```



Loop was not vectorized.

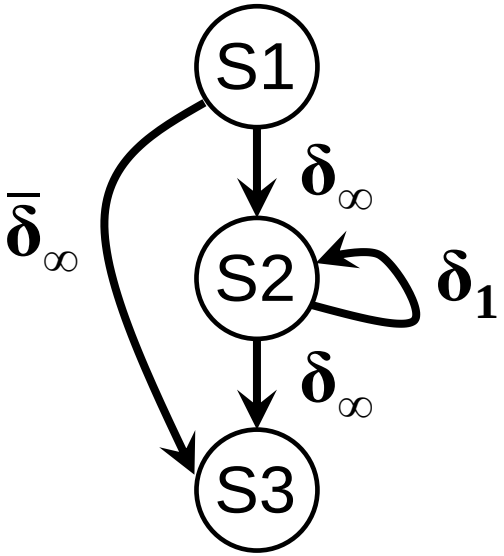
```
b[1:n-1] = b[1:n-1] + c[1:n-1];
for (i=1; i<n; i++)
    a[i] = a[i-1] + b[i];
c[1:n-1] = a[1:n-1] + 1;
```

Loop 1 and 3 was vectorized.
Loop 2 was not vectorized.

Loop Distribution (2/4)

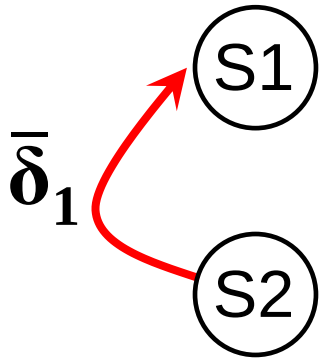
- Validity Condition for loop distribution:
  - Sufficient (but not necessary) condition: a loop with two statements can be distributed if there are no dependences from any instance of the later statement to any instance of the earlier one.

```
for (i=1; i<n; i++)  {
S1:      b[i] = b[i] + c[i];
S2:      a[i] = a[i-1] + b[i];
S3:      c[i] = a[i] + 1;
}
```



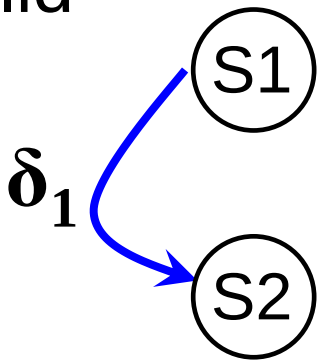
Loop Distribution (3/4)

- Example: loop distribution is not valid (executing all S1 first and then all S2):




```
for (i=1; i<n; i++)  {
S1:      a[i] = b[i] + c[i];
S2:      e[i] = a[i+1] * d[i];
}
```

- Example: loop distribution is valid




```
for (i=1; i<n; i++)  {
S1:      a[i] = b[i] + c[i];
S2:      e[i] = a[i-1] * d[i];
}
```

Loop Distribution (4/4)

```
for (i=1; i<n; i++)  {
      b[i] = b[i] + c[i];
      a[i] = a[i-1] + b[i];
      c[i] = a[i] + 1;
}
```

```
b[1:n-1] = b[1:n-1] + c[1:n-1];
for (i=1; i<n; i++)
      a[i] = a[i-1] + b[i];
c[1:n-1] = a[1:n-1] + 1;
```

- Since the loop distribution decreases cache and/or register locality, it may result in limited performance benefit or even slowdowns.
- Profitability analysis should determine if the overhead of this transformation can be amortized by the speedups obtained.

Strip Mining (1/2)

- Strip-mining turns one loop into two nested loops, which is used to enable vectorization and improve memory locality in the inner loop.

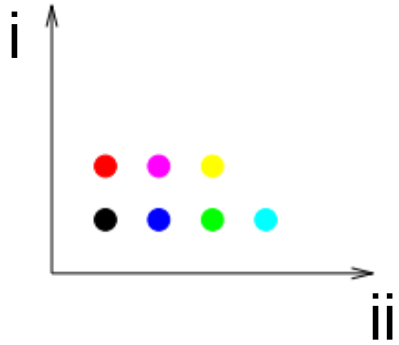
```
for (i=0; i<n; i++)  {
    //...   do work
}
```

```
const  int  STRIP = 1024;
for (ii=0; ii<n; ii += STRIP)
    for (i=ii; i<ii + STRIP; i++)  {
    //...   do work
 }
```

original loop

strip-mined loop

Original Loop



Stripmined Loop: strip size = 2

Strip Mining (2/2)

```
for (i=1; i<LEN; i++) {
    a[i]= b[i];
    c[i] = c[i-1] + a[i];
}
```

Distribution

```
for (i=1; i<LEN; i++)
    a[i]= b[i];
for (i=1; i<LEN; i++)
    c[i] = c[i-1] + a[i];
```

Loop distribution increases the cache miss ratio

Strip mining

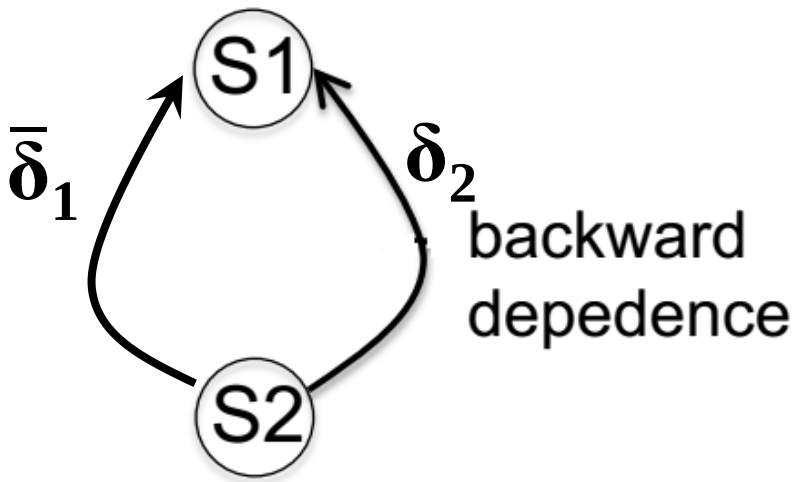
vectorization factor

```
for (i=1; i<LEN; i+=strip_size) {
    a[i:i+strip_size-1] = b[i:i+strip_size-1];
    for (j=i; j<strip_size; j++)
        c[j] = c[j-1] + a[j];
}
```

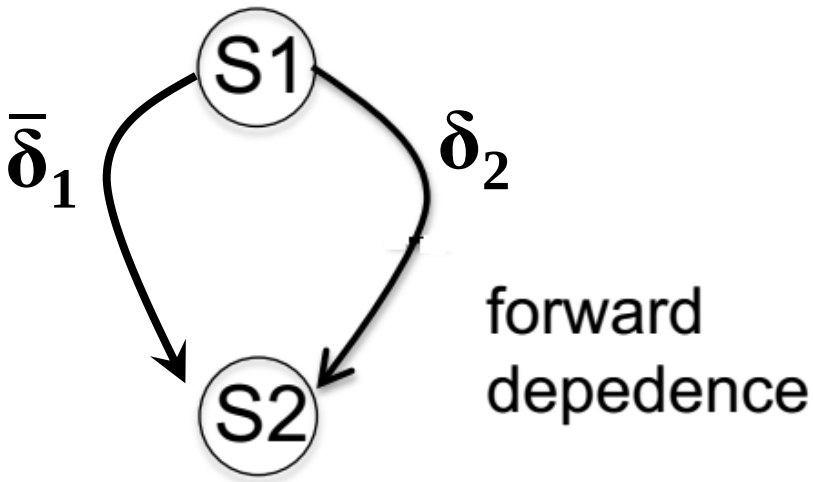
Reordering Statements

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: A[i] = B[i];
    S2: B[i+2] = A[i+1];
}
```

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    S1: B[i+2] = A[i+1];
    S2: A[i] = B[i];
}
```



Not Vectorizable



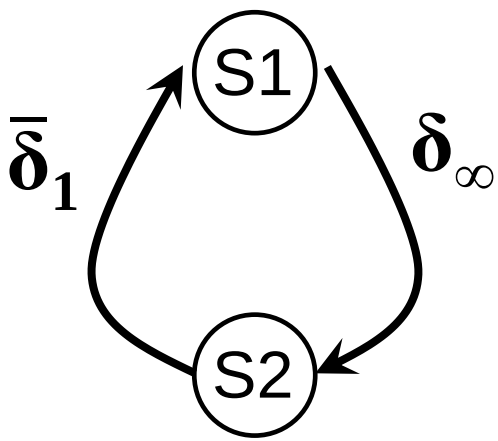
Vectorizable

A reordering transformation enables vectorization of codes with backward dependences within a loop body.

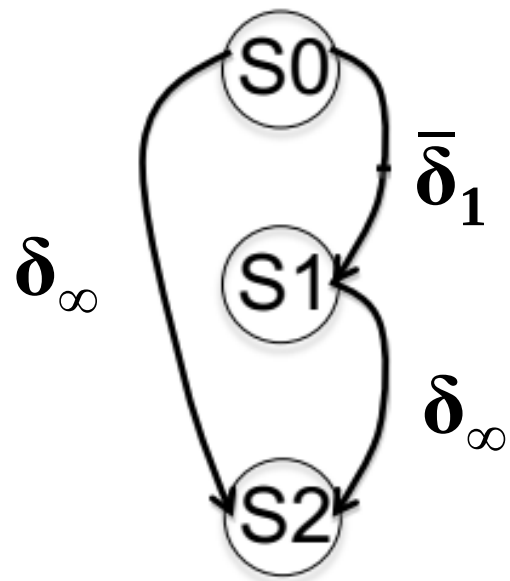
Node Splitting

```
for (i=1; i<n; i++)  {
S1:     a[i] = b[i] + c[i];
S2:     d[i] = a[i]+a[i+1];
}
```

```
for (i=1; i<n; i++)  {
S0:     t[i]  = a[i+1];
S1:     a[i] = b[i] + c[i];
S2:     d[i] = a[i] + t[i];
}
```



Not Vectorizable



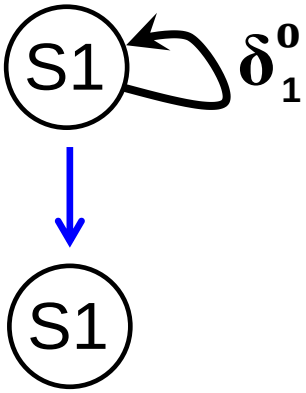
Vectorizable

Scalar Expansion (1/2)

- The idea of scalar expansion is to allocate an array with one element for each iteration and replace a single scalar reference in the loop.
- Scalar expansion can
  - eliminate dependences due to reuse of memory locations.
  - help break the cycle dependence with Write After Write dependences.

```
int  A[N], B[N], i
for (i=0; i<N; i++)
S1:   T = A[i] + B[i];
```

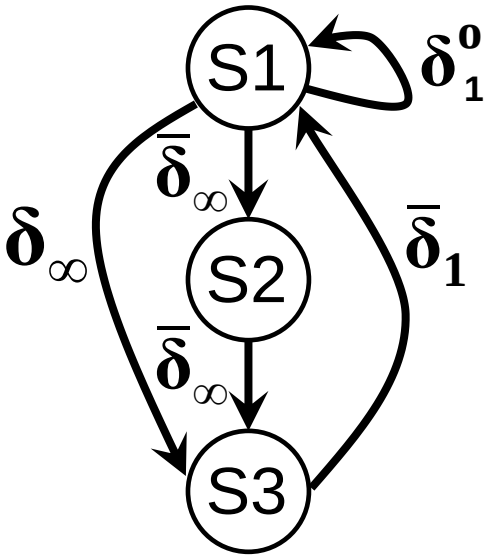
T[i]



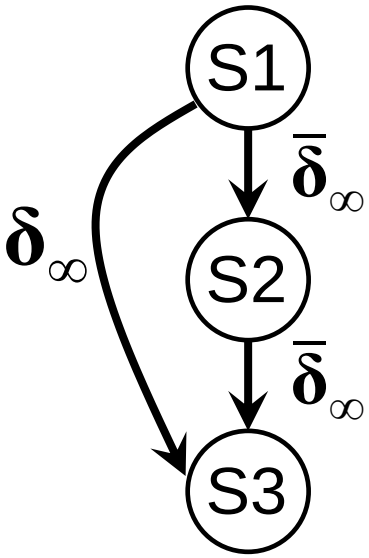
Scalar Expansion (2/2)

```
for (i=1; i<n; i++)  {
S1:        t = a[i];
S2:     a[i] = b[i];
S3:     b[i] = t;
}
```

```
for (i=1; i<n; i++)  {
S1:     t[i]  = a[i];
S2:     a[i] = b[i];
S3:     b[i] = t[i];
}
```



Not Vectorizable

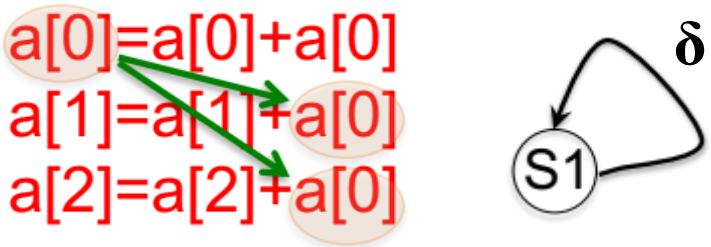


Vectorizable

Loop Peeling

- Loop peeling
  - removes the first or last few iterations from the loop body into separate code outside the loop.
  - is helpful to break loop-carried dependences which only exist for first or last few iterations.

```
for (i=0; i<n; i++){
S1:     a[i] = a[i] + a[0];
}
```




```
a[0] = a[0] + a[0];
for (i=1; i<n; i++)  {
      a[i] = a[i] + a[0];
}
```

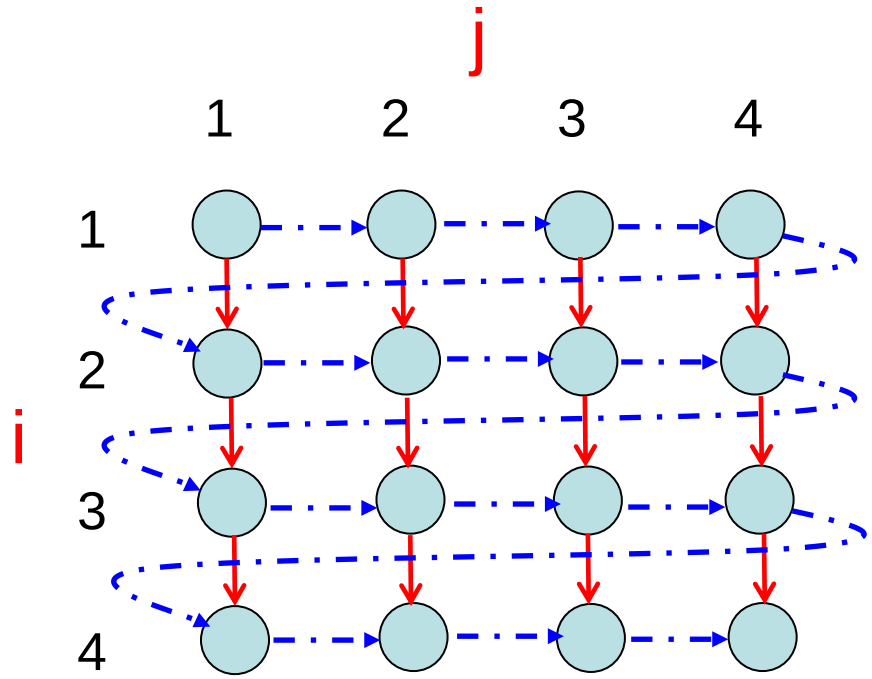
After loop peeling, there are no loop-carried dependences, and the loop can be vectorized.

Freezing Loops

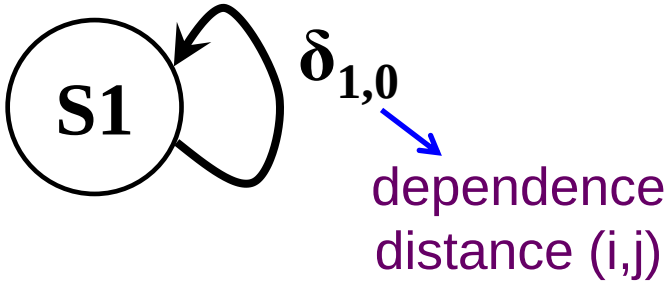
```
for (i=1; i<n; i++)
    for (j=1; j<n; j++)
        a[i][j]=a[i][j]+a[i-1][j];
```

Freezing the outer loop

```
for (i=1; i<n; i++)
    a[i][1:n-1]=a[i][1:n-1] +
                a[i-1][1:n-1];
```



Dependence graph for the loop



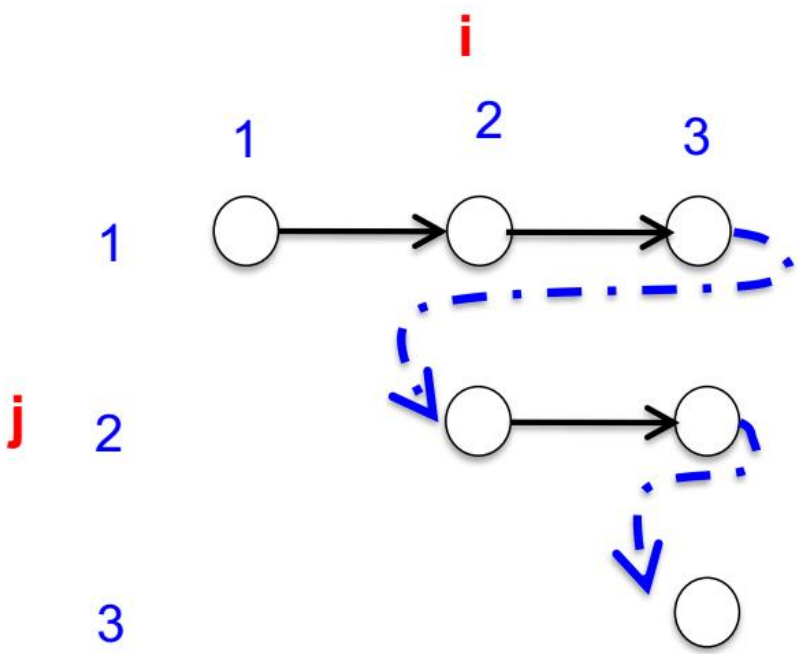
Using inner loop vectorization while freezing the outer loop.

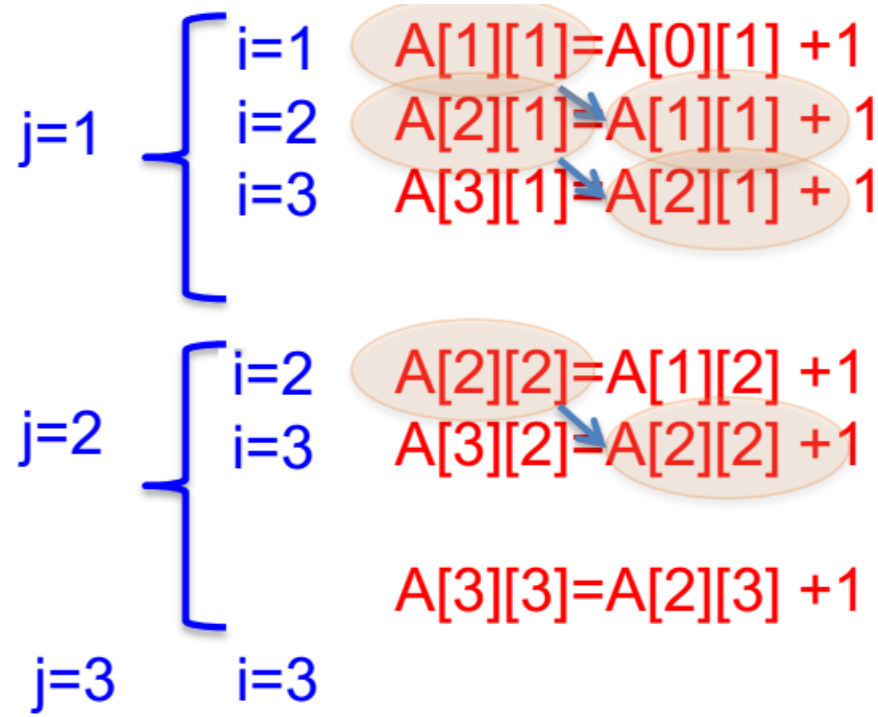
Loop Interchanging (1/3)

- This transformation switches the positions of one loop that is tightly nested within another loop.

```
for (j=1; j<n; j++)
    for (i=j; i<n; i++)
        a[i][j]=a[i-1][j] + 5;
```

note: the array stored in row-major order

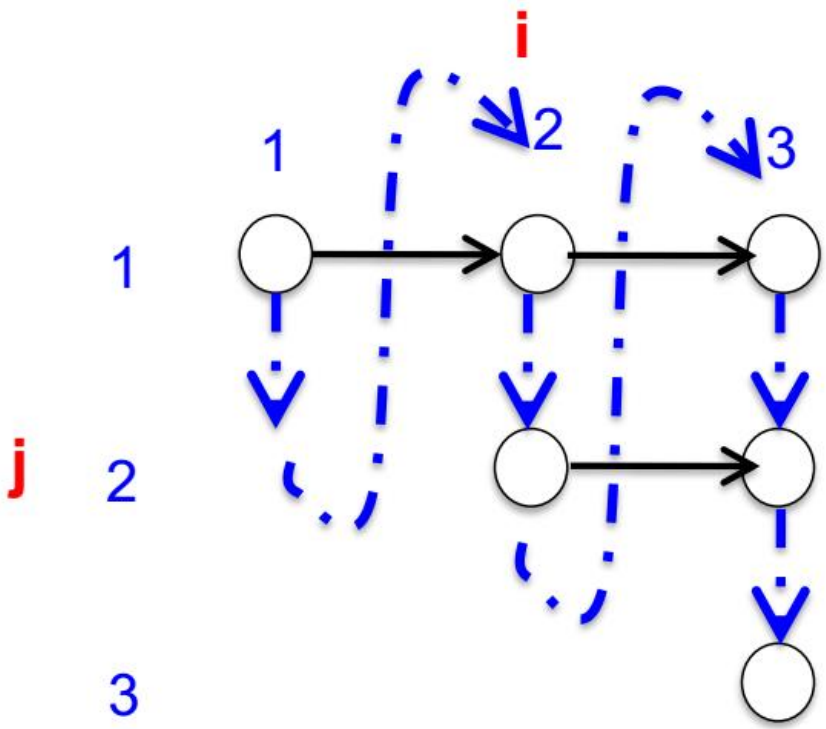


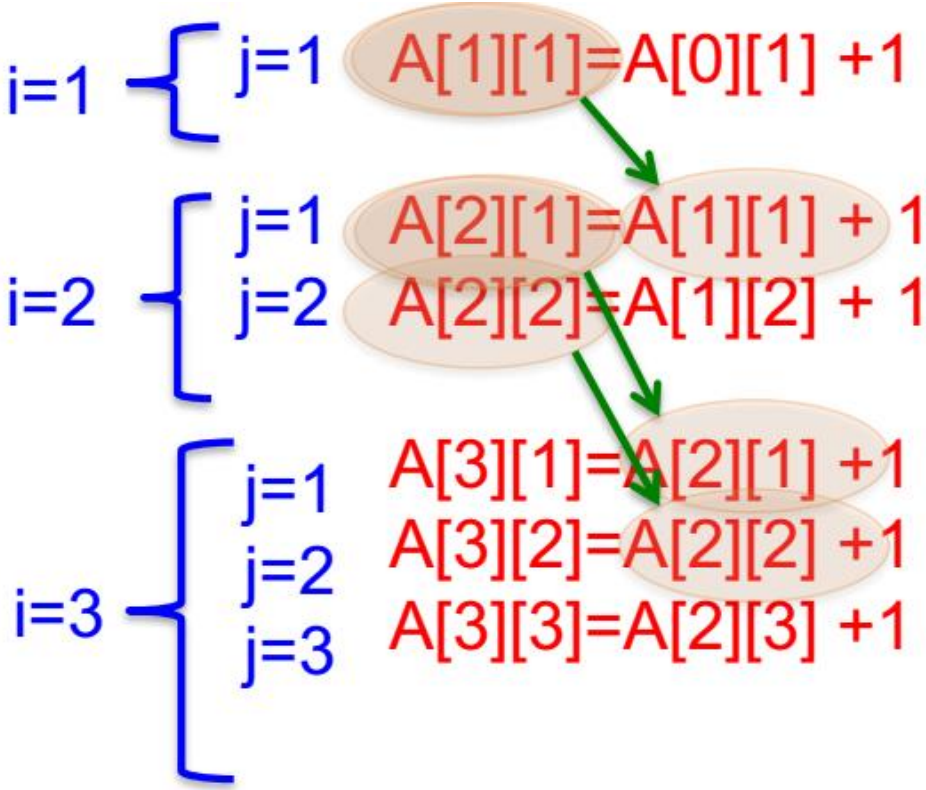


Inner loop cannot be vectorized because of data dependence (Read After Write).

Loop Interchanging (2/3)

```
for (i=1; i<n; i++)
    for (j=1; j<i+1; j++)
        a[i][j]=a[i-1][j] + 5;
```





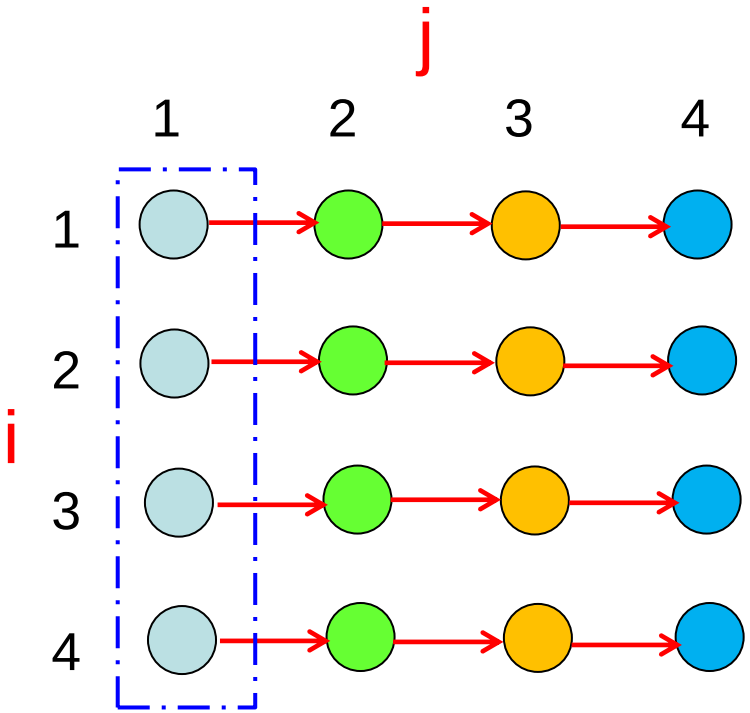
After loop interchanging, no dependences exist in inner loop. The loop can be vectorized by freezing the outer loop.

Loop Interchanging (3/3)

- Loop interchanging is also called outer loop vectorization.

```
for (i=1; i<n; i++)
    for (j=1; j<n; j++)
        a[i][j]=a[i][j]+a[i][j-1];
```

```
for (j=1; j<n; j++)
    for (i=1; i<n; i++)
        a[i][j]=a[i][j]+a[i][j-1];
```



Dependence graph for the loop

Note: In this case, gather and scatter instructions are needed in the target machine to support outer loop vectorization.

Removing Dependences (1/2)

- Counting number of matches in two strings

```
int count(char* string1, char* string2, int size)
{
    int r = 0;
    for (int j = 0; j < size; ++j)    {
        if (string1[j] == string2[j])
            ++r;
    }
    return r;
}
```

Not Vectorizable

- In this case, the vectorization is inhibited due to the presence of a conditional branch in the loop.

Removing Dependences (2/2)

- Counting number of matches in two strings

```
int count(char* string1, char* string2, int size)
{
    int    r = 0;
    bool b;
    for (int j = 0; j < size; ++j)      {
            b = (string1[j] == string2[j]);
            r += b;
    }
    return r;
 }
```

Vectorizable

- The transformation removes the control dependence to generate a vector code.

Take Home Message

- Vectorization is the holy grail of software optimizations, if the hot loop is efficiently vectorized.
- SIMD Advantages
  - available on almost all CPUs
  - the cheapest way to perform parallelization without “warm-up” cost
  - compilers can automatically generate SIMD instructions
- SIMD Drawbacks
  - can efficiently process only simple types (e.g., integers, floats, shorts)
  - memory layout is important
  - cannot vectorize code with loop carried dependences and pointer aliasing
  - management of conditionals

References

- Auto-vectorization in GCC: https://gcc.gnu.org/projects/tree-ssa/vectorization.html
- Auto-vectorization in LLVM: https://llvm.org/docs/Vectorizers.html
- Intel Intrinsics Guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide/
- Practical SIMD Programming: http://www.cs.uu.nl/docs/ vakken/magr/2017-2018/files/SIMD%20Tutorial.pdf
- SIMD for C++ Developers: http://const.me/articles/simd/simd.pdf
- Improving performance with SIMD intrinsics in three use cases: https://stackoverflow.blog/2020/07/08/improving-performance-with-simd-intrinsics-in-three-use-cases/
- Crunching Numbers with AVX and AVX2: https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX
- The ARM scalable vector extension. IEEE micro, 2017
- A sneak peek into SVE and VLA programming: https://developer.arm.com/solutions/hpc/resources/hpc-white-papers/a-sneak-peek-into-sve-and-vla-programming

Appendix A: Removing Dependences

```
for (i=0; i<n; i++){
      a = b[i] + 1;
      c[i] = a + 2;
}
```

Scalar Expansion

```
for (i=0; i<n; i++){
      y[i] = b[i] + 1;
      c[i] = y[i] + 2;
}
a=y[n-1]
```

Loop Distribution

```
y[0:n-1] = b[0:n-1] + 1;
c[0:n-1] = y[0:n-1] + 2;
a=y[n-1]
```

- This example includes a Write After Write data dependence, which can be broken by scalar expansion.
- The basic idea is to allocate an array with one element for each iteration and replace each scalar reference in the loop with a reference to the array.

Predicate-Driven Loop Control (2/4)

- Sum the elements in two integer arrays

```
void sum(int *a, int *b, int *c, int N)
{
        int i;
        for (i = 0; i < N; ++i)
            a[i] = b[i] + c[i];
}
```

Scalar Code

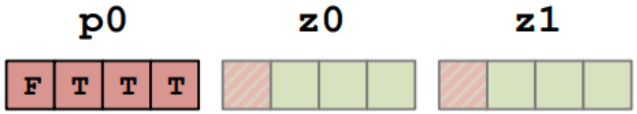
Predicate-Driven Loop Control (3/4)

```
void sum(int *a, int *b, int *c, int N) {
    int i;                                  i+3 ≤ N-1
    for (i = 0; i <= N - 4; i += 4)     {    // vector loop
            __m128i     vb = _mm_loadu_si128(b+i);
            __m128i     vc = _mm_loadu_si128(c+i);
            __m128i     va = _mm_add_epi32(vb,vc);
            _mm_store_si128(a+i,va);
    }
    for (; i < N; ++i)                          // loop tail
        a[i] = b[i] + c[i];
}
```

Vectorized Version (x86/SSE2)

Predicate-Driven Loop Control (4/4)

```
        # x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'i', x4 is 'N'
        mov     x3, 0                          # set 'i=0'
        b       cond                           # branch to 'cond'
loop_body:
        ld1w    z0.s, p0/z, [x1, x3, lsl 2]    # load vector z0 from 'b + i'
        ld1w    z1.s, p0/z, [x2, x3, lsl 2]    # load vector z1 from 'c + i'
        add     z0.s, p0/m, z0.s, z1.s         # add the vectors
        st1w    z0.s, p0, [x0, x3, lsl 2]      # store vector z0 at 'a + i'
        incw    x3                             # increment 'i'
  cond:
```




```
        whilelt p0.s, x3, x4                   # build the loop predicate p0
                                               # p0.s[idx] = (x3 + idx) < x4
        b.first loop_body                      # branch to 'loop_body'
        ret
```

Vectorized Version (ARM/SVE)