



Cornell University
Center for Advanced Computing

Vectorization on KNL

Steve Lantz

Senior Research Associate

Cornell University Center for Advanced Computing (CAC)

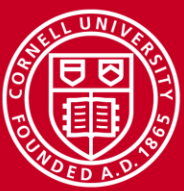
steve.lantz@cornell.edu

High Performance Computing on Stampede 2, with KNL, Jan. 23, 2017



What is Vectorization?

- **Hardware Perspective:** Specialized instructions, registers, or functional units to 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.



Vectorization via SIMD: Motivation

- CPU speeds reach a plateau
 - Power limitations!
 - Many “slow” transistors more efficient than fewer “fast” transistors
- Process improvements make physical space cheap
 - Moore’s law, 2x every 18–24 months
 - Easy to add more “stuff”
- One solution: more cores
 - First dual-core Intel CPUs appeared in 2005
 - Counts have grown rapidly, e.g., 8 in Sandy Bridge, 61–72 on Xeon Phi
- Another solution: more FPU units per core – vector operations
 - First appeared on a Pentium with MMX in 1996
 - Vector widths have ballooned recently, e.g., 512-bit (8 doubles) on MIC

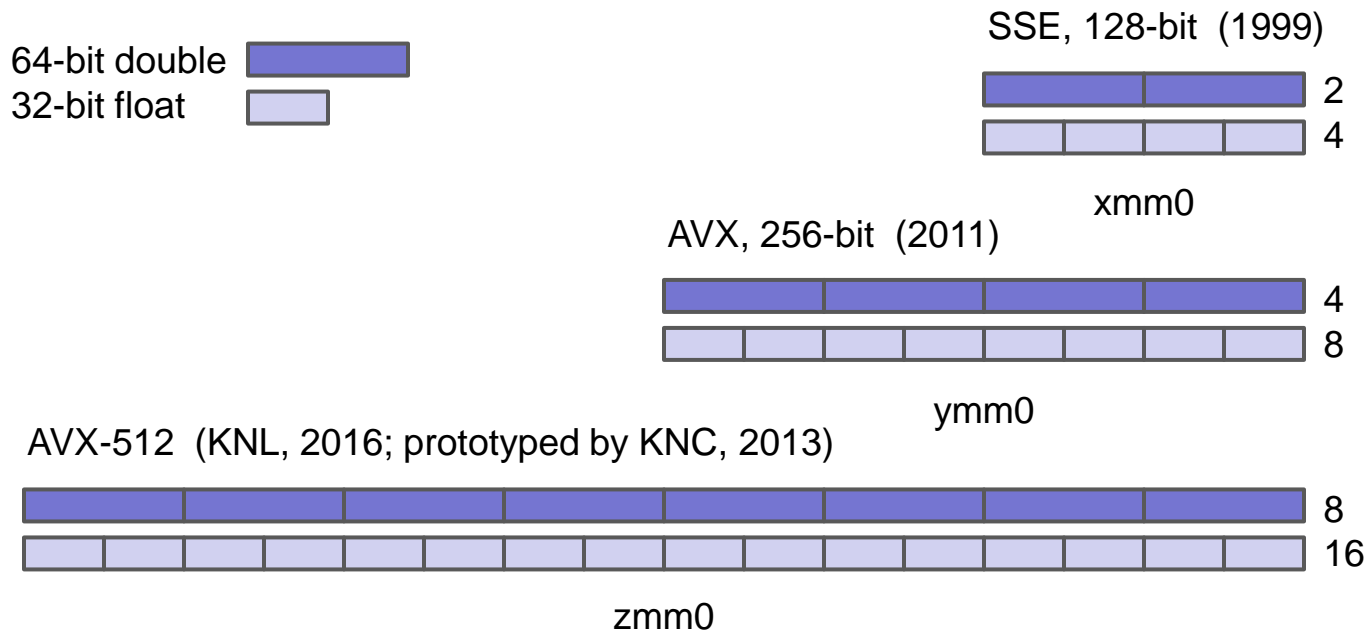


Vectorization: Hardware

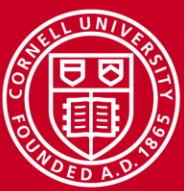
- Goal: parallelize computations over vector arrays
- SIMD: Single Instruction Multiple Data
- Many instances of a single operation executing simultaneously
 - Late '90s – present, commodity CPUs (x86, x64, PowerPC, etc.)
 - Small vectors, few cycles per instruction
 - Newer CPUs (since Sandy Bridge) can pipeline some SIMD instructions as well – best of both worlds
- KNL has two VPU's per core, both of which can execute an add and a multiply operation every cycle (FMA = Fused Multiply-Add)



Evolution of Vector Registers and Instructions



- Core has 16 (SSE, AVX) or 32 (AVX-512) separate vector registers
- In 1 cycle, both ADD and MUL units can do operations with registers



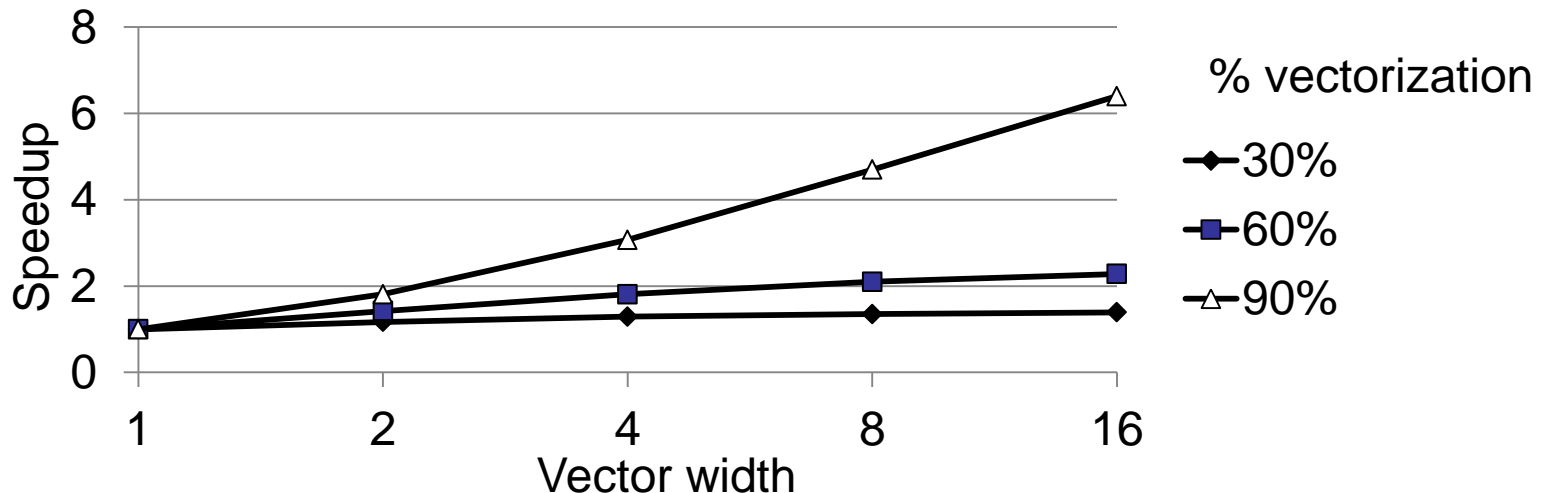
Speed

- True SIMD parallelism – Can take 1 cycle per floating point computation on all values in a vector register.
 - Exception: Slow operations like division, square roots
- Speedup (compared to no vector) proportional to vector width
 - 128-bit SSE – 2x double, 4x single
 - 256-bit AVX – 4x double, 8x single
 - 512-bit MIC – 8x double, 16x single
- Hypothetical AVX example: 8 cores/CPU * 4 doubles/vector * 2.0 GHz = 64 Gflops/CPU DP
- Gets even better with FMA (Fused Multiply Add), $a=a+(b*c)$; multiply and add in the same clock cycle (Haswell+, Xeon Phi)



Speed

- Clearly memory bandwidth is potential issue, we'll explore this later
 - Poor cache utilization, alignment, memory latency all detract from ideal
- SIMD is parallel, so Amdahl's law is in effect!
 - Serial/scalar portions of code or CPU are limiting factors
 - Theoretical speedup is only a ceiling





User Perspective

Let's take a step back – how can we leverage this power

- 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 but risky
- Let the compiler figure it out
 - Relatively “easy” for user, “challenging” for compiler
 - Less expressive languages like C make compiler’s job more difficult
 - Compiler may need some guidance
- Link to an optimized library that does the actual work
 - e.g. Intel MKL, written by people who know all the tricks
 - Get benefits “for free” when running on supported platform



Vector-Aware Coding

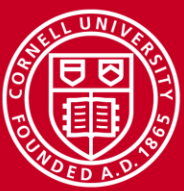
- Know what makes codes vectorizable at all
 - “for” loops (in C) or “do” loops (in fortran) that meet certain constraints
- Know where vectorization will help
- Evaluate compiler output
 - Is it really vectorizing where you think it should?
- Evaluate execution performance
 - Compare to theoretical speedup
- Know data access patterns to maximize efficiency
- Implement fixes: directives, compilation flags, and code changes
 - Remove constructs that make vectorization impossible/impractical
 - Encourage/force vectorization when compiler doesn't, but should
 - Better memory access patterns



Writing Vectorizable Loops

Basic requirements of vectorizable loops:

- Countable at runtime
 - Number of loop iterations is known before loop executes
 - No conditional termination (break statements)
- Have single control flow
 - No switch statements
 - 'if' statements are allowable when implemented as masked assignments
- Must be the innermost loop if nested
 - Note, the compiler may reverse loop order as an optimization!
- No function calls
 - Basic math is allowed: pow(), sqrt(), sin(), etc
 - Some inline functions allowed



Conceptualizing Compiler Vectorization

- Think of vectorization in terms of loop unrolling
 - Unroll N interactions of loop, where N elements of data array fit into 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
```



Compiling Vector loops

- Intel Compiler:
 - Vectorization starts at optimization level `-O2`
 - Will default to SSE instructions and 128-bit vector width
 - Use `-xAVX` to use AVX and 256-bit vector width
 - Embed SSE *and* AVX instructions in the same binary with `-axAVX`
 - Use `-xMIC-AVX512` for KNL
 - Get a vectorization report (in `.optrpt` file) with `-qopt-report=<n>`
- GCC 4.9 or higher:
 - Auto-vectorization is generally enabled at `-O3` optimization level
 - Previously needed `-ftree-vectorize` flag and optimization `> -O2`
 - For Xeon Phi: `-mavx512f -mavx512cd -mavx512er -mavx512pf`
 - Vectorization reports: `-fopt-info-vec -fopt-info-vec-missed`



Vectorization Lab

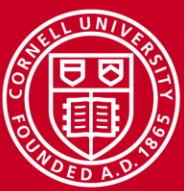
1. Compile **vector.c** as a native MIC application:

```
icc -qopenmp -O3 -xMIC-AVX512 ./vector.c -o vec
```

2. And also as a MIC application but disabling vectorization:

```
icc -qopenmp -O3 -xMIC-AVX512 -no-vec ./vector.c -o novvec
```

3. Run both executables and take note of the timing difference. How much speedup comes from the vectorization?
4. Does this make sense given what you have learned about the KNL architecture?



Optimization Reports

Let's get some information about vectorization in this example code.

Compile the code again, but add a basic optimization report option to the compilation line:

```
icc -qopenmp -O3 -xMIC-AVX512 -qopt-report=2 ./vector.c \  
-o vec
```

This will generate a report file called **vector.optrpt**

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

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



Optimization Report: What Vectorized, What Didn't

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 an external function (RAND) in lines 34 and 35 or the example:

```
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, `-qoptprt=4`, in order 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 8
remark #15399: vectorization support: unroll factor set to 8
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: 0.620
remark #15478: estimated potential speedup: 12.800
remark #15488: --- end vector loop cost summary ---
remark #25015: Estimate of max trip count of loop=4
```




Challenge: Loop Dependencies

- Vectorization changes the order of computation compared to sequential case
- Compiler must be able to prove that vectorization will produce correct result
- Need to consider independence of *unrolled* loop operations – depends on vector width
- Compiler performs dependency analysis



Loop Dependencies: Read After Write

Consider the loop:

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

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

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

Applying each operation sequentially:

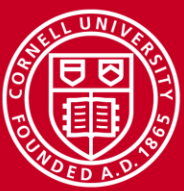
$$a[1] = a[0] + b[1] \rightarrow a[1] = 0 + 6 \rightarrow a[1] = 6$$

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

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

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

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



Loop Dependencies: Read After Write

Consider the loop:

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

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

```
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: Write after Read

Consider the loop:

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

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

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

Applying each operation sequentially:

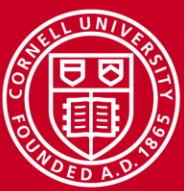
$$a[0] = a[1] + b[0] \rightarrow a[0] = 1 + 5 \rightarrow a[0] = 6$$

$$a[1] = a[2] + b[1] \rightarrow a[1] = 2 + 6 \rightarrow a[1] = 8$$

$$a[2] = a[3] + b[2] \rightarrow a[2] = 3 + 7 \rightarrow a[2] = 10$$

$$a[3] = a[4] + b[3] \rightarrow a[3] = 4 + 8 \rightarrow a[3] = 12$$

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



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; i++)  
    a[i] = a[i+1] + b[i];
```

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

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

$a = \{0, 6, 8, 10, 12\} = \{0, 6, 8, 10, 12\}$ 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?

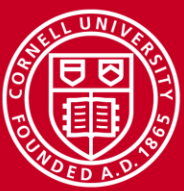
```
void compute(double *a,  
             double *b, double *c) {  
    for (i=1; i<N; i++) {  
        a[i]=b[i]+c[i];  
    }  
}
```

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



Vectorization Reports

- Shows which loops are or are not vectorized, and why
- Intel: `-vec-report=<n>`
 - 0: None
 - 1: Lists vectorized loops
 - 2: Lists loops not vectorized, with explanation
 - 3: Outputs additional dependency information
 - 4: Lists loops not vectorized, without explanation
 - 5: Lists loops not vectorized, with dependency information
- Reports are essential for determining where the compiler finds a dependency
- Compiler is conservative, you need to go back and verify that there really is a dependency.



Loop Dependencies: Vectorization Hints

- Compiler must prove there is no data dependency that will affect correctness of result
- Sometimes, this is impossible
 - e.g. unknown index offset, complicated use of pointers
- Intel compiler solution: IVDEP (Ignore Vector DEpendencies) hint.
 - Tells compiler “Assume there are no dependencies”

```
subroutine  
vec1(s1,M,N,x)  
...  
!DEC$ IVDEP  
do i = 1,N  
  x(i) = x(i+M) + s1  
end do
```

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



Compiler Hints Affecting Vectorization

- For Intel compiler only
- Affect whether loop is vectorized or not
- `#pragma ivdep`
 - Assume no dependencies.
 - Compiler may vectorize loops that it would otherwise think are not vectorizable
- `#pragma vector always`, `#pragma simd`
 - Always vectorize if technically possible to do so.
 - Overrides compiler's decision to not vectorize based upon cost
- `#pragma novector`
 - Do not vectorize

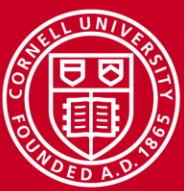


Loop Dependencies: Language Constructs

- 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=1; i<N; i++) {  
        a[i]=b[i]+c[i];  
    }  
}
```

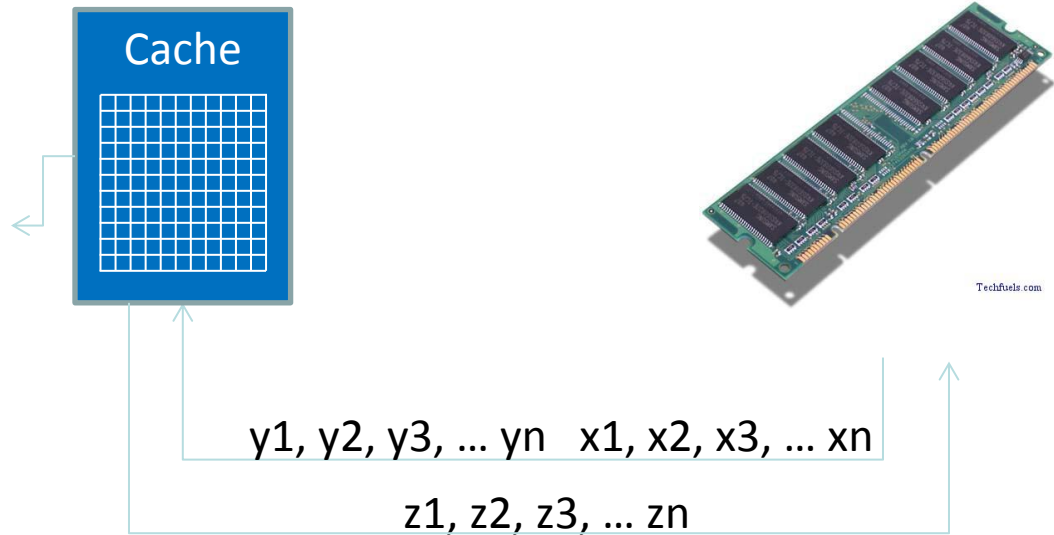
- May need compiler flags to use, e.g. `-restrict`, `-std=c99`



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 requires concerns beyond SIMD unit!
 - Registers: Alignment of data on 16-, 32-, or 64-byte boundaries
 - Cache: Cache is fast, memory is slow
 - Memory: Sequential access much faster than random/strided



Strided Access

- Fastest usage pattern is “stride 1”: perfectly sequential
- Best performance when CPU can load L1 cache from memory in bulk, sequential manner
- Stride 1 constructs:
 - Iterating structs of arrays vs. arrays of structs
 - Multi dimensional array:
 - Fortran: stride 1 on “inner” dimension
 - C/C++: Stride 1 on “outer” dimension

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

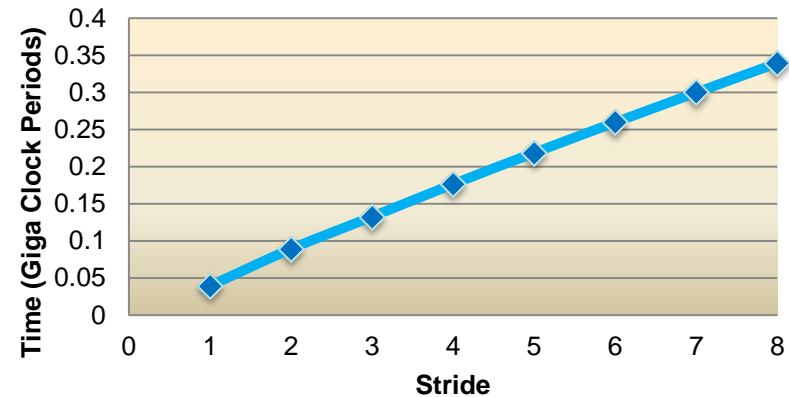
```
for (j=0;j<n;j++)  
  for (i=0;i<n;i++)  
    a[j][i]=b[j][i]*s;
```



Strided Access

- Striding through memory reduces effective memory bandwidth!
 - For DP, roughly 1-stride/8
- Worse than non-aligned access. Lots of memory operations to populate a cache line, vector register

Cost of
Memory-Strided Summation



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




Diagnosing Cache and Memory Deficiencies

- Obviously bad stride patterns may prevent vectorization at all:
 - In vector report: “vectorization possible but seems inefficient”
- Otherwise, may be difficult to detect
 - No obvious assembly instructions, other than a proliferation of loads and stores
 - Vectorization performance farther away from ideal than expected
- Profiling tools can help
 - Intel VTune, PerfExpert (available at TACC)
 - Visualize CPU cycles wasted in data access (L1 cache miss, TLB misses, etc.)
- Intel Advisor can make recommendations for you based on your source code



Conclusion

- Vectorization occurs in tight loops “automatically” by the compiler
- Need to know where vectorization should occur, and verify that compiler is doing that.
- Need to know if a compiler’s failure to vectorize is legitimate
 - Fix code if so, use `#pragma` if not
- Need to be aware of caching and data access issues
 - Very fast vector units need to be well fed