

SIMD Programming

Moreno Marzolla
Dip. di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna

Copyright © 2017–2024
Moreno Marzolla, Università di Bologna, Italy
<https://www.moreno.marzolla.name/teaching/HPC/>



Except where stated otherwise, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0). To view a copy of this license, visit <https://creativecommons.org/licenses/by-sa/4.0/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

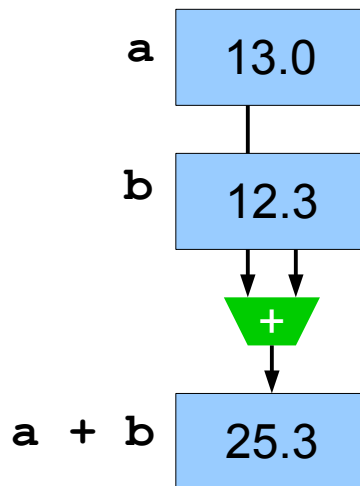
Credits

- Marat Dukhan (Georgia Tech)
- Salvatore Orlando (Univ. Ca' Foscari di Venezia)

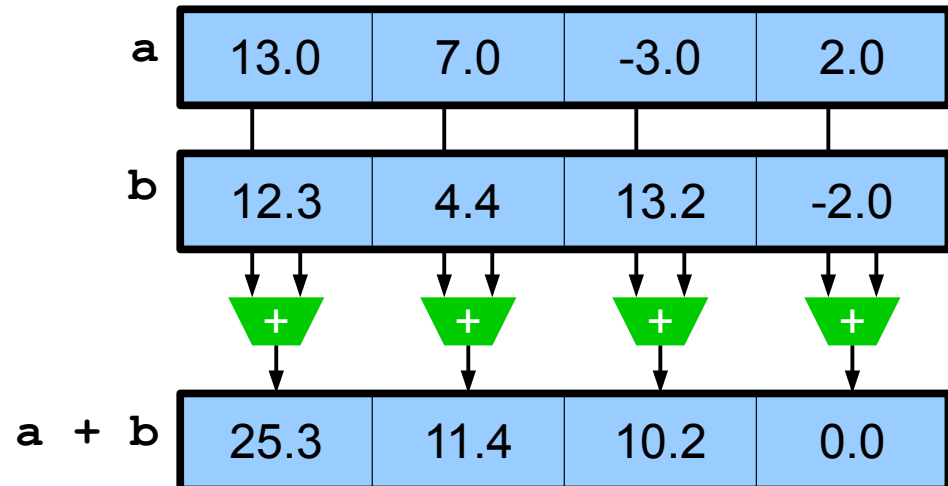
Single Instruction Multiple Data

- In the SIMD model the **same** operation is applied to multiple data items
- This is usually realized through special instructions that work with short, fixed-length arrays
 - E.g., SSE and ARM NEON can work with 4-element arrays of 32-bit floats

Scalar instruction

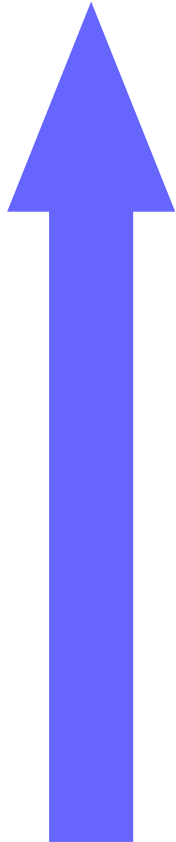


SIMD instruction



Programming for SIMD

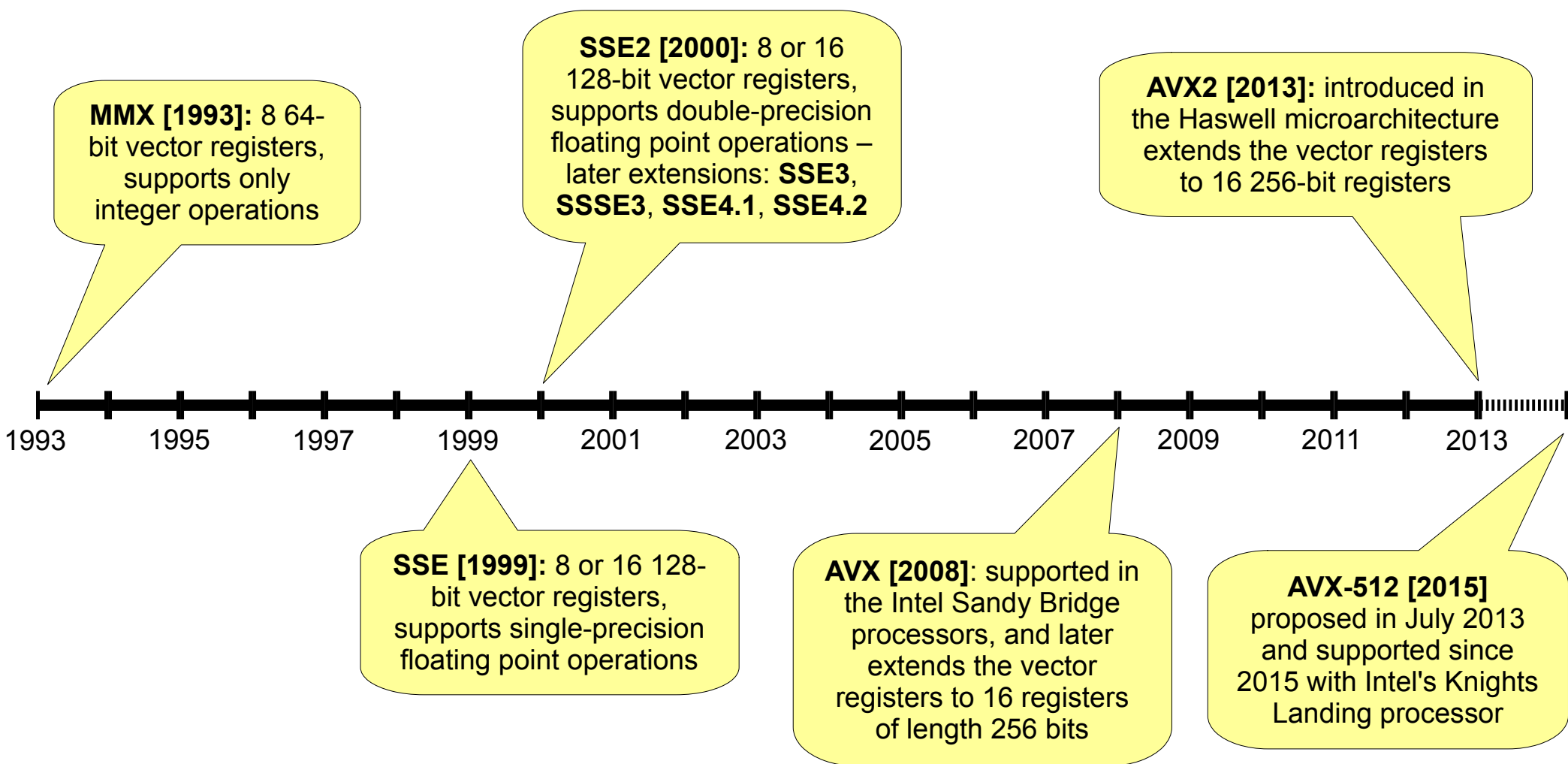
High level



Low level

- Compiler auto-vectorization
- Optimized SIMD libraries
 - ATLAS, FFTW
- Domain Specific Languages (DSLs) for SIMD programming
 - E.g., Intel SPMD program compiler
- Compiler-dependent vector data types
- Compiler SIMD intrinsics
- Assembly language

Intel SIMD extensions timeline



x86 SSE/AVX

- SSE**

- 16 × 128-bit SIMD registers XMM0—XMM15 (in x86-64 mode)

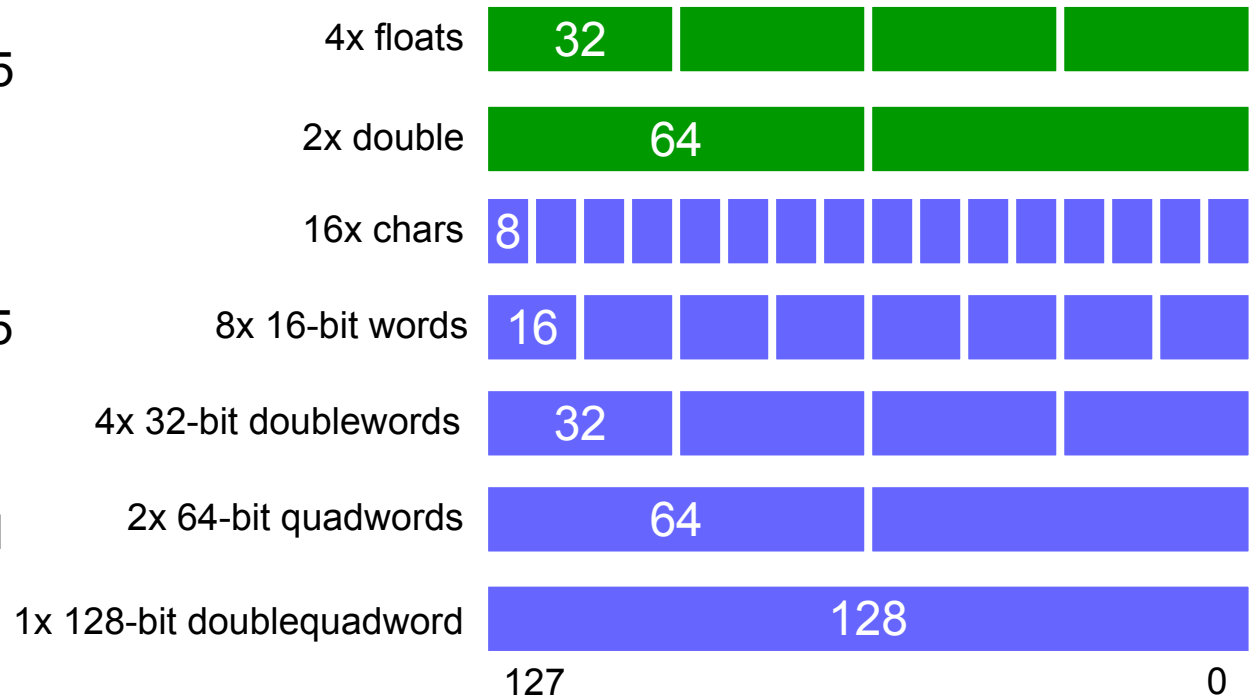
- AVX2**

- 16 × 256-bit SIMD registers YMM0—YMM15

- AVX-512**

- 32 × 512-bit SIMD registers ZMM0—ZMM31

SSE/AVX types

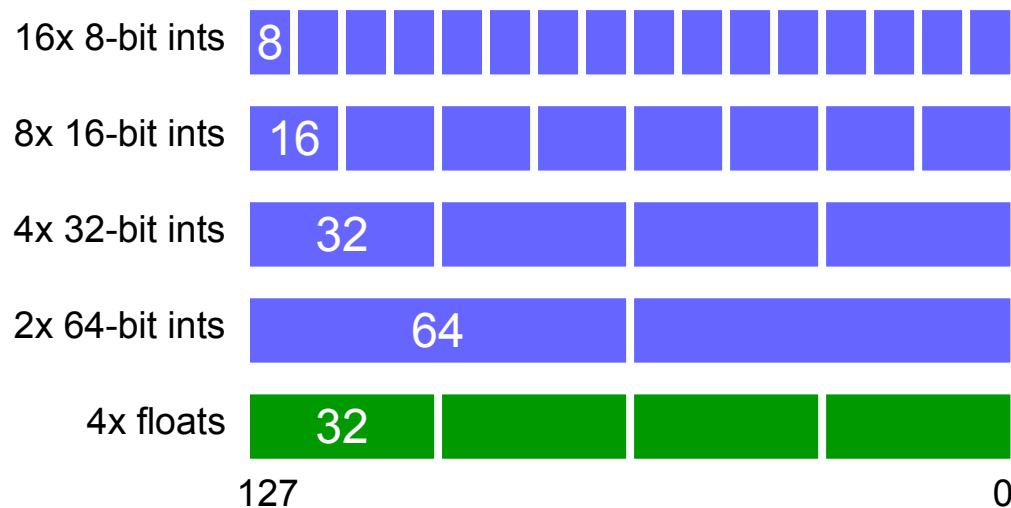


AVX2 types



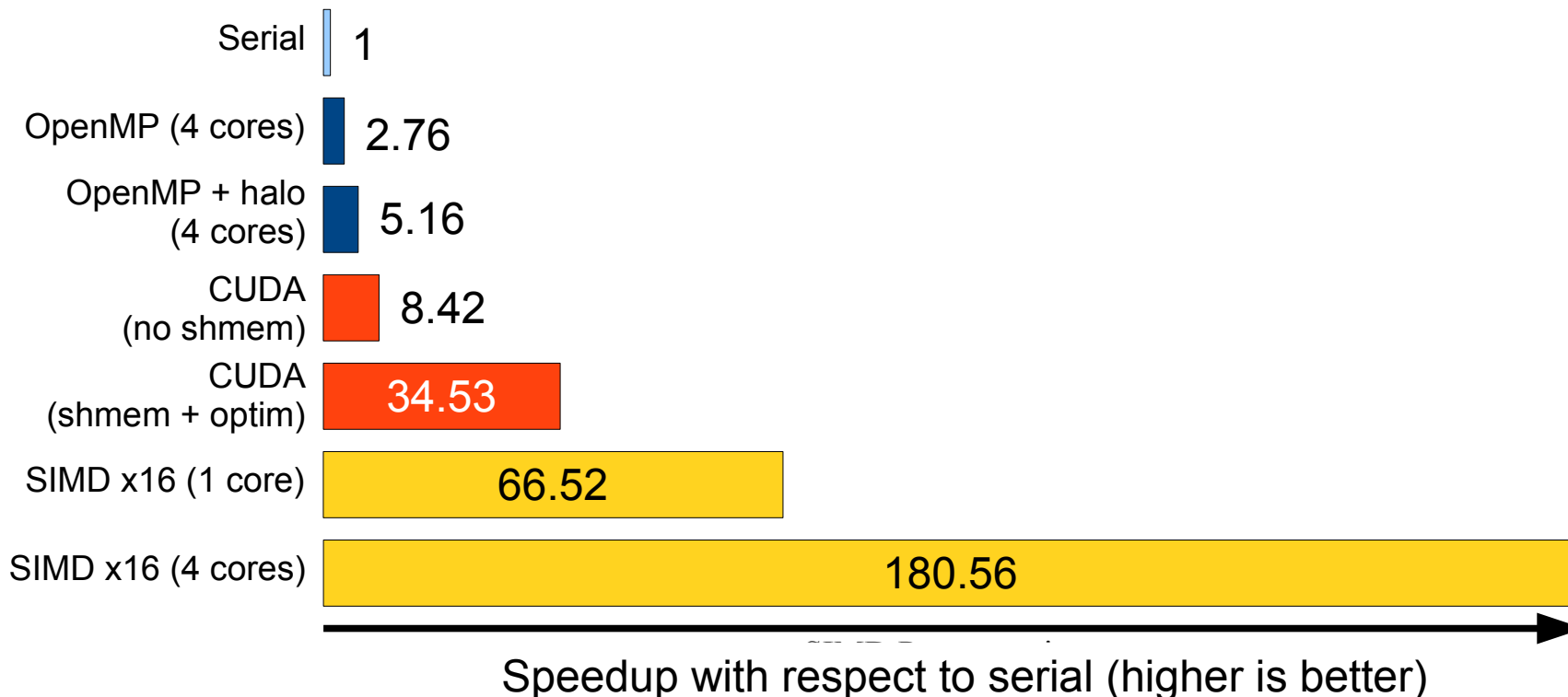
ARM NEON

- The NEON register bank consists of 32 64-bit registers
- The NEON unit can view the same register bank as:
 - 16 × 128-bit quadword registers Q0—Q15
 - 32 × 64-bit doubleword registers D0—D31



Is it worth the effort?

- In some cases, yes
 - Example: BML cellular automaton ($N = 1024$, $\rho = 0.3$, steps = 1024), CPU Xeon E3-1220 @ 3.10GHz 4 cores (no ht) + GCC 4.8.4, 16GB RAM; GPU nVidia Quadro K620 (384 CUDA cores) + nvcc V8.0.61



Superlinear speedup?

Init
Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test

SIMD Programming

```
for (i=0; i<n; i++)  
    Compute(i)  
}
```

Superlinear speedup?

Init
Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test

Init
Test
Compute
Compute
Compute
Compute
Inc & Test
Compute
Compute
Compute
Compute
Inc & Test

SIMD Programming

```
for (i=0; i<n-3; i+=4)
    Compute(i)
    Compute(i+1)
    Compute(i+2)
    Compute(i+3)
}
handle leftovers...
```

Superlinear speedup?

Init
Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test
Compute
Inc & Test

Init
Test
Compute
Compute
Compute
Compute
Inc & Test
Compute
Compute
Compute
Compute
Inc & Test

Init
Test
SIMD_Compute
Inc & Test
SIMD_Compute
Inc & Test

SIMD Programming

```
for (i=0; i<n-3; i+=4)
    SIMD_Compute(i, ..., i+3)
}
handle leftovers...
```

Checking for SIMD support on Linux

`cat /proc/cpuinfo`

- On x86

```
flags      : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs
bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni
pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma
cx16 xtrp pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm
3dnowprefetch epb intel_pt tpr_shadow vnmi flexpriority ept vpid
fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm
rdseed adx smap xsaveopt cqm_llc cqm_occup_llc cqm_mbm_total
cqm_mbm_local dtherm arat pln pts
```

- On ARM

```
Features   : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4
idiva idivt vfpd32 lpae evtstrm crc32
```

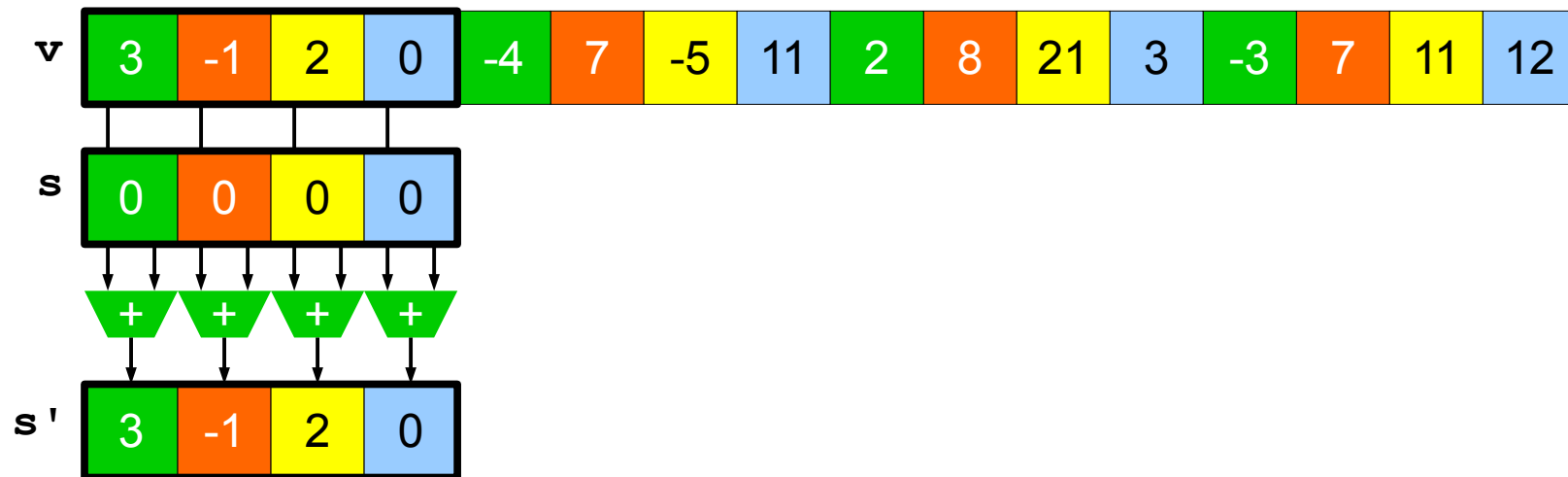
Vectorization opportunities

```
float vsum(float *v, int n)
{
    float s = 0.0;
    for (int i=0; i<n; i++)
        s += v[i];
    return s;
}
```

3	-1	2	0	-4	7	-5	11	2	8	21	3	-3	7	11	12
---	----	---	---	----	---	----	----	---	---	----	---	----	---	----	----

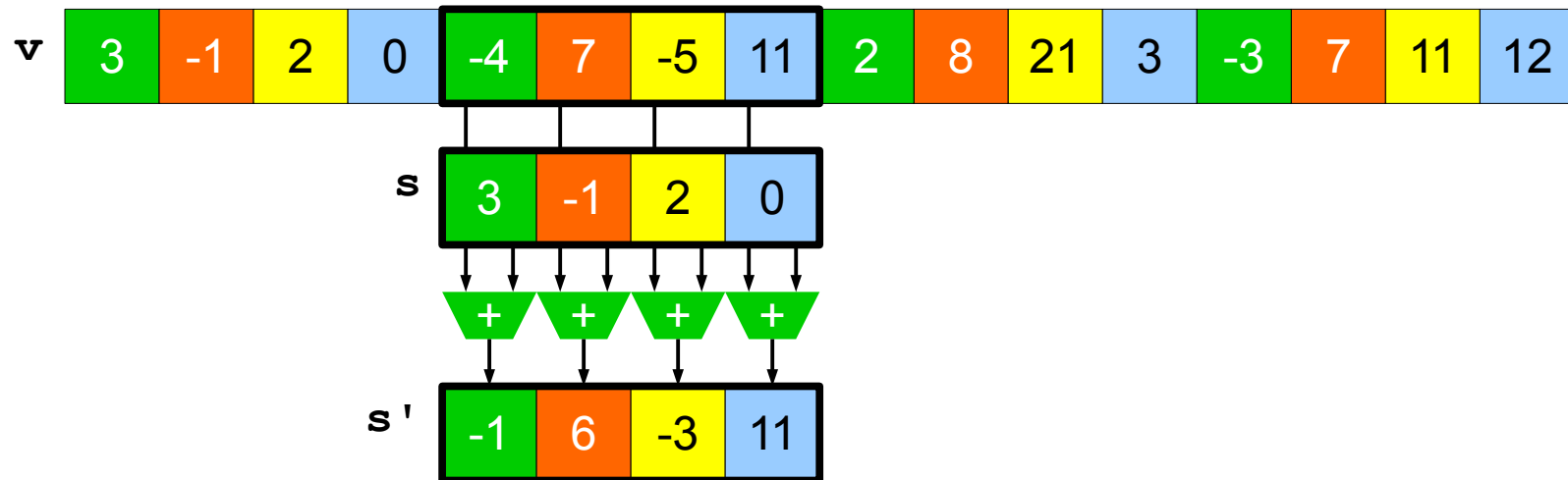
Vectorization opportunities

```
float vsum(float *v, int n)
{
    float s = 0.0;
    for (int i=0; i<n; i++)
        s += v[i];
    return s;
}
```



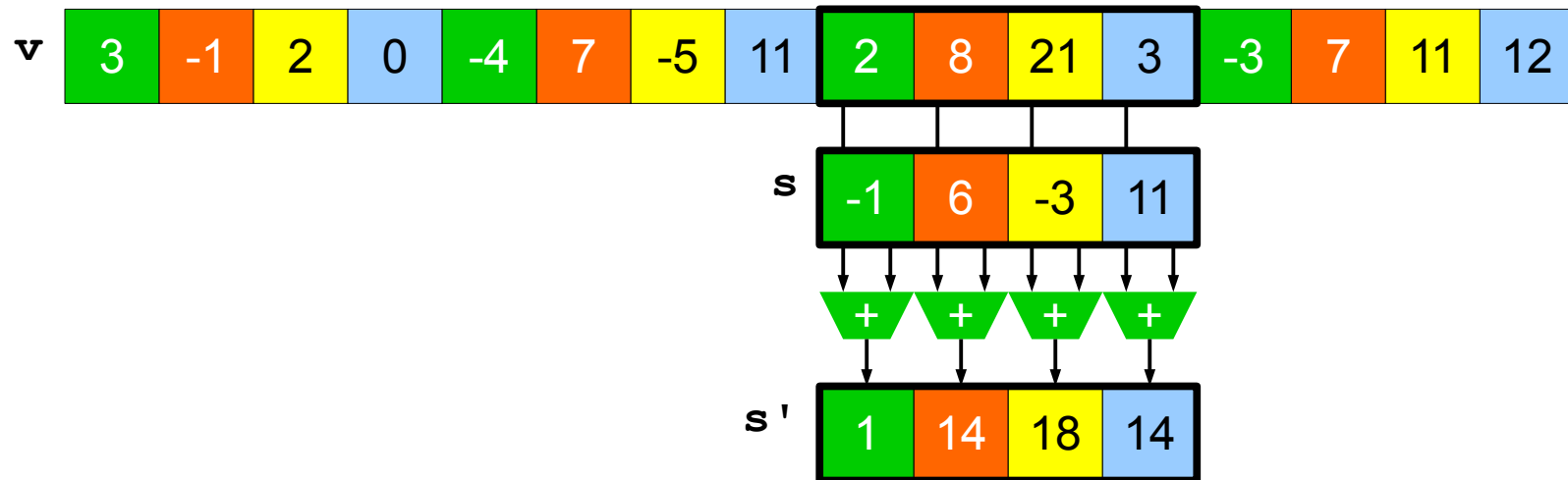
Vectorization opportunities

```
float vsum(float *v, int n)
{
    float s = 0.0;
    for (int i=0; i<n; i++)
        s += v[i];
    return s;
}
```



Vectorization opportunities

```
float vsum(float *v, int n)
{
    float s = 0.0;
    for (int i=0; i<n; i++)
        s += v[i];
    return s;
}
```



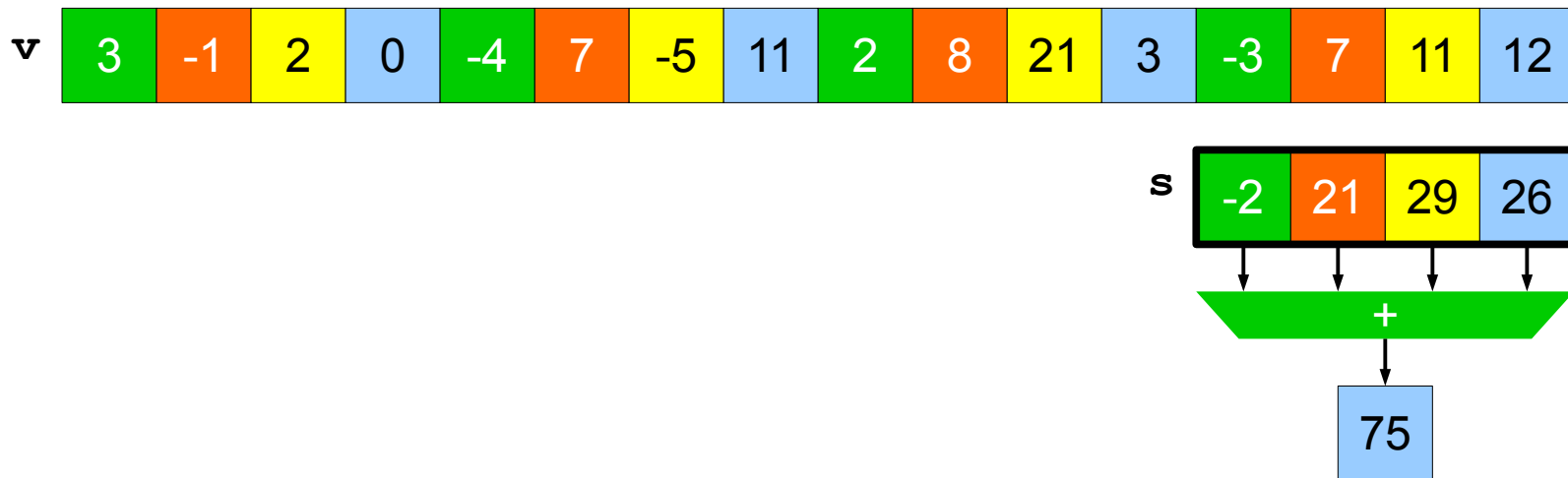
Vectorization opportunities

```
float vsum(float *v, int n)
{
    float s = 0.0;
    for (int i=0; i<n; i++)
        s += v[i];
    return s;
}
```



Vectorization opportunities

```
float vsum(float *v, int n)
{
    float s = 0.0;
    for (int i=0; i<n; i++)
        s += v[i];
    return s;
}
```

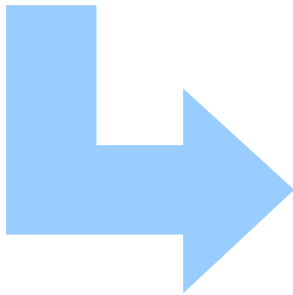


Vectorization opportunities

- Care must be taken if the array length is not multiple of the SIMD vector length
- **Padding**
 - Add “dummy” elements at the end to make the array length multiple of the SIMD vector length
 - Not always feasible (requires to modify the input; choosing the values of the extra elements can be tricky and problem-dependent)
- **Handle the leftovers with scalar operations**
 - Small performance hit
 - Redundant – therefore error-prone – code if done by hand

```
float vsum(float *v, int n)
{
    float s = 0.0; int i;
    for (i=0; i<n; i++)
        s += v[i];
    return s;
}
```

Not really SIMD;
assumes 4-lanes SIMD



```
float vsum(float *v, int n)
{
    float vs[4] = {0.0, 0.0, 0.0, 0.0};
    float s = 0.0;
    int i;
    for (i=0; i<n-3; i += 4) {
        vs[0] += v[i];
        vs[1] += v[i+1];
        vs[2] += v[i+2];
        vs[3] += v[i+3];
    }
    s = vs[0] + vs[1] + vs[2] + vs[3];
    /* Handle leftover */
    for (; i<n; i++) {
        s += v[i];
    }
    return s;
}
```

vs[0:3] += v[i:i+3];

Vector data type

Vector data types

- Some compilers support **vector data types**
 - Vector data types are **non portable**, **compiler-specific** extensions
- Vector data types as (as the name suggests) small vectors of some numeric type
 - typically, **char**, **int**, **float**, **double**
- Ordinary arithmetic operations (sum, product...) can be applied to vector data types
- The compiler emits the appropriate SIMD instructions for the target architecture if available
 - If no appropriate SIMD instruction is available, the compiler emits equivalent scalar code

Definition

- Defining vector data types

```
/* v4i is a vector of elements of type int; variables of type v4i occupy 16 bytes of contiguous memory */  
typedef int v4i __attribute__((vector_size(16)));  
  
/* v4f is a vector of elements of type float; variables of type v4f occupy 16 bytes of contiguous memory */  
typedef float v4f __attribute__((vector_size(16)));
```

- The length (num. of elements) of **v4f** is

```
#define VLEN (sizeof(v4f)/sizeof(float))
```

Definition

- Is it possible to define vector types or “arbitrary” length

```
/* v8f is a vector of elements of type float; variables of type v8f occupy  
32 bytes of contiguous memory */  
typedef float v8f __attribute__((vector_size(32)));
```

- If the target architecture does not support SIMD registers of the specified length, the compiler takes care of that, e.g., using multiple SIMD instructions on shorter vectors

Usage

```
/* simd-vsum-vector.c */

typedef float v4f __attribute__((vector_size(16)));
#define VLEN (sizeof(v4f)/sizeof(float))

float vsum(float *v, int n)
{
    v4f vs = {0.0f, 0.0f, 0.0f, 0.0f};
    v4f *vv = (v4f*)v;
    int i; float s = 0.0f;
    for (i=0; i<n-VLEN+1; i += VLEN) {
        vs += *vv;
        vv++;
    }
    s = vs[0] + vs[1] + vs[2] + vs[3];
    for ( ; i<n; i++) {
        s += v[i];
    }
    return s;
}
```

*Variables of type **v4f**
can be treated as
standard arrays*

*Variables of type **v4f**
can be treated as
standard arrays*

Vector data types

- GCC allows the following operators on vector data types
 - $+$, $-$, $*$, $/$, unary minus, $^$, $|$, $\&$, $\sim$, $\%$
- It is also possible to use a binary vector operation where one operand is a scalar

```
typedef int v4i __attribute__((vector_size (16)));

v4i a, b, c;
long x;

a = b + 1;      /* OK: a = b + {1,1,1,1};          */
a = 2 * b;      /* OK: a = {2,2,2,2} * b;          */
a = 0;          /* Error: conversion not allowed          */
a = x + a;      /* Error: cannot convert long to int      */
```

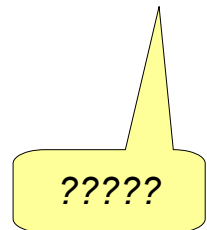
Vector data types

- Vector comparison is supported with standard comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`
- SIMD vectors are compared element-wise
 - `0` when comparison is `false`
 - `-1` when comparison is `true`

```
typedef int v4i __attribute__((vector_size (16)));  
  
v4i a = {1, 2, 3, 4};  
v4i b = {3, 2, 1, 4};  
v4i c;  
  
c = (a > b);          /* Result {0, 0, -1, 0} */  
c = (a == b);         /* Result {0, -1, 0, -1} */
```

Note on memory alignment

- Some versions of GCC emit assembly code for dereferencing a pointer to a vector datatype that only works if the memory address is 16B aligned
 - More details later on
- `malloc()` may or may not return a pointer that is properly aligned
 - From the man page: *“The malloc() and calloc() functions return a pointer to the allocated memory, which is suitably aligned for any built-in type.”*



Ensuring proper alignment

- For data on the stack:

```
/* __BIGGEST_ALIGNMENT__ is 16 for SSE, 32 for AVX; it is therefore the preferred choice since it is automatically defined to suit the target */
```

```
float v[1024] __attribute__((aligned(__BIGGEST_ALIGNMENT__))) ;
```

- For data on the heap:

```
#define _XOPEN_SOURCE 600  
#include <stdlib.h>
```

```
float *v;  
posix_memalign(&v, __BIGGEST_ALIGNMENT__, 1024);
```

*Do this at the very beginning (before including anything); better yet, compile with the **-D_XOPEN_SOURCE=600** flag to define this symbol compilation-wide*

Alignment

Number of bytes to allocate

Vectorizing branches

- Branches (if-then-else) are difficult to vectorize, since the SIMD programming model assumes that the same operation is applied to **all** elements of a SIMD register
 - How can we vectorize the following code fragment?

```
int a[4] = { 12, -7, 2, 3 };

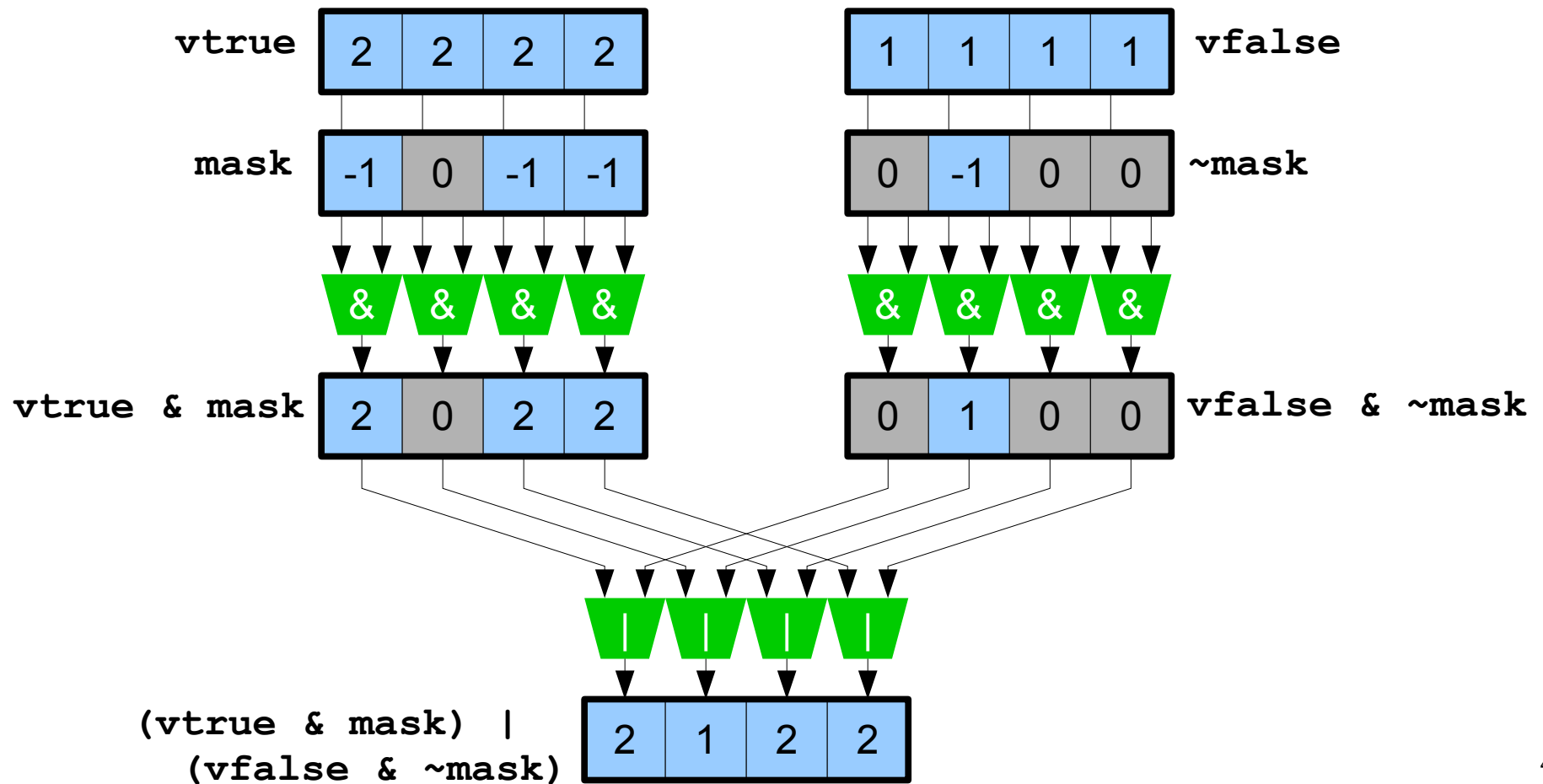
for (int i=0; i<4; i++) {
    if ( a[i] > 0 ) {
        a[i] = 2;
    } else {
        a[i] = 1;
    }
}
```

```

v4i a = { 12, -7, 2, 3 };
v4i vtrue = {2, 2, 2, 2};
v4i vfalse = {1, 1, 1, 1};

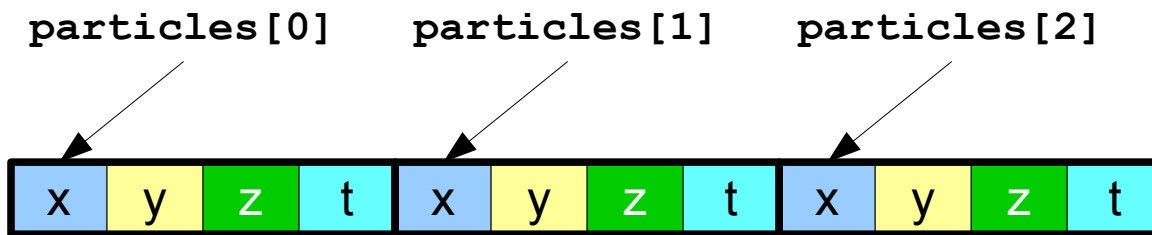
v4i mask = (a > 0); /* mask = {-1, 0, -1, -1} */
a = (vtrue & mask) | (vfalse & ~mask);

```



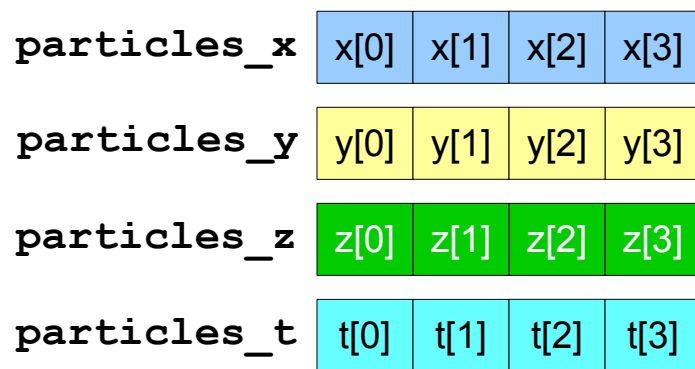
Data layout: SoA vs AoS

- Arrays of Structures (AoS)



```
typedef struct point3d {  
    float x, y, z, t;  
} point3d;  
#define NP 1024  
point3d particles[NP];
```

- Structures of Arrays (SoA)



```
#define NP 1024  
float particles_x[NP];  
float particles_y[NP];  
float particles_z[NP];  
float particles_t[NP];
```