

# Vector Parallelism on Multi-Core Processors

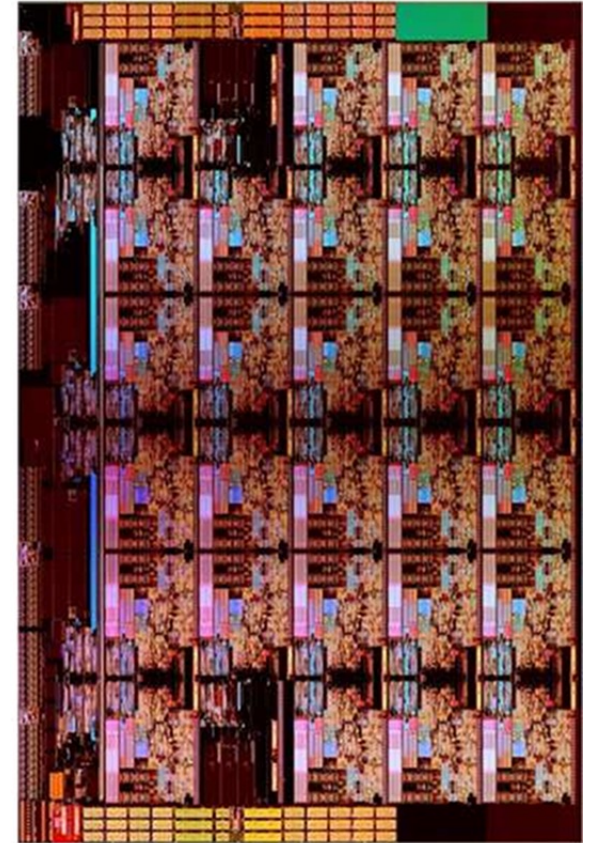
Steve Lantz, Cornell University

*CoDaS-HEP Summer School, August 1, 2022*



# Vector Parallelism: Motivation

- CPUs are no faster in GHz than they were 15 years ago
  - Power limits! “Slow” transistors are more efficient, cooler
- Yet process improvements have made CPUs denser
  - Moore’s Law! Add 2x more “stuff” every 18–24 months
- One way to use extra transistors: **more cores**
  - Dual-core Intel chips arrived in 2005; counts keep growing
  - Up to 40 in Intel Xeon “Ice Lake”, 64 in AMD EPYC “Milan”
- Another solution: **SIMD or vector operations**
  - First appeared on Intel Pentium with MMX in 1996
  - Vectors have ballooned: 512 bits (16 floats) in Intel Xeon
  - Can *vectorization* increase speed by an order of magnitude?

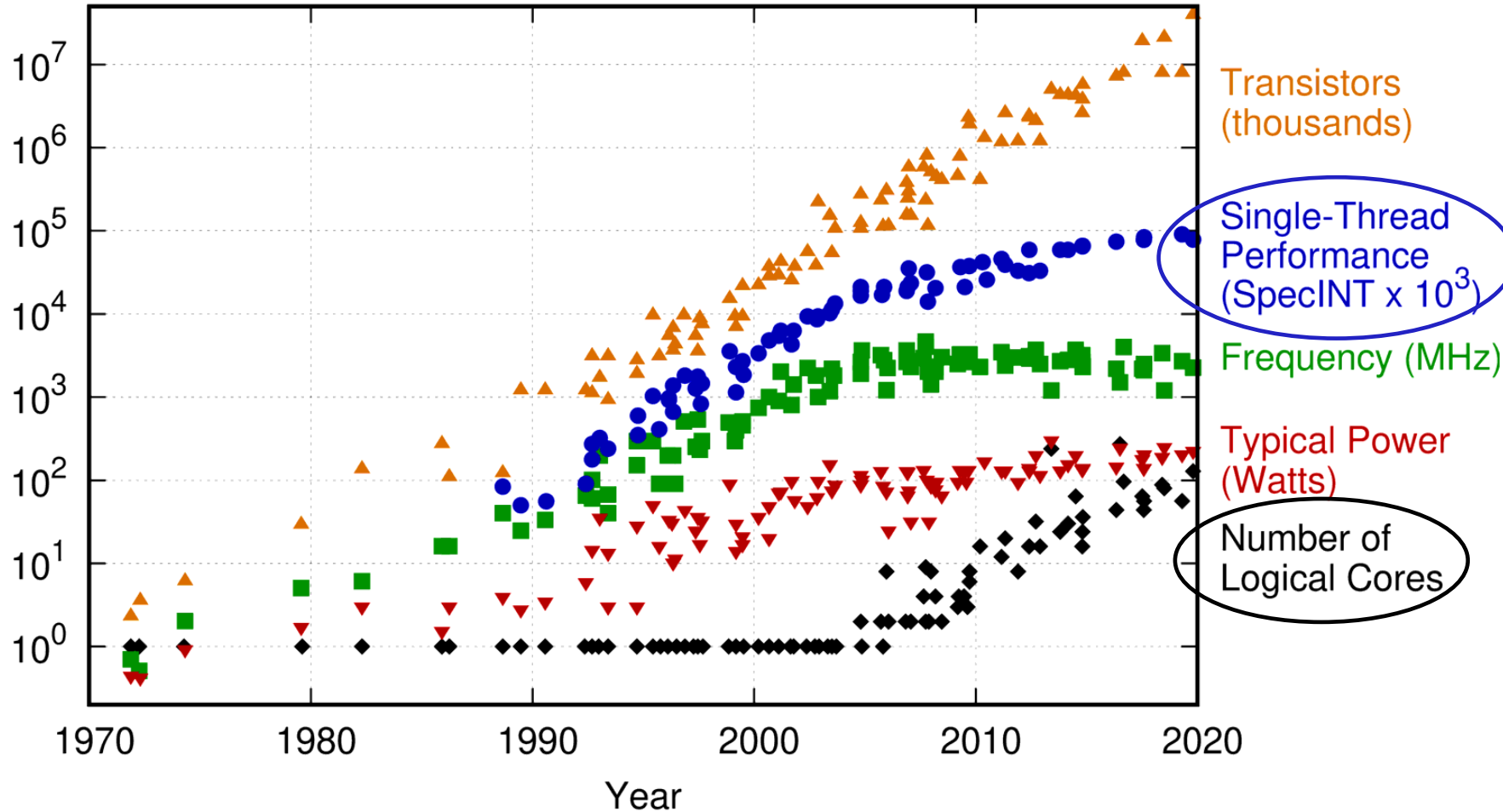


Die shot of 28-core Intel Skylake-SP  
Source: wikichip.org



# What Moore's Law Buys Us, These Days...

48 Years of Microprocessor Trend Data

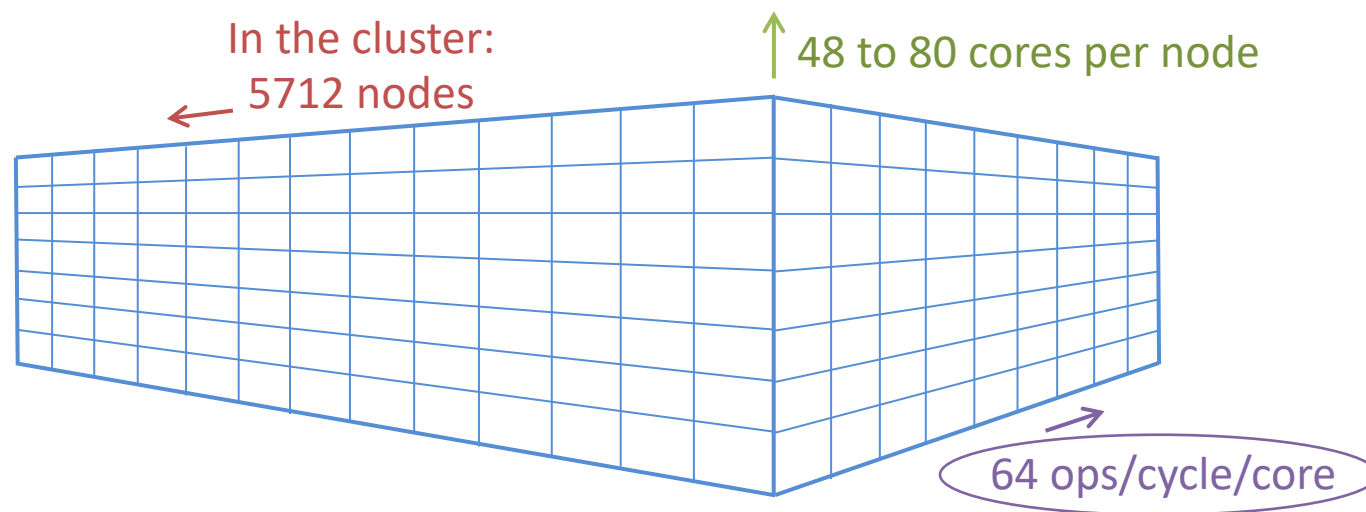


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2019 by K. Rupp [GitHub link](#)



# A Third Dimension of Scaling

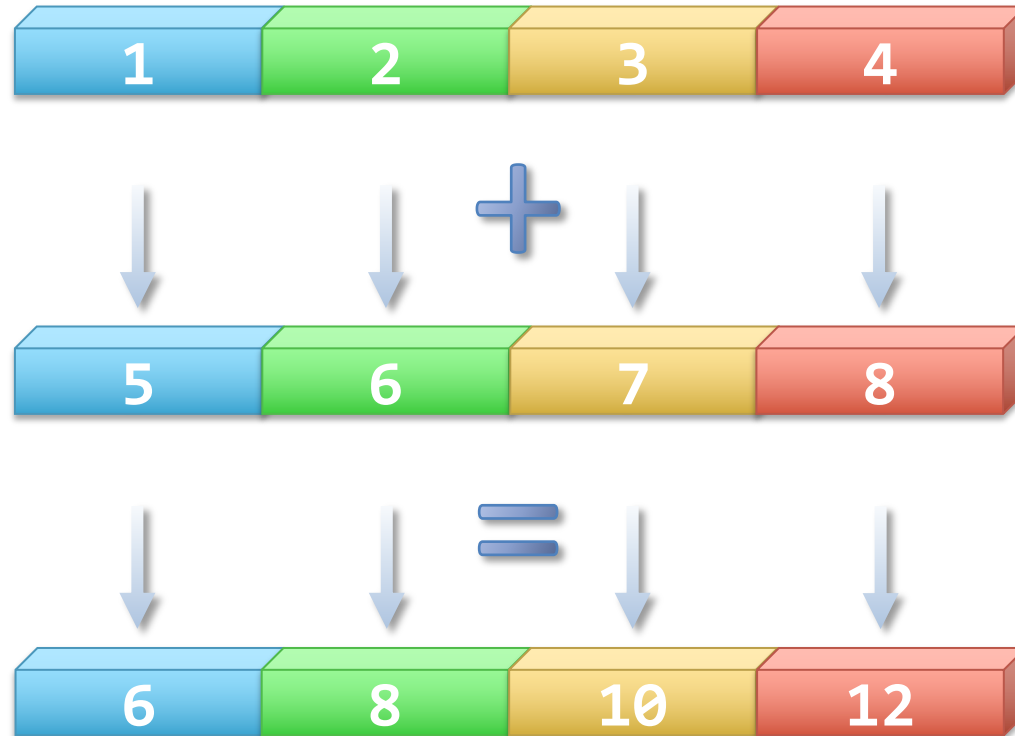
- Along with scaling *out* and *up*, you can “*scale deep*”
  - Arguably, vectorization may be as important as multithreading
- Example: Intel processors in TACC Stampede2 cluster



- 1960 Xeon (SKX, ICX) + 3752 Xeon Phi (KNL) nodes; 48–80 cores
- Each core does up to **64 operations/cycle** on vectors of 16 floats



# How It Works, Conceptually

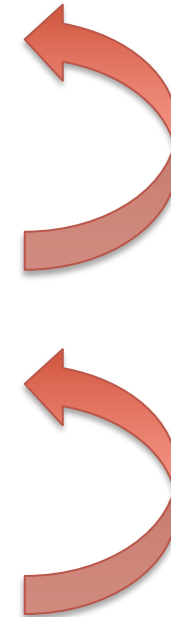


SIMD: Single Instruction, Multiple Data



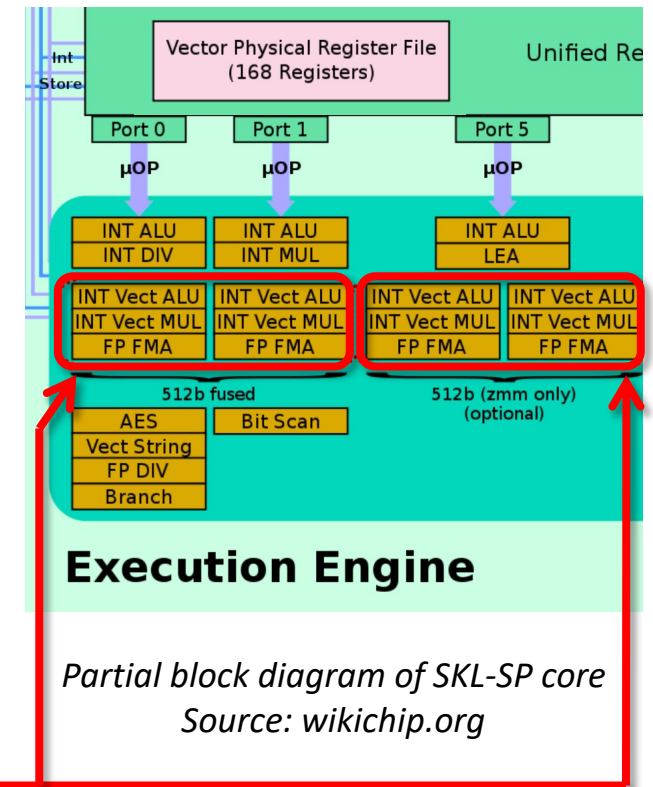
# Three Ways to Look at Vectorization

1. **Hardware Perspective**: Run vector instructions involving special registers and functional units that allow in-core parallelism for operations on arrays (vectors) of data.
2. **Compiler Perspective**: Determine how and when it is possible to express computations in terms of vector instructions.
3. **User Perspective**: Determine how to write code with SIMD in mind; e.g., in a way that allows the compiler to deduce that vectorization is possible.

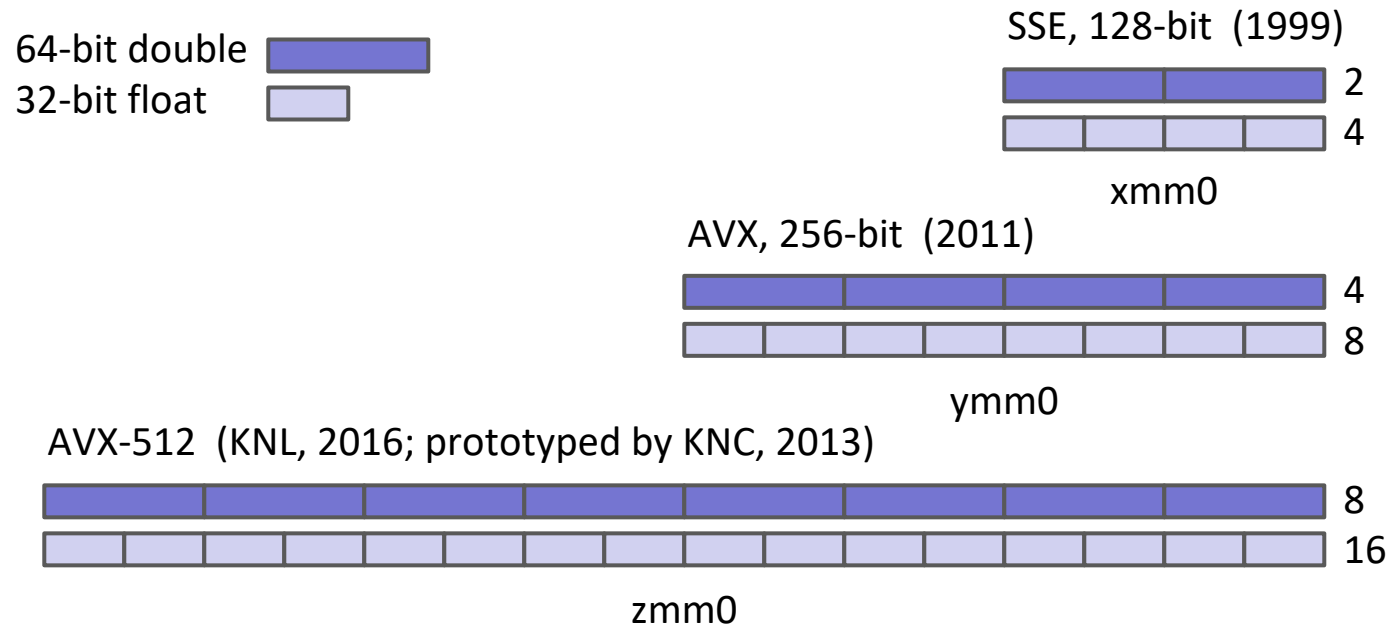


# Hardware Perspective

- SIMD = Single Instruction, Multiple Data
  - Part of commodity CPUs (x86, x64, PowerPC) since late '90s
- Goal: parallelize computations on vector arrays
  - Line up operands, execute one op on all simultaneously
- SIMD instructions have gotten speedier over time
  - Initially: several cycles for execution on small vectors
  - Intel AVX introduced pipelining of some SIMD instructions
  - Now: multiply-and-add large vectors on every cycle
- Intel's latest: Skylake-SP, Cascade Lake, Ice Lake...
  - 2 VPU (vector processing units) per core, in most models
  - 2 ops/VPU if they do FMAs (Fused Multiply-Add) every cycle



# Evolution of Vector Registers, Instructions



- A core has 16 (SSE, AVX) or 32 (AVX-512) vector registers
- In each cycle, VPU can access registers, do FMAs (e.g.)





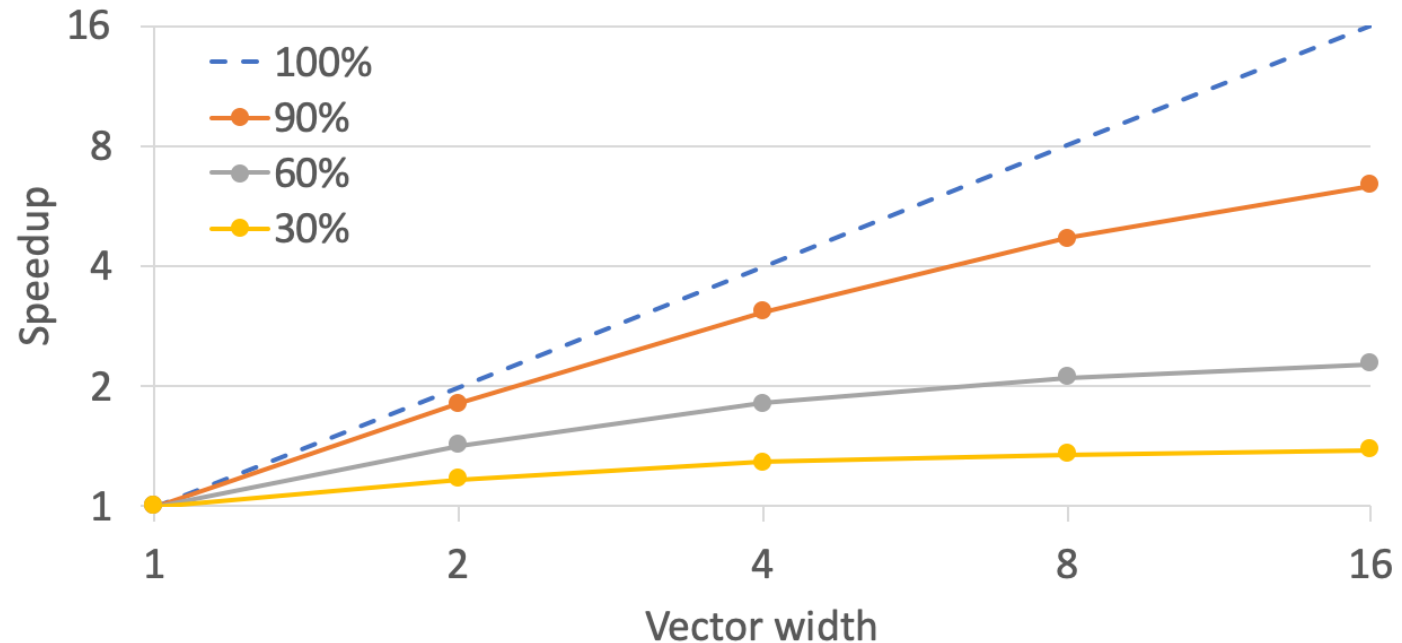
# Peak Flop/s, and Why It's Basically a Fiction

- Peak flop/s (Floating-point OPs per second) is amplified **2x** by vector FMAs
- Example: floats on Intel Xeon Gold 6130 “Skylake-SP” @ 2.1 GHz
  - $(2 \times 16 \text{ flop/VPU}) \times (2 \text{ VPUs/core}) \times (16 \text{ cores}) \times 2.1 \text{ GHz} = 2150 \text{ Gflop/s}$  (really?)
- *Dubious assumption #1: no slow operations like division or square root*
  - Peak rate assumes *exactly* 1 add and 1 multiply (= 2 flops) per VPU per cycle
- *Dubious assumption #2: data are loaded and stored with no delay*
  - Implies heavy reuse of data in vector registers, perfect prefetching into L1 cache
- *Dubious assumption #3: clock rate is fixed*
  - In reality: if all cores are active, Xeon will slow AVX-512 by ~10% to prevent overheating
- *Dubious assumption #4: every instruction in the code is vectorized*
  - In reality: serial fraction of work  $S$  limits the factor in blue to  $1/S$  (Amdahl's Law)



# A Quick Word on Amdahl's Law

- SIMD means parallel, so Amdahl's Law is in effect!
  - Linear speedup is possible only for *perfectly* parallel code
  - Amdahl's asymptote of the speedup curve is  $1/(\text{serial fraction})$
  - Speedup of 16x is unattainable even if 99% of work is vector



# Instructions *Must* Do More Than Just Flops...

- *Data Access*: Load/Store, Pack/Unpack, Gather/Scatter
- *Data Prefetch*: Fetch, but don't load into a register
- *Vector Rearrangement*: Shuffle, Bcast, Shift, Convert
- *Vector Initialization*: Random, Set
- *Logic*: Compare, AND, OR, etc.
- *Math*: Arithmetic, Trigonometry, Cryptography, etc.
- *Variants of the Above...* Mask, Swizzle, Implicit Load...
  - Combine an operation with data selection or movement
- This is why AVX-512 comprises over 4000 instructions

Extension	ICX	SKX	KNL
AVX512F <i>Foundation</i>	X	X	X
AVX512CD <i>Conflict Det.</i>	X	X	X
AVX512BW <i>Byte &amp; Word</i>	X	X	
AVX512DQ <i>Dble. &amp; Quad.</i>	X	X	
AVX512VL <i>Vector Length</i>	X	X	
AVX512PF <i>Prefetch</i>			X
AVX512ER <i>Exp. &amp; Recip.</i>			X
AVX512VNNI <i>Neural Net.</i>	X		
AVX512...etc. <i>ICX additions</i>	X		



# How Do You Get Vector Speedup?

- Program the key routines in assembly?
  - Ultimate performance potential, but only for the brave
- Program the key routines using intrinsics?
  - Step up from assembly; useful in spots, but risky
- ✓ Link to an optimized library that does the heavy lifting
  - Intel MKL, e.g., written by people who know all the tricks
  - BLAS is the portable interface for doing fast linear algebra
- ✓ Let the compiler figure it out
  - Relatively “easy” for user, “challenging” for compiler
  - Compiler may need some guidance through directives
  - Programmer can help by using simple loops and arrays



# Compiler Perspective

- Vectorization is effectively loop unrolling
  - In effect, the compiler unrolls by 4 iterations, if 4 elements fit into a vector register

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



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



```
Load a(i..i+3)  
Load b(i..i+3)  
Do 4-wide a+b->c  
Store c(i..i+3)
```



# Loops That the Compiler Can Vectorize

Basic requirements of vectorizable loops:

- Number of iterations is known on entry
  - No conditional termination (“break” statements, while-loops)
- Single control flow; no “if” or “switch” statements
  - Note, the compiler may convert “if” to a masked assignment!
- Must be the innermost loop, if nested
  - Note, the compiler may reorder loops as an optimization!
- No function calls but basic math: `pow()`, `sqrt()`, `sin()`, etc.
  - Note, the compiler may inline functions as an optimization!
- All loop iterations must be independent of each other



# Compiler Options and Optimization

- GCC vectorizes with `-O2 -ftree-vectorize` or `-O3`
  - Default for x86\_64 is SSE (see output from `gcc -v`)
  - To tune vectors to the host machine: `-march=native`
  - To optimize across objects (e.g., to inline): `-flto`
  - For AVX-512, you *must* add `-mprefer-vector-width=512`
- Intel Classic Compilers vectorize with simply `-O2`
  - Default is SSE instructions, 128-bit vector width (4 floats)
  - To tune vectors to the host machine: `-xHost`
  - To optimize across objects (e.g., to inline functions): `-ipo`
  - For AVX-512, you *must* add `-qopt-zmm-usage=high`
  - Says AVX-512 isn't a great default; AMD doesn't even have it



# Architecture-Specific Compiler Options

- GCC compilers (+ LLVM-based, like Clang, Intel oneAPI,...)
  - Use `-mavx2` `-mfma` or `-march=haswell` to compile for AVX2
  - Careful! Don't get FMAs unless `-mfma` accompanies `-mavx2`
  - GCC 4.9+ has specific options for most AVX-512 extensions
  - GCC 5.3+ has `-march=skylake-avx512`
  - GCC 8.1+ has `-march=icelake-server` (Intel released ICX late)
  - GCC 9.1+ has `-march=cascadelake`
- Intel Classic compilers: most GCC options work, plus...
  - Use `-xCORE-AVX2` or `-xHASWELL` to compile for AVX2
  - For SKL-SP and later: `-xCORE-AVX512`
  - For Xeon + KNL: `-xCOMMON-AVX512` (pure KNL is `-xKNL`)





# Example Code that Does 2 Billion FMAs

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#define ARRAY_SIZE 1024
#define NUMBER_OF_TRIALS 1000000

double dtime() {
    double tseconds = 0.0;
    struct timeval my_t;
    gettimeofday(&my_t, NULL);
    tseconds = (double)(my_t.tv_sec + my_t.tv_usec * 1.0e-6);
    return (tseconds);
}

int get_model_name(char *mname) {
    FILE *fp;
    fp = fopen("/proc/cpuinfo", "r");
    if (fp == NULL) {
        strcpy(mname, "(/proc/cpuinfo is not readable)\n");
        return(1);
    }
    /* model name should be on the fifth line */
    for (int i=0; i < 5; i++) fgets(mname, 80, fp);
    fclose(fp);
    return(0);
}
```

```
int main(int argc, char *argv[]) {

    /* Declare arrays small enough to stay in L1 cache.
       Assume the compiler aligns them correctly. */
    double a[ARRAY_SIZE], b[ARRAY_SIZE], c[ARRAY_SIZE];
    int i, t, rc;
    double m = 1.5, w1, w2, d = 0.0;
    char modelname[80];

    /* Initialize a, b and c arrays */
    for (i=0; i < ARRAY_SIZE; i++) {
        a[i] = 0.0; b[i] = i*1.0e-9; c[i] = i*0.5e-9;
    }

    /* Perform operations with arrays many, many times */
    w1 = dtime();
    for (t=0; t < NUMBER_OF_TRIALS; t++) {
        for (i=0; i < ARRAY_SIZE; i++) {
            a[i] += m*(m*b[i] + c[i]);
        }
    }
    w2 = dtime();

    /* Print total time and processor type used in the run.
       Print a result so array ops aren't optimized away. */
    for (i=0; i < ARRAY_SIZE; i++) d += a[i];
    printf("d = %f   time = %f\n", d, w2 - w1);
    rc = get_model_name(modelname);
    if (rc == 0) printf("%s", modelname);
}
```



# Exercise 1 (to do later)

- The code on the preceding slide is available at these links:
  - <https://godbolt.org/z/z5jecaae8>
  - <https://indico.cern.ch/event/1151367/timetable/>
- The top link takes you to the Compiler Explorer website, a great resource that lets you try lots of compilers and their options and view assembler output
  - DEMO showing how different compiler flags affect vectorization
  - You can execute code on the site, too, but it's not great for benchmarking
- Exercise to try later: using either link above (or the preceding slide), save or copy-paste the code into a file named `abc_fma.c` for benchmarking
- The next two slides guide you through a series of compile-and-run steps to show the performance effects of enabling optimization and vectorization



# Exercise 1 (cont'd.)

1. Invoke your compiler with no special flags and time a run:

```
gcc-11 abc_fma.c -o abc_fma
./abc_fma
```

2. Repeat this process for the following sets of options:

```
gcc-11 -O2 abc_fma.c -o abc_fma
gcc-11 -O3 -fno-tree-vectorize abc_fma.c -o abc_fma
gcc-11 -O3 abc_fma.c -o abc_fma
gcc-11 -O3 -msse3 abc_fma.c -o abc_fma
gcc-11 -O3 -march=native abc_fma.c -o abc_fma
gcc-11 -O3 -march=??? abc_fma.c -o abc_fma #take a guess
```



# Exercise 1 (still cont'd.)

3. Your best result should be from `-march=native`. Why?
  - Here is the current [list of architectures that gcc knows about](#)
  - On a laptop, `-mavx2 -mfma` may be slightly better or worse
4. Do you get the expected speedup factors?
  - SSE registers hold 2 doubles; AVX registers hold 4 doubles
  - Recent laptops should be able to do AVX (but not AVX-512)
5. Other things to note:
  - Optimization `-O3` is degraded by `-fno-tree-vectorize`
  - Not specifying an architecture at `-O3` is equivalent to `-msse3`
  - In icc, vectorization is disabled by `-no-vec` (after `-O2` or `-O3`)
  - Why disable or downsize vectors? To gauge their benefit!



# Why Not Use an Optimized Library?

- Optimized libraries like OpenBLAS may not have the exact function you need
- The kernel of `abc_fma.c` looks like a DAXPY, or  $(aX + Y)$  with doubles... but it isn't quite...
- The inner loop must be replaced by two DAXPY calls, not one, and with function overhead, the resulting code runs several times slower

```
for (t=0; t < NUMBER_OF_TRIALS; t++) {  
    for (i=0; i < ARRAY_SIZE; i++) {  
        a[i] += m*(m*b[i] + c[i]);  
    }  
}
```



```
for (t=0; t < NUMBER_OF_TRIALS; t++) {  
    cblas_daxpy(ARRAY_SIZE, m*m, b, 1, a, 1);  
    cblas_daxpy(ARRAY_SIZE, m, c, 1, a, 1);  
}
```



# Optimization Reports

Use optimization report options for info on vectorization:

```
gcc -c -O3 -fopt-info-vec -fopt-info-vec-missed myvec.c
```

```
icc -c -O3 -qopt-report=2 -qopt-report-phase=vec myvec.c
```

The above are equivalent. For GCC, tack on more flags (or `-a11`) for more info.

For Intel, use `=n` to control the amount of detail in `myvec.optrpt`

n	Description of information presented
0	No vector report
1	Lists the loops that were vectorized
2	(default level) Adds the loops that were not vectorized, plus a short reason
3	Adds summary information from the vectorizer about all loops
4	Adds verbose information from the vectorizer about all loops
5	Adds details about any data dependencies encountered (proven or assumed)



# Exercise 2 (to do later)

Let's examine optimization reports for the `abc_fma.c` code.

1. Recompile the code with `-O3`, along with optimization reporting (`-fopt-info-vec`) from the vectorizer.
  - Confirm that the inner loops were vectorized as expected.
2. Repeat (1), but this time with vectorization turned off (i.e., `-fno-tree-vectorize`). Do you get any output?
3. Repeat (1), but now add `-fopt-info-vec-missed` (loops that missed out on vectorization) to see what else the compiler tried to do with this code.
  - Considering that the main loops ultimately vectorized, you may find that gcc gives way too much information here.



# User Perspective

- User's goal is to supply code that runs well on hardware
- Thus, you need to know the *hardware perspective*
  - Think about how instructions will run on vector hardware
  - Try also to combine additions with multiplications
  - Furthermore, try to reuse everything you bring into cache!
- And you need to know the *compiler perspective*
  - Look at the code like the compiler looks at it
  - At a minimum, set the right compiler options!





# Vector-Aware Coding

- Know what makes codes vectorizable at all
  - The “for” loops (C) or “do” loops (Fortran) that meet constraints
- Know where vectorization ought to occur
- Arrange vector-friendly data access patterns (unit stride)
- Study compiler reports: do loops vectorize as expected?
- Implement fixes: directives, compiler flags, code changes
  - Remove constructs that hinder vectorization
  - Encourage/force vectorization when compiler fails to do it
  - Engineer better memory access patterns
- Turn to performance tools, if further speedup is desired



# Challenge: Loop Dependencies

- Vectorization changes the order of computation compared to sequential case
  - Groups of computations now happen simultaneously
- Compiler must be able to prove that vectorization will produce correct results
- Key criterion: “unrolled” loop iterations must be **independent** of each other
  - Wider vectors means that more iterations must be independent
  - Not everything that looks like a dependency truly is one
- Compiler must perform dependency analysis prior to vectorizing
  - It will make conservative assumptions about dependencies, unless guided by directives



# Loop Dependencies: Read After Write

Consider adding the following vectors in a loop,  $N=5$ :

$a = \{0,1,2,3,4\}$

$b = \{5,6,7,8,9\}$

```
for(i=1; i<N; i++)  
    a[i] = a[i-1] + b[i];
```

Applying each operation sequentially:

$$a[1] = a[0] + b[1] \rightarrow a[1] = 0 + 6 \rightarrow a[1] = 6$$

$$a[2] = a[1] + b[2] \rightarrow a[2] = 6 + 7 \rightarrow a[2] = 13$$

$$a[3] = a[2] + b[3] \rightarrow a[3] = 13 + 8 \rightarrow a[3] = 21$$

$$a[4] = a[3] + b[4] \rightarrow a[4] = 21 + 9 \rightarrow a[4] = 30$$

$a = \{0, 6, 13, 21, 30\}$



# Loop Dependencies: Read After Write

Consider adding the following vectors in a loop,  $N=5$ :

$a = \{0,1,2,3,4\}$

$b = \{5,6,7,8,9\}$

```
for(i=1; i<N; i++)  
    a[i] = a[i-1] + b[i];
```

Applying each operation sequentially:

$a[1] = a[0] + b[1] \rightarrow a[1] = 0 + 6 \rightarrow a[1] = 6$

$a[2] = a[1] + b[2] \rightarrow a[2] = 6 + 7 \rightarrow a[2] = 13$

$a[3] = a[2] + b[3] \rightarrow a[3] = 13 + 8 \rightarrow a[3] = 21$

$a[4] = a[3] + b[4] \rightarrow a[4] = 21 + 9 \rightarrow a[4] = 30$

$a = \{0, 6, 13, 21, 30\}$



# Loop Dependencies: Read After Write

Now let's try vector operations:

$a = \{0,1,2,3,4\}$

$b = \{5,6,7,8,9\}$

```
for(i=1; i<N; i++)  
    a[i] = a[i-1] + b[i];
```

Applying vector operations,  $i=\{1,2,3,4\}$ :

$a[i-1] = \{0,1,2,3\}$  (load)

$b[i] = \{6,7,8,9\}$  (load)

$\{0,1,2,3\} + \{6,7,8,9\} = \{6, 8, 10, 12\}$  (operate)

$a[i] = \{6, 8, 10, 12\}$  (store)

$a = \{0, 6, 8, 10, 12\} \neq \{0, 6, 13, 21, 30\}$  **NOT VECTORIZABLE**



# Loop Dependencies: Synopsis

- Read After Write
  - Also called “flow” dependency
  - Variable written first, then read
  - Not vectorizable
  
- Write After Read
  - Also called “anti” dependency
  - Variable read first, then written
  - Vectorizable

```
for(i=1; i<N; i++)  
    a[i] = a[i-1] + b[i];
```

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



# Loop Dependencies: Synopsis

- Read After Read

- Not really a dependency
- Vectorizable

```
for(i=0; i<N; i++)  
    a[i] = b[i%2] + c[i];
```

- Write After Write

- a.k.a “output” dependency
- Variable written, then re-written
- Not vectorizable
- Exception: array sums and products (`+=`, `*=`) are vectorizable

```
for(i=0; i<N; i++)  
    a[i%2] = b[i] + c[i];
```



# Loop Dependencies: Pointer Aliasing

- In C, pointers can hide data dependencies!
  - The memory regions that they point to may overlap
- Is this vectorizable?

```
void compute(double *a, double *b, double *c) {  
    for (i=1; i<N; i++) {  
        a[i] = b[i] + c[i];  
    }  
}
```

- ...Not if we give it the arguments `compute(a, a-1, c)`
- In effect, `b[i]` is really `a[i-1]` → Read After Write dependency
- Compilers can usually cope, at some cost to performance





# Dependencies and Optimization Reports

- Loop-carried dependencies are a common reason for vectorization failure
- Optimization reports say where the compiler found apparent dependencies
  - Choose a report level that gives info about places where vectorization was missed
- Remember, the compiler is conservative about dependencies
  - Dig into the details, see if the claimed dependencies really exist in the code
  - The Intel compiler is generally better than gcc for this because it is more concise
- Even with no dependencies, vectorization is not guaranteed!
  - Compiler may fail to vectorize a loop if it has complicated indexing
  - Compiler may decline to vectorize a loop if no performance gain is projected
  - Reports give information about these situations too



# Exercise 3 (to do later)

1. Make a copy of `abc_fma.c` called `abc_fma_shift.c`. Edit it and change the innermost of the nested loops to look like this:

```
for (i=0; i < ARRAY_SIZE-1; i++) {  
    a[i+1] += m*(m*b[i] + c[i]);  
}
```

2. The above loop has no dependencies. (Why not?) Compile the code with vectorization enabled, and request info on loops that missed out:

```
gcc-11 -O3 abc_fma_shift.c -o abc_fma_shift -fopt-info-vec-missed
```

3. Did gcc vectorize the loop? Look for any “missed” remarks directed at the loop on line `abc_fma_shift.c:46` – or grep for “complicated access pattern”



# Loop Dependencies: Vectorization Hints

- Sometimes, it is impossible for the compiler to prove that there is no data dependency that will affect correctness
  - e.g., unknown index offset, complicated use of pointers
- To stop the compiler from worrying, you can give it the IVDEP (Ignore Vector DEpendencies) hint
  - It assures the compiler, “It’s safe to assume no dependencies”
  - Compiler may still choose not to vectorize based on cost
  - Example: assume we know  $M >$  vector width in doubles...

```
void vec1(double s1, int M,  
          int N, double *x) {  
    #pragma GCC ivdep      // for Intel, omit GCC  
    for(i=M; i<N; i++) x[i] = x[i-M] + s1;
```



# OpenMP 4.0 and Vectorization

- #pragma omp simd
  - Motivates the compiler to try harder to vectorize a particular loop
  - Is enabled by the special compiler flag `-fopenmp-simd` (Intel: `-qopenmp-simd`)
  - Can be combined with other OpenMP constructs; use flag `-fopenmp` (Intel: `-qopenmp`)
  - Has its own set of OpenMP clauses
  - May not be required in all instances; in order to vectorize the multithreaded OpenMP example below, GCC needs “simd”, Intel doesn’t

```
#pragma omp for simd private(x) reduction(+:sum)
for (j=1; j<=num_steps; j++) {
    x = (j-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
```



# Loop Dependencies: Language Constructs

- C99 introduced 'restrict' keyword to language
  - Instructs compiler to assume addresses will not overlap, ever

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

- Intel compiler may need extra flags: `-restrict -std=c99`

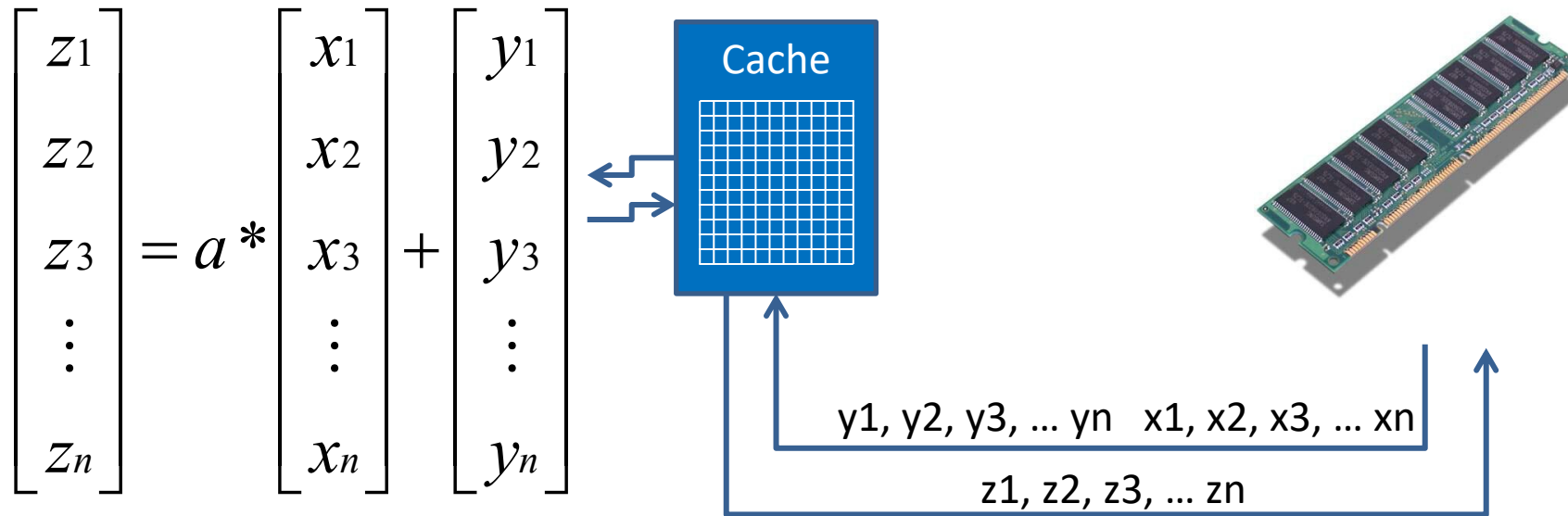


# Memory Performance and Vectorization

- We have mostly been focusing on faster flop/s, but flop/s don't happen unless data are present
  - Moving data from memory is often the rate-limiting step!
- Data (including scalar data + neighbors) travel between RAM and caches in groups called “cache lines” that are the exact same size as vectors
- But wait... if data movement is “vectorized”, just like adds and multiplies are vectorized, then everything is getting the same speedup, right?
  - Um, no. The data rate for RAM is slow, even if it is always “vectorized” in a sense
  - Well... loads from L1 cache to registers, and stores from registers to L1, do get vectorized. But that's just the final short step if the data start way out in RAM



# Cache and Alignment



- Optimal vectorization takes you beyond the SIMD unit!
  - Cache lines start on 16-, 32-, or 64-byte boundaries in memory
  - Sequential, aligned access is much faster than random/strided



# Strided Access

- Fastest usage pattern is “stride 1”: perfectly sequential
  - Cache lines arrive in L1d as full, ready-to-load vectors
- Stride-1 constructs:
  - Storing data in structs of arrays vs. arrays of structs
  - Looping through arrays so their “fast” dimension is innermost
    - C/C++: stride 1 on last index (columns)
    - Fortran: stride 1 on first index (rows)

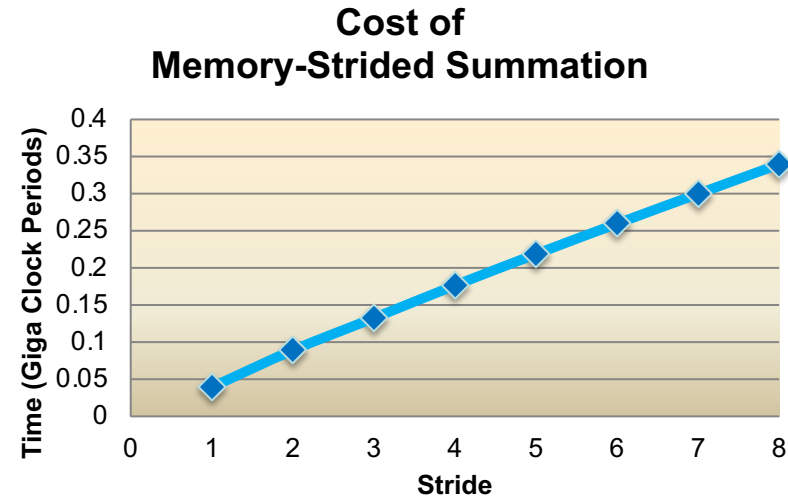
```
for (j=0; j<n; j++) {  
    for (i=0; i<n; i++) {  
        a[j][i]=b[j][i]*s;  
    }  
}  
  
do j=1,n  
    do i=1,n  
        a(i,j)=b(i,j)*s  
    end do  
end do
```





# Penalty for Strided Access

- Striding through memory reduces effective memory bandwidth!
  - Roughly by  $1/(\text{stride})$
- Why? For some stride  $s$ , data must be “gathered” from  $s$  cache lines to fill a vector register
- It’s worse than non-aligned access



```
for (i=0; i<4000000*istride;  
     i+=istride) {  
    a[i] = b[i] + c[i]*sfactor;  
}
```



# Diagnosing Cache & Memory Deficiencies

- Really bad stride patterns may prevent vectorization
  - The GCC vector info might say, “not vectorized: vectorization is not profitable.”
  - The Intel vector report might say, “vectorization possible but seems inefficient”
- Bad stride and other problems may be difficult to detect
  - The result is merely poorer performance than might be expected
- Profiling tools like Intel VTune can help
- Intel Advisor makes recommendations based on source



# Conclusions: Vectorization Basics

- The compiler “automatically” vectorizes tight loops
- Write code that is vector-friendly
  - Innermost loop accesses arrays with stride one
  - Loop bodies consist of simple multiplications and additions
  - Data in cache are reused; loads are stores are minimized
- Write code that avoids the potential issues
  - No loop-carried dependencies, branching, aliasing, etc.
- This means you know where vectorization should occur
- Optimization reports will tell you if expectations are met
  - See whether the compiler’s failures are legitimate
  - Fix code if the compiler is right; use `#pragma` if it is not

