

---

# Assembly Language and Microcomputer Interface

## Vectorization

Ming Cai

cm@zju.edu.cn

College of Computer Science and Technology,  
Zhejiang University

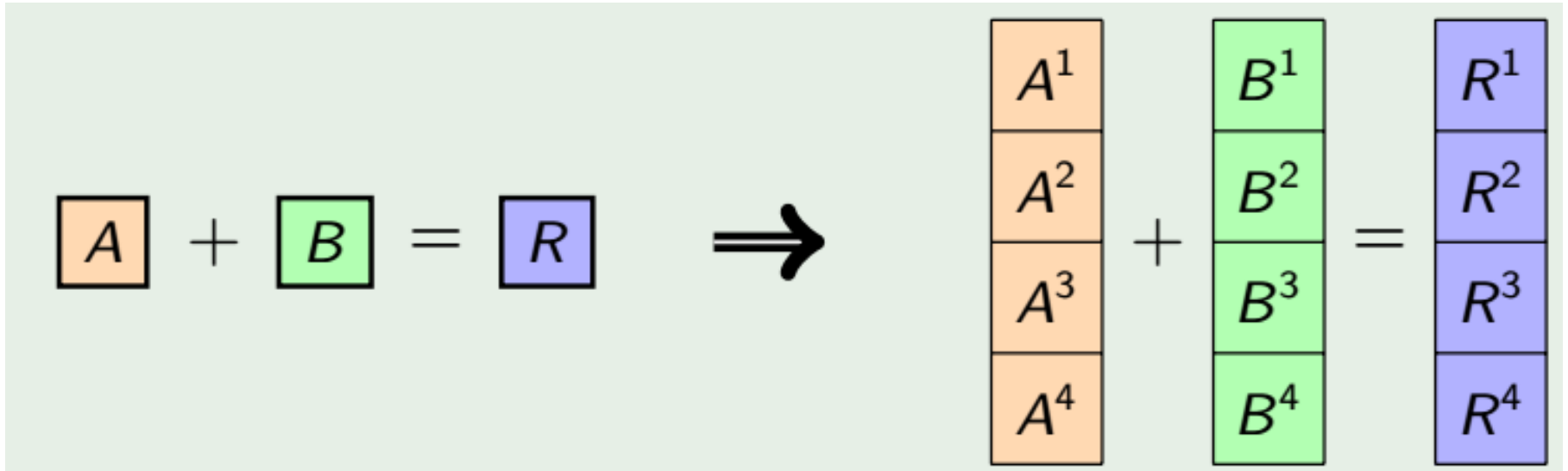
# Outline

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

# 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.

# 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)

# 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

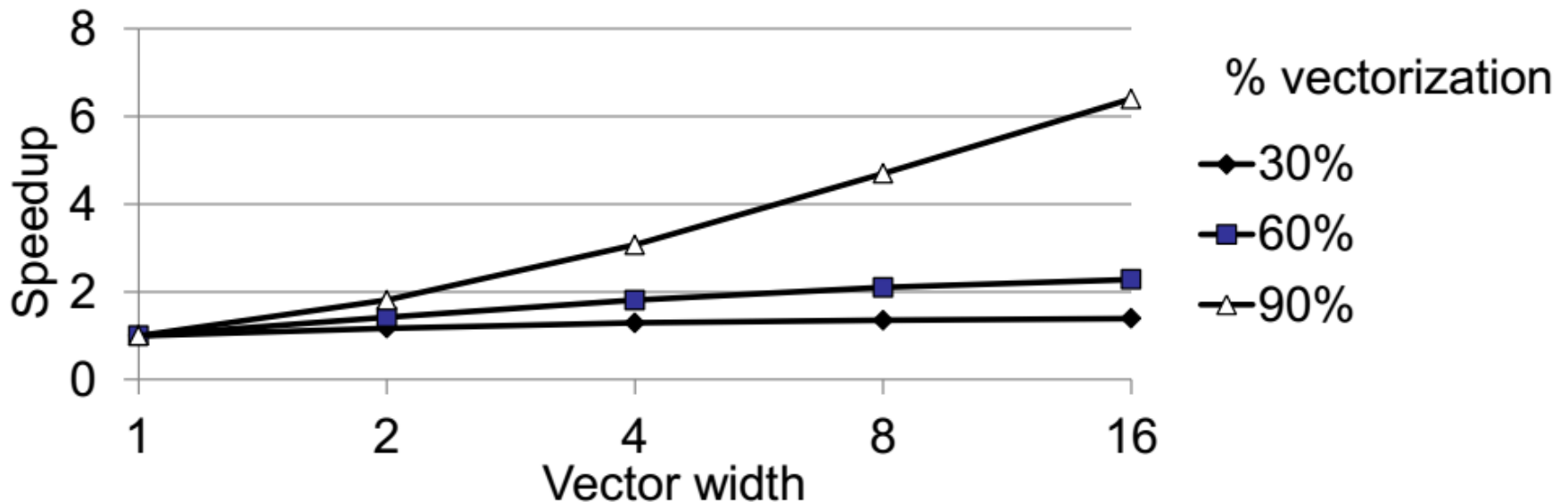
vectorized code

```
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.

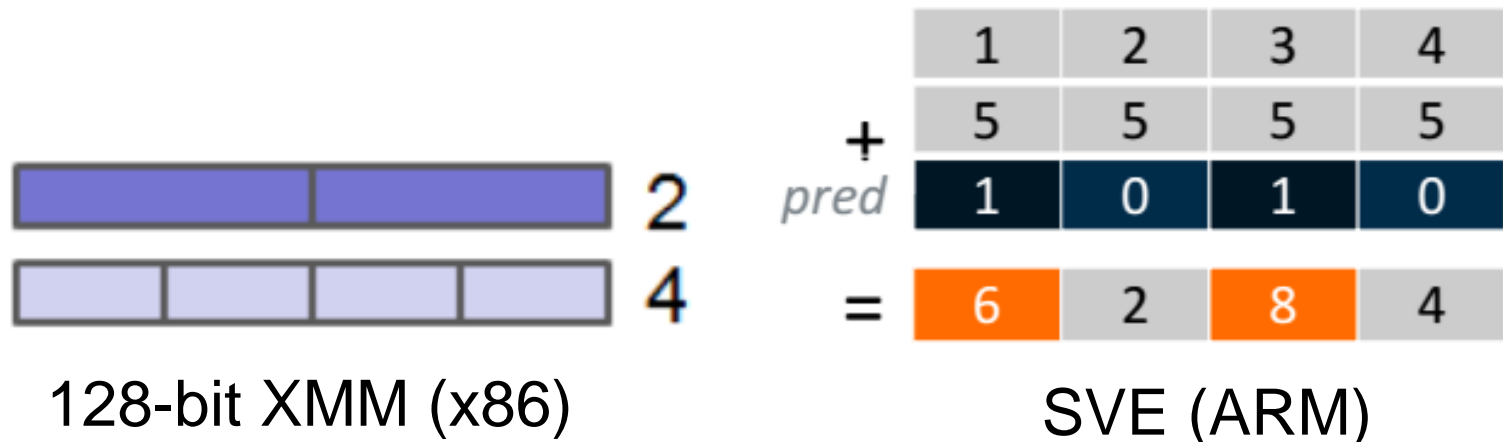


# Vectorization versus Parallelism

- **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.





# 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

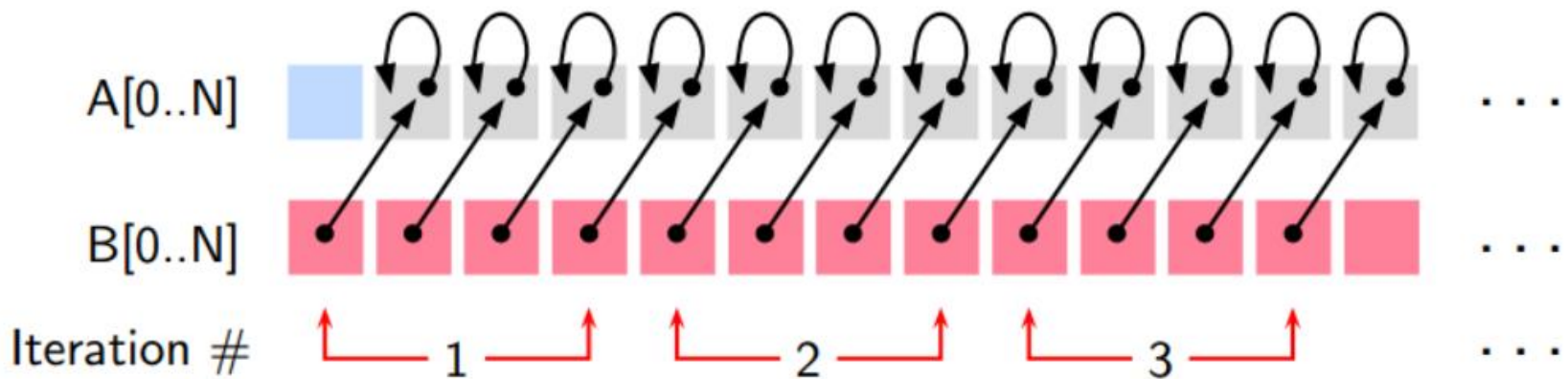
Original Code

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

Vectorized Code

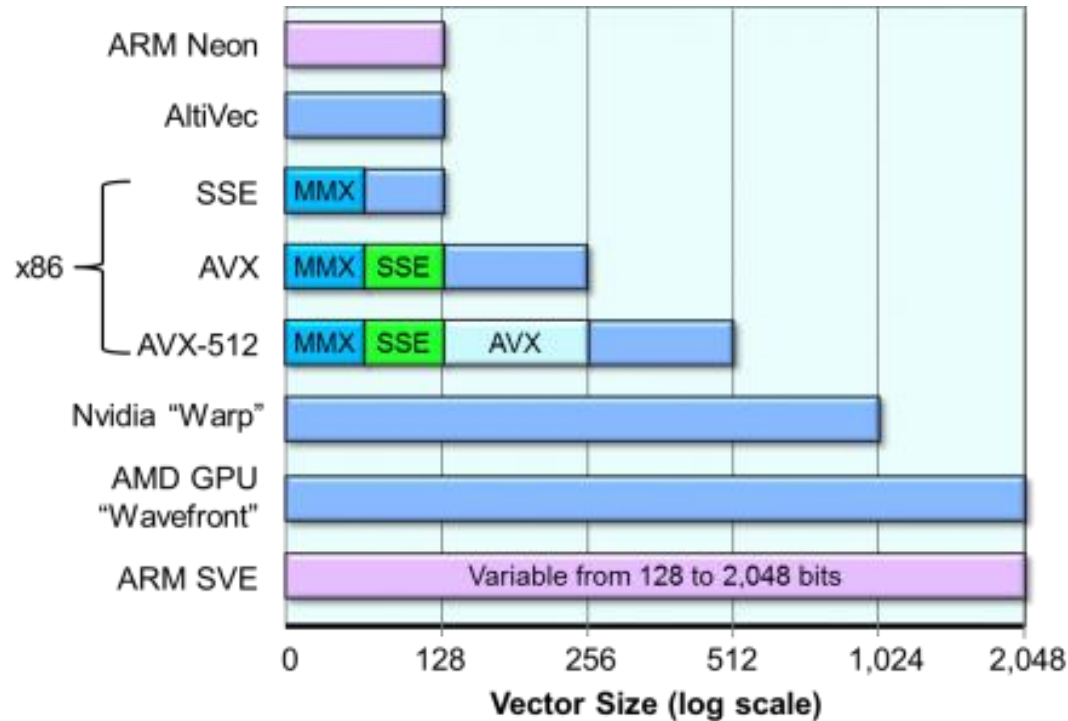
```
int A[N], B[N], i;  
for (i=1; i<N; i=i+4)  
    A[i:i+3] = A[i:i+3] + B[i-1:i+2];
```

Vectorization  
Factor



## 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

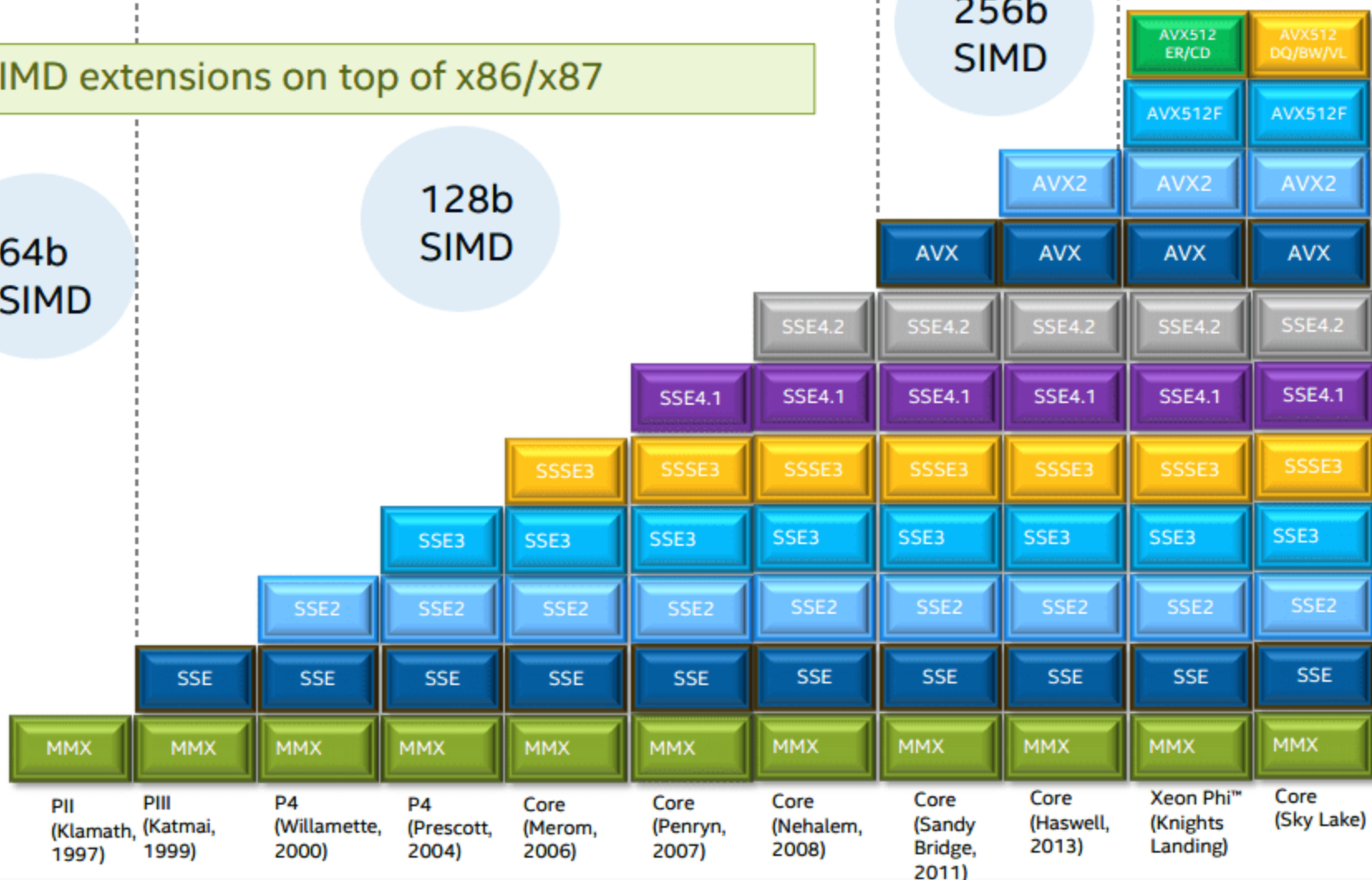
SIMD extensions on top of x86/x87

64b  
SIMD

128b  
SIMD

256b  
SIMD

512b  
SIMD




# Identification of Instruction Set

**CPU-Z**

**CPU** | Caches | Mainboard | Memory | SPD | Graphics | Bench | About

**Processor**

Name	Intel Core i7 7500U				
Code Name	Skylake	Max TDP	15.0 W		
Package	Socket 1515 FCBGA				
Technology	14 nm	Core VID	0.758 V		
Specification	Intel® Core™ i7-7500U CPU @ 2.70GHz				
Family	6	Model	E	Stepping	9
Ext. Family	6	Ext. Model	8E	Revision	B0
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3				

**Clocks (Core #0)**

Core Speed	1994.14 MHz
Multiplier	x 20.0 ( 4 - 35 )
Bus Speed	99.71 MHz
Rated FSB	

**Cache**

L1 Data	2 x 32 KBytes	8-way
L1 Inst.	2 x 32 KBytes	8-way
Level 2	2 x 256 KBytes	4-way
Level 3	4 MBytes	16-way

Selection: Socket #1 | Cores: 2 | Threads: 4

**CPU-Z** Ver. 1.92.2.x64 | Tools | Validate | Close

# How to access the SIMD units?

- **Four choices**
  - C/C++ code and a vectorizing compiler
  - SIMD intrinsics
  - vector library (e.g., xSIMD, VCL)
  - assembly language

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

```
__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)
```



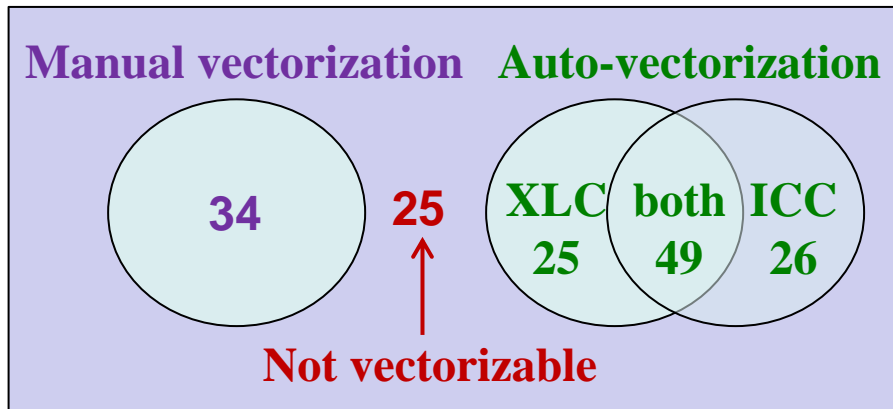
C++ wrappers for SIMD intrinsics

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

# 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)

# Challenge: Loop Dependencies

- Vectorization changes the order of computation compared to sequential case.
- Compiler must be able to prove that vectorization will produce correct result.
- Need to consider independence of unrolled loop operations – depends on vector width.
- Compiler performs **dependency analysis**.



# 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**

# 3. Data Dependencies

- The notion of dependence is the foundation of the process of vectorization.
- It is used to build a calculus of program transformations that can be applied manually by the programmer or automatically by a compiler.

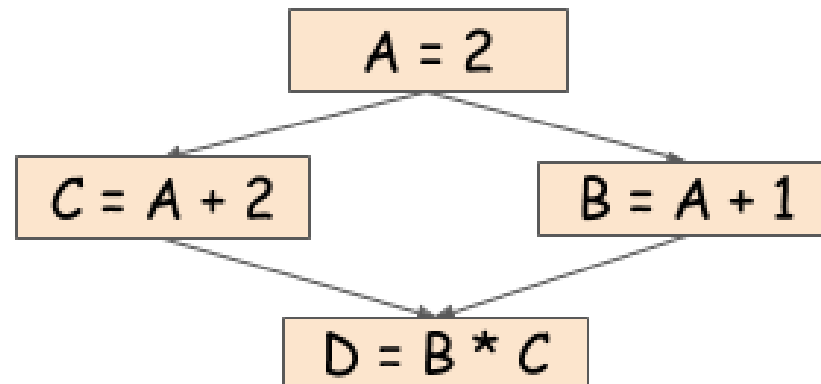
# Definition of Dependence

- A **statement S** is said to be data dependent on **statement T** if
  - T executes before S in the original sequential or scalar program
  - S and T access the same data item
  - At least one of the accesses is a write

code

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

dependence



→ dependence

statement

# Types of Dependences (1/2)

Access in $S_i$	Access in $S_j$	Dependence
Write m	Read m	Flow (true) Dependence
Read m	Write m	Anti (Pseudo) dependence
Write m	Write m	Output (Pseudo) dependence
Read m	Read m	Input Dependence Does not matter

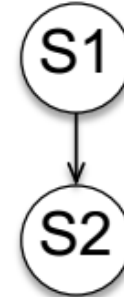
- True dependence cannot be eliminated.
- Pseudo dependences may be eliminated by some transformations.

# Types of Dependences (2/2)

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

S1:  $X = A + B$

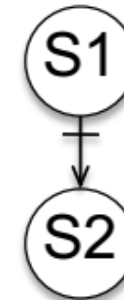
S2:  $C = X + A$



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

S1:  $A = X + B$

S2:  $X = C + D$



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

S1:  $X = A + B$

S2:  $X = C + D$



# No Dependencies Exist

- Vectorization : Yes
- Parallelization: Yes

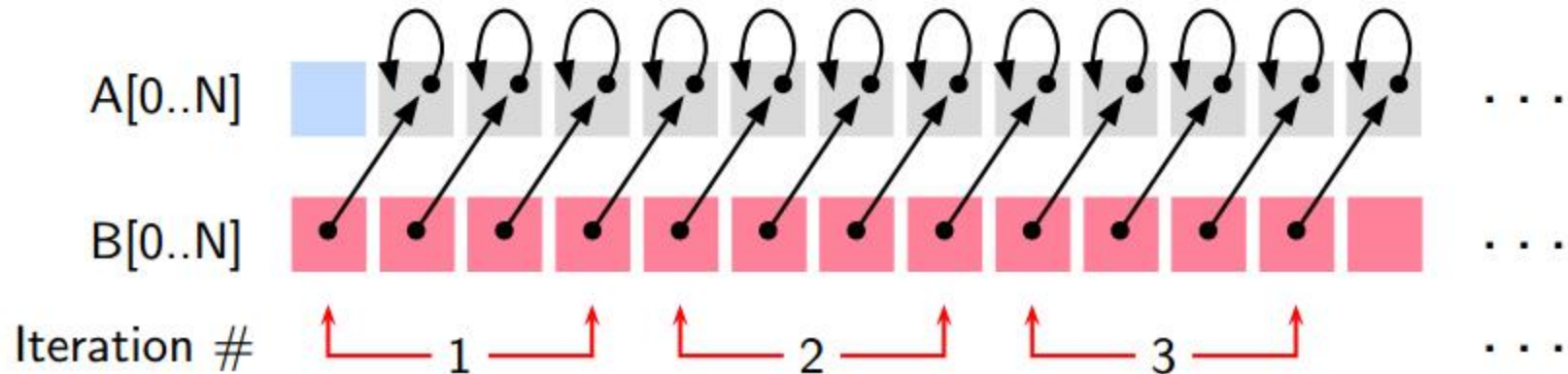
Original Code

```
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];
```

Vectorization  
Factor



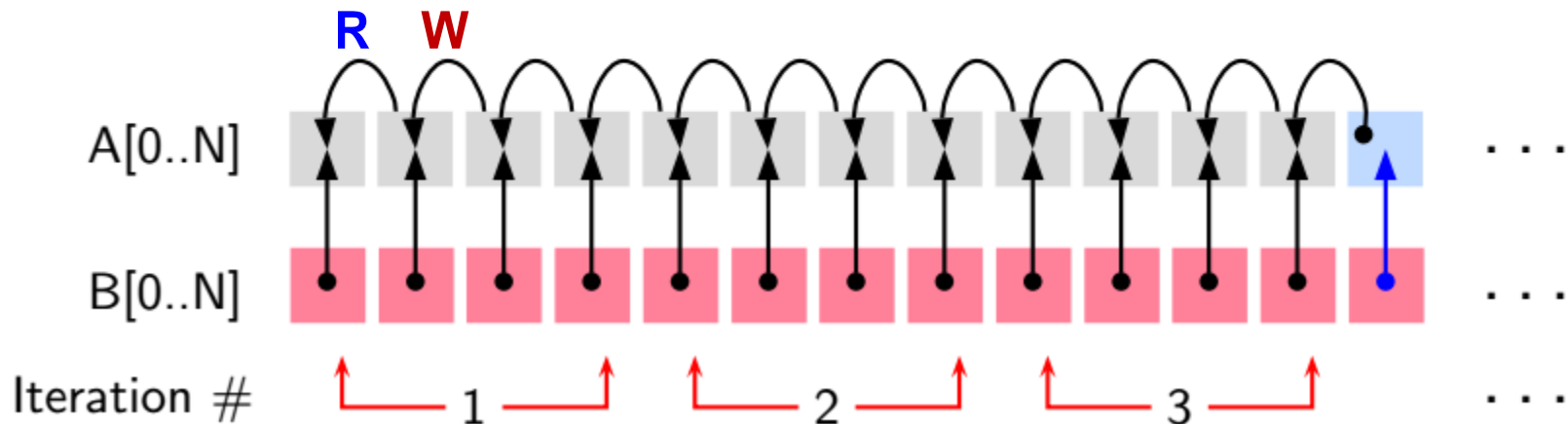
# Dependencies: Write After Read (1/3)

- Vectorization : Yes
- Parallelization: No

## 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



# Dependencies: 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:

$$a[0] = a[1] + b[0] \rightarrow a[0] = 1 + 5 \rightarrow a[0] = 6$$

$$a[1] = a[2] + b[1] \rightarrow a[1] = 2 + 6 \rightarrow a[1] = 8$$

$$a[2] = a[3] + b[2] \rightarrow a[2] = 3 + 7 \rightarrow a[2] = 10$$

$$a[3] = a[4] + b[3] \rightarrow a[3] = 4 + 8 \rightarrow a[3] = 12$$

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



# Dependencies: 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)

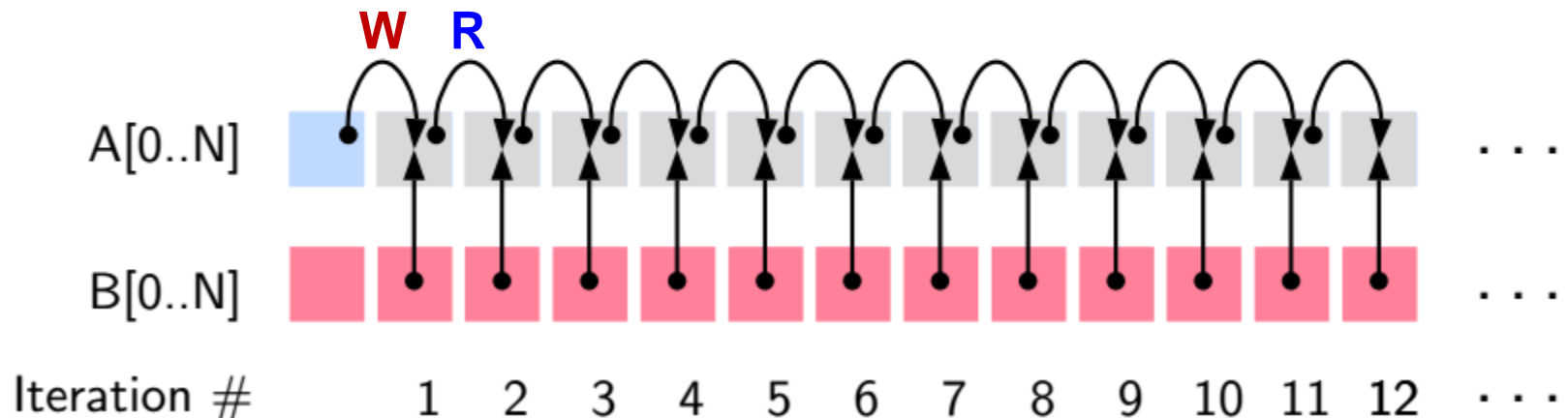
$a = \{6, 8, 10, 12, 4\} = \{6, 8, 10, 12, 4\}$       **Vectorizable**

# Dependencies: Read After Write (1/3)

- Vectorization : No
- Parallelization: No

```
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



# Dependencies: 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:

$$a[1] = a[0] + b[1] \rightarrow a[1] = 0 + 6 \rightarrow a[1] = 6$$

$$a[2] = a[1] + b[2] \rightarrow a[2] = 6 + 7 \rightarrow a[2] = 13$$

$$a[3] = a[2] + b[3] \rightarrow a[3] = 13 + 8 \rightarrow a[3] = 21$$

$$a[4] = a[3] + b[4] \rightarrow a[4] = 21 + 9 \rightarrow a[4] = 30$$

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

# Dependencies: 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)

$A = \{0, 6, 8, 10, 12\} \neq \{0, 6, 13, 21, 30\}$

**Not Vectorizable**

# Dependencies: Write After Write

- Vectorization : No
- Parallelization: No

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

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.

# Dependencies: 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  
dependency here.

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

# Data Dependencies

- Read After Write
  - Not vectorizable
  - Not parallelizable
- Write after Read
  - Vectorizable
  - Not parallelizable
- Read After Read
  - Vectorizable
  - Parallelizable
- Write after Write
  - Not Vectorizable
  - Not parallelizable
- If parallelization is possible then vectorization is trivially possible.
- Vectorization does not imply parallelization.

# 4. Auto-vectorization

- **Main issues with auto-vectorization**
  - Aliasing, alignment, data dependencies, 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 dependencies



# 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, -xAVX2 -xAVX2 on icc
- For avx512 on gcc/clang : -march=skylake-avx512
- For avx512 on icc: -xCORE-AVX512
- For optimal vectorization depending on your CPU: -march=native on gcc/clang, -xHOST on icc

# Visualizing the Auto-vectorization



Add... More...

Sponsors **intel** **PC-lint** **Solid Sands**

Share Policies

intro-sum.c

x86-64 gcc 11.2 (C, Editor #1, Compiler #1)

A Save/Load + Add new... Vim

C

x86-64 gcc 11.2 -O3

A Output... Filter... Libraries + Add new... Add tool...

```
1 int Sum1ToN(int n){
2     int sum = 0;
3     for (int i = 0; i < n; i++){
4         sum +=i;
5     }
6     return sum;
7 }
8
9 int Sum(int n){
10     int sum = 0,a[n],b[n];
11
12     for (int i = 0; i < n; i++){
13         sum += a[i];
14         sum += b[i];
15     }
16     return sum;
17 }
18
```

```
101     shr     edi, 2
102     sal     rdi, 4
103 .L15:
104     movdqu  xmm2, XMMWORD PTR [rsi+rax]
105     movdqu  xmm3, XMMWORD PTR [rcx+rax]
106     add     rax, 16
107     padd    xmm0, xmm2
108     padd    xmm0, xmm3
109     cmp     rax, rdi
110     jne     .L15
111     movdqa  xmm1, xmm0
112     mov     edi, edx
113     psrldq  xmm1, 8
114     and     edi, -4
115     padd    xmm0, xmm1
116     movdqa  xmm1, xmm0
117     psrldq  xmm1, 4
118     padd    xmm0, xmm1
119     movd    eax, xmm0
120     test    dl, 3
121     je     .L12
122 .L14:
123     movsx   r8, edi
124     sal     r8, 2
125     add     rsi, r8
126     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 dependencies in the loop could prevent vectorization.
- Non-contiguous memory 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

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

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

# Pointer Aliasing (1/4)

- Two seemingly different pointers may point to storage locations in the same array (**aliasing**).
- Data dependencies 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 dependencies (overlaps) are possible. This is difficult to do when using pointers.

# Pointer Aliasing (2/4)

- 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 (3/4)

- If we invoked it as follows:

`compute(p, p-1, q);`

- We could substitute these values and construct an equivalent loop:

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

There is a read after  
write dependency.



**Not Vectorizable**

# Pointer Aliasing (4/4)

- We can add directives to guide the compiler.
- C99 introduced “restrict” keyword to inform the compiler that there is no memory overlap (aliasing).

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

# 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.

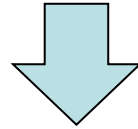
```
void func1(float *a, float *b, float *c) {  
    __assume_aligned(a, 16);  
    __assume_aligned(b, 16);  
    __assume_aligned(c, 16);  
    for (int i=0; i<LEN; i++)  
        a[i] = b[i] + c[i];  
}
```

# 5. Transformations for Vectorization

- When data dependencies are present, vectorization can be achieved by:
  - Separating (distributing) the statements
  - Freezing loops
  - Strip mining
  - Removing dependences
  - Changing the algorithm

# Distributing

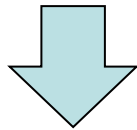
```
for (i=1; i<n; i++){  
    b[i] = b[i] + c[i];  
    a[i] = a[i-1]*a[i-2]+b[i];  
    c[i] = a[i] + 1;  
}
```



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

# Freezing Loops

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



Ignoring (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];  
}
```

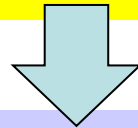
# Strip Mining (1/2)

- Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling vectorization and improving memory locality.
- By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure in two ways:
  - reduces the number of iterations of the loop by a vectorization factor (VF)
  - increases the temporal and spatial locality in the data cache.



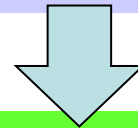
# Strip Mining (2/2)

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



**Distributing**

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



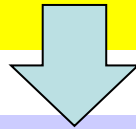
**Strip mining**

```
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];  
    }  
}
```

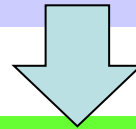
Loop distribution  
increases the  
cache miss ratio

# Removing Dependences

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



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



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

# Changing the Algorithm (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

# Changing the Algorithm (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

# 6. 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
<b>Integers</b>	<b>__m128i</b>	<b>__m256i</b>	<b>__m512i</b>
<b>16-bit float</b>	<b>__m128h</b>	<b>__m256h</b>	<b>__m512h</b>
<b>32-bit float</b>	<b>__m128</b>	<b>__m256</b>	<b>__m512</b>
<b>64-bit float</b>	<b>__m128d</b>	<b>__m256d</b>	<b>__m512d</b>

- 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

intrinsic: `_mm_add_ps (__m128 a, __m128 b)` `_mm256_sqrt_pd (__m256d a)`



instruction: `addps xmm, xmm`



`vsqrtpd ymm, ymm`

# **7. Examples with Intrinsic**

**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

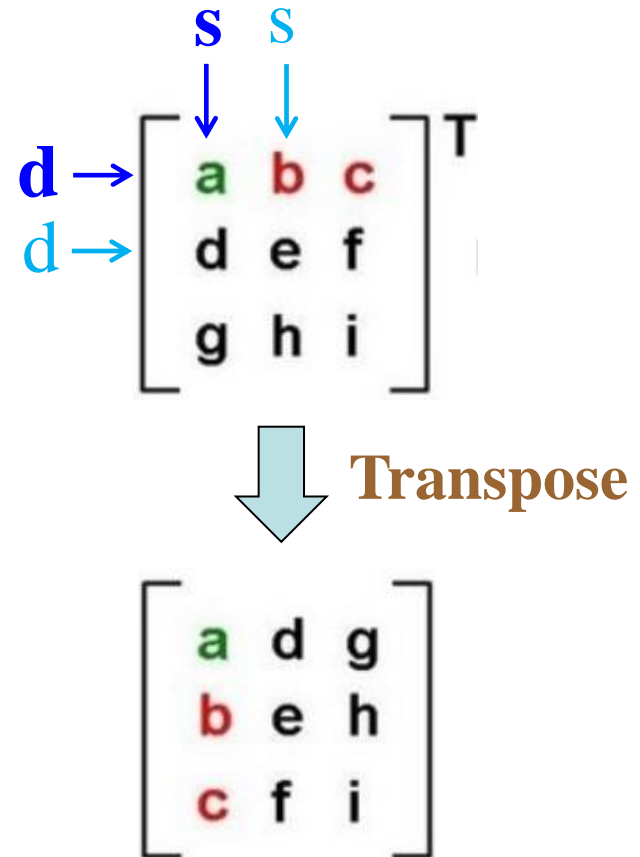
```
float dotProduct( float* p1, float* p2, int count ) {  
    assert( 0 == count % 4 );  
    __m128 acc = _mm_setzero_ps();           //Return vector with zero  
    float* const p1End = p1 + count;  
    for( ; p1 < p1End; p1 += 4, p2 += 4 )    {  
        // Load 2 vectors, 4 floats / each  
        const __m128 a = _mm_loadu_ps( p1 );  
        const __m128 b = _mm_loadu_ps( p2 );  
        // 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  
}
```

using SSE Intrinsics

# Matrix Transpose——Scalar Code

```
void transpose(double *x, double *y, int n)
```

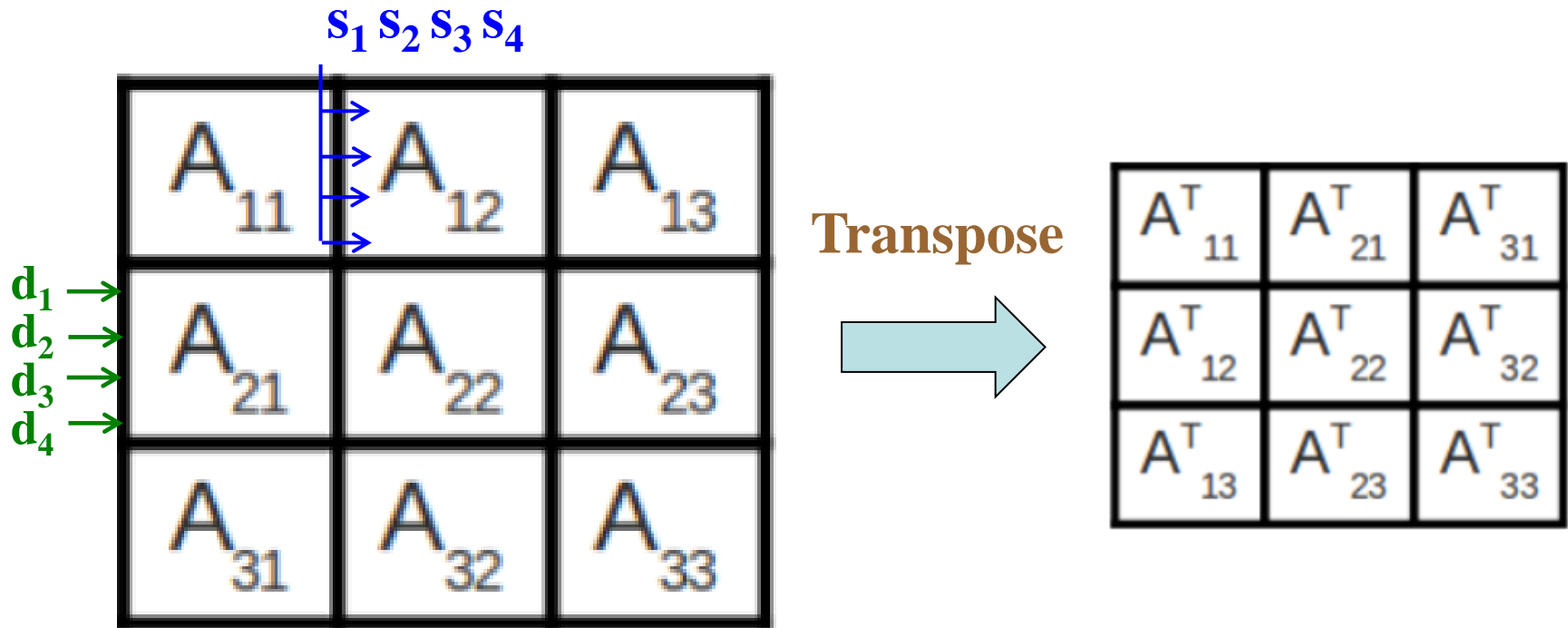
```
{  
    double *s=x,*d=y;  
    for(int i=0;i<n;i++) {  
        d=y+i;  
        for(int j=0;j<n;j++) {  
            *d=*(s++);  
            d+=n;  
        }  
    }  
}
```



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

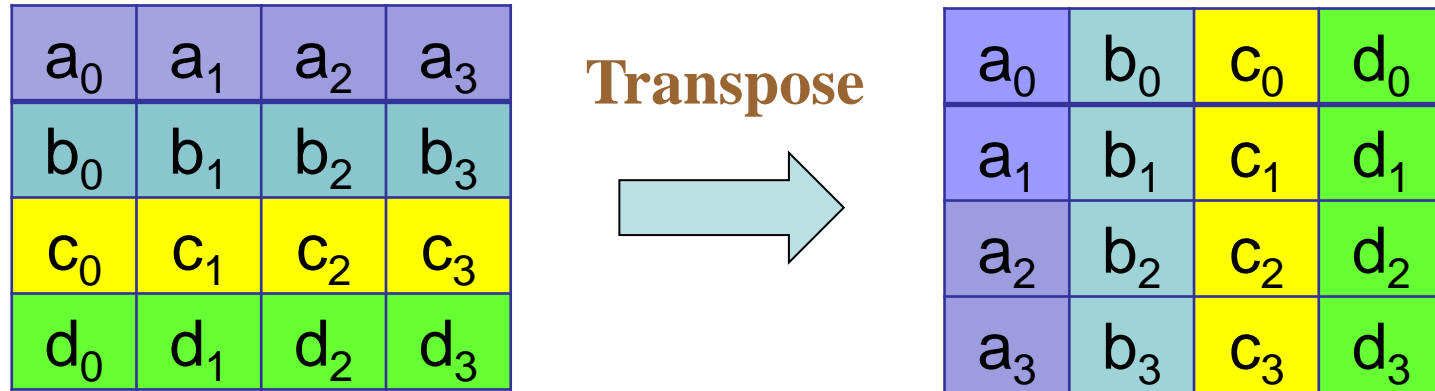
# Matrix Transpose——Vectorized Version

- 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



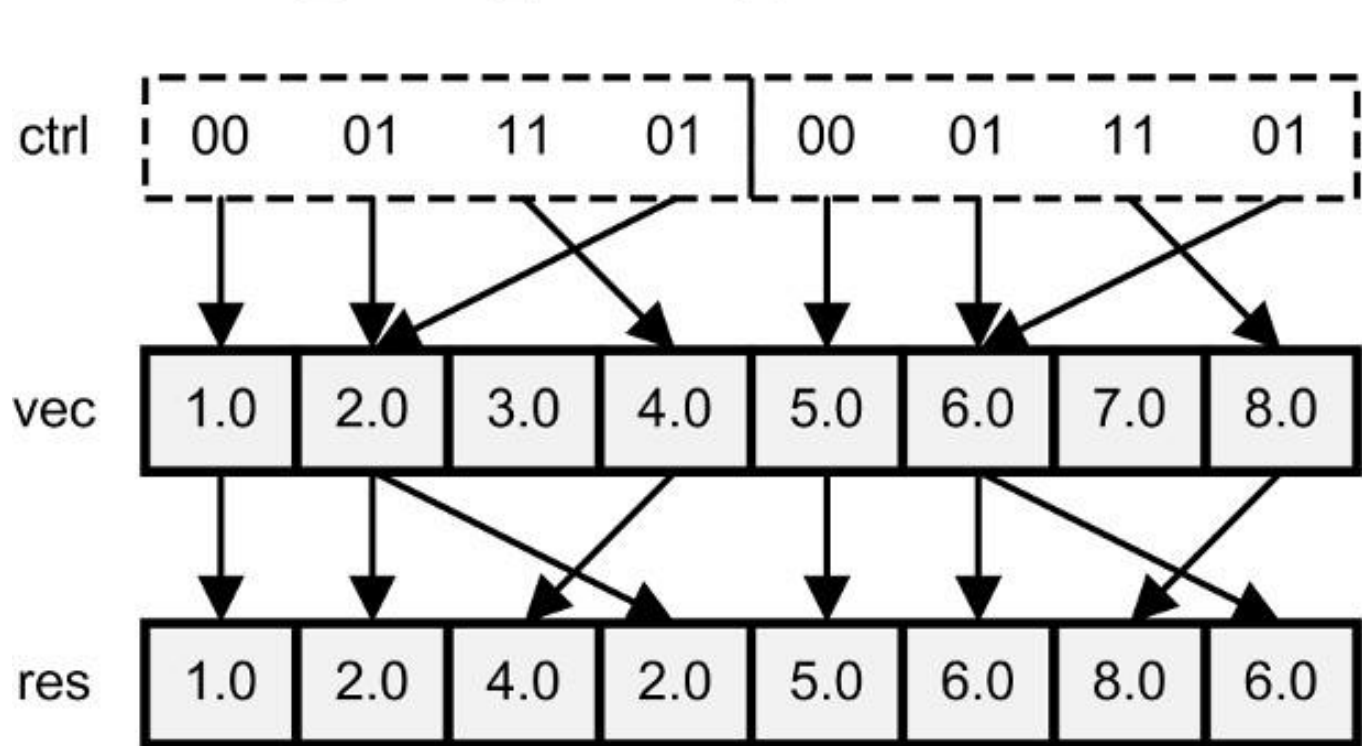
- Transpose of a  $4 \times 4$  matrix
  - **Permute instruction**: `_mm256_permute4x64_pd`
  - **Shuffle instruction**: `_mm256_blend_ps`
  - **Unpack instructions**: `_mm256_unpacklo_pd`, `_mm256_unpackhi_ps`



# Matrix Transpose——Vectorized Version

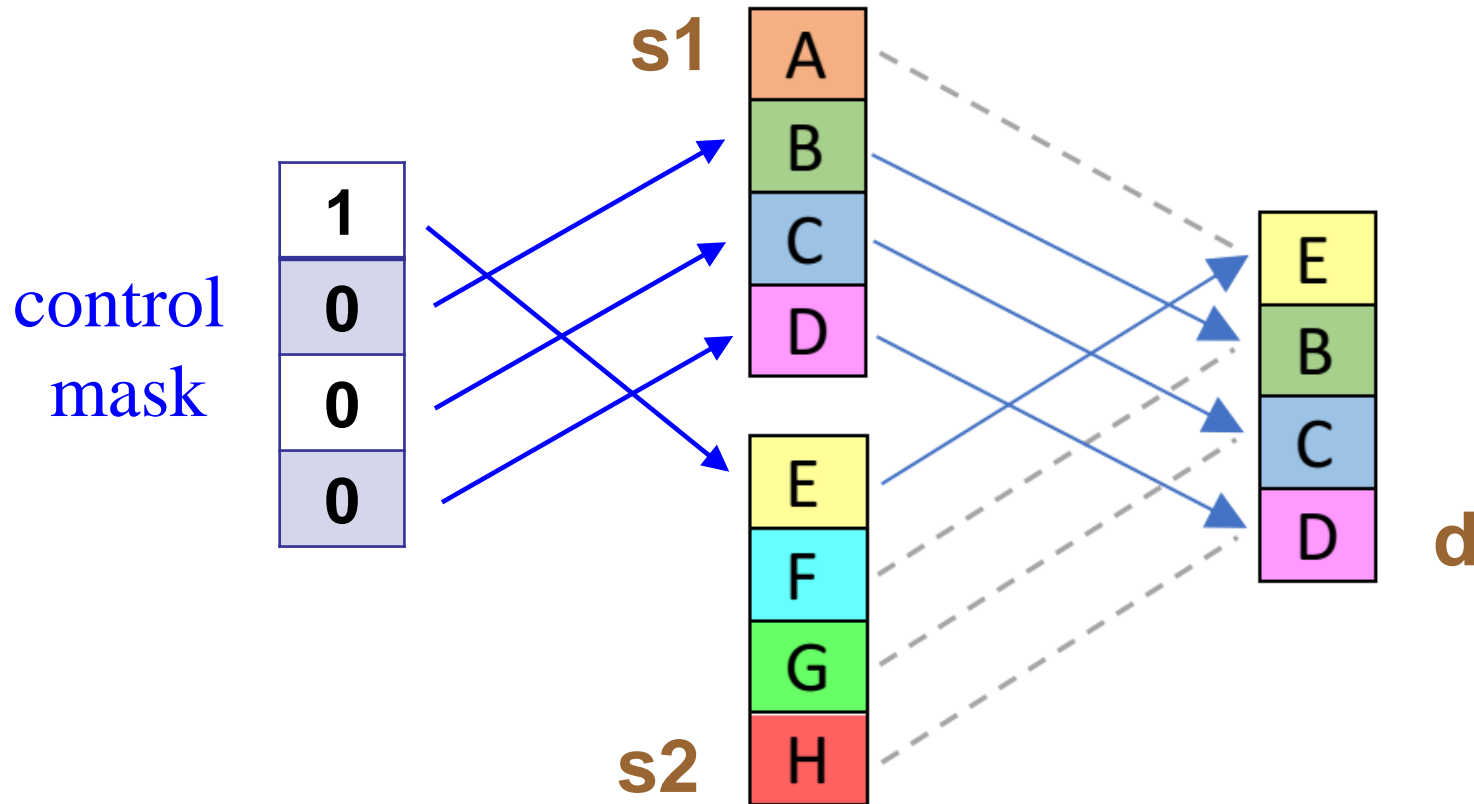
- **Permute instructions** provide intrinsics that return a vector containing the rearranged elements of a vector.

**control mask** `res = _mm256_permute_ps(vec, 0b01110100)`



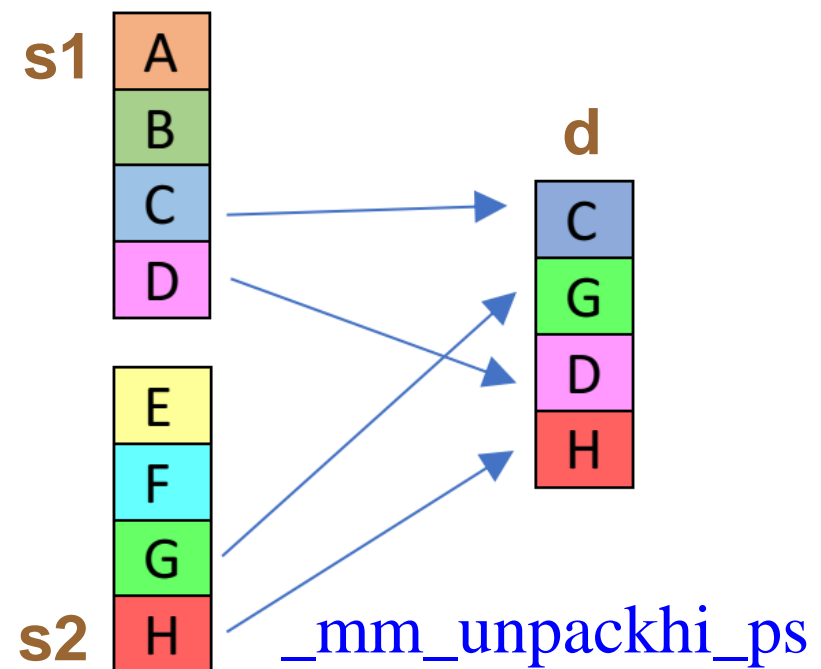
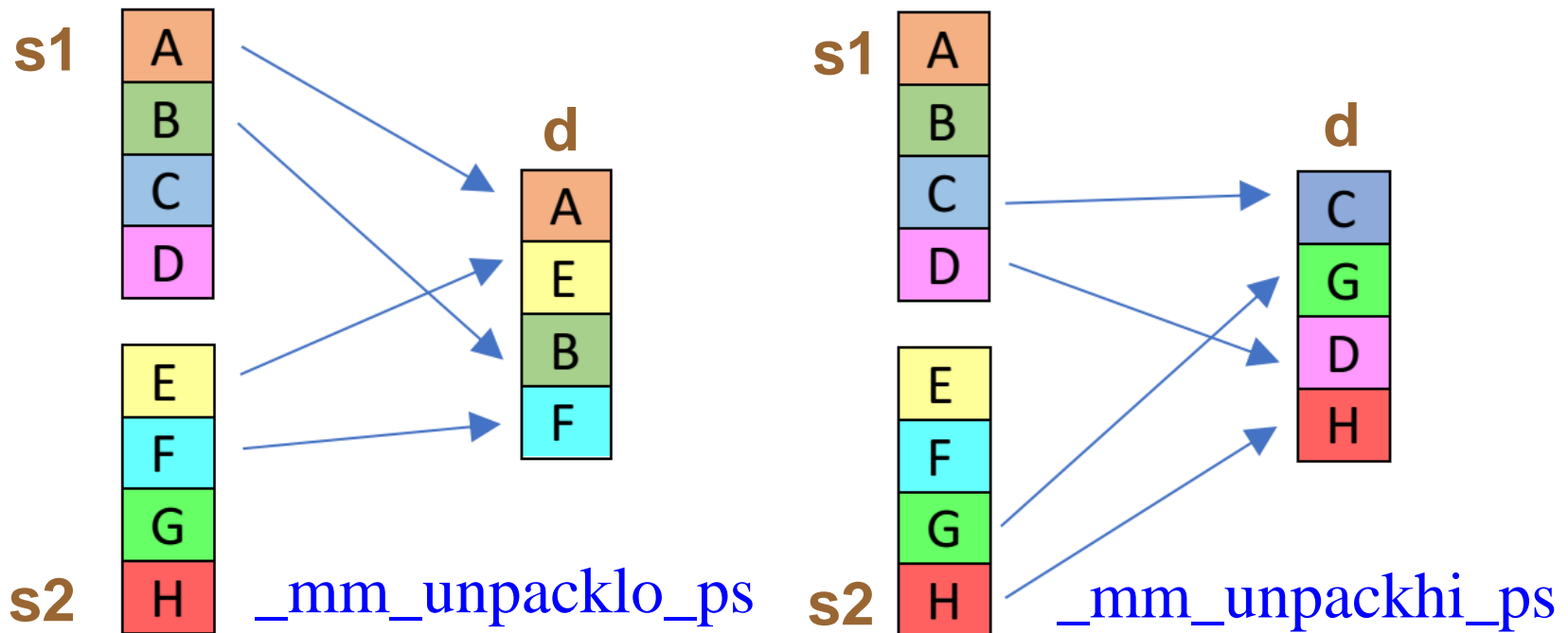
# Matrix Transpose——Vectorized Version

- **Shuffle intrinsics** select elements from two input vectors and place them in the output vector.
- For example,  $d = \text{\_mm\_blend\_ps}(s1, s2, 0001B)$



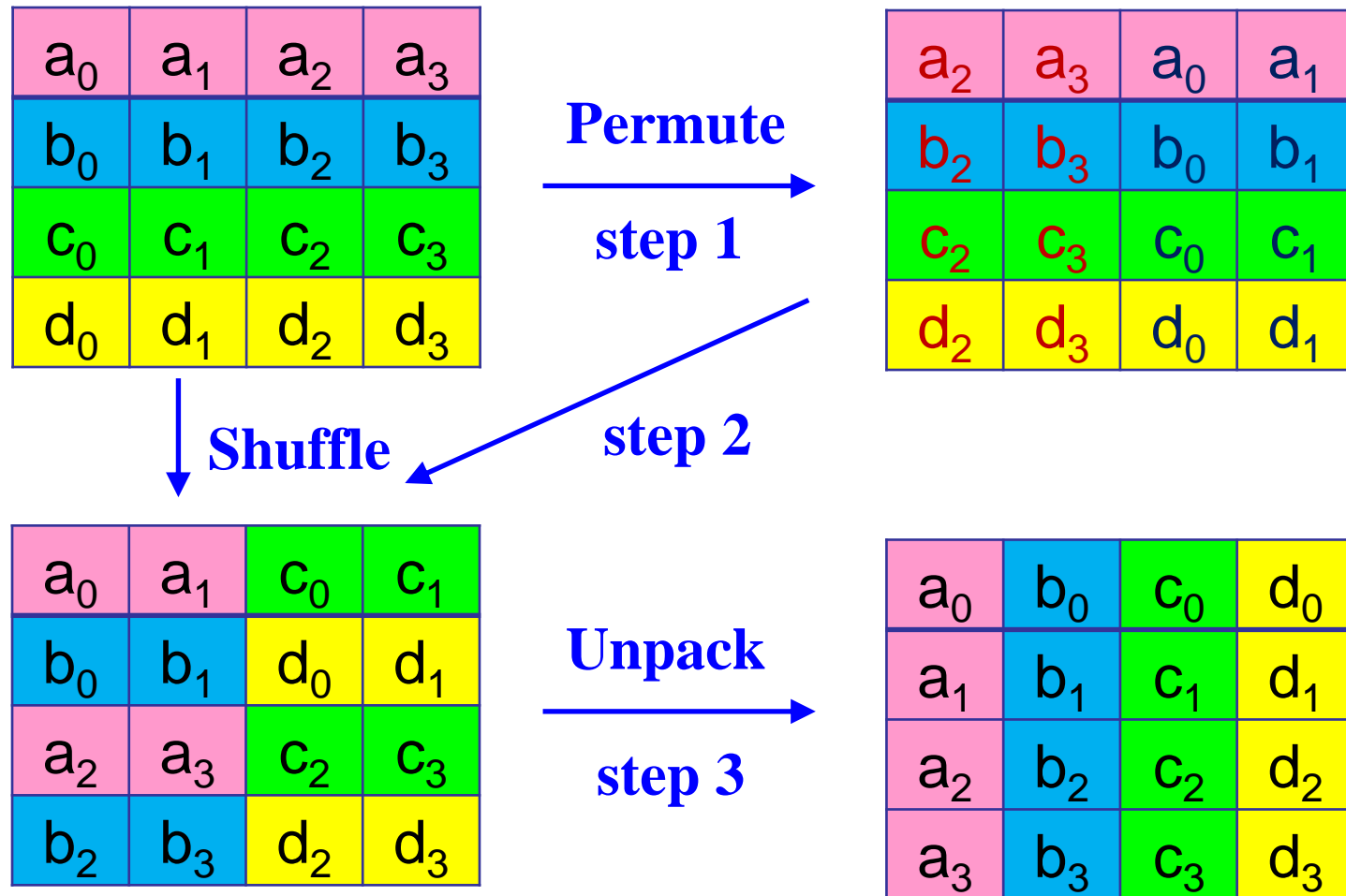
# Matrix Transpose——Vectorized Version

- **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

- Transpose of a  $4 \times 4$  matrix



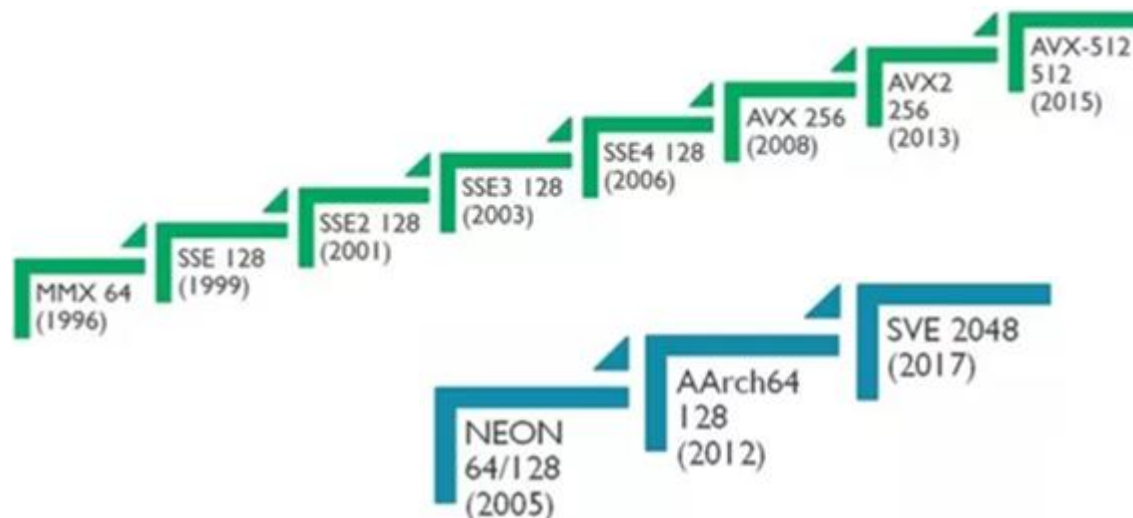
# Matrix Transpose——Vectorized Version

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
}
```

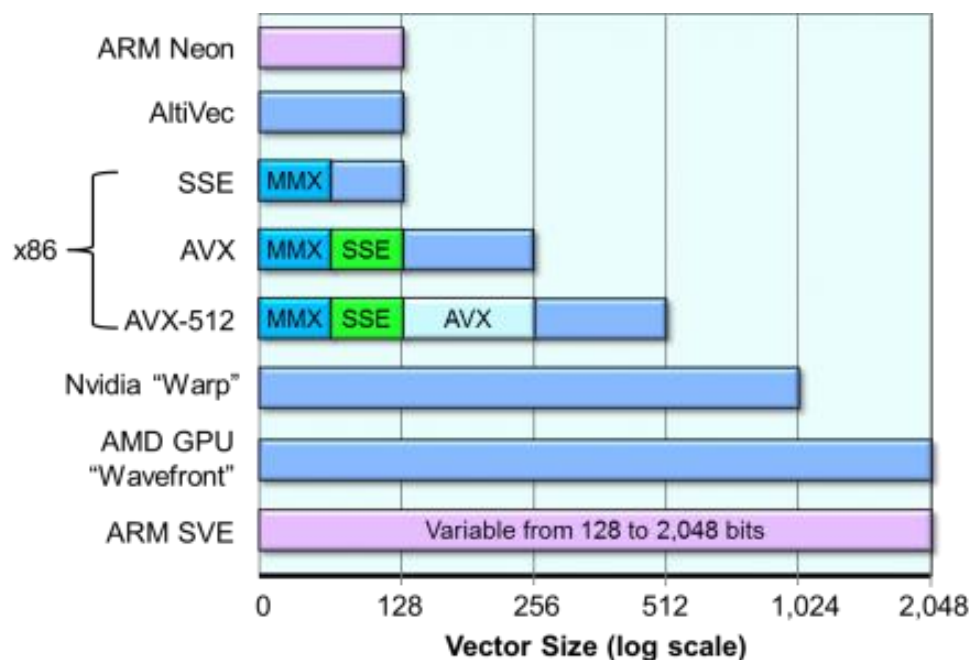
<https://www.cxyzjd.com/article/artorias123/90513600>

# 8. Overview of ARM SVE



Intel SIMD

ARM SIMD



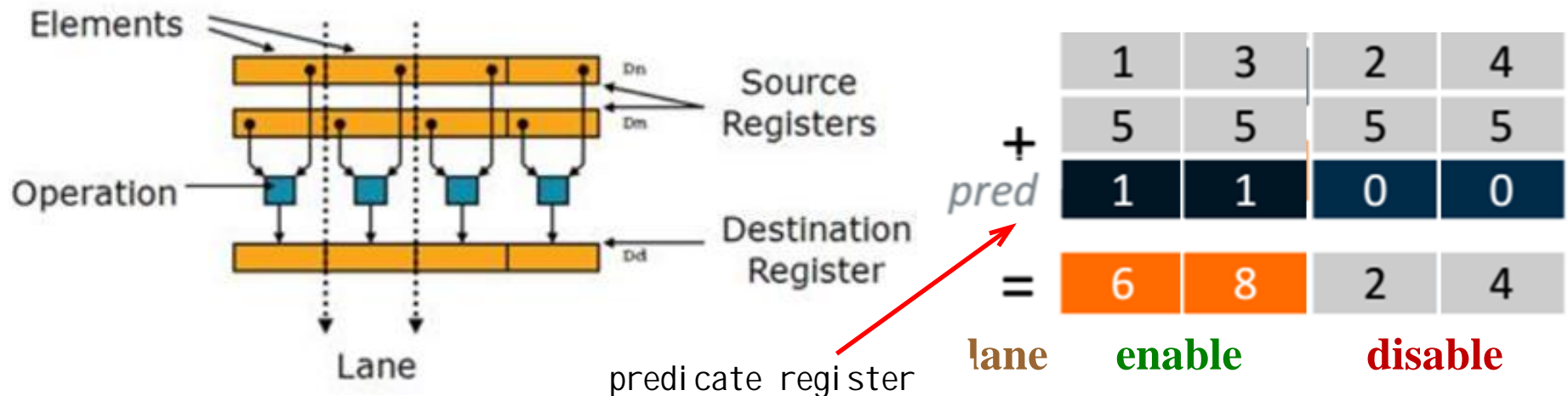
ARM NEON

X86  
(MMX/SSE/AVX)

ARM SVE

# ARM SVE (Scalable Vector Extension)

- SVE is vector length agnostic (VLA)
  - Vector length (VL) is a hardware implementation choice from 128 up to 2048 bits.
  - SVE operates on individual lanes of vector controlled by of a governing **predicate register**.
  - SVE eliminates loop heads and tails and other overhead by processing partial vectors.



# SVE versus Traditional Methods

- For example, how do we compute data which has ten chunks of 4-bytes?
- **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



16-byte

4-byte



16-byte



# Vector Length Agnostic (VLA) Programming (1/3)

- 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

# Vector Length Agnostic (VLA) Programming (2/3)

```
void sum(int *a, int *b, int *c, int N) {  
    int i;  
    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)

# Vector Length Agnostic (VLA) Programming (3/3)

# 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)

# References

- [Intel Intrinsics Guide](https://software.intel.com/sites/landingpage/IntrinsicsGuide/): <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>
- [Practical SIMD Programming](http://www.cs.uu.nl/docs/vakken/magr/2017-2018/files/SIMD%20Tutorial.pdf): <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): <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/): <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-AVX2): <https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX2>
- [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): <https://developer.arm.com/solutions/hpc/resources/hpc-white-papers/a-sneak-peek-into-sve-and-vla-programming>