# Vectorization

Paul Springer

Aachen Institute for Advanced Study in
Computational Engineering Science

Aachen, 20.04.14



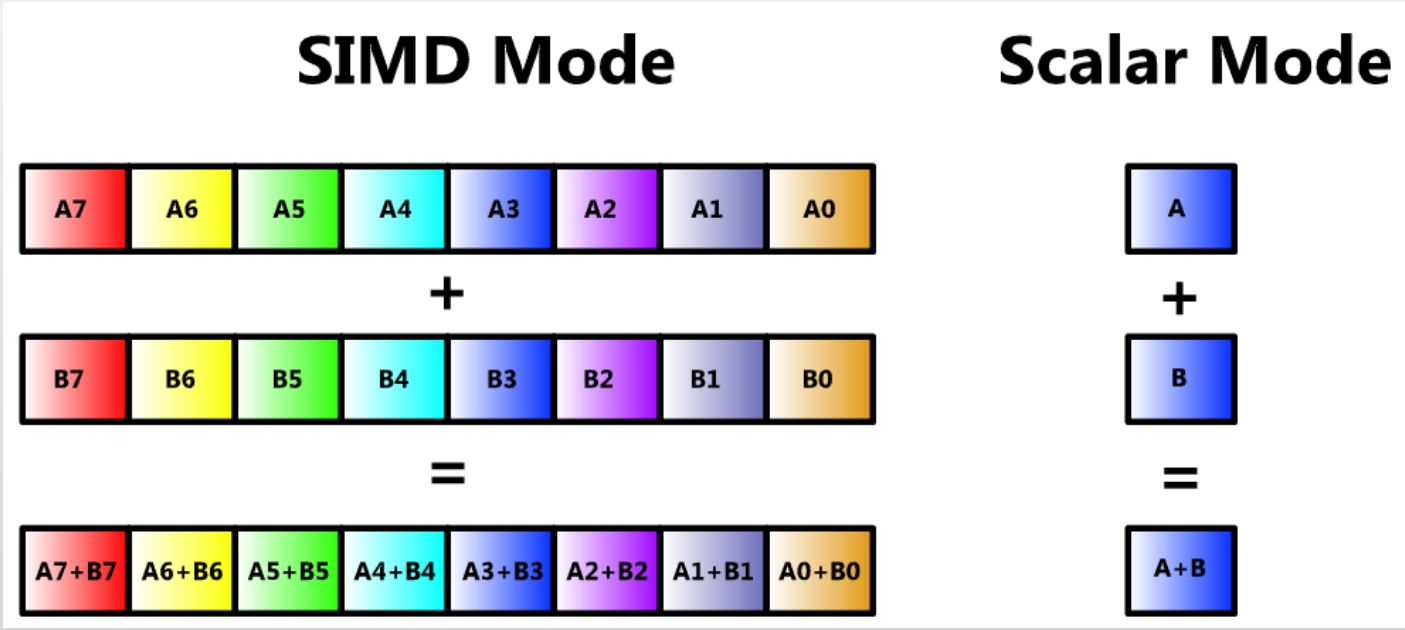
# Outline

# Outline

# Resources

- `software.intel.com/sites/landingpage/IntrinsicsGuide/`
- Introduction to Intel Advanced Vector Extensions by Chris Lomont
- A Guide to Vectorization with Intel C++ Compilers

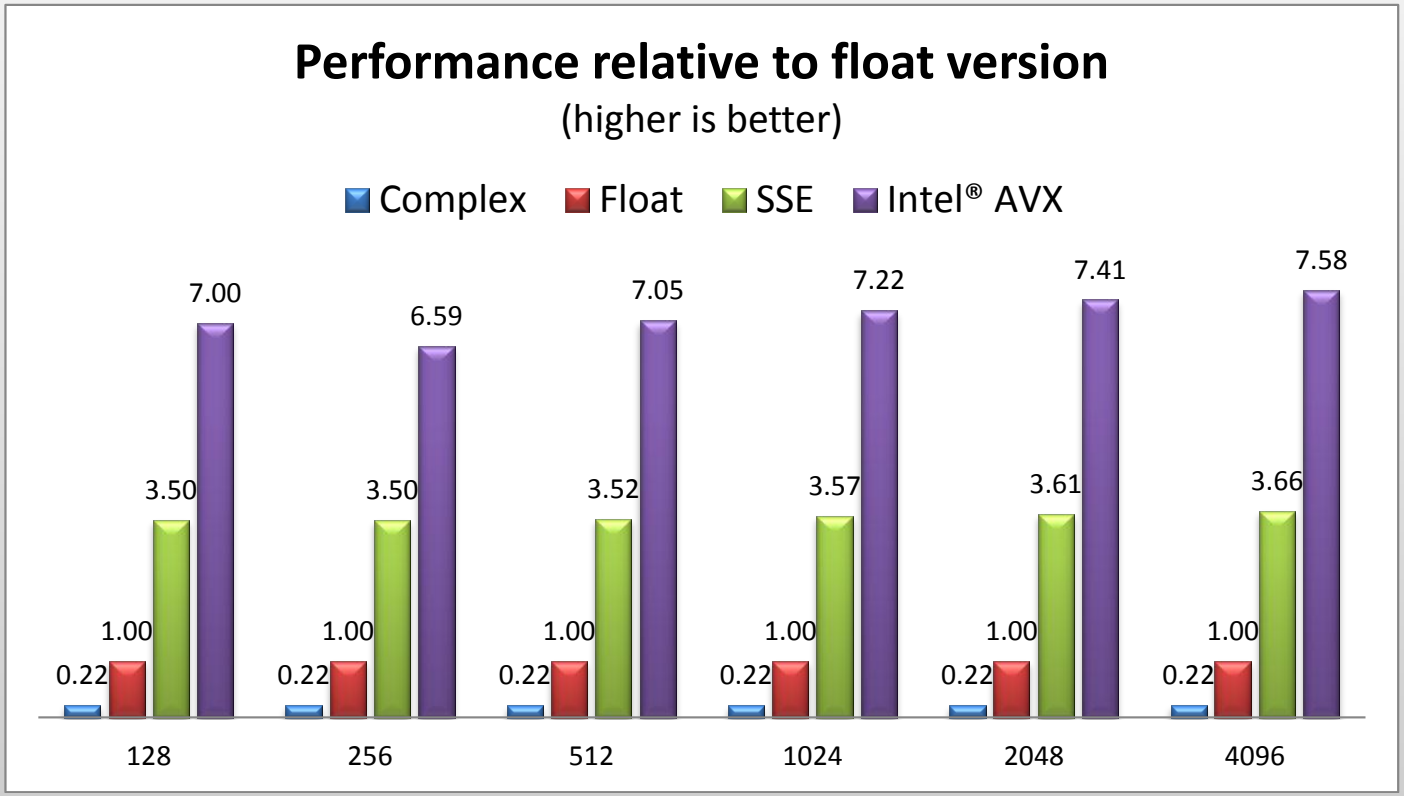
# Outline



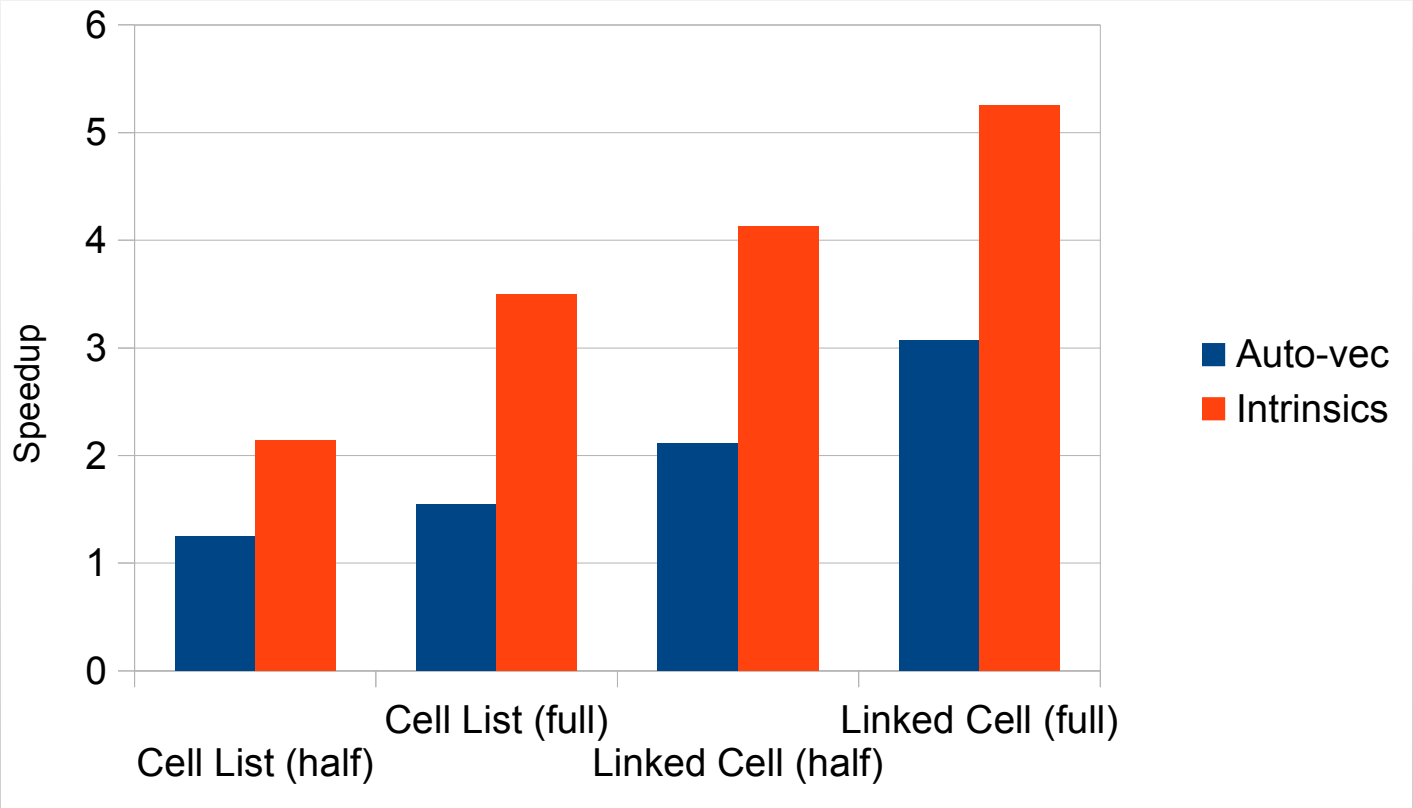

- Theoretical[1] speedup of 8x over scalar version

[1] For SP using AVX

# Single Instruction Multiple Data



# Single Instruction Multiple Data



Frequency wall + Memory wall + Energy wall

=

| | |
|---|---|
| 1997 | Intel MMX |
| 1998 | AMD 3DNow! |
| 1999 | Streaming SIMD Extensions (SSE) |
| 2001 | Streaming SIMD Extensions 2 (SSE2) |
| 2003 | Streaming SIMD Extensions 3 (SSE3) |
| 2006 | Streaming SIMD Extensions 4 (SSE4) |
| 2011 | Advanced Vector Instruction (AVX) |
| 2013 | Advanced Vector Instruction 2 (AVX2) |
| (2015 | AVX-512) |

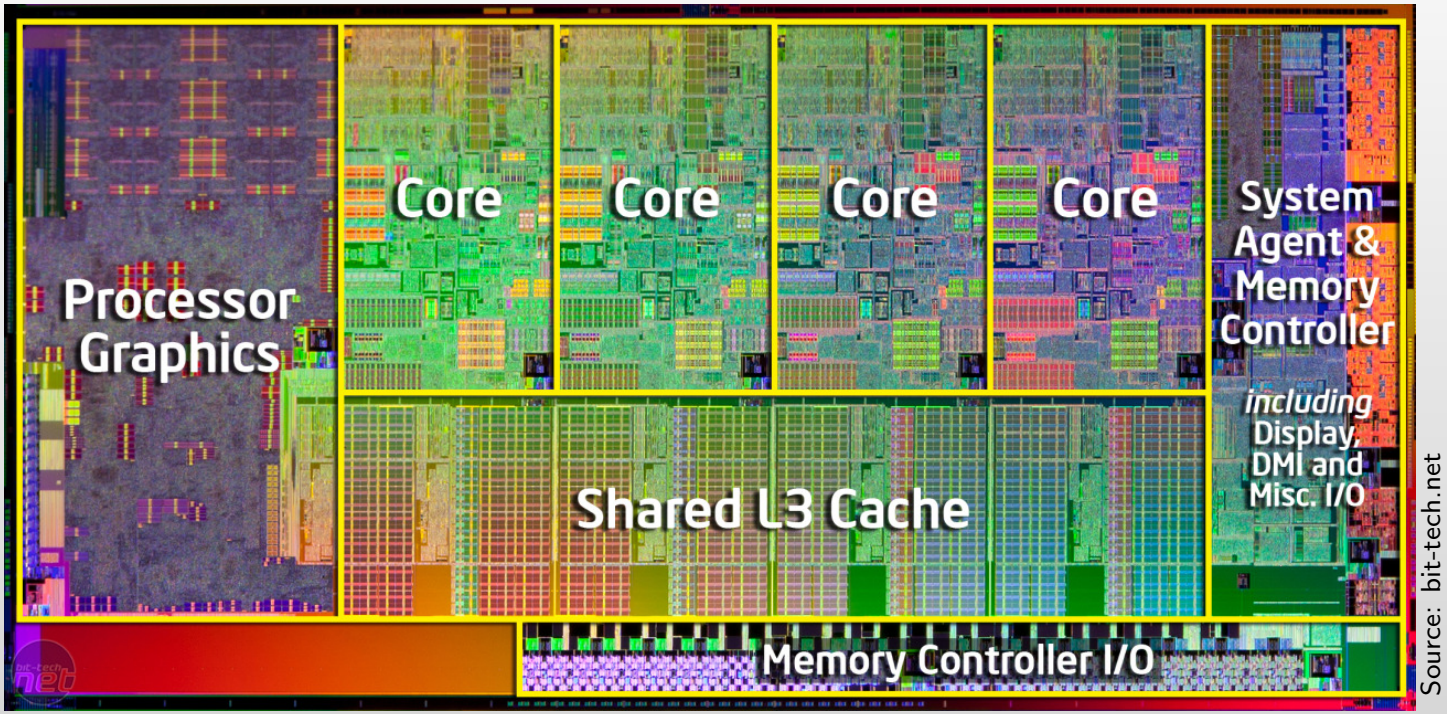



Figure: Intel Sandy Bridge Die.

- 32nm feature size
- Over 2 billion transistors (vs Nvidia Kepler 7.1B)

# CPU Execution Units

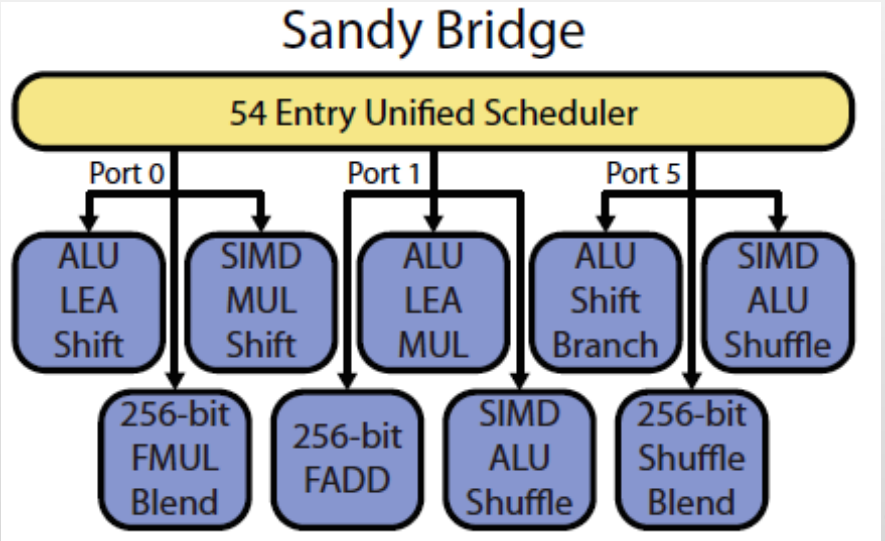



Figure: Intel Sandy Bridge Execution Units.

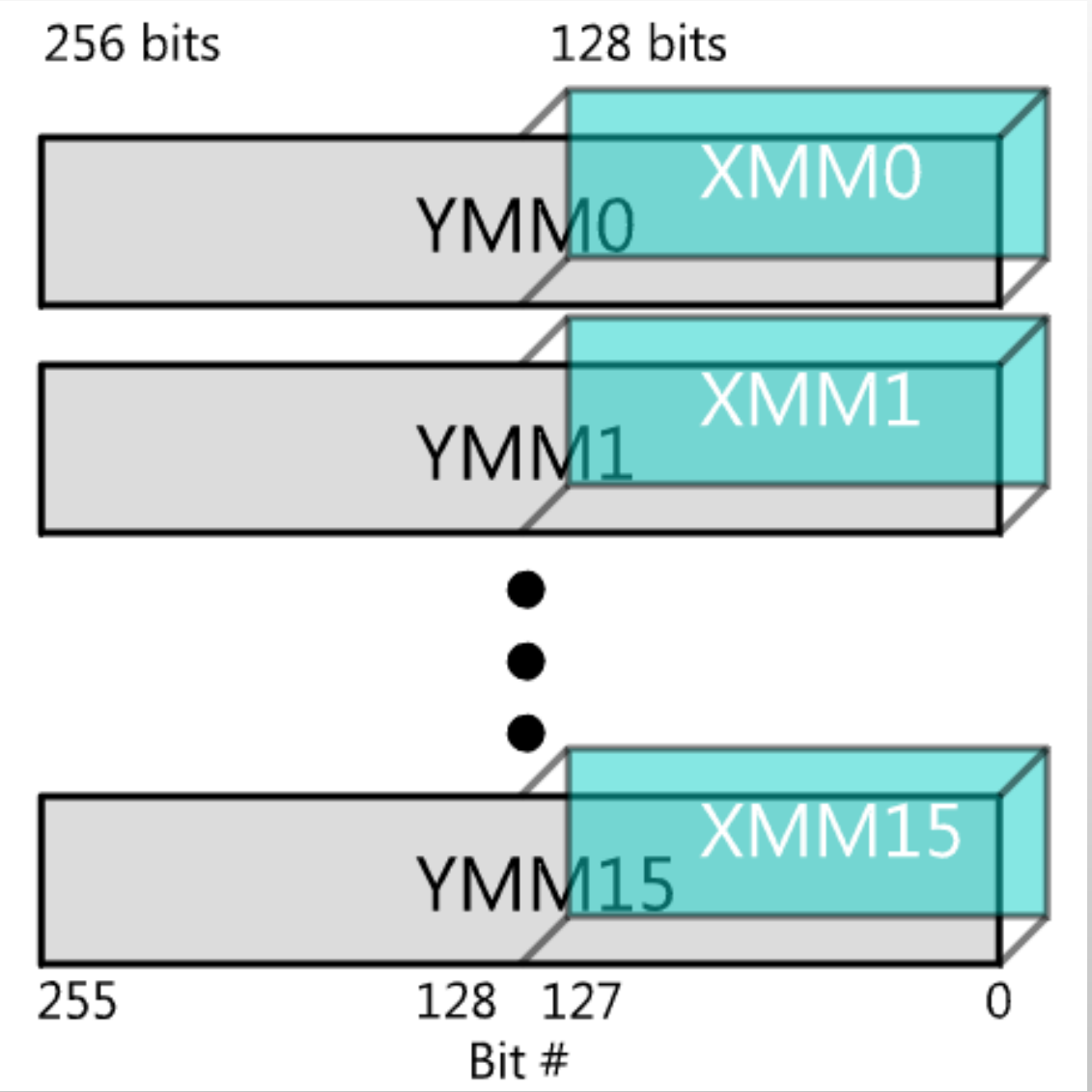
- Sustain 16 SP or 8 DP FP operations per cycles
- 1x Mul, 1x Add and 1x shuffle per cycle

# Outline

# Advanced Vector Extension

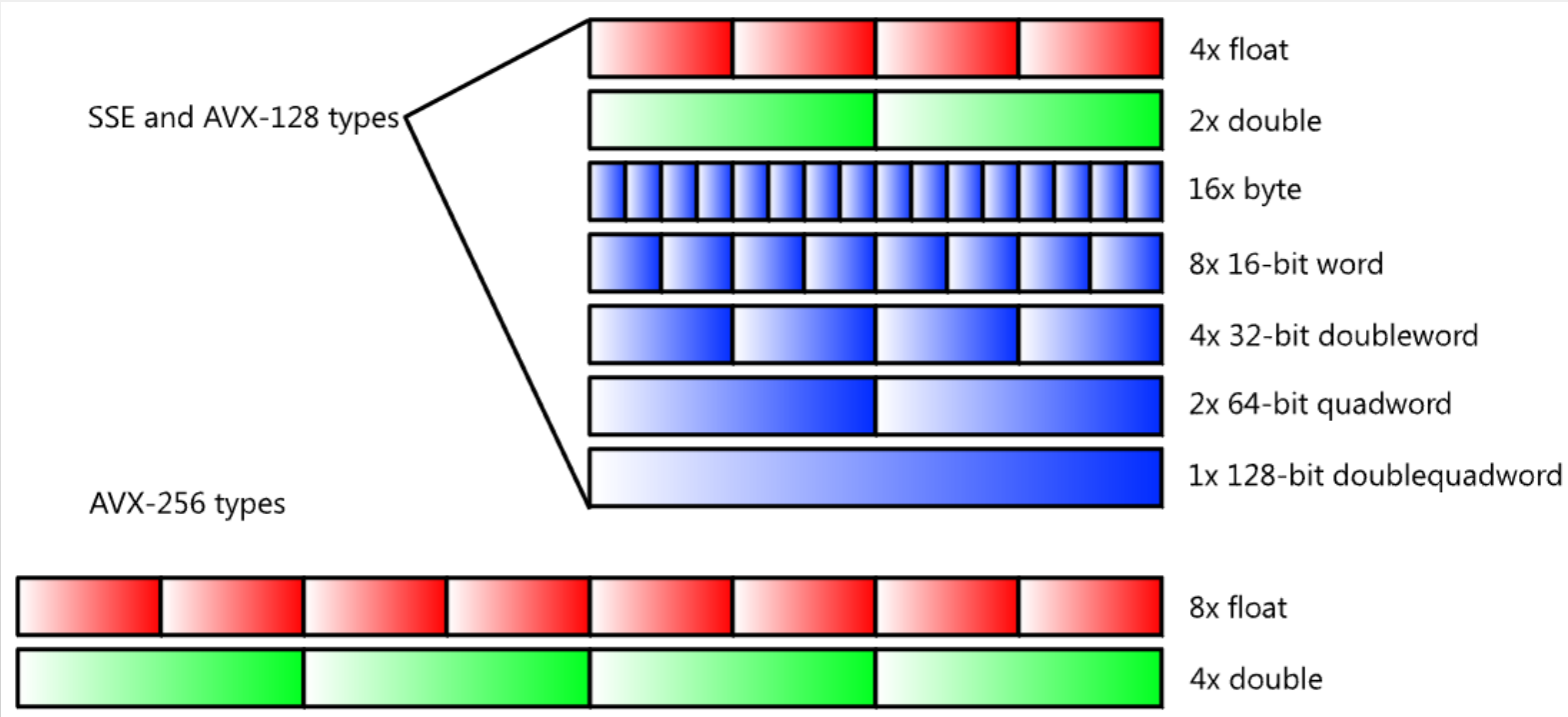
- Introduced in 2011
  - Intel Sandy Bridge
  - AMD Bulldozer
- Increased vector width to 256 bit
- 16 vector registers
- Three operand SIMD instructions
- New instructions
  - VBROADCASTSS
  - VPERM2F128
  - ...

# Advanced Vector Extension

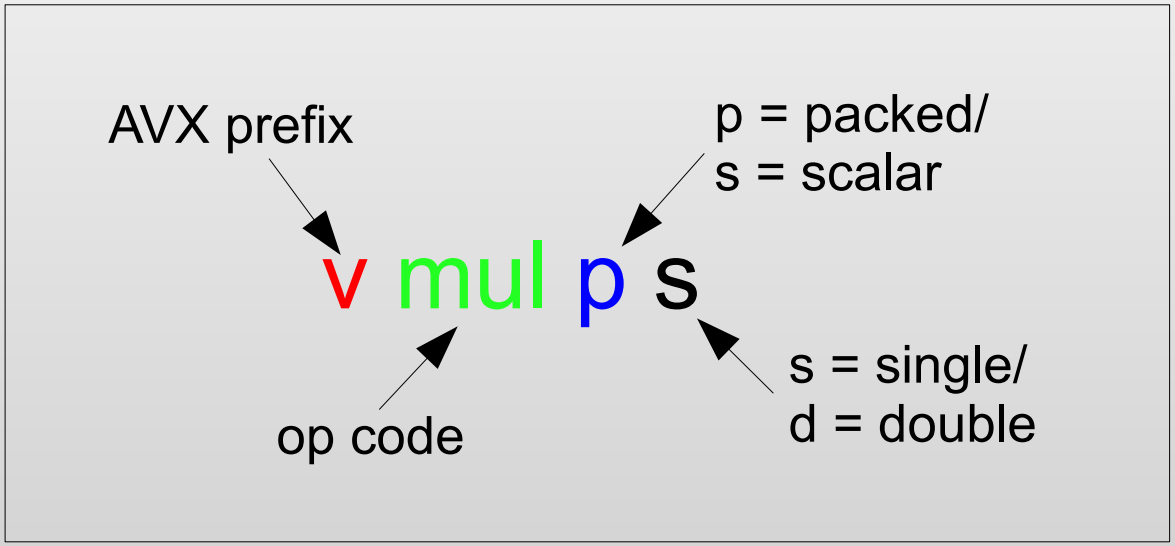



© Chris Lomont

# Advanced Vector Extension




© Chris Lomont


AVX prefix
p = packed/
s = scalar
v mul p s
op code
s = single/
d = double

# Outline

# Auto-Vectorization with ICC

- Auto-vec is activated by default
- Optional compiler flags:
  - *-vec-report3*
  - *-xAVX*
- Implicit vectorization

Let the compiler do the work for you.

# Exercise

# Limitations

- Only vectorizes innermost loop
- Pay attention to the following guidelines:
  - Avoid divergence based on the loop counter
  - Avoid branching
  - Use single-entry, single-exit loops
  - Avoid non-contiguous and indirect memory accesses
  - Avoid inter-loop dependencies
  - No function calls allowed[2]
  - Align memory to 32 byte boundaries
  - Often Structure of Arrays is preferable over Array of Structures

[2] except for two exceptions

# SoA vs AoS



AoS: X Y Z T X Y Z T X Y Z T ...

SoA:

| | | | |
|---|---|---|---|
| X | X | X | X |
| Y | Y | Y | Y |
| Z | Z | Z | Z |
| T | T | T | T |

...

| SoA | AoS |
|---|---|
| Vectorization friendly if continuous memory accesses | Better cache behaviour than SoA if random memory accesses |
| Better cache behaviour if only single elements of the structure are used | Programmer friendly |

# Alignment

- `__attribute__((aligned(32)))`
  - e.g.: `float x[128] __attribute__((aligned(32)))`
- `void* _mm_malloc (size_t size, size_t align )`
- `void _mm_free (void *p)`
- `#pragma vector aligned`

| Compiler Hint | Description |
|---|---|
| `#pragma ivdep` | Ignore potential (unproven) data dep. |
| `#pragma vector always` | Override efficiency heuristic |
| `#pragma vector nontemporal` | Hint to use streaming stores |
| `#pragma vector [un]aligned` | assert [un]aligned property |
| `#pragma novector` | disable vectorization for this loop |
| `#pragma loop count (<int>)` | Hint for likely trip count |