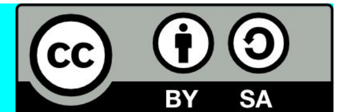
SIMD Programming

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

moreno.marzolla@unibo.it

Copyright © 2017–2022
Moreno Marzolla, Università di Bologna, Italy
https://www.moreno.marzolla.name/teaching/HPC/



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 http://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

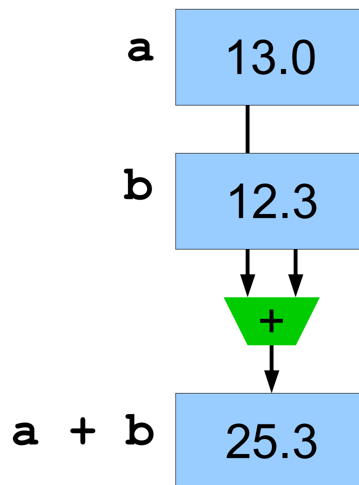
Credits

- Marat Dukhan (Georgia Tech)
  - http://www.cc.gatech.edu/grads/m/mdukhan3/
- Salvatore Orlando (Univ. Ca' Foscari di Venezia)

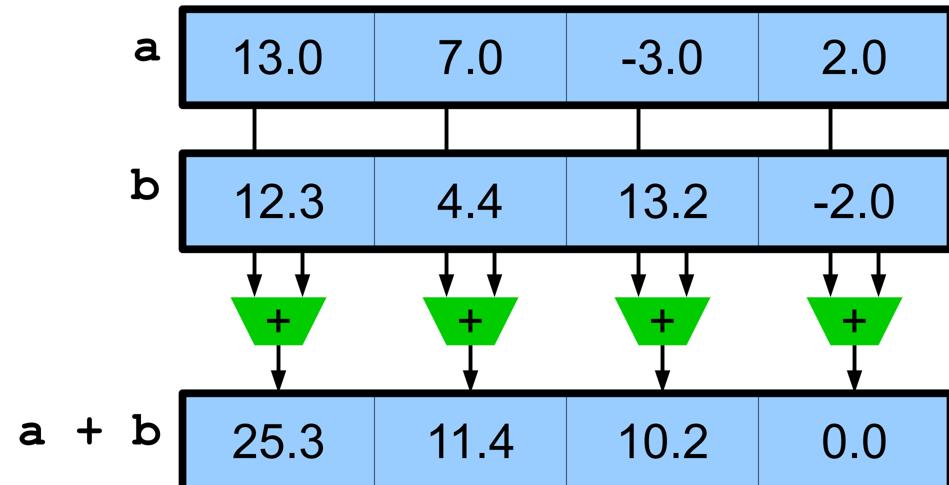
Single Instruction Multiple Data

- 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

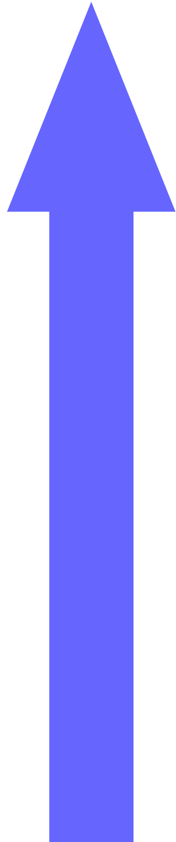


SIMD instruction



Programming for SIMD

High level

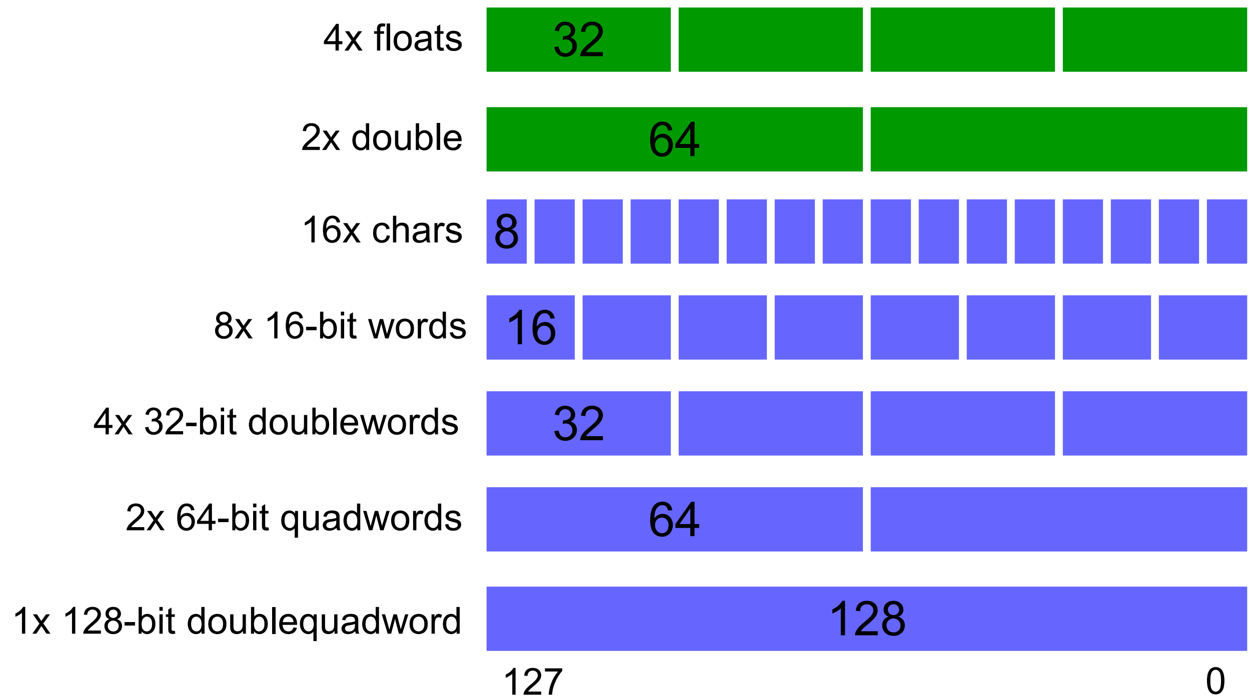


Low level

- 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

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

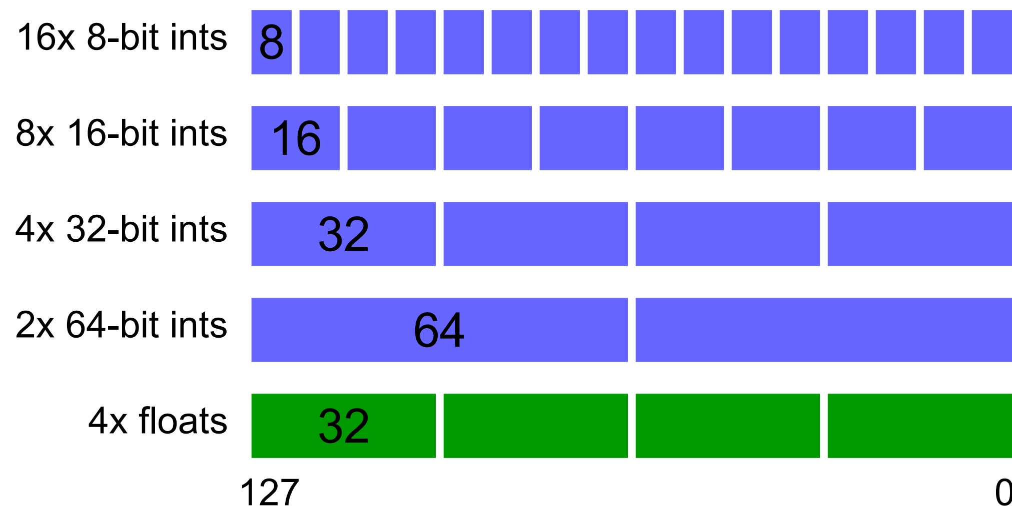


Source: https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf

ARM NEON

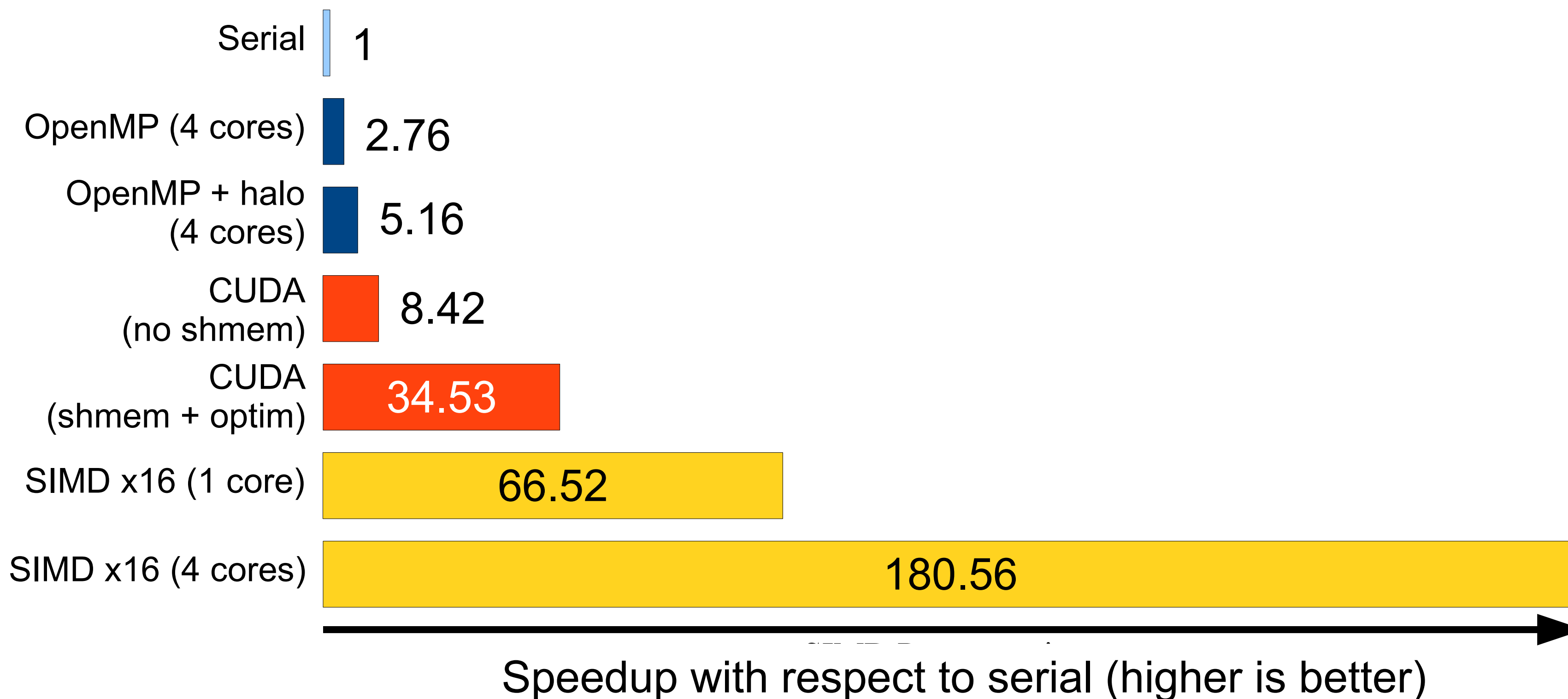
- The NEON register bank consists of 32 64-bit registers
- The NEON unit can view the same register bank as:
  - 16 × 128-bit quadword registers Q0—Q15
  - 32 × 64-bit doubleword registers D0—D31

See http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002a/index.html

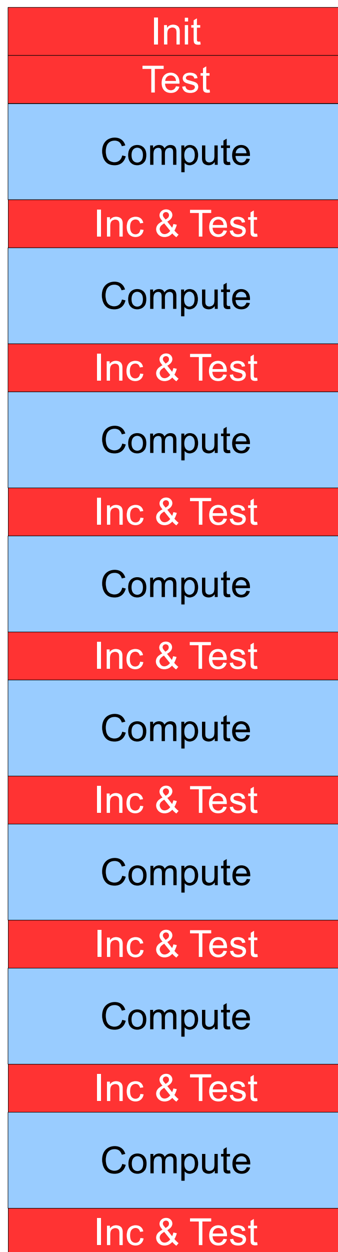


Is it worth the effort?

- In some cases, yes
  - Example: BML cellular automaton (N = 1024, ρ = 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

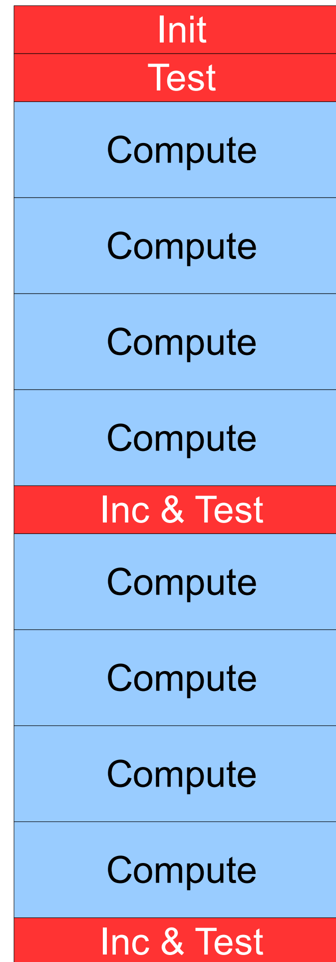


Superlinear speedup?



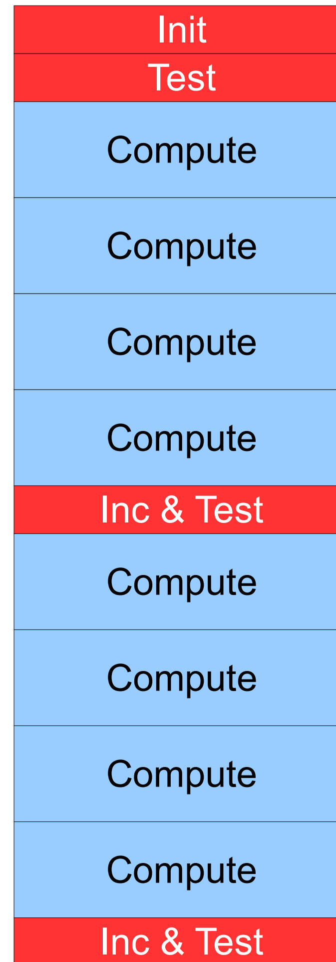

```
for (i=0; i<n; i++)
    Compute(i)
}
```

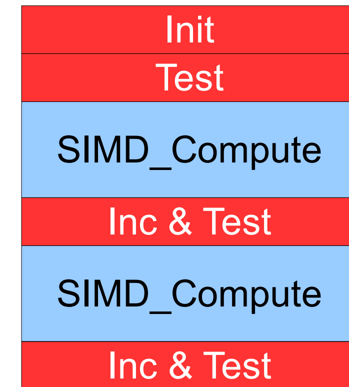
Superlinear speedup?




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

Superlinear speedup?






```
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 xtpr 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
```

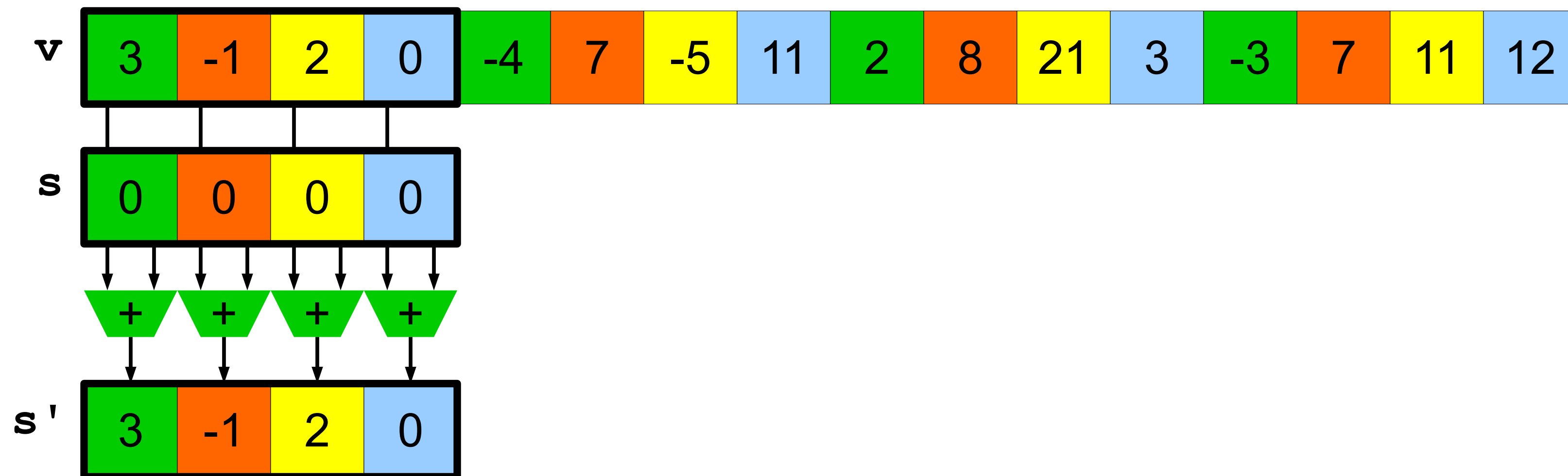
Vectorization opportunities

```
float vsum(float *v, int n)
{
    float s = 0.0; int i;
    for (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

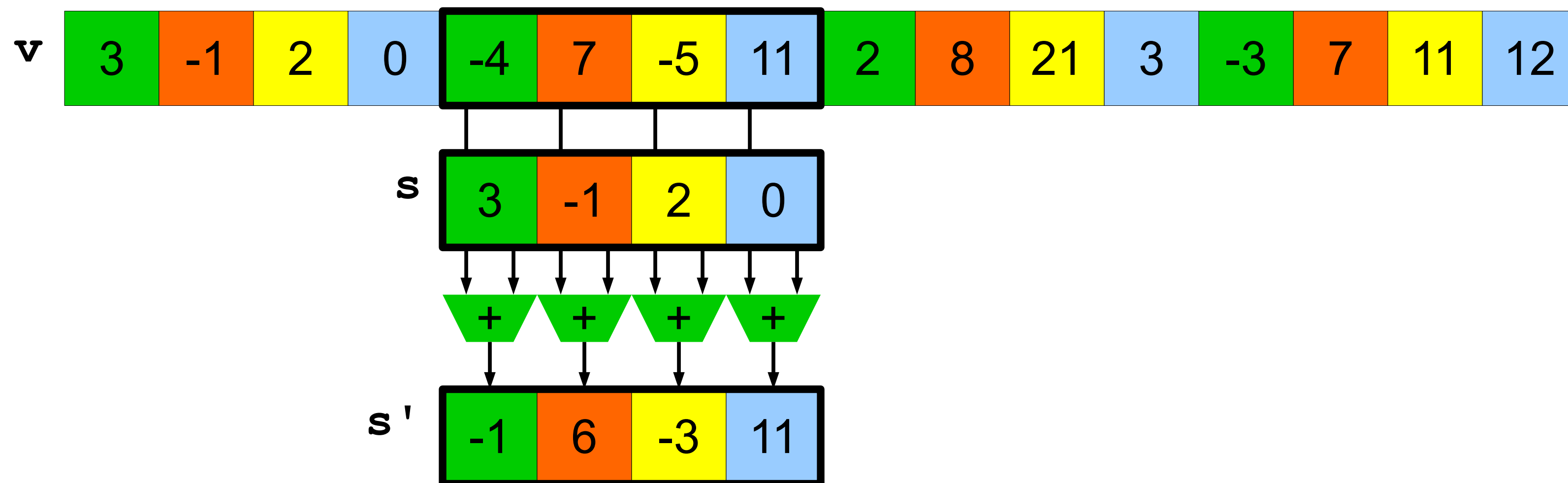
Vectorization opportunities

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



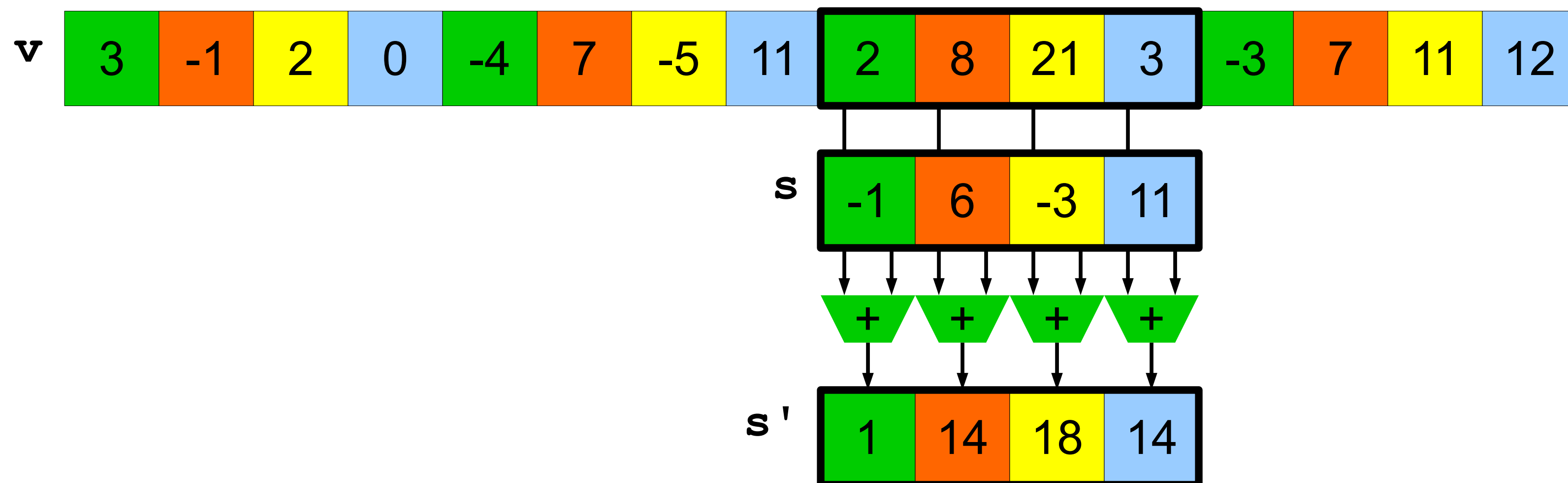
Vectorization opportunities

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



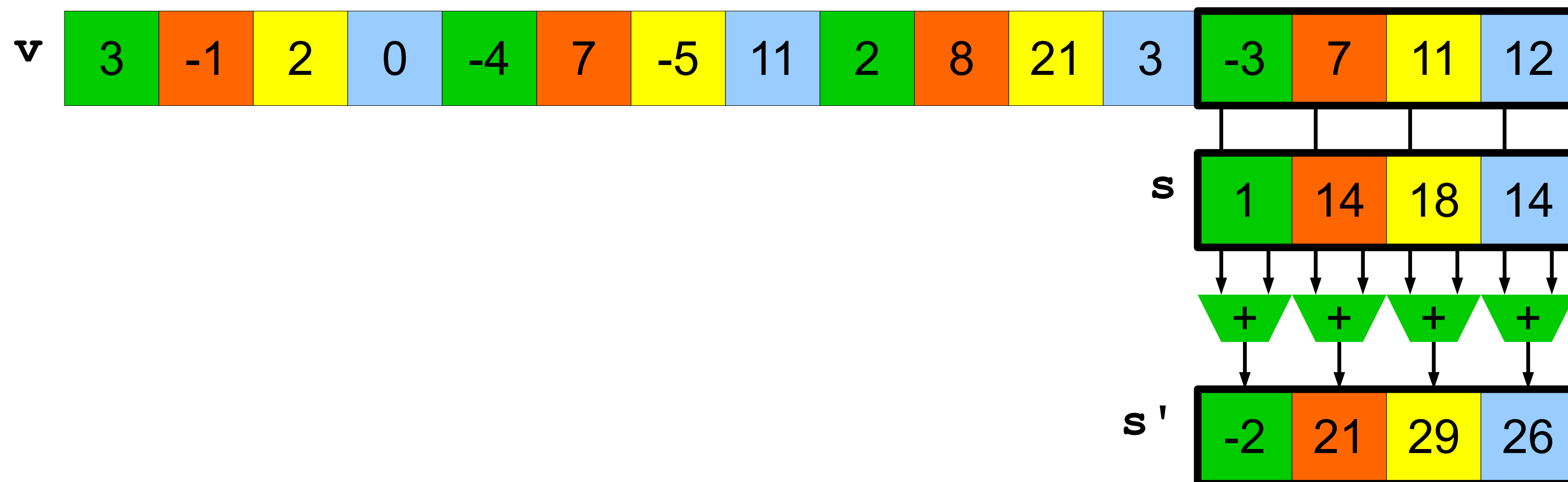
Vectorization opportunities

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



Vectorization opportunities

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



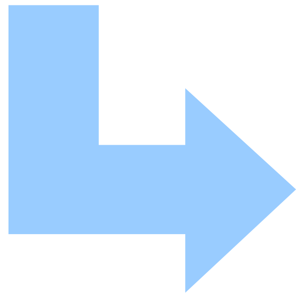
Vectorization opportunities

```
float vsum(float *v, int n)
{
     float s = 0.0; int i;
     for (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;
}
```



```
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];

Not really SIMD; assumes 4-lanes SIMD

Auto-vectorization

Compiler auto-vectorization

- To enable auto-vectorization with GCC

```
-O2
-ftree-vectorize
-fopt-info-vec
-march=native
```

You might need to turn on optimization to enable auto-vectorization (-O2)

Enable debugging output to see what gets vectorized and what does not (-fopt-info-vec)

Autodetect and use SIMD instructions available on your platform (-march=native)

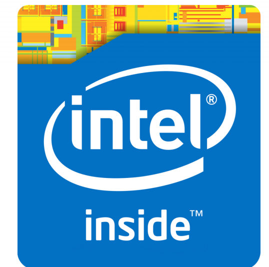
- Warning: if you enable platform-specific optimizations, your binary will not run on less capable processors!
- To see which flags are enabled with **`-march=native`**

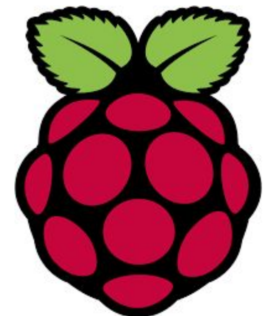
```
gcc -march=native -Q --help=target
```

Selection of the target architecture

the hard way

- On x86:
  - **`-msse`** emits SSE instructions
  - **`-msse2`** emits SIMD instructions up to SSE2
  - ...
  - **`-msse4.2`** emits SIMD instructions up to SSE4.2
  - **`-mavx`** emits SIMD instructions up to AVX (incl. SSEx)
  - **`-mavx2`** emits SIMD instructions up to AVX2
- On ARM (RaspberryPI2/3)
  - **`-march=armv7-a -mfpu=neon \`**
    `-mvectorize-with-neon-quad`
- On ARM (NVIDIA Jetson TK1 dev. board)
  - **`-march=armv7 -mfpu=neon \`**
    `-mvectorize-with-neon-quad`





Auto-vectorization

- Let's try GCC 7.5.0 auto-vectorization capabilities

```
gcc -march=native -O2 -ftree-vectorize -fopt-info-vec \
      simd-vsum-auto.c -o simd-vsum-auto
```

```
simd-vsum-auto.c:45:5: note: step unknown.
simd-vsum-auto.c:45:5: note: reduction: unsafe fp math optimization:
s_10 = _9 + s_14;
…
simd-vsum-auto.c:45:5: note: not vectorized: unsupported use in stmt.
…
simd-vsum-auto.c:54:5: note: Unknown misalignment, is_packed = 0
simd-vsum-auto.c:54:5: note: virtual phi. skip.
simd-vsum-auto.c:54:5: note: loop vectorized
```

Note: GCC 9.3.0 vectorizes the loop even without -funsafe-math-optimizations

Auto-vectorization

- GCC 7.5.0 was unable to vectorize this loop:

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

- The output gives us a hint:

```
simd-vsum-auto.c:45:5: note: reduction: unsafe fp math
optimization: s_10 = _9 + s_14;
```

Auto-vectorization

- In this case, GCC is smarter than we are
- The familiar rules of infinite-precision math do not hold for finite-precision math
  - FP addition is not associative, i.e., $(a + b) + c$ could produce a different result than $a + (b + c)$
  - Therefore, GCC by default does not apply optimizations that violate FP safety
- We can tell GCC to optimize anyway with **`-funsafe-math-optimizations`**

Bingo!

```
gcc -funsafe-math-optimizations -march=native -O2 \
   -ftree-vectorize -fopt-info-vec \
   simd-vsum-auto.c -o simd-vsum-auto
```

```
…
simd-vsum-auto.c:45:5: note: loop vectorized
…
simd-vsum-auto.c:54:5: note: loop vectorized
…
```

Note: GCC 9.3.0 vectorizes the loop even without -funsafe-math-optimizations

Auto-vectorization

- You can look at the generated assembly output with

```
gcc -S -c -funsafe-math-optimizations \
   -march=native -O2 -ftree-vectorize \
   simd-vsum-auto.c -o simd-vsum-auto.s
```

```
.L4:
        movq     %rcx, %r8
        addq     $1, %rcx
        salq     $5, %r8
        vaddps   (%rdi,%r8), %ymm0, %ymm0
        cmpl     %ecx, %edx
        ja       .L4
        vhaddps %ymm0, %ymm0, %ymm0
        vhaddps %ymm0, %ymm0, %ymm1
        vperm2f128      $1, %ymm1, %ymm1, %ymm0
        vaddps  %ymm1, %ymm0, %ymm0
        cmpl     %esi, %eax
        je       .L17
```

xxxps *instructions are those dealing with **P**acked **S**ingle-precision SIMD registers*

Auto-Vectorization

the Good, the Bad, the Ugly

***In theory**, auto-vectorization is totally transparent.*
***However**, in might be beneficial to provide hints to the compiler (e.g., alignment info, the C99 `restrict` keyword) to help it do a better job*

*Even when auto-vectorization is possible, the compiler often generates **sub-optimal code***

*Regardless how good the compiler is, there are many cases where **auto-vectorization fails.** Then you are on your own.*

IL BUONO IL BRUTTO IL CATTIVO

SERGIO LEONE

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, ^, |, &, ~, %

- 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 */
```


https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

Vector data types

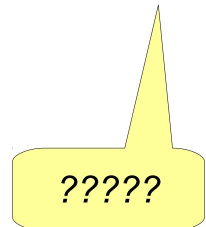
- Vector comparison is supported with standard comparison operators: ==, !=, <, <=, >, >=
- SIMD vectors are compared element-wise
  - 0 when comparison is false
  - -1 when comparison is true

```
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}  */
```


https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

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:

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*

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

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

Alignment

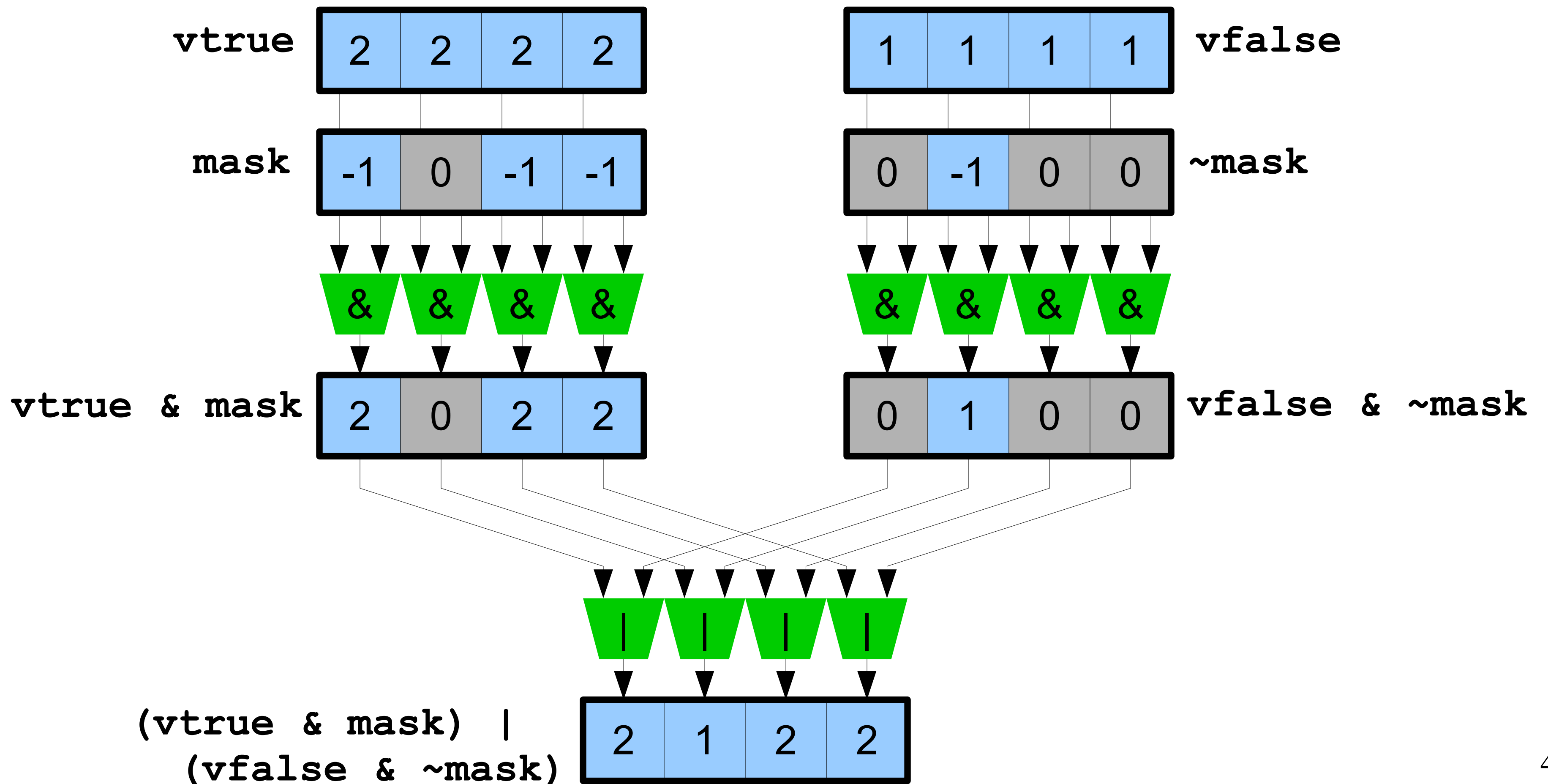
Number of bytes to allocate

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?

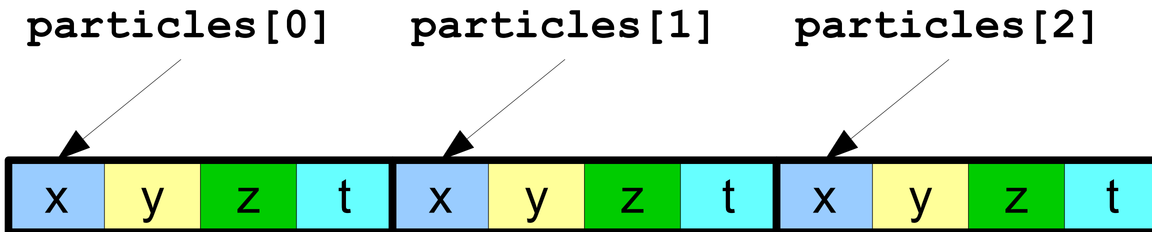
```
int a[4] = { 12, -7, 2,  3 };
int i;

for (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);
```



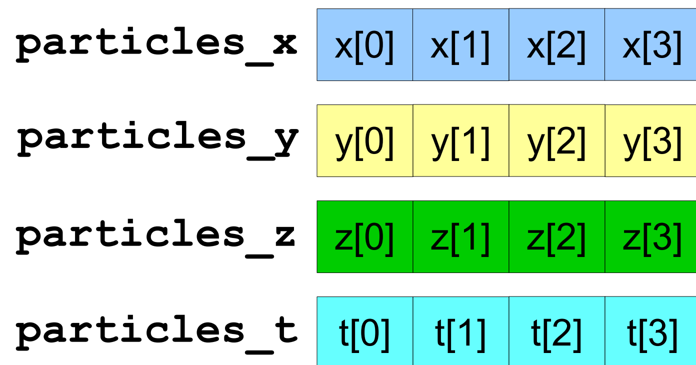
Data layout: SoA vs AoS

- Arrays of Structures (AoS)




```
typedef point3d {
    float x, y, z, t;
};
#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];
```

Programming with SIMD intrinsics

Optional Material

SIMD intrinsics

- The compiler exposes all low-level SIMD operations through C functions
  - Each function maps to the corresponding low-level SIMD instruction
  - In other words, you are programming in assembly
- SIMD intrinsics are platform-dependent
  - since different processors have different instruction sets
- However
  - SIMD intrinsics are machine-dependent but compiler-independent
  - Vector data types are machine-independent but compiler-dependent

Optional Material

Vector sum with SSE intrinsics

```
/* simd-vsum-intrinsics.c */
#include <x86intrin.h>

float vsum(float *v, int n)
{
    __m128 vv, vs;
    float s = 0.0f;
    int i;

    vs = _mm_setzero_ps();
    for (i=0; i<n-4+1; i += 4) {
        vv = _mm_loadu_ps(&v[i]);
        vs = _mm_add_ps(vv, vs);
    }
    s = vs[0] + vs[1] + vs[2] + vs[3];
    for ( ; i<n; i++) {
        s += v[i];
    }
    return s;
}
```

vs = {0.0f, 0.0f, 0.0f, 0.0f}

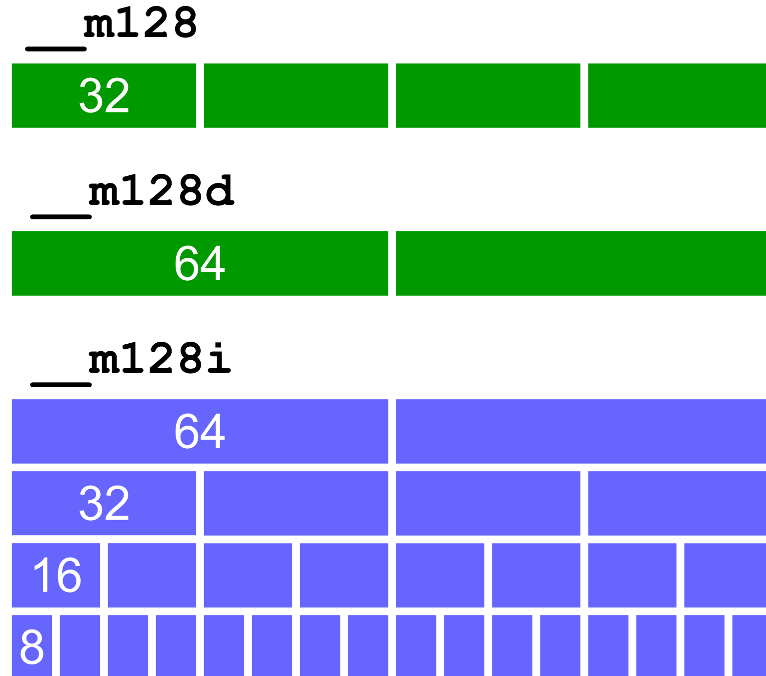
_mm_loadu_ps = Load Unaligned Packed Single-precision

variables of type __m128 can be used as vectors

Optional Material

SSE intrinsics

- `#include <x86intrin.h>`
- Three new datatypes
  - `__m128`
    - Four single-precision floats
  - `__m128d`
    - Two double-precision floats
  - `__m128i`
    - Two 64-bit integers
    - Four 32-bit integers
    - Eight 16-bit integers
    - Sixteen 8-bit integers



Optional Material

SSE memory operations

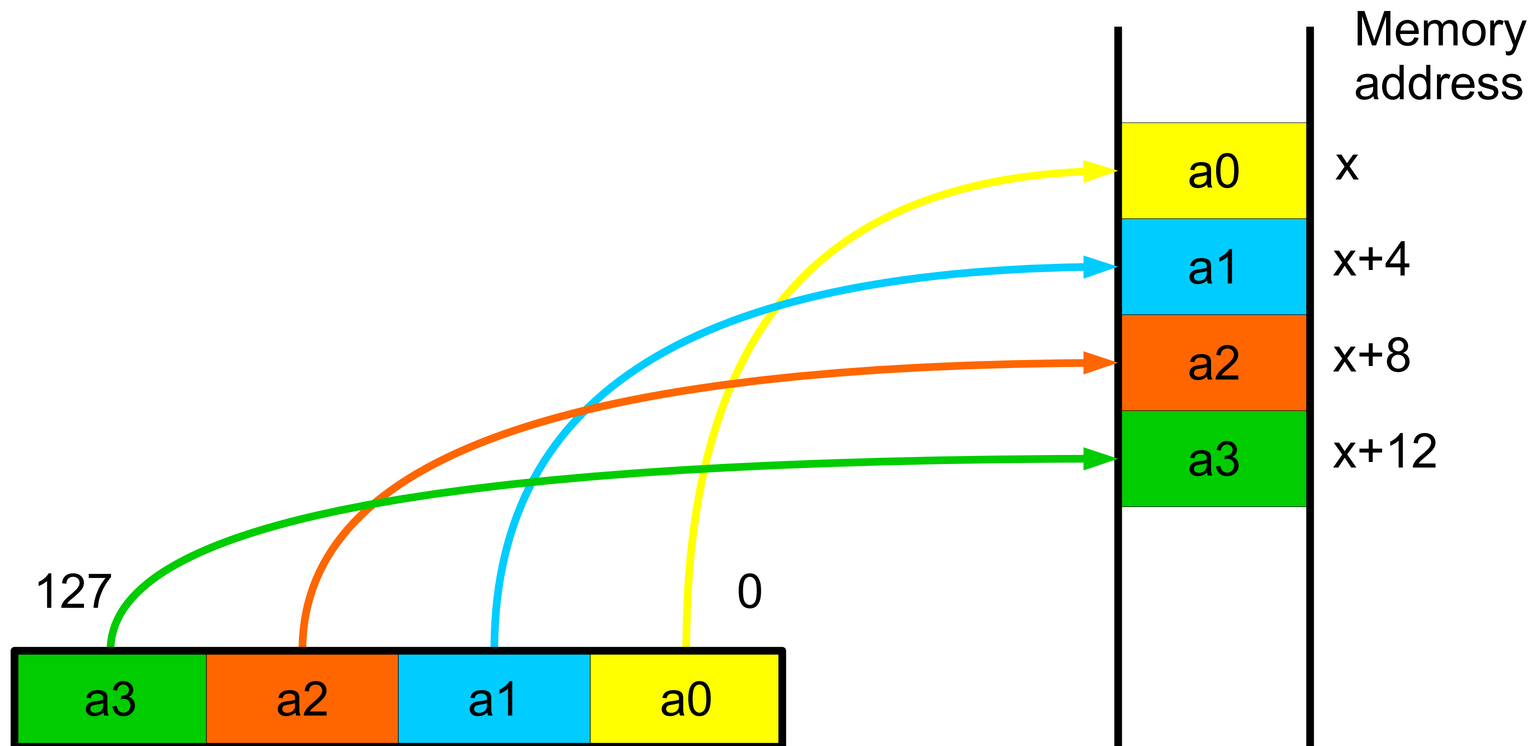
u = unaligned

- `__m128 _mm_loadu_ps(float *aPtr)`
  - Load 4 floats starting from memory address aPtr
- `__m128 _mm_load_ps(float *aPtr)`
  - Load 4 floats starting from memory address aPtr
  - aPtr must be a multiple of 16
- `_mm_storeu_ps(float *aPtr, __m128 v)`
  - Store 4 floats from v to memory address aPtr
- `_mm_store_ps(float *aPtr, __m128 v)`
  - Store 4 floats from v to memory address aPtr
  - aPtr must be a multiple of 16
- There are other intrinsics for load/store of doubles/ints

How values are stored in memory

Optional Material

- SSE SIMD registers are stored in memory as a0, a1, a2, a3 in increasing memory locations



Optional Material

Some SSE arithmetic operations

- Do operation <op> on a and b, write result to c:

```
__m128 c = _mm_<op>_ps(a, b);
```

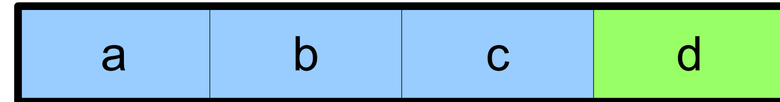
- Addition: c = a + b
  - c = _mm_add_ps(a, b);
- Subtraction: c = a – b
  - c = _mm_sub_ps(a, b);
- Multiplication: c = a * b
  - c = _mm_mul_ps(a, b);
- Division: c = a / b
  - c = _mm_div_ps(a, b);
- Minimum: c = fmin(a, b)
  - c = _mm_min_ps(a, b);
- Maximum: c = fmax(a, b)
  - c = _mm_max_ps(a, b);
- Square root: c = sqrt(a)
  - c = _mm_sqrt_ps(a);

Optional Material

Load constants

```
v = _mm_set_ps(a, b, c, d)
```

```
float d = _mm_cvtss_f32(v)
```




```
v = _mm_set1_ps(a)
```

```
v = _mm_setzero_ps();
```

Resources

- Intel Intrinsics Guide
  https://software.intel.com/sites/landingpage/IntrinsicsGuide/