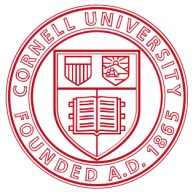
Vector Parallelism for Kalman-Filter-Based Particle Tracking on Multi- and Many-Core Processors

Steve Lantz, Cornell University

Matevž Tadel, UC San Diego

CoDaS-HEP Summer School, July 13, 2017



Cornell University
Center for Advanced Computing

UC San Diego

Vector Parallelism: Motivation

- CPU speeds hit a plateau a decade ago
  - Power limitations! "Slow" transistors are more efficient, cooler
- But process improvements keep making space cheaper
  - Moore's Law! Easy to add 2x more "stuff" every 18–24 months
- One solution: more cores
  - First dual-core Intel CPUs appeared in 2005
  - Counts have grown rapidly: 8 in Sandy Bridge, 61–72 in Xeon Phi
- Another solution: SIMD or vector operations
  - First appeared on Pentium with MMX in 1996
  - Vector sizes are ballooning: 512 bits (8 doubles) in Xeon Phi
  - *Vectorization* can thus increase speed by an order of magnitude

What Is Vectorization?

- **Hardware Perspective:** Run vector instructions involving special registers and functional units that allow in-core parallelism for operations on arrays (vectors) of data.

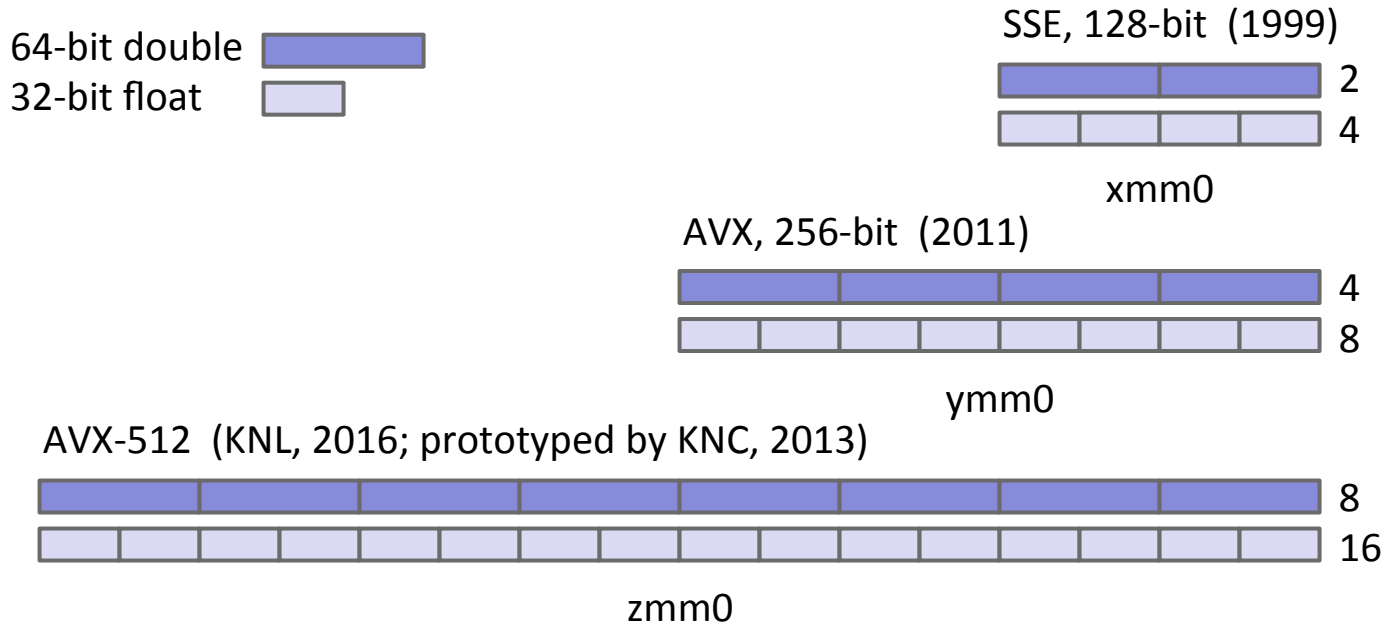
- **Compiler Perspective:** Determine how and when it is possible to express computations in terms of vector instructions.

- **User Perspective:** Determine how to write code in a manner that allows the compiler to deduce that vectorization is possible.

Hardware Perspective

- SIMD = Single Instruction Multiple Data
  - Part of commodity CPUs (x86, x64, PowerPC, etc.) since late ’90s
- Goal: parallelize computations on vector arrays
  - Line up operands, execute one operation on all simultaneously
- SIMD instructions have gotten quicker over time
  - Initially, several cycles for execution on small vectors
  - With Intel AVX, pipelining of some SIMD instructions
  - Now, multiply-and-add with large vectors on every cycle
- Intel’s latest: Knights Landing (KNL)
  - Two VPUs (vector processing units) per core
  - Each VPU can execute an FMA (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, ADD and MUL units can access registers

Peak Flop/s

- Vectorization is essential for attaining peak flop/s
  - Flop/s = floating point operations per second
- Speedup (vs. no vector) is proportional to vector width
- Extra factor of 2 from pipelined/fused multiply-add:
  - 128-bit SSE – 4x double, 8x single (pipelined)
  - 256-bit AVX – 8x double, 16x single (pipelined, or FMA in AVX2)
  - 512-bit AVX – 16x double, 32x single (FMA)
- Example: Intel Xeon E5-2670 v2 "Ivy Bridge"
  - 10 cores, each with 256-bit AVX vector unit = 8 flop/cycle DP
  - 10 cores * 8 flop/cycle * 2.5 GHz = 200 Gflop/s peak DP
  - Effective rate may be 80-90% of nominal due to throttling (heat)

Why Peak Flop/s Is (Almost) a Fiction

- Assumes all code is perfectly vectorized
  - SIMD is parallel, so Amdahl's law is in effect!
  - Serial/scalar portions of code would limit the speedup
- Assumes no slow operations like division, square root
- Assumes data are loaded and stored with no delay
  - Only true for data in L1 cache and vector registers
  - Implies either the dataset is tiny, or prefetching is ideal
- Assumes there are no issues with access to RAM
  - Bandwidth is sufficient; latency is hidden by other operations
  - All data are aligned properly, e.g., on 64-byte boundaries
- Only a near-trivial code satisfies all the above criteria!

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 actual work
  - Intel MKL, e.g., written by people who know all the tricks
  - Get benefits “for free” when running on supported platform
- 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

- Think of vectorization in terms of loop unrolling
  - Unroll N iterations, where N elements fit into a vector register

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

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

```
Load b(i..i+3)
Load c(i..i+3)
Operate b+c->a
Store a
```

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; masked assignments are OK
- 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

- Intel Compiler:
  - Vectorization starts at optimization level `-O2`
  - Default is SSE instructions and 128-bit vector width
  - Use `-xAVX` or `-xhost` to enable AVX and 256-bit vector width
  - Use `-xMIC-AVX512` or `-xhost` for Xeon Phi KNL
  - Vectorization report (in .optrpt file): `-qopt-report=<n>`
- GCC 4.9 or higher:
  - Vectorization starts at optimization level `-O3`
  - Use `-march=native -fwhole-program` (like Intel `-xhost -ipo`)
  - KNL, `-mavx512f -mavx512cd -mavx512er -mavx512pf`
  - Reports, `-fopt-info-vec -fopt-info-vec-missed`

Optimization Reports

To get information about vectorization, compile the code with an optimization report option on the compilation line:

```
icc -O3 -xAVX -qopt-report=2 -qopenmp ./vector.c -o vec
```

This generates a detailed report file called `vector.optrpt`
(The `-qopenmp` just enables calls to the OpenMP timer)

Open the optimization report file with your favorite text editor, or simply cat the contents to your screen:

```
cat ./vector.optrpt
```

What Was and Wasn't Vectorized

There is a lot of information in the optimization report file. We find out that our array initialization can't be vectorized because we call external function `rand` in lines 34 and 35:

```
LOOP BEGIN at ./vector.c(34,2)
   remark #15527: loop was not vectorized: function call to rand(void)
   cannot be vectorized [ ./vector.c(34,33) ]
LOOP END
LOOP BEGIN at ./vector.c(35,2)
   remark #15527: loop was not vectorized: function call to rand(void)
   cannot be vectorized [ ./vector.c(35,33) ]
```

But the main loop has been vectorized:

```
LOOP BEGIN at ./vector.c(45,3)
   remark #15300: LOOP WAS VECTORIZED
```

Optimization Report: More Detail

Try a higher reporting level, `-qopt-report=4`, to find out more about the quality of the main loop vectorization:

```
LOOP BEGIN at ./vector.c(45,3)
   ...
   remark #15305: vectorization support: vector length 4
   remark #15399: vectorization support: unroll factor set to 4
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 2
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 8
   remark #15477: vector loop cost: 1.250
   remark #15478: estimated potential speedup: 6.400
   remark #15488: --- end vector loop cost summary ---
   remark #25015: Estimate of max trip count of loop=16
```

Super-Simple Main Loop in vector.c

```
#define N 256
...
    tstart = omp_get_wtime();
    // External loop to measure a reasonable time interval
    for(i = 0; i < 1000000; i++){
        #pragma vector aligned
        for( j = 0; j < N; j++){
            x[j] =  y[j] + z[j];
        }
    }
    tend = omp_get_wtime();
```

- Want to time this on Intel Sandy Bridge, 1 thread
- Will it achieve the maximum expected flop/s?

Note: All Arrays Fit in L1 Cache

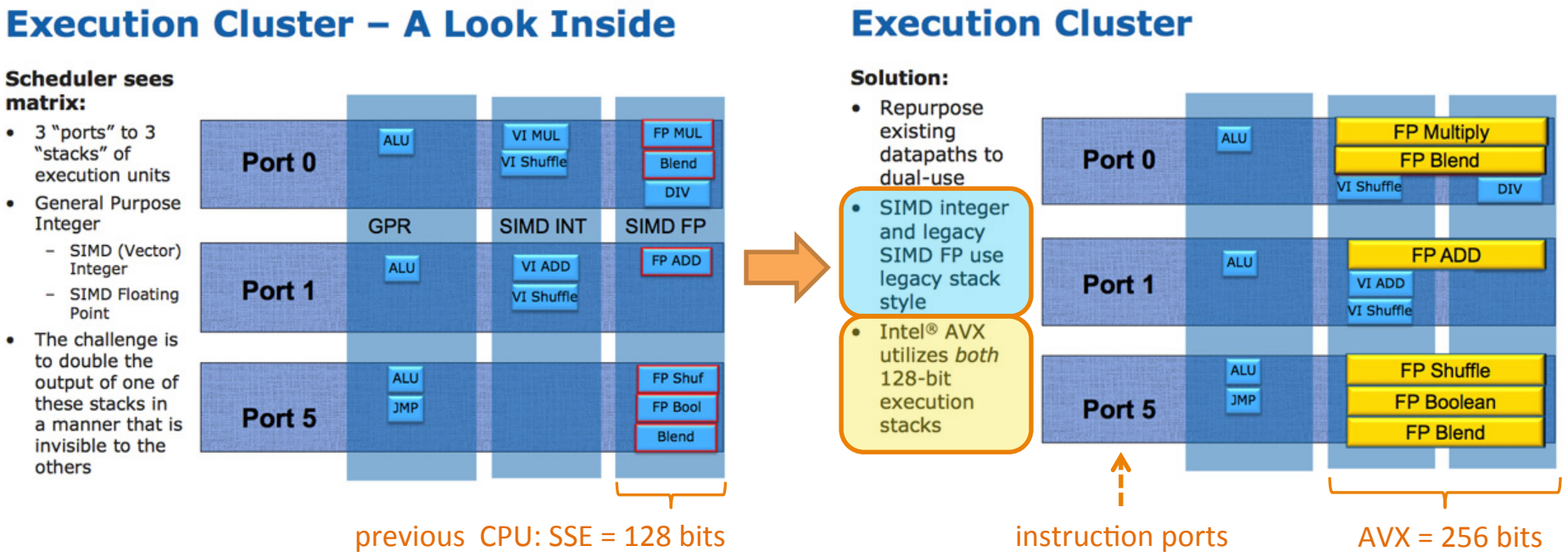
```
#define N 256
...
    double *x, *y, *z;

    // Allocate memory aligned to a 64 byte boundary
    x = (double *)memalign(64,N*sizeof(double));
    y = (double *)memalign(64,N*sizeof(double));
    z = (double *)memalign(64,N*sizeof(double));
```

- Arrays are allocated with 64-byte boundary alignment
- Total storage is 3*256*8 bytes = 6 KB
- Sandy Bridge L1 data cache is 32 KB

What Do We Expect? Do We Get It?

- Sandy Bridge has AVX capability, which supports vector operations up to 256 bits = 32 bytes = 4 doubles
- Expectation (fantasy) is 1 vector add/cycle...
  - The Xeon E5-2620's on the test machine run at 2 GHz
  - The vector.c code adds 256 doubles 1M times
  - 256 M dble / (4 dble/cycle) / (2000 M cycle/s) = **0.032 s**
- Measured time for the loop is **0.068s**, or roughly 2x slower than predicted max...
- Why?

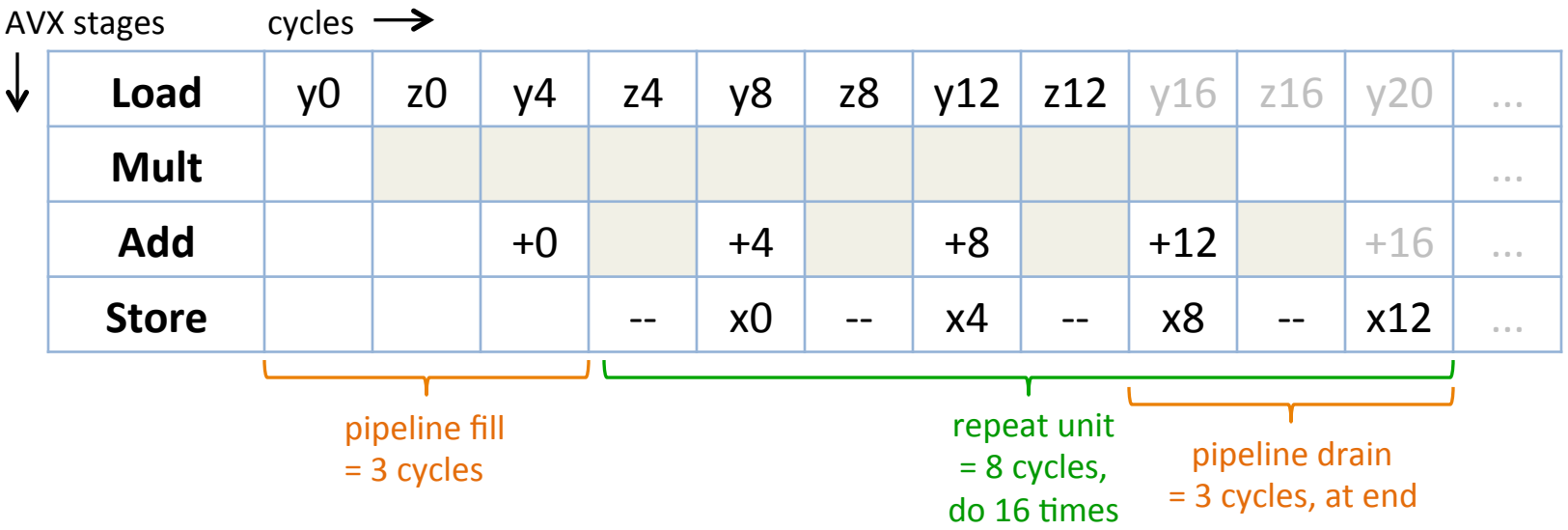
Sandy Bridge (SNB) Architecture



- Memory Unit services **three** data accesses per cycle
  - 2 read requests of up to 16 bytes [2 x 128 bits, 1 AVX load]
  - AND 1 store of up to 16 bytes [128 bits, 0.5 AVX store]

http://www.anandtech.com/show/3922/intels-sandy-bridge-architecture-exposed/3

Under-Filled AVX Pipeline in SNB



- Compiler unrolls j loop by 16, creates 4-stage pipeline
  - Two AVX loads take **2 cycles**; one store also takes **2 cycles**
  - Every *2 cycles*, we get one AVX vector sum of 4 doubles
  - Adding 256 doubles takes 3 + **2 x** (256/4) + 3 = 134 cycles
  - Do 1M times: 134 M cycles / (2000 M cycles/s) = **0.067 s**

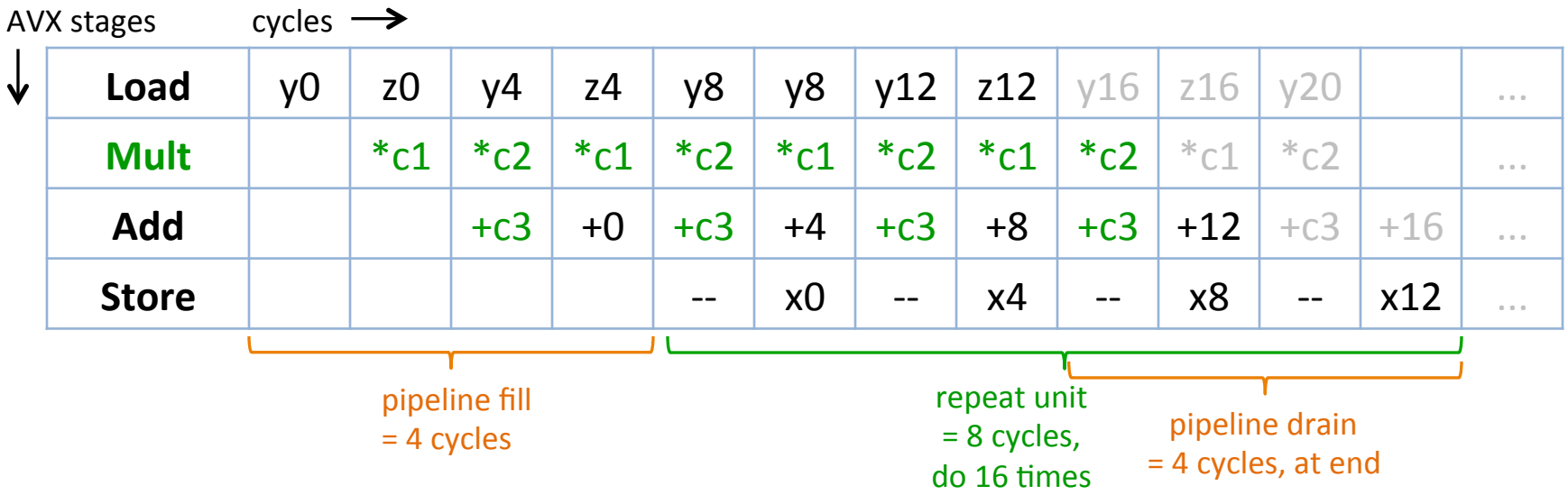
Arithmetic Intensity

- Our very simple loop does not get peak flop/s rate!
- The obvious explanation is that the *arithmetic intensity* (AI = flop/byte) of the code is too low
- As a result, there are unfilled slots in the ADD and MUL pipelines because the VPU must wait for loads and stores to complete
- Even though the right vectors are sitting in L1 cache, low AI makes it impossible to load and store operands fast enough, due to the limited bandwidth to the registers
- Need to introduce more flops between loads and stores

Code with Higher AI: vector2m2a.c

```
#define N 256
...
    tstart = omp_get_wtime();
    // External loop to measure a reasonable time interval
    for(i = 0; i < 1000000; i++){
        #pragma vector aligned
        for( j = 0; j < N; j++){
            x[j] =  2.5*y[j] + 3.6*z[j] + 4.7;
        }
    }
    tend = omp_get_wtime();
```

Filling the AVX Pipeline in SNB?

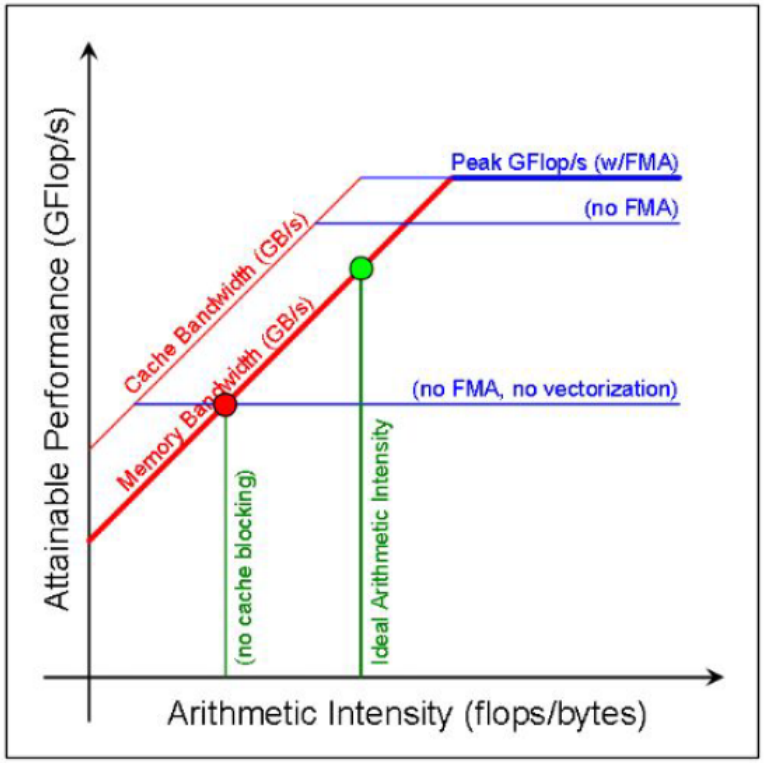


- Compiler fills the 4-stage pipeline for vector2m2a.c
  - SNB can do one AVX add *and* one AVX multiply, per cycle!
  - With low AI, we got only one AVX add every *other* cycle
  - Here, doing 1024 flop takes 4 + 2 x 256/4 + 4 = 136 cycles?
  - If correct: 136 M cycles / (2000 M cycles/s) = **0.068 s**

OK, Prove It! (Umm...)

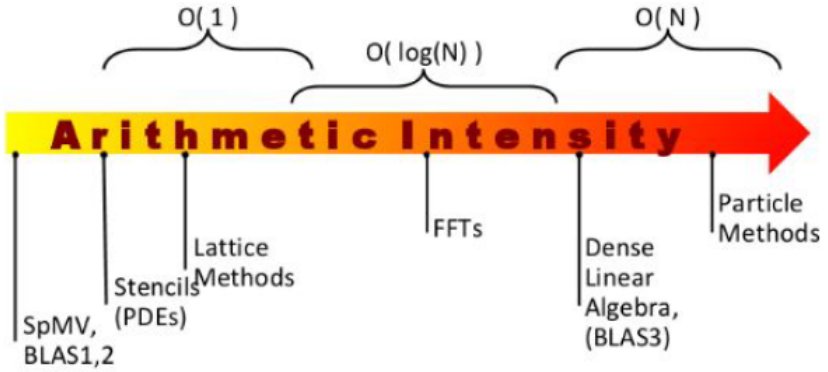
- Use performance models to predict total loop time...
- For vector.c:
  - 0.067 s, predicted
  - 0.068 s, measured= 136 M cycles, 2 cycles/(AVX store)
- For vector2m2a.c:
  - 0.068 s, predicted
  - **0.083 s**, measured = 166 M cycles, 2.5 cycles/(AVX store)
- Turns out that when the denser loop is unrolled to depth 4 for pipelining (j += 16), some registers must be reused!
  - This disrupts the smooth progression of results in the diagram
- *Still* not getting the max flops/s! Can we ever succeed?...

Roofline Analysis



$$\text{Attainable FLOPs/sec} = \min \begin{cases} \text{Peak FLOPs/sec}, \\ \text{Peak Memory Bandwidth} \times \text{Arithmetic Intensity} \end{cases}$$

$$\text{Arithmetic Intensity} = \frac{\text{Total FLOPs}}{\text{Total Bytes}}$$



Deslippe et al., "Guiding Optimization Using the Roofline Model," tutorial presentation at IXPUG2016, Argonne, IL, Sept. 21, 2016. https://anl.app.box.com/v/IXPUG2016-presentation-29

What's the Point?

- Roofline analysis is a way of telling whether a code is compute bound or *memory bound*
- The "roofline" is actually a performance ceiling which is determined by *hardware characteristics*
- The key parameter is the *arithmetic intensity* or AI (flop/byte) of the code: it tells you whether data can be loaded [stored] fast enough from [to] memory
- **Follow-up question:** can a high-AI code actually hit the peak Gflop/s, i.e., what is the *empirical* peak?
- Again want to test Intel Sandy Bridge, 1 core, but with floats instead of doubles

Creating a Good Example Is Tricky

- "High AI" means loads and stores are infrequent
  - Ideally every cycle has 2 vector flops (1 add, 1 multiply)
  - Main vectorized loop must do at least 2 flops *per load*
  - Main vectorized loop must do at least 4 flops *per store*
  - Therefore, 50% of operands must be vectors of constants, or variables that aren't reloaded on every iteration
- Code must be simple enough that the compiler has no trouble vectorizing everything in sight
- Code must not be so simple that the compiler can optimize away all planned work through cleverness

About as Good as It Gets: vecflops.c

```
// These global arrays are aligned on 64-byte boundaries in memory
alignas(64) static float a[NARRAY], b[NARRAY], c[NARRAY];
  ...
  accum = 1.0f;
  // Outer loop does multiple trials to guard against anomalies
  for (k = 0; k < NTIMES; k++) {
    times[k] = mysecond();
    // Middle loop ensures the overall time interval is measurable
    for (j = 0; j < NITERS; j++) {
      accum *= 1.0f/(float)NITERS;
      // Inner loop is the kernel of the flop/s test
      for (i = 0; i < NARRAY; i++)
        accum += 2.5f*(b[i] + 3.6f*c[i]);
    }
    times[k] = mysecond() - times[k];
  } // Define NTIMES = 10, NITERS = 1000000, NARRAY = 2048 to fit in L1d
```

Disassembled Code from SDE

```
XDIS 0000000000400cd0:  AVX C57459148D40806000      vmulps ymm10, ymm1, ymmword ptr [rcx*4+0x608040]
XDIS 0000000000400cd9:  AVX C574592C8D60806000      vmulps ymm13, ymm1, ymmword ptr [rcx*4+0x608060]
XDIS 0000000000400ce2:  AVX C52C581C8D405E6000      vaddps ymm11, ymm10, ymmword ptr [rcx*4+0x605e40]
XDIS 0000000000400ceb:  AVX C51458348D605E6000      vaddps ymm14, ymm13, ymmword ptr [rcx*4+0x605e60]
XDIS 0000000000400cf4:  AVX C57459148D80806000      vmulps ymm10, ymm1, ymmword ptr [rcx*4+0x608080]
XDIS 0000000000400cfd:  AVX C574592C8DA0806000      vmulps ymm13, ymm1, ymmword ptr [rcx*4+0x6080a0]
XDIS 0000000000400d06:  AVX C4414C59E3              vmulps ymm12, ymm6, ymm11
XDIS 0000000000400d0b:  AVX C4414C59FE              vmulps ymm15, ymm6, ymm14
XDIS 0000000000400d10:  AVX C52C581C8D805E6000      vaddps ymm11, ymm10, ymmword ptr [rcx*4+0x605e80]
XDIS 0000000000400d19:  AVX C51458348DA05E6000      vaddps ymm14, ymm13, ymmword ptr [rcx*4+0x605ea0]
XDIS 0000000000400d22:  AVX C4C15C58E4              vaddps ymm4, ymm4, ymm12
XDIS 0000000000400d27:  AVX C58458D2                vaddps ymm2, ymm15, ymm2
XDIS 0000000000400d2b:  AVX C4414C59E3              vmulps ymm12, ymm6, ymm11
XDIS 0000000000400d30:  AVX C4414C59FE              vmulps ymm15, ymm6, ymm14
XDIS 0000000000400d35:  AVX C59C58DB                vaddps ymm3, ymm12, ymm3
XDIS 0000000000400d39:  AVX C4410458C0              vaddps ymm8, ymm15, ymm8
XDIS 0000000000400d3e: BASE 4883C120                add rcx, 0x20
XDIS 0000000000400d42: BASE 4881F900080000          cmp rcx, 0x800
XDIS 0000000000400d49: BASE 7285                    jb 0x400cd0
```

- Intel Software Development Emulator counts *real instructions*
- Unrolled loop contains 16 AVX flops, 8 AVX loads (ptr source), 0 AVX stores; accum = ymm{2,3,4,8}; 14 of 16 registers used

Observations

- Main loop satisfies 2 AVX flops per 1 AVX load
  - Can't really have *fewer* loads: compiler just precomputes operations involving constants, and the flops go away
  - There are enough registers to hold intermediate results
- SDE says the main loop has 640M executions (98.4%)
  - Main loop is unrolled, vectorized: 4 x 8 = 32 results per trip
  - Original inner loop becomes 2048/32 = 64 main-loop trips
  - Outer loops multiply the 64 main-loop trips by 10M
- Code's built-in timer gives **27.7 Gflop/s**, 87% of "peak"*
  - Ignores periodic rescaling of accum and the final reduction

*For full AVX on one core of E5-2620 v3, e.g, clock drops to 87% of 2.4 GHz rate to avoid overheating : https://www.microway.com/knowledge-center-articles/detailed-specifications-intel-xeon-e5-2600v3-haswell-ep-processors/ (refer to figure with "Clock Speeds for Single-Core Operation")

A Look at Some Other Kernels

Kernel	Gflop/s	Notes
`accum += 2.5f*(b[i] + 3.6f*c[i]);`	**27.7**	Most straightforward
`accum += 2.5f*(b[i] + 3.6f*b[i+8]);`	29.9	Reuses previous load
`accum += 2.5*(b[i] + 3.6*c[i]);`	4.2	Constants aren't floats!
`accum += 2.5f*(b[i+1] + 3.6f*c[i+1]);`	23.6	Imperfect alignment
`accum += 2.5f*(b[i+1] + 3.6f*c[i+2]);`	16.6	Worse alignment
`a[i] = 2.5f*b[i] + 3.6f*c[i] + 4.7f;`	23.9	Compiler acts tricky*
`a[i] = b[i] + c[i];`	7.3	Low AI

Compile line: `icc -O3 -xAVX -g`

* Compiler somehow cuts iterations from 1 million to 500,000 if the (unneeded!) reset statement `accum *= 1.0f/(float)NITERS;` is left in the timing loop and accum is printed at the end!

Speedups from Vectorization

Kernel	-no-vec	-vec	Speedup
`accum += 2.5f*(b[i] + 3.6f*c[i]);`	3.76	**27.7**	7.4
`accum += 2.5f*(b[i] + 3.6f*b[i+8]);`	2.27	29.9	13.1?
`accum += 2.5*(b[i] + 3.6*c[i]);`	0.87	4.2	4.8*
`accum += 2.5f*(b[i+1] + 3.6f*c[i+1]);`	3.76	23.6	6.3*
`accum += 2.5f*(b[i+1] + 3.6f*c[i+2]);`	3.76	16.6	4.4*
`a[i] = 2.5f*b[i] + 3.6f*c[i] + 4.7f;`	3.18	23.9	7.5
`a[i] = b[i] + c[i];`	1.31	7.3	5.6

* These kernels do not vectorize optimally, due to either a violation of 64-byte alignment or mixing of types

? This kernel has >8x speedup because loading twice from one array (with index offset) mucks up the non-vectorized code

Vector Speedup: Hard to Predict!

Aspects of SNB architecture make the vector performance and speedup hard to predict with precision…

- Complication #1: per cycle, SNB can do 1 AVX load and 0.5 AVX store, but 2 scalar loads and 1 scalar store
  - Therefore, low-AI, bandwidth-limited kernels operating in single precision realize a speedup that is more like 4x, not 8x
- Complication #2: Turbo Boost technology adjusts the clock rate according to load, even on a single core
  - Slows the clock down for heavy AVX; speeds it up for scalars
- As we saw with SDE, performance ultimately depends on how the compiler turns code into instructions – in detail!

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 hardware
  - At a minimum, try to reuse everything you bring into cache!
  - Try also to combine additions with multiplications
- 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
- Study compiler reports: is it vectorizing where it should?
- Evaluate execution performance: is it near the roofline?
- 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

Challenge: Loop Dependencies

- Vectorization changes the order of computation compared to sequential case
- Compiler must be able to prove that vectorization will produce correct results
- Need to consider independence of *unrolled* loop operations – it may depend on vector width
- Compiler performs dependency analysis, unless it is prevented 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] → 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

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

Loop Dependencies: Write After Read

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

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

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

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

Applying each operation sequentially:

a[0] = a[1] + b[0] → a[0] = 1 + 5 → a[0] = 6

a[1] = a[2] + b[1] → a[1] = 2 + 6 → a[1] = 8

a[2] = a[3] + b[2] → a[2] = 3 + 7 → a[2] = 10

a[3] = a[4] + b[3] → a[3] = 4 + 8 → a[3] = 12

a = {6, 8, 10, 12 , 4}

Loop Dependencies: Write After Read

Now let's try vector operations:

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

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

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

Applying vector operations, i={0,1,2,3}:

a[i+1] = {1,2,3,4} (load)

b[i] = {5,6,7,8} (load)

{1,2,3,4} + {5,6,7,8} = {6, 8, 10, 12} (operate)

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

a = {6, 8, 10, 12 , 4} = {6, 8, 10, 12 , 4} VECTORIZABLE

Loop Dependencies: Summary

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

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

- Write After Read
  - Also called “anti” dependency
  - Variable read first, then written
  - vectorizable

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

Loop Dependencies: Summary

- 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

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

Loop Dependencies: Aliasing

- In C, pointers can hide data dependencies!
  - Memory regions 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

Optimization Reports

- They show whether loops are vectorized or not , and why
- Intel: `-qopt-report=<n> <-qopt-report-phase=vec>`
  - 0: No report
  - 1: Reports which loops were vectorized
  - 2: (default level) Adds loops not vectorized, plus a short reason
  - 3: Adds loop summary information from the vectorizer
  - 4: Adds more detail about vectorized and non-vectorized loops
  - 5: Adds details about any proven or assumed data dependencies
- Levels 2+ tell you where dependencies were found
- Compiler is conservative: you need to dig into the .optrpt files and see if the dependencies really exist in the code

Loop Dependencies: Vectorization Hints

- Compiler must prove to itself that there is no data dependency that will affect correctness of the result
- Sometimes, this is impossible
  - e.g., unknown index offset, complicated use of pointers
- To stop the Intel compiler from worrying, you can give it the IVDEP (Ignore Vector DEPendencies) hint
  - It assures the compiler, "It's safe to assume no dependencies"
  - Example: assume we know M > vector width in doubles...

```
void vec1(double s1, int M,
    int N, double *x) {
#pragma ivdep
for(i=0; i<N-M; i++) x[i] = x[i+M] + s1;
```

Compiler Hints Affecting Vectorization

- Theses are for the Intel compiler only
- They affect whether loop is vectorized or not
- #pragma ivdep
  - Says to assume no dependencies
  - Allows compiler to vectorize loops that otherwise seem unsafe
- #pragma vector always, #pragma simd
  - Always vectorize if technically possible to do so
  - Overrides compiler's decision not to vectorize based upon cost
- #pragma novector
  - Do not vectorize

Loop Dependencies: Language Constructs

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

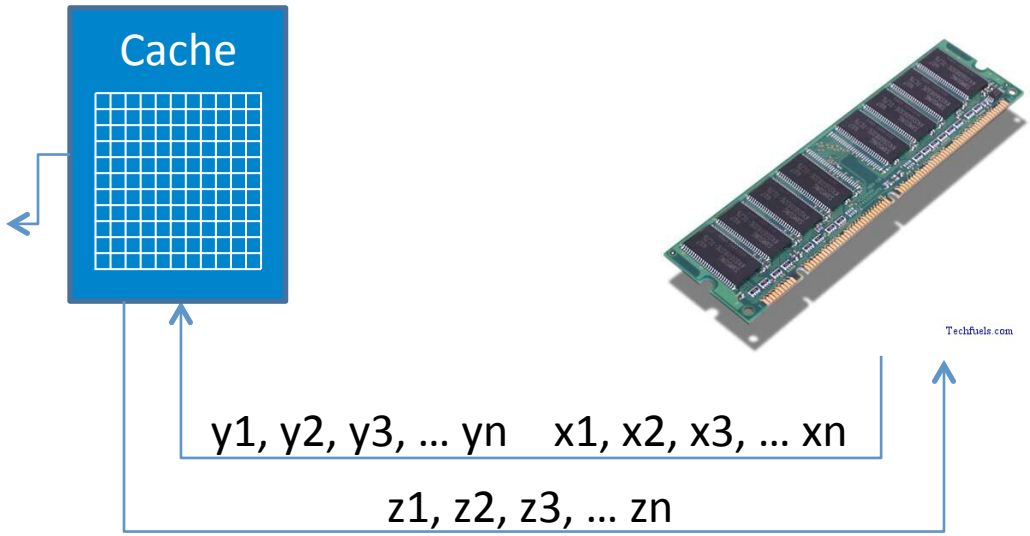
```
void compute(double * restrict a,
    double * restrict b, double * restrict c) {
     for (i=0; i<N; i++) {
       a[i] = b[i] + c[i];
     }
}
```

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

Cache and Alignment

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

ymm2 ymm0 ymm1



- 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

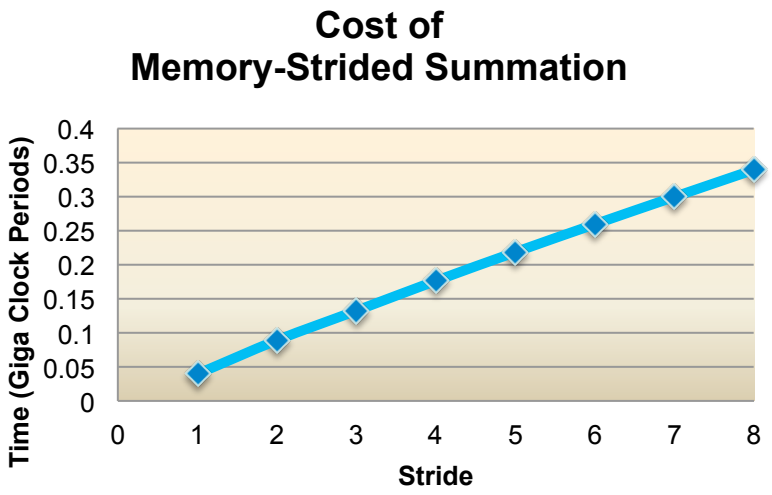
- Fastest usage pattern is “stride 1”: perfectly sequential
  - Cache lines arrive in L1d as full, ready-to-load vectors
- Stride-1 constructs:
  - Storing data in structs of arrays vs. arrays of structs
  - Looping through arrays so their “fast” dimension is innermost
    - Fortran: stride 1 on first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

```
do j=1,n
   do i=1,n
      a(i,j)=b(i,j)*s
end do; end do
```

```
for(j=0;j<n;j++)
   for(i=0;i<n;i++)
      a[j][i]=b[j][i]*s;
```

Strided Access

- Striding through memory reduces effective memory bandwidth!
  - Roughly by 1/stride
- It's worse than non-aligned access, as data must be "gathered" by hardware to fill a vector register




```
do i=1,4000000*istride,istride
   a(i) = b(i) + c(i) *
sfactor
end do
```

Diagnosing Cache & Memory Deficiencies

- Really bad stride patterns may prevent vectorization
  - In vector report: “vectorization possible but seems inefficient”
- Bad stride and other problems may be difficult to detect
  - Merely result in poorer performance than expected
- Profiling tools like Intel VTune can help
- Intel Advisor makes recommendations based on source

Conclusion

- The compiler “automatically” vectorizes tight loops
- Write code that is vector-friendly
  - Innermost loop accesses arrays with stride one
  - Data in cache are reused; loads are stores are minimized
  - Loop bodies consist of simple multiplications and additions
- Write code that avoids the potential issues
  - No loop-carried dependencies, branching, 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 so; use #pragma if not