Vector Parallelism on Multi-Core Processors

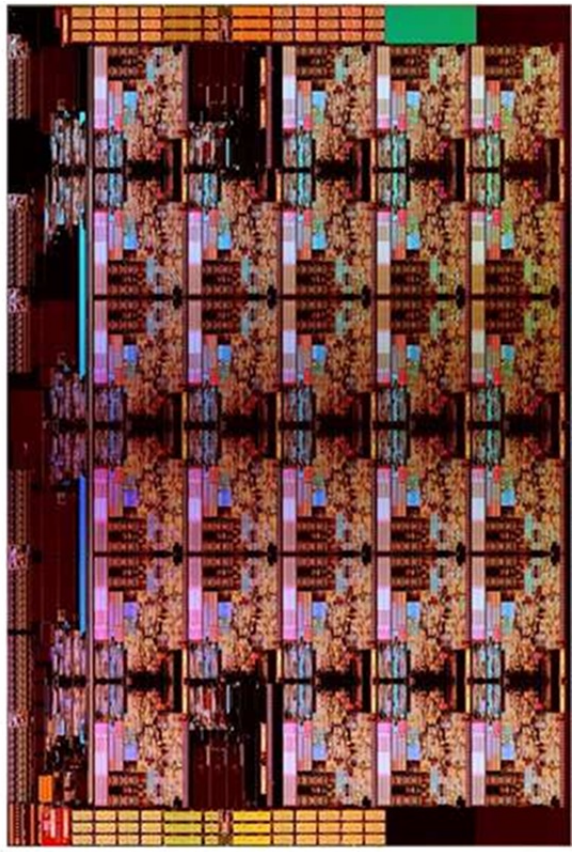
Steve Lantz, Cornell University

CoDaS-HEP Summer School, July 17, 2023

Cornell University
Center for Advanced Computing

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 60 in Intel Xeon “Sapphire Rapids”, 128 in AMD EPYC
- 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?



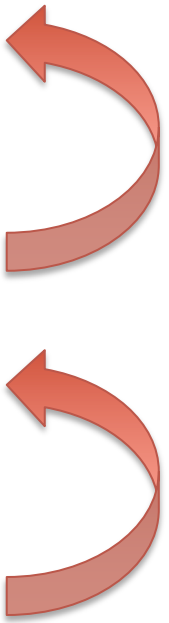
Die shot of 28-core Intel Skylake-SP
Source: wikichip.org

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.

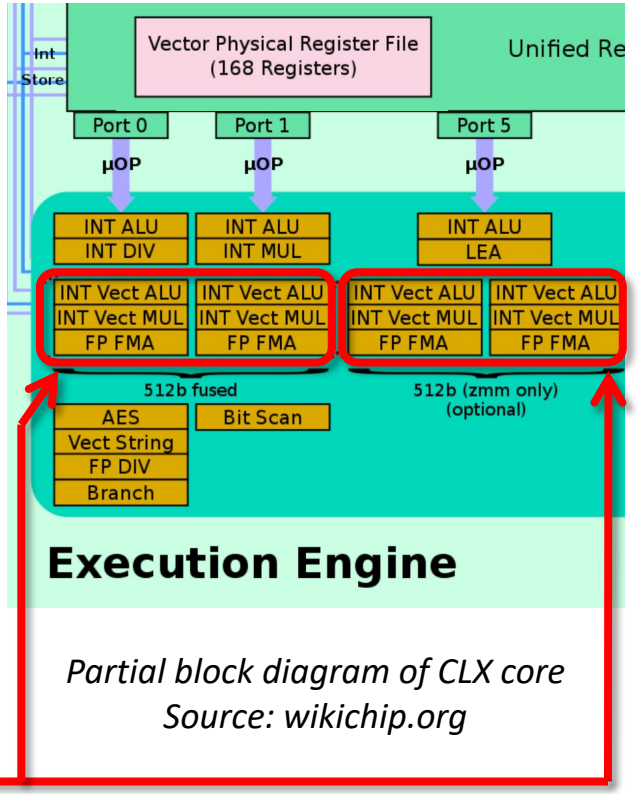
2. **Compiler Perspective:** Determine how and when it is possible to express computations in terms of vector instructions.

3. **Programmer Perspective:** Write code with SIMD in mind; e.g., in a way that allows the compiler to deduce that vectorization is possible.



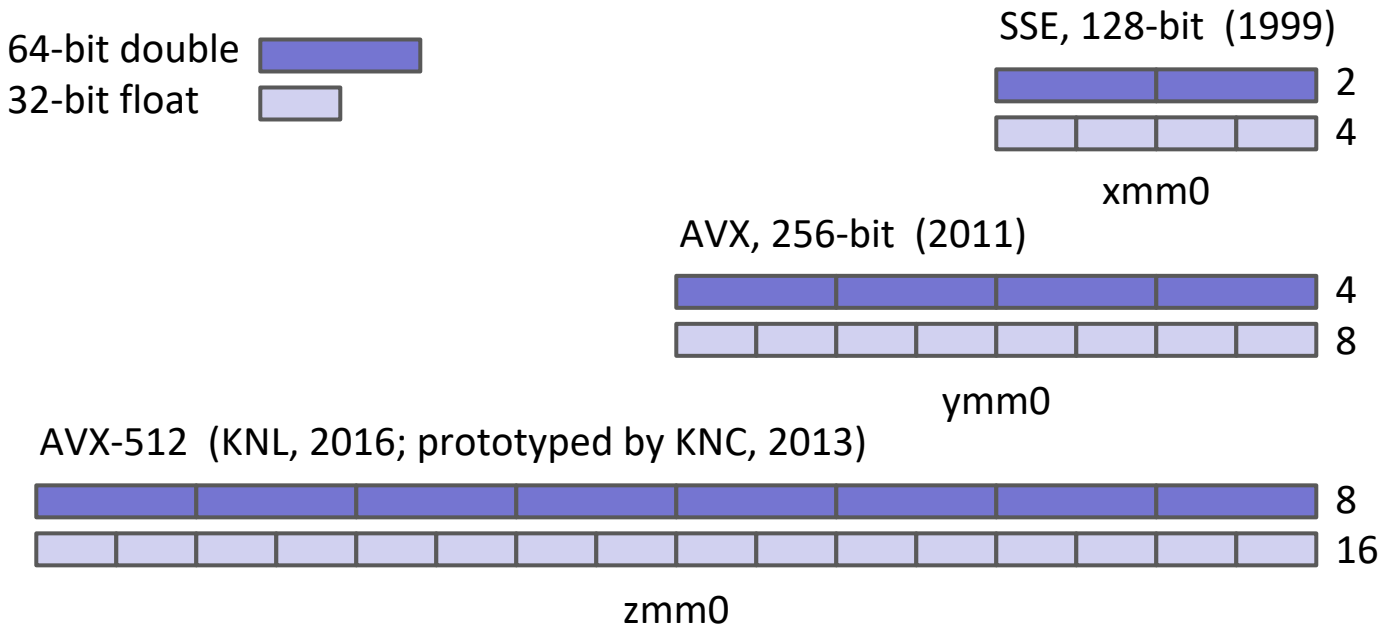
Hardware Perspective

- 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: Cascade Lake, Ice Lake, Sapphire Rapids…
  - 2 VPUs (vector processing units) available per core
  - 2 ops/VPU if they do FMAs (Fused Multiply-Add) every cycle



Partial block diagram of CLX core
Source: wikichip.org

Evolution of Vector Registers, Instructions



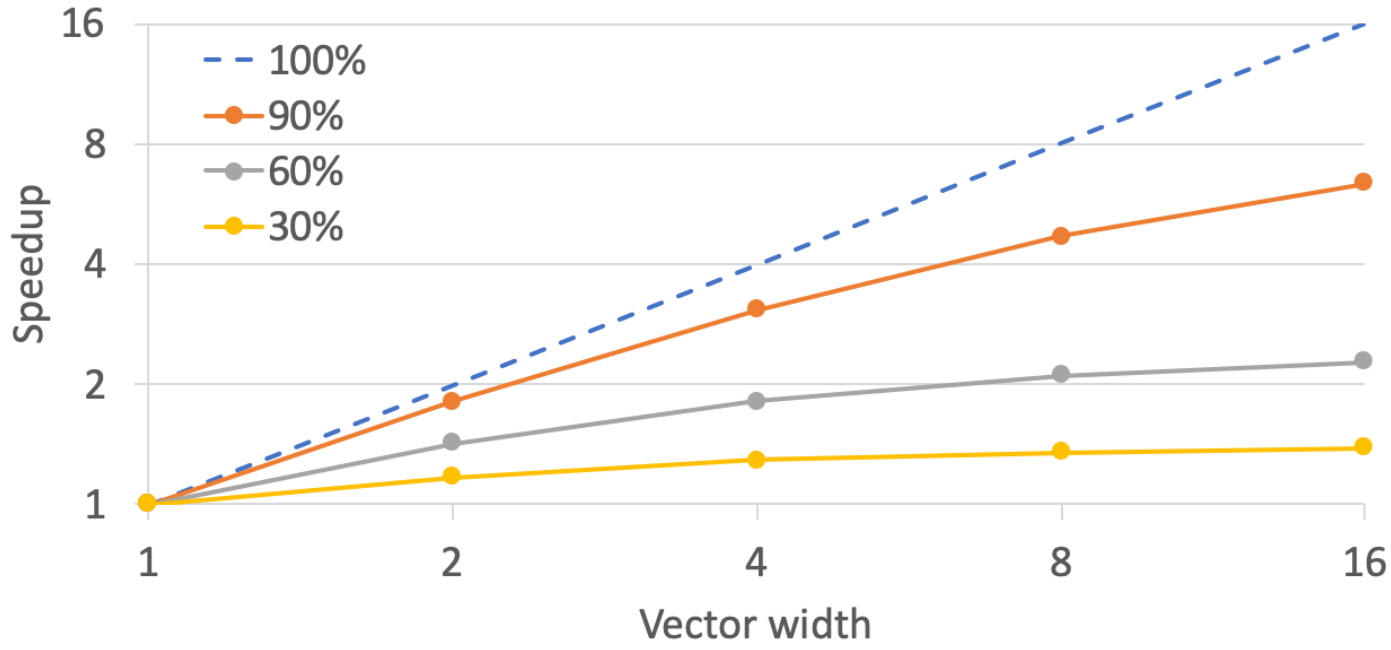
- A core has 16 (SSE, AVX) or 32 (AVX-512) vector registers
- In each cycle, VPUs can access registers, do FMAs (e.g.)

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 x 16 flop/VPU) x (2 VPUs/core) x (16 cores) x 2.1 GHz = 2150 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/(serial fraction)
  - Speedup of 16x is unattainable even if 99% of work is vector



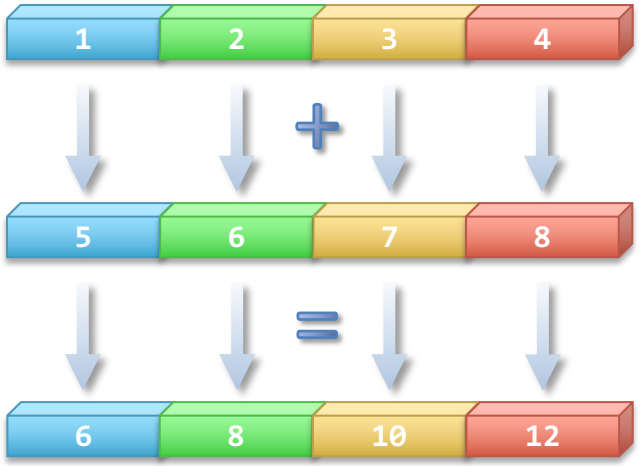
Instructions *Must* Do More Than Just Flops…

- *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 *Foundation*	X	X	X
AVX512CD *Conflict Det.*	X	X	X
AVX512BW *Byte & Word*	X	X	
AVX512DQ *Dble. & Quad.*	X	X	
AVX512VL *Vector Length*	X	X	
AVX512PF *Prefetch*			X
AVX512ER *Exp. & Recip.*			X
AVX512VNNI *Neural Net.*	X		
AVX512…etc. *ICX additions*	X		

How Do We Get Vector Speedup?

- Program the key routines in assembly…
  - Ultimate performance potential, but only for the brave
- Program the key routines using SIMD intrinsics...
  - Step up from assembly; useful in spots, but risky
- ✓ Link to an optimized high-level library
  - 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 find vectorizable loops and calls!
  - Compiler may need some guidance through directives
  - Programmer can help by using simple loops and arrays



Compiler Perspective

- Vectorization is essentially loop unrolling
  - In effect, the compiler unrolls by 4 iterations, if 4 elements fit in 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)
```

Basic requirements of vectorizable loops

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

Compiler Options and Optimization

- GCC, clang vectorize 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`
  - Note: starting with GCC 12, vectorization occurs with just `-O2`
- 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`

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` …and so on
- Intel Classic compilers: most GCC options work, plus…
  - Use `-xCORE-AVX2` or `-xHASWELL` to compile for AVX2
  - For SKL-SP and later: `-xCORE-AVX512`

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

- The code on the preceding slide is available at this link:
  - https://godbolt.org/z/z5jecaae8
- This 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-13 abc_fma.c -o abc_fma
./abc_fma
```

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

```
gcc-13 -O2 abc_fma.c -o abc_fma
gcc-13 -O3 -fno-tree-vectorize abc_fma.c -o abc_fma
gcc-13 -O3 abc_fma.c -o abc_fma
gcc-13 -O3 -msse3 abc_fma.c -o abc_fma
gcc-13 -O3 -march=native abc_fma.c -o abc_fma
gcc-13 -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`
   - With Intel's `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);
}
```

Programmer Perspective

- Programmer's goal is to supply code that runs well on hardware
- Thus, you need to start with 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 add the *compiler perspective*
  - Look at the code like the compiler looks at it
  - At a minimum, set the right compiler options!
  - But you also have to consider how to lower barriers for the compiler…

Challenge: Loop Dependencies

- Vectorization changes the order of computation from sequential order
  - Groups of computations now happen simultaneously
- Compiler must be able to prove that vectorization yields correct results
- The key: “unrolled” loop iterations must be **independent** of each other
  - Wider vectors means that bigger groups of iterations must be independent
  - Not everything that looks like a dependency truly is one
- Compiler must perform a dependency analysis prior to vectorizing
  - It must make conservative assumptions about dependencies
  - You can give guidance by inserting compiler directives and keywords

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] → a[1] = 0 + 6 → a[1] = 6

a[2] = a[1] + b[2] → a[2] = 6 + 7 → a[2] = 13

a[3] = a[2] + b[3] → a[3] = 13 + 8 → a[3] = 21

a[4] = a[3] + b[4] → a[4] = 21 + 9 → 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} ≠ {0, 6, 13, 21, 30} NOT VECTORIZABLE

Dependencies and Optimization Reports

- Loop-carried dependencies are a common reason for non-vectorization
- Optimization reports say where the compiler found dependencies
  - Choose a report level with 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 2 (to do later)

Let's examine optimization reports for the abc_fma.c code from Exercise 1.

1. Recompile the code with `-O3`, along with optimization reporting from the vectorizer: `-fopt-info-vec`
   - Confirm that the inner loops were vectorized as expected.
2. Repeat (1), but with vectorization disabled: `-fno-tree-vectorize`
   - Do you get any output at all?
3. Repeat (1), but now add: `-fopt-info-vec-missed`
   - This reports on the loops that missed out on vectorization
   - Considering that the main loops ultimately vectorized, you may find that gcc gives way too much information here…

Exercise 2 (cont'd.)

4. 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]);
}
```

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

```
gcc-13 -O3 abc_fma_shift.c -o abc_fma_shift -fopt-info-vec-missed
```

6. 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"

Compiler Directives for Vectorization

- Sometimes, it is impossible for the compiler to prove that there is no data dependency, even if there isn't one
  - e.g., unknown loop index offset, complicated use of pointers
- In this case, 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

- OpenMP 4.0 directive: `#pragma omp simd`
  - Motivates the compiler to try harder to vectorize a particular loop
  - Can be refined with its own set of OpenMP clauses
  - Is enabled by the special compiler option `-fopenmp-simd` (Intel: `-qopenmp-simd`)
  - Can be combined with other OpenMP constructs; use `-fopenmp` (Intel: `-qopenmp`)
  - To vectorize the multithreaded 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);
}
```

Challenge: 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

Getting Past Pointer Aliasing

- C99 introduced the “restrict” keyword to the C language
  - Instructs compiler to assume addresses will not overlap, ever

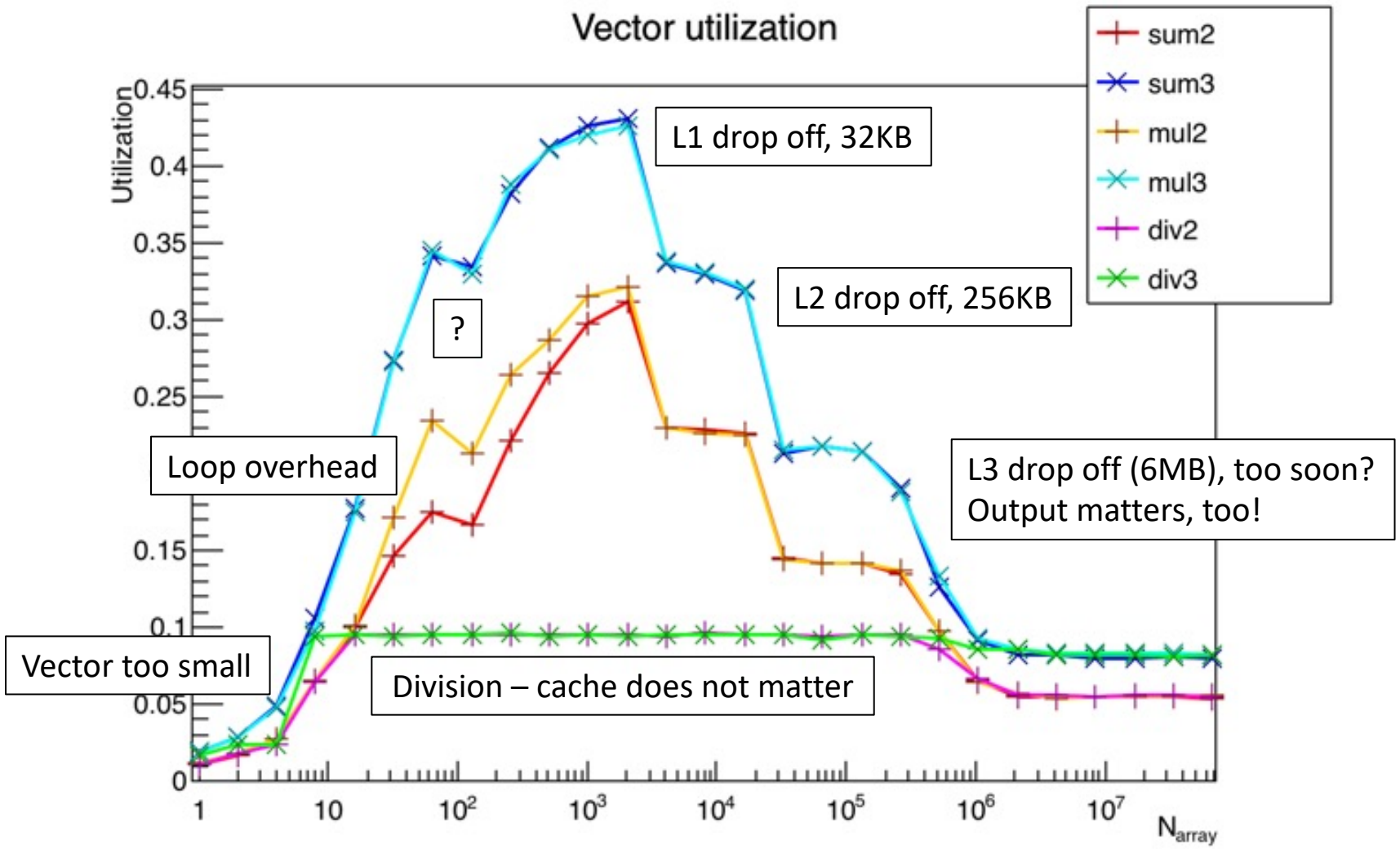
```
void compute(double * restrict a, double * restrict b, double * restrict c) {
    for (i=0; i<N; i++) {
        a[i] = b[i] + c[i];
    }
}
```

- Intel’s `icc` 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

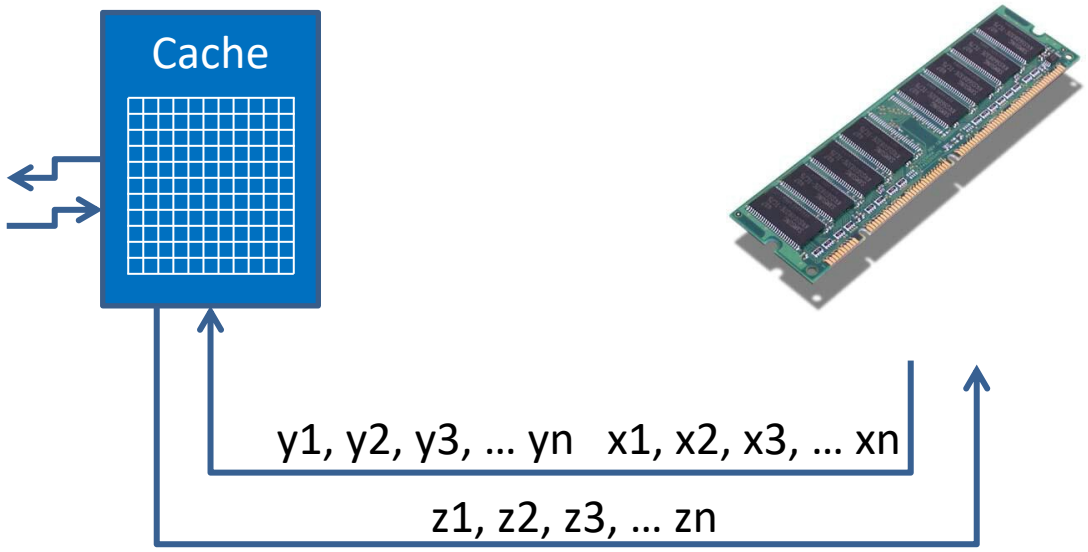
Laptop Vector Utilization as a Function of Array Size



"mtorture" code by Matevž Tadel, UCSD

Cache and Alignment

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_n \end{bmatrix} = a * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$



- 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

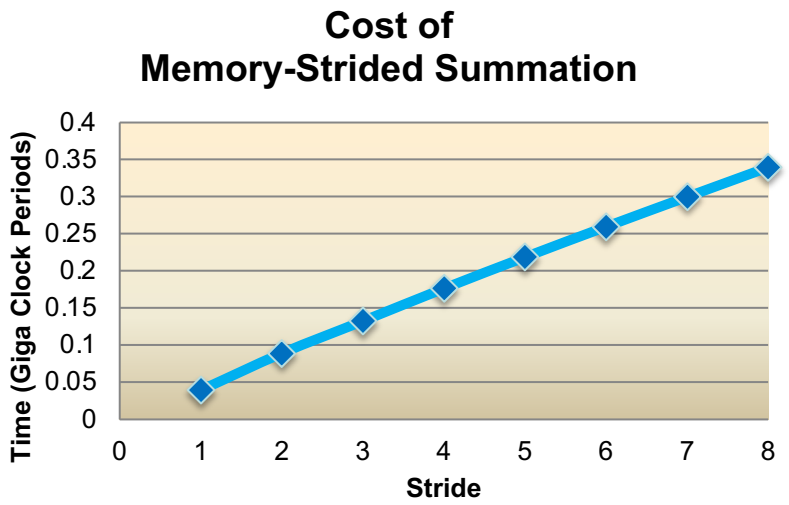
- Fastest usage pattern is “unit stride”: perfectly sequential
  - Cache lines arrive in L1d as full, ready-to-load vectors
- To create unit-stride accesses:
  - Store data in structs of arrays (SoA), rather than arrays of structs
  - Loop through arrays so their “fast” dimension is innermost
  - C/C++ is fastest on the last index; Fortran is fastest on the first

```
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/(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 and 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
  - Loop bodies consist of simple multiplications and additions
  - Innermost loop accesses arrays with unit stride
  - Data in cache are reused; reads from and writes to RAM 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
  - Fix code if the compiler is right about it; insert a directive if it is not

A Few Words on Using Multiple Cores

- Partition workload into coarse-grained tasks
  - Assign tasks to different CPU threads
  - Try to find SIMD opportunities within each task
  - Same strategy works for GPU programming
- Task parallelization is not done automatically
  - Multithreading can be done by the compiler if the code has OpenMP directives
  - Other APIs exist for multithreading, e.g., Threading Building Blocks (TBB)
  - To split up work among processes, use a message passing interface like MPI
  - Multithreading assumes shared memory; MPI works for distributed memory, too
  - Use MPI if the application will need multiple nodes connected over a network

Parallelization plan for multiple cores

```
# multithread this loop...
For t in [ tasks ]
  #pragma omp simd
  for i in [ task t ]
    # vectorized calculations
```

What About Python? Or MATLAB?

- Use array syntax as much as possible; avoid low-level looping
  - Assume that the underlying libraries use SIMD operations on arrays
  - In Python, this entails making good use of NumPy and SciPy
  - This is actually what “vectorization” means in Python!
  - It’s almost the opposite of what to do for compiled code, but the goal is the same
- High-level tasks must be defined using language features or extensions
  - Python has multiprocessing rather than multithreading
  - MATLAB has the Parallel Computing Toolbox
  - Extra coding is required to identify the tasks to be done in parallel
- Entire presentations could be made on each of these subjects!

A Dirty Little Secret of Scripting Languages

- Interpreted languages like Python and MATLAB are built on compiled code!
  - It's the main way they can get competitive performance
  - (Note, Julia may be an exception–I've heard it's Julia all the way down)
- They rely on runtime libraries that are written in C/C++ and (gulp) Fortran!
  - The libraries build in many of the tricks and compiler options I've described
- Why? NumPy and MATLAB are SIMD-friendly: they're built for linear algebra
  - NumPy is often linked against an optimized implementation of BLAS (e.g., MKL)
  - The core libraries can even be multithreaded (e.g., MKL is)
- There are tricks to make these languages behave more like compiled code
  - Just-in-time (JIT) compiling, which has an up-front cost (e.g., Numba)
  - Source conversion to C/C++, which is then compiled (e.g., Cython, mcc)