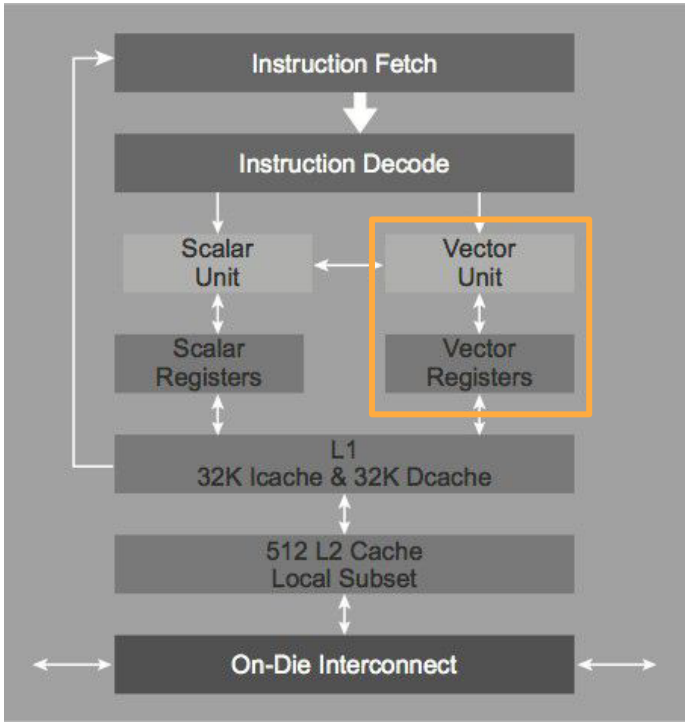
Multiple Levels of Parallelism

- Distributed Machines with their own local memory connected via network
- Multiple CPUs within a single compute node/
  Cores within a single CPU accessing the same local memory
- Data-level parallelism within the Cache of a single CPU Core
- Instruction level parallelism within a CPU Core

Vectorization

- hardware perspective happens when code makes use of hardware vector units
- developer perspective the process of turning scalar code into vectorized code
- form of data-level parallelism, where a single-thread operates on N array-elements simultaneously in **SIMD** fashion:
  Single **I**nstruction **M**ultiple **D**ata
- requires special hardware instructions

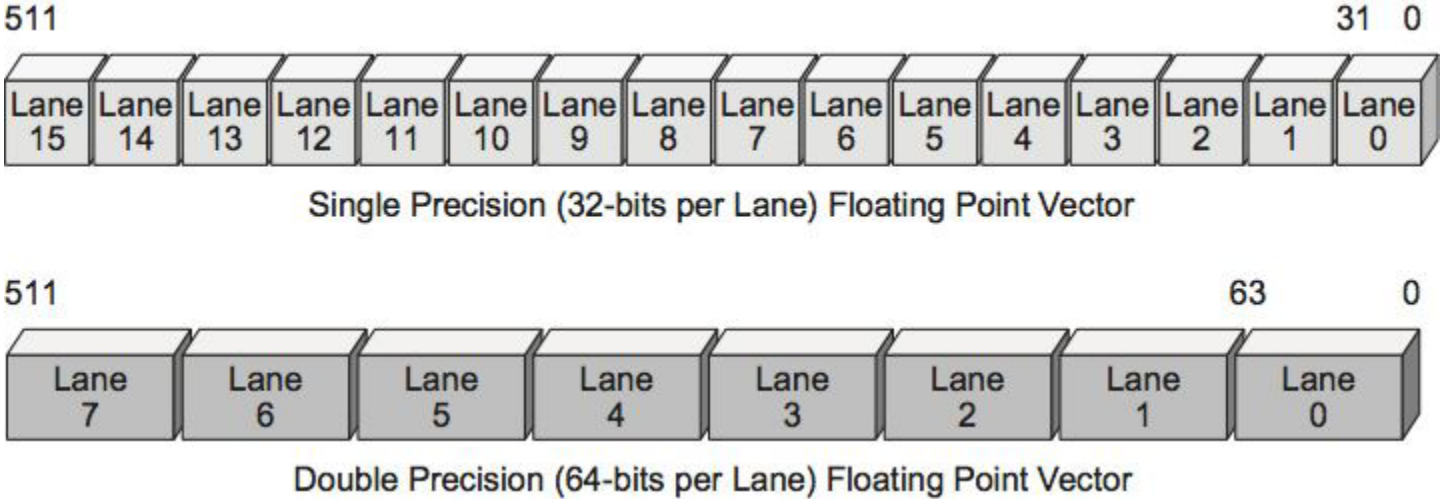
Current Vectorization Hardware



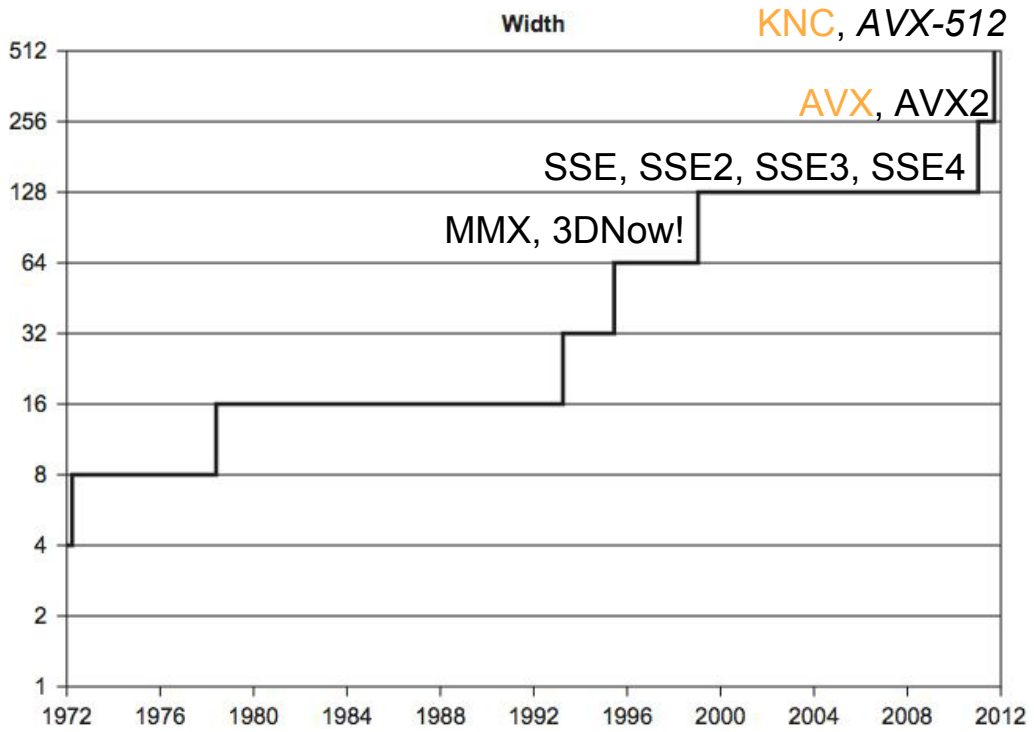
- 512-bit wide vector registers
- can accommodate 16/8 single/double-precision floats
- standard arithmetic, transcendental functions (exp, log)
- packed load/store

Vector Registers

A single register



History


Width
KNC, AVX-512
AVX, AVX2
SSE, SSE2, SSE3, SSE4
MMX, 3DNow!
512
256
128
64
32
16
8
4
2
1
1972
1976
1980
1984
1988
1992
1996
2000
2004
2008
2012

Why vectorization?

- Performance!
  vectorized instruction has a throughput of 1 instruction / cycle
  speedup: 2x (SSE), 4x (AVX), 8x (KNC)
- ROI (Return of Investment) / Performance p. $
  New CPUs are expensive due to more compute power → better use all of it
- Energy efficiency
  scalar execution leaves registers mostly empty
  power consumption is not that much lower though
  more instructions necessary for same work
  if speedup not critical: potential to underclock, lower voltage

So what? The compiler will do...

- not so easily
- Vectorization poses restrictions on data layout
- needs regular program structure (no heavy branching code)
- compiler conservatism (correctness first)
  usually language restrictions
  developer has to give hint the compiler

So how do we *vectorize* then?

- program in assembly → ultimate performance (potentially)
- not recommended

```
float a[1000];
float b[1000];
for (int i = 0; i < 1000; ++i) {
  // some more code to set rdx
  __asm__(
    vmovups a(,%rdx,4), %ymm0
    vaddps  b(,%rdx,4), %ymm0, %ymm1
    vmovups %ymm1, c(,%rdx,4)
  );
}
```

So how do we *vectorize* then?

- program in Intrinsics
- special compiler-header with function-level abstraction from assembly instructions

```
#include "immintrin.h"
float a[1024], b[1024], c[1024];
for (int i = 0; i < 1024; i+=16) {
  __m512 ar, br, cr;
  ar = _mm512_load_ps(a + i);
  br = _mm512_load_ps(b + i);
  cr = _mm512_add_ps(ar, br);
  _mm512_store_ps(c + i, cr);
}
```

Intrinsics Review

- easier to use than assembler
- can achieve good performance, BUT…
- large number of possible intrinsic functions to choose from → requires expertise to achieve good performance
- order of instructions dictated by user
- only 1(!) particular architecture

So how do we *vectorize* then?

- don't do it yourself 😉
- trust someone else who has done the *hard work* and use specialized libraries
- BLAS library for linear algebra
  e.g.: OpenBLAS, Intel MKL
- OpenCV for computer vision
- any other domain specific vectorized library

So you have to do it yourself...

- let the compiler figure it out
- portable code with future-ready performance when the next generation of CPUs arrive
- challenging for the compiler
- less expressive languages like C/C++ make it difficult
- can be done with the user (you!) assisting the compiler

Steps to vectorization

1. know what can be vectorized
   for-loops in C/C++
2. know where vectorization leads to speedup
3. evaluate compiler output
   vectorization-report
   assembly if necessary
4. evaluate performance - compare to theoretical speedup
5. know data access patterns
6. apply fixes : directives, flags, code changes

Vector-loop requirements

- countable at runtime
  number of iterations known before loop executes
  (doesn't change for the duration of the loop)
  no conditional termination (no breaks or for-loop conditions)
- regular (single) control flow
  no switch statements
  if statements only if they can be implemented as *masked assignments*
  (again: better check vectorization report)
- only the innermost nested-loop can be vectorized
  the compiler may reverse the loop order (if possible) as an optimization
- No function calls but basic math: sin(), cos(), exp(), ...

How to think of auto-vectorization

- loop-unrolling + packed instruction generation
- Load a(i .. i+3)
  Load b(i .. i+3)
  Load c(i .. i+3)
  c = a + b
  Store c(i .. i+3)
- never(!) unroll by hand

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

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

How to instruct the compiler

- Intel compiler:
  vectorization start at optimization level -O2
  will default to SSE → use -xhost
  use -qopt-report=5 -qopt-report-phase=vec to generate reports
- GCC
  vectorization is disabled by default
  activate via -ftree-vectorize, together with -O2 or -O3
  again: SSE by default, use -march=native
  use -ftree-vectorizer-verbose to generate reports

Challenge: loop dependencies

- compiler changes the order of computation compared to the sequential case
- the compiler must prove(!) that this will not change the program's semantics
- this is the reason why, in many cases, without the help of the programmer, the compiler will not vectorize, or not optimally
- the compiler has to perform dependency analysis

Loop-Dependency – Read After Write

```
float a[] = {0, 1, 2, 3, 4};
float b[] = {5, 6, 7, 8, 9};

for (std::int64_t i = 1; i < 5; ++i)
  a[i] = a[i - 1] + b[i];
```

- evaluating this position sequentially
  a[1] = 0 + 6;
  a[2] = 6 + 7;
  a[3] = 13 + 8;
  a[4] = 21 + 9;
  → a = {0, 6, 13, 21, 30};
- evaluating it vectorized

a[i - 1]	= {0, 1, 2, 3}	load
b[i]	= {6, 7, 8, 9}	load
a[i - 1] + b[i]	= {6, 8, 10, 12}	operate
a[i]	= {6, 8, 10, 12}	store

- {0, 6, 13, 21, 30} != {0, 6, 8, 10, 12}
  loop is not vectorizable
- also called flow-dependency

Loop-Dependency – Write After Read

- sequentially:
  a = {6, 8, 10, 12, 4}
- packed:
  a = {6, 8, 10, 12, 4}
- VECTORIZABLE

```
float a[] = {0, 1, 2, 3, 4};
float b[] = {5, 6, 7, 8, 9};

for (int i = 0; i < (5 - 1); ++i)
  a[i] = a[i + 1] + b[i];
```

Loop-Dependencies

- Read-after-Read (not really a dependency)
  a[i] = b[i % 2] + c[i]
  vectorizable
- Write-after-Write
  a[i%2] = b[i] + c[i]
  not vectorizable

Aliasing

- in C, pointers can hide data dependencies
- Is that safe?

```
void compute(const double* a,
             const double* b,
             double* c,
             int size) {
  for (int i = 0; i < size; ++i)
    c[i] = a[i] + b[i];
}
```

- not, if called by **compute**(a, b + 1, b)
- effectively, within compute `c` is actually `b - 1`
  → Read-After-Write Dependency

Aliasing and Compilers

- check report output: -vec-report=5
- sometimes compilers add bounds checking to cope with aliasing
- when a compiler can inline a function, the context might reveal there is no dependency
- if compiler finds a dependency that it cannot resolve, check if there is really a dependency
- compilers are very conservative
  they have to be → correctness first!

Resolving Dependencies

- sometimes a compiler cannot prove that there are no dependencies → assist!
- if you know there is no dependency between function arguments, add restrict keyword
- if you want to force vectorization (regardless off cost effectiveness/dependency checks) use #pragma omp simd (part of the OpenMP 4.0 standard)
- can lead to incorrect result if not careful!

Example

```
void compute(const double* restrict a,
             const double* restrict b,
             double* restrict c,
             int size) {
  // version relying on restrict keyword
  for (int i = 0; i < size; ++i)
    c[i] = a[i] + b[i];
  // version that doesn't need restrict and is portable
  #pragma omp simd
  for (int i = 0; i < size; ++i)
    c[i] = a[i] + b[i];
}
```

Caches (again...) and alignment

- caches are even more important when considering vectorization with wide vector units! -> data starvation
- data has to be fetched from main memory into caches before we can compute on them
- when they finally reached the L1 data cache, a vectorized load can be used to fill the vector registers to actually start computing
- vectorized loads come in 2 flavors: aligned and unaligned
- aligned loads have lower latency → good to keep enough data flowing into wide vector units

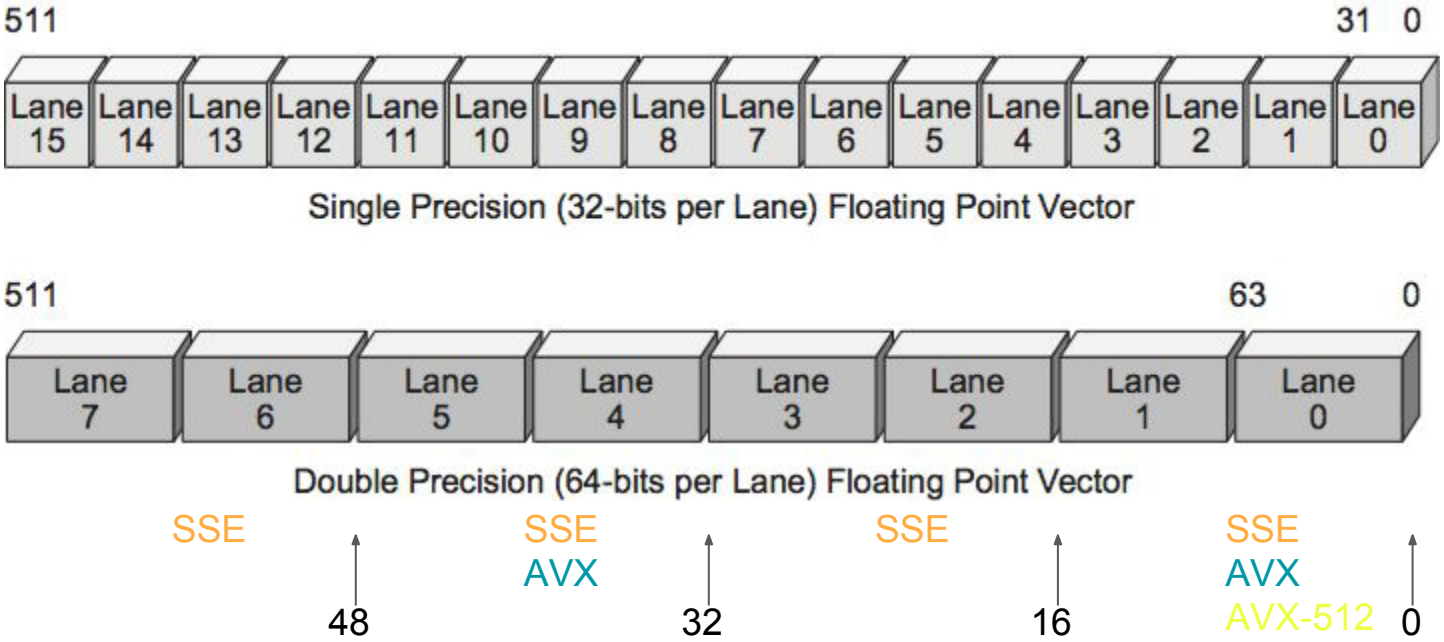
Alignment for vectorization

- especially important when using intrinsics/asm:
  when calling aligned loads on unaligned data your program will crash
- alignment is always of the size of the vector registers
  SSE on 16 byte boundary
  AVX on 32 byte boundary
  KNC on 64 byte boundary → cache line
- recommendation: always align to cache line boundary

Alignment on 512-bit wide registers

theoretical: 64 possible memory boundaries,

only 4 good for vectorization



How to align

```
// compiler-dependent, non-portable way
float a[100] __attribute__((aligned(64)));
// including boost-library header
#include <boost/align/aligned_allocator.hpp>
// align a std::vector
#include <vector>
// declare our 64-byte aligned allocator
template <typename T>
using aligned_alloc = boost::alignment::aligned_allocator<T, 64>;
// declare our aligned vector
template <typename T>
using aligned_vector = std::vector<T, aligned_alloc<T>>;
// allocate aligned memory, and use just like std::vector
aligned_vector<float> b(100);
```

Tell the compiler about it

- on static arrays within context the compiler can infer alignment
- not so on dynamic arrays
- also not on function parameters
- use #pragma omp simd aligned(a:64)
  this will tell the compiler to assume 64 byte alignment of “a” within the subsequent loop
  part of the OpenMP 4.0 standard
- only Intel compiler:
  __assume_aligned(a, 64);

What happens without alignment

- slow unaligned loads
- compiler generation of peel loops
  dynamically skip the beginning of the array in a separate unaligned loop
- subsequent elements are then processed in
- vectorized main loop
- if the total size of the array is unknown at compile time,
  the compiler will also generate a remainder loop (often not avoidable)