



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Programming

Prof. Marco Bertini



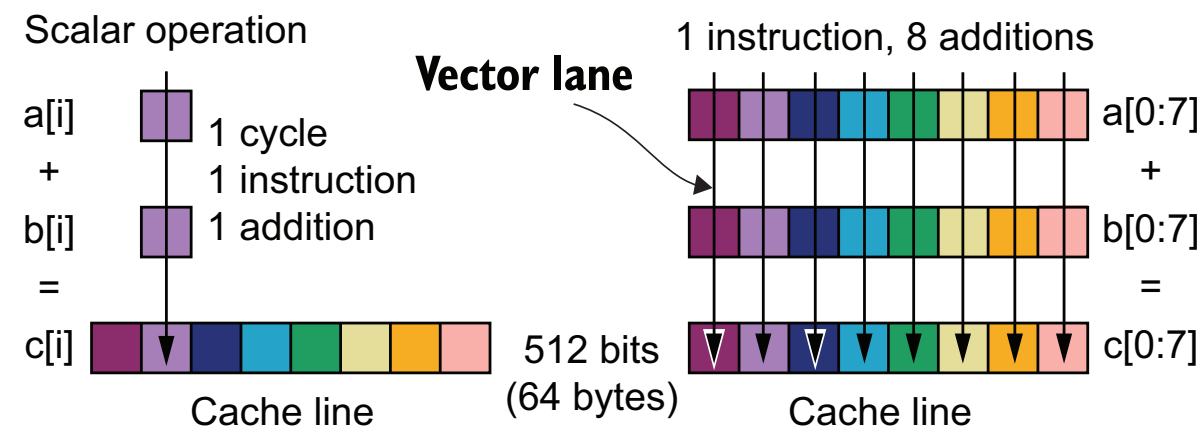
UNIVERSITÀ
DEGLI STUDI
FIRENZE

Vectorization



Vectorization and SIMD

- SIMD: single instruction that is executed across multiple data streams.
- One vector add instruction replaces eight individual scalar add instructions in the instruction queue
 - reduced pressure on the instruction queue and cache.
- The biggest benefit is that it takes about the same power to perform eight additions in a vector unit as one scalar addition.

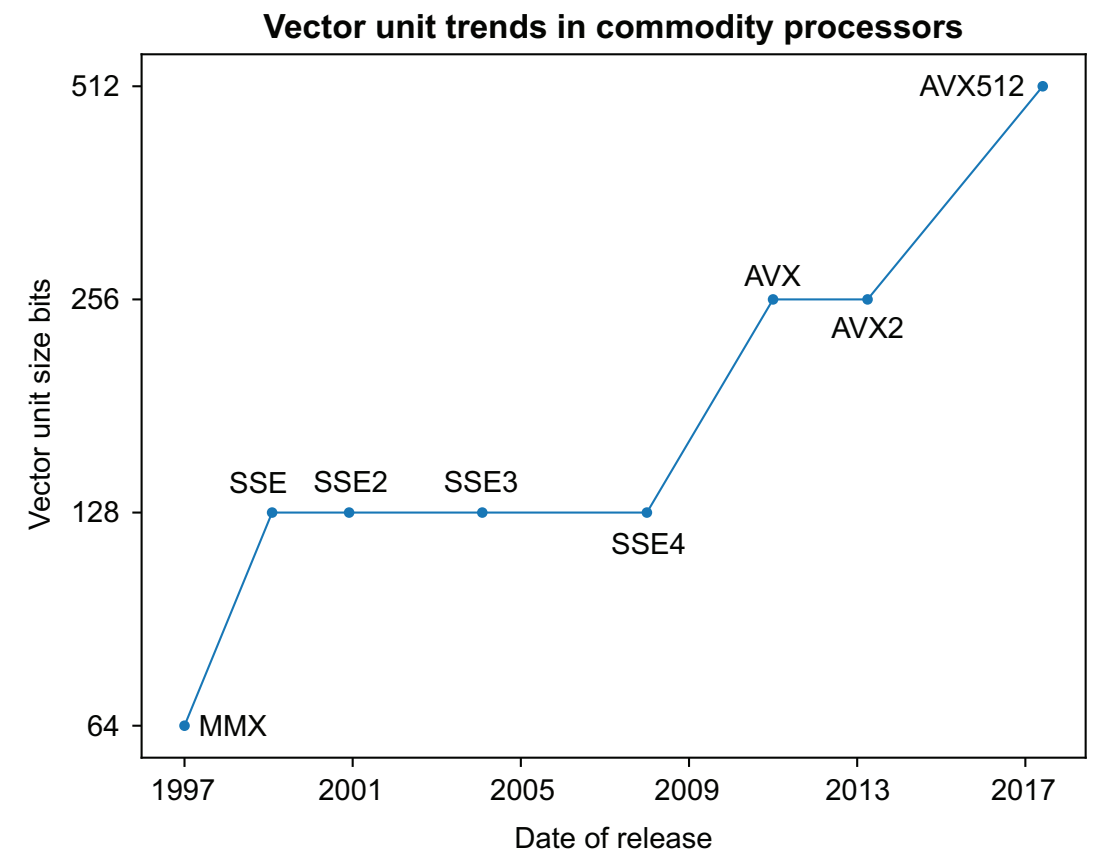


Vectorization: **h/w** & **s/w**

- Generate instructions: the vector instructions must be generated by the compiler or manually specified through intrinsics or assembler coding.
- Use recent compilers
- Match instructions to the vector unit of the processor: newer hardware can usually process older SIMD instructions but older h/w fails with newer ones
- Use recent CPUs (i.e. 5-7 years old is enough!)

Vectorization: h/w

- MMX: first Intel SIMD instruction set
- SSE (Streaming SIMD Extensions): First Intel vector unit to offer floating-point operations with single-precision support
- SSE2: Double precision instructions
- **AVX (Advanced Vector Extensions)**: double vector length. AMD added a fused multiply-add FMA vector instruction in its CPUs, doubling the performance for some loops.
- AVX2: Intel added FMA
- AVX512: increased vector size (bits)





How to vectorize


- Use optimized libraries
- Auto-vectorization
- Hints to the compiler
- Vector intrinsics
- Manual coding using assembler instructions



Optimized libraries

- **Easy approach**: use tested low-level libraries 
- BLAS (Basic Linear Algebra System): a base component of high-performance linear algebra software
- LAPACK: A linear algebra package 
- SCALAPACK: A scalable linear algebra package
- SIMD JSON: parse JSON files
- Intel Math Kernel Library (MKL): optimized versions of the BLAS, LAPACK, SCALAPACK, **FFTs**, sparse solvers, and mathematical functions for Intel processors.
- etc...

Auto-vectorization

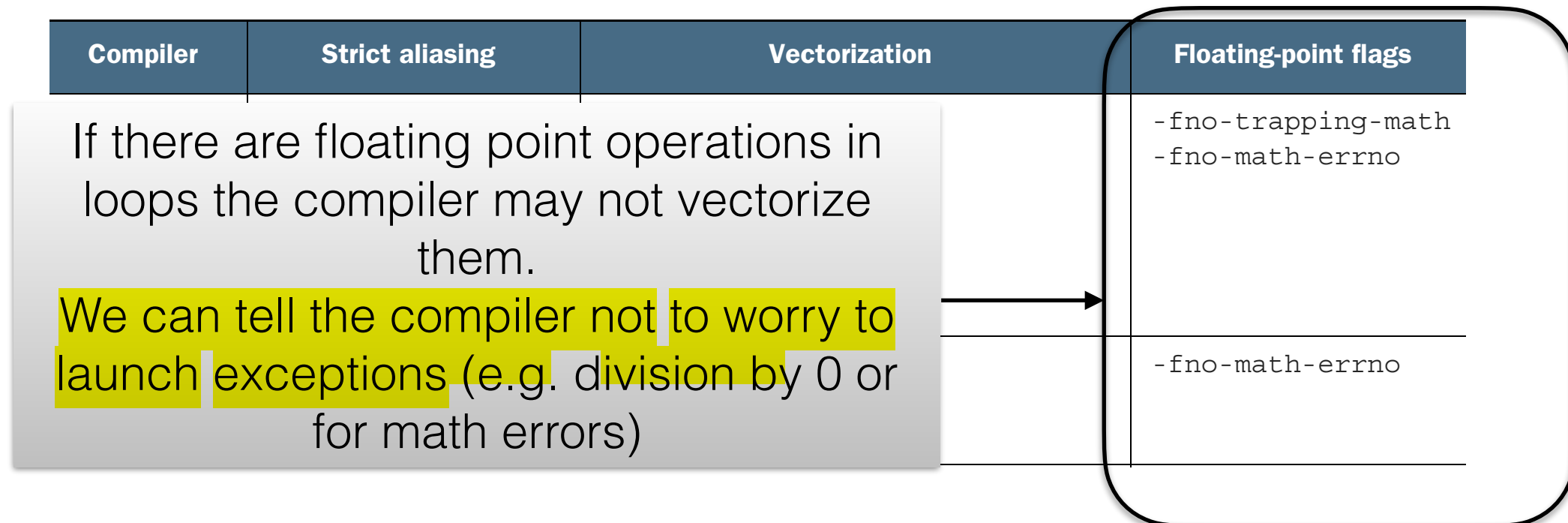
- Auto-vectorization is the vectorization of the source code by the compiler for standard C, C++, or Fortran languages.
- 
- Least amount of programming effort
 - It's up to the compiler to recognize what can be vectorized
 - Help providing some hints in code or using compiler options

Auto-vectorization: compiler flags

| Compiler | Strict aliasing | Vectorization | Floating-point flags |
|--------------------------|--------------------------------|---|---|
| GCC, G++, GFortran v9 | <code>-fstrict-aliasing</code> | <code>-ftree-vectorize</code> <code>-march=native</code> <code>-mtune=native</code> ver 8.0+: <code>-mprefer-vector</code> <code>-width=512</code> | <code>-fno-trapping-math</code> <code>-fno-math-errno</code> |
| Clang v9 | <code>-fstrict-aliasing</code> | <code>-fvectorize</code> <code>-march=native</code> <code>-mtune=native</code> | <code>-fno-math-errno</code> |

- `march=native` tells the compiler to use the SIMD instruction set of the CPU used for compiling. It's possible to specify SIMD instruction sets.
- Depending on the compiler vectorization may be on by default for certain optimization levels (e.g. `-O3` for GCC) or always on (e.g. MSVC)
 - It's possible to turn off vectorization to test results, to be sure that vectorization does not introduce errors
- There are many more flags: RTFM !

Auto-vectorization: compiler flags




- march=native tells the compiler to use the SIMD instruction set of the CPU used for compiling. It's possible to specify SIMD instruction sets.
- Depending on the compiler vectorization may be on by default for certain optimization levels (e.g. -O3 for GCC) or always on (e.g. MSVC)
 - It's possible to turn off vectorization to test results, to be sure that vectorization does not introduce errors
- There are many more flags: RTFM !

Auto-vectorization: reports

| Compiler | Vectorization report |
|--------------------------|---|
| GCC, G++, GFortran v9 | <code>-fopt-info-vec-optimized[=file]</code> <code>-fopt-info-vec-missed[=file]</code> <code>-fopt-info-vec-all[=file]</code> Same for loop optimizations replace vec with loop |
| Clang v9 | <code>-Rpass-analysis=loop-vectorize</code> |

- Check your compiler manual to get the flags needed to activate **vectorization reports**, to understand what was vectorized
- E.g.: <https://llvm.org/docs/Vectorizers.html>

Pointer aliasing

- Aliasing is where pointers point to overlapping regions of memory.
- In this situation, the compiler cannot tell if it is the same memory, and it would be unsafe to generate vectorized code or other optimizations. 



Aliasing

```
#define STREAM_ARRAY_SIZE 80000000
```

```
double a[STREAM_ARRAY_SIZE];
```

```
double b[STREAM_ARRAY_SIZE];
```

```
double c[STREAM_ARRAY_SIZE];
```

```
for (int i=0; i<STREAM_ARRAY_SIZE; i++){
```

```
    c[i] = a[i] + scalar*b[i];
```

```
}
```

This code is easily vectorized
The compiler perfectly understands
the arrays it's operating on

Aliasing

```
#define STREAM_ARRAY_SIZE 80000000
```

```
double a[STREAM_ARRAY_SIZE];  
double b[STREAM_ARRAY_SIZE];  
double c[STREAM_ARRAY_SIZE];
```

```
void stream_triad(double* a, double* b, double* c, double scalar){
```

```
    for (int i=0; i<STREAM_ARRAY_SIZE; i++){
```

```
        c[i] = a[i] + scalar*b[i];
```

```
    }
```


```
}
```

```
double scalar = 3.0;  
stream_triad(a, b, c, scalar);
```



- This code is more critical.
- The compiler cannot tell if the arguments to the function point to the same or to overlapping data.
- This causes the compiler to create more than one version and produces code that tests the arguments to determine which to use.

Aliasing

- To tell the compiler that the pointer arguments of a function do **not point to overlapping** data use
 - C99: restrict attribute as part of the arguments:
`void stream_triad(double* restrict a,
double* restrict b, double* restrict c,
double scalar)`
 - C++: no standard restrict keyword, but different compilers have extensions like `__restrict` (GCC and Clang)
- By using the `restrict` attribute, you make a promise to the compiler that there is no aliasing. 

Aliasing

- We can tell the compiler to aggressively generate code with the assumption that there is no aliasing.
- In recent years, the strict aliasing option has become the default with GCC and other compilers (optimization levels -O2 and -O3 set -fstrict-aliasing).
This broke a lot of code where aliased variables actually existed.
As a result, compilers have dialed back how aggressively they generate more efficient code.
- Thus, check the manual of your compiler...


Use both the `restrict` attribute and the `-fstrict-aliasing` compiler flag. The attribute is portable with the source across all architectures and compilers. Instead, you'll need to apply the compiler flags for each compiler.

- We can tell the compiler to aggressively generate code with the assumption that there is no aliasing.
- In recent years, the strict aliasing option has become the default with GCC and other compilers (optimization levels `-O2` and `-O3` set `-fstrict-aliasing`). This broke a lot of code where aliased variables actually existed. As a result, compilers have dialed back how aggressively they generate more efficient code.
- Thus, check the manual of your compiler...

Hints to the compiler

- A pragma is an instruction to a C or C++ compiler to help it interpret the source code. The form of the instruction is a preprocessor statement starting with `#pragma`.
- We can use pragmas to provide suggestions to the compiler to vectorize code (if it can't auto vectorize it)
 - Check the vectorization reports to see if some loop was not vectorized, e.g. due to a condition in the loop code

Hints to the compiler

- `#pragma clang loop` directive allows loop vectorization
- `#pragma simd` is used by Intel compiler
- Other pragmas depend on the compiler
- Use: `#pragma omp simd` for portability, thanks to the common support of OpenMP by compilers 

Hints to the compiler: example

```
#pragma omp simd
for (i = 0; i < count; i++) {
    a[i] = a[i-1] + 1; // carried dep.
    b[i] = *c + 1;
    bar(i);
}
```

- The loop can not be parallelized because of the dependence on `a[]` in the first instructions but the following instructions can be vectorized

Programming styles for vectorization: general and data structures

- Use the **restrict** attribute on pointers in function arguments and declarations (C and C++).
- Use **pragmas or** directives where needed **to inform the compiler** (check the reports)
- Try to use a data structure with a long length **for the innermost loop.**
- Use the **smallest data type needed** (short rather than int).
- Use contiguous memory accesses. Some newer instruction sets implement gather/scatter memory loads, but these are less efficient.
- Use **Structure of Arrays (SOA)** rather than Array of Structures (AOS).
- Use **memory-aligned data structures** where possible.

Programming styles for vectorization: loop structures and body

- Use simple loops without special exit conditions.
- Use the loop index for array addresses when possible.
- Expose the loop bound size so it is known to the compiler. If the loop short, the compiler might unroll it rather than vectorizing it.
- Define local variables within a loop so that it is clear that these are not carried to subsequent iterations (C and C++).
- Variables and arrays within a loop should be write-only or read-only (except for reductions).
- Don't reuse local variables for a different purpose in the loop—create a new variable.
- Avoid function calls and inline instead (manually or with the compiler).
- Limit conditionals within the loop and, where necessary, use simple forms that can be masked (i.e. the compiler inserts a mask that uses only part of the vector results).

Books

- Parallel and High Performance Computing, R. Robey, Y. Zamora, Manning - Chapt. 6