# Vectorization

Aaron Birkland

Consultant

Cornell CAC

*Vectorization: Ranger to Stampede Transition*

*December 11, 2012*

Special thanks to Dan Stanzione, Lars Koesterke, Bill Barth, and Kent Milfield at TACC for materials and inspiration.

## 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: Hardware

- Goal: parallelize computations over vector arrays
- Two major approaches: pipelining, SIMD (Single Instruction Multiple Data)
- Pipelining: Several different tasks executing simultaneously
  - Popular through 1990s in supercomputing contexts
  - Large vectors, Many cycles per “instruction”
- SIMD: Many instances of a single task executing simultaneously
  - Late ‘90s – present, commodity CPUs (x86, x64, PowerPC, etc)
  - Small vectors, few cycles per instruction
  - Newer CPUs (Sandy Bridge) can pipeline some SIMD instructions as well – best of both worlds.

## Vectorization: Pipelining

| Clock Cycle | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| 1 | $X_1$ | | | |
| 2 | $X_2$ | $X_1^2$ | | |
| 3 | $X_3$ | $X_2^2$ | $X_1^2+8$ | |
| 4 | $X_4$ | $X_3^2$ | $X_2^2+8$ | $(X_1^2+8)/2$ |
| 5 | $X_5$ | $X_4^2$ | $X_3^2+8$ | $(X_2^2+8)/2$ |
| 6 Load | | $X_5^2$ | $X_4^2+8$ | $(X_3^2+8)/2$ |
| 7 | Square | | $X_5^2+8$ | $(X_4^2+8)/2$ |
| 8 | | Add | | $(X_5^2+8)/2$ |

Divide

Hypothetical pipelined operations

## Vectorization: Hardware: SIMD

| Clock | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| 1 | $[X_1, X_2 \ldots X_5]$ | | | |
| 2 | | $[X_1^2, X_2^2 \ldots X_5^2]$ | | |
| 3 | | | $[X_1^2+8, X_2^2+8 \ldots X_5^2+8]$ | |
| 4 | Load | Square | Add | $[(X_1^2+8)/2,$ $(X_2^2+8)/2$ … $(X_5^2+8)/2]$ |

Divide

Hypothetical SIMD operations

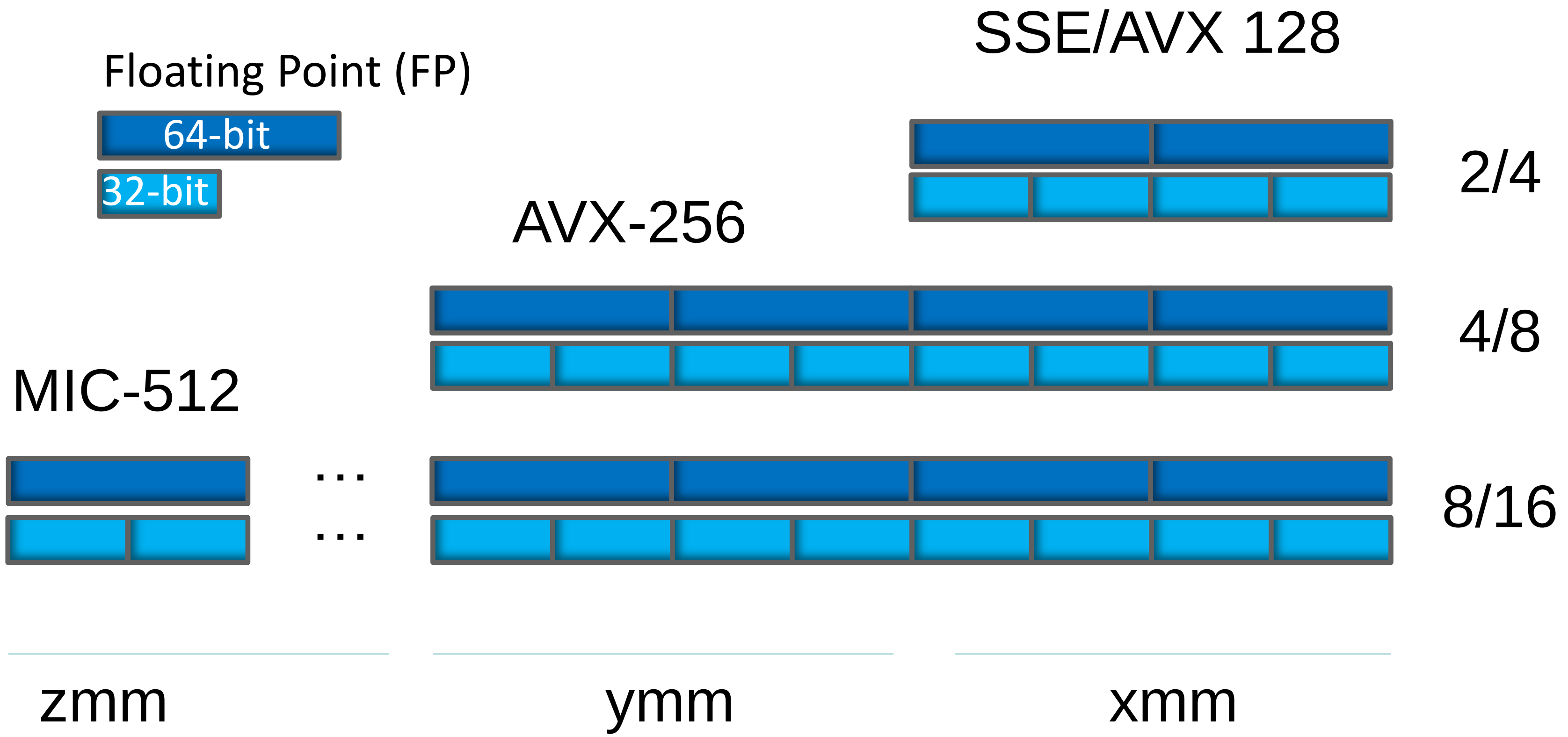
## Vectorization via SIMD: Motivation

- 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 appear in 2005
  - Increasing in number rapidly (e.g. 8 in Stampede, 60+ on MIC)
- Another Solution: More FPU units per core – vector operations
  - First appeared on a Pentium with MMX in 1996
  - Increasing in vector width rapidly (e.g. 512-bit [8 doubles]) on MIC

## Vectorization via SIMD: History

| Year | Registers | Instruction Set | |
|---|---|---|---|
| ~1997 | 80-bit | MMX | Integer SIMD (in x87 registers) |
| ~1999 | 128-bit | SSE1 | SP FP SIMD (xMM0-8) |
| ~2001 | 128-bit | SSE2 | DP FP SIMD (xMM0-8) |
| --- | 128-bit | SSEx | |
| ~2010 | 256-bit | AVX | DP FP SIMD (yMM0-16) |
| ~2012 | 512-bit | (MIC) | |
| ~2014 | 512-1024-bit | (Haswell) | |

## Vector Registers



## SIMD Instructions

- Loading
  - movupd xmm0 … (SSE move unaligned packed double into 128-bit )
  - vmovaps ymm0 … (AVX move aligned packed single into 256-bit)
- Operating
  - vaddpd ymm1 ymm2 (AVX add packed double 256-bit)
  - addsd (SSE Add scalar doubles – SSE, but NOT vector op!)
- KEY:
  - v = AVX
  - p, s = packed, scalar
  - u, a = unaligned, aligned
  - s, d = single, double

## AVX Instructions

- Optimal for 64-bit operation
- Uses Vex prefix (V)
  - Extendable to 512-bit or 1024-bit SIMD
  - Can Reference 3 or 4 registers
  - New instructions, broadcast to registers, mask, permute, etc
- FMA (Fused Multiply Add)  available soon (Haswell/AVX2)

## AVX Instructions

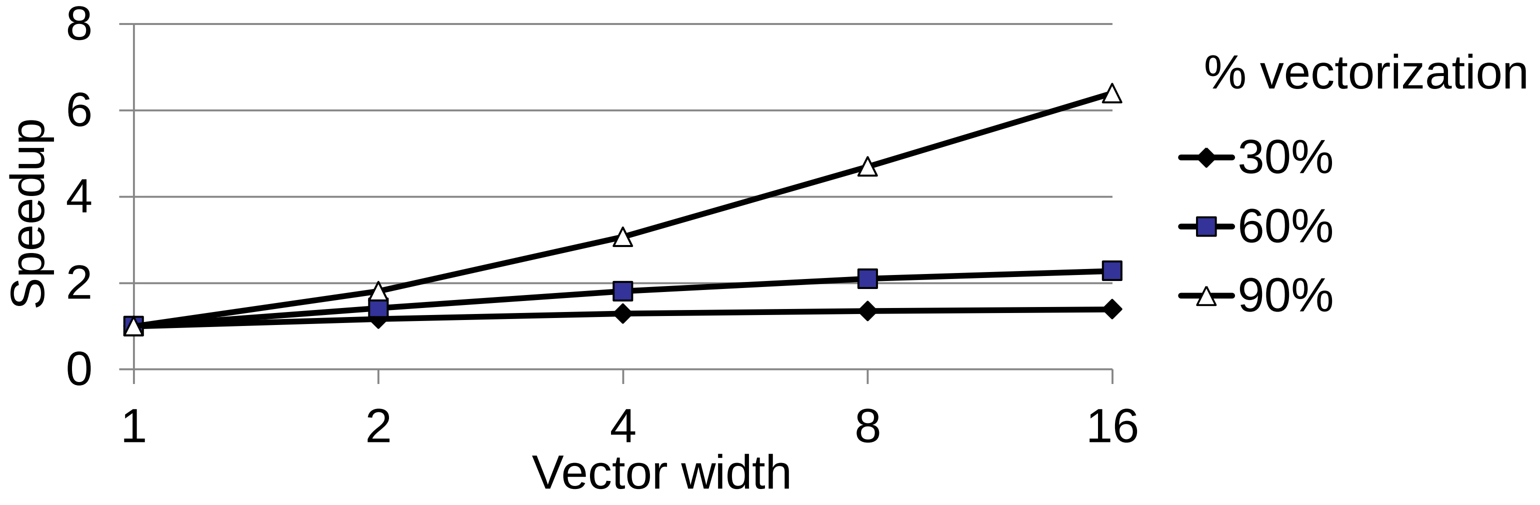
| | AVX 128-bit VEX Prefix | AVX 256-bit Vex Prefix |
|---|---|---|
| Legacy SIMD | | |
| Scalar | Yes | No |
| Vector Data Movement | Yes | Yes |
| Vector FP | Yes | Yes |
| Vector Int | No | No |
| Int | No | No |
| New Functionality | | |
| Permute (v) | Yes | Yes |
| Mask (v) | Yes | Yes |
| Broadcast (v) | Yes | Yes |
| Insert/Extract/Zero | No | Yes |

## Speed

- True SIMD parallelism – typically 1 cycle per floating point computation
  - 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
  - Pipelining could make this even greater!

## 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 in assembly
  - Ultimate performance potential, but only for the brave
- Program in 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 hand holding.
- 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  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 Vector 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 they can be implemented as masked assignments
  - Must be the innermost loop if nested
    - 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
  - Can embed SSE *and* AVX instructions in the same binary with `-axAVX`
    - Will run AVX on CPUs with AVX support, SSE otherwise
  - `-vec-report=<n>` for a vectorization report
- GCC
  - Vectorization is disabled by default, regardless of optimization level
  - Need `-ftree-vectorize` flag, combined with optimization > `-O2`
  - SSE by default, `-mavx -march=corei7-avx` for AVX
  - `-ftree-vectorizer-verbose` for a vectorization report

## Lab: Simple Vectorization

In this lab you will

- Use the Intel and gcc compilers to create vectorized with non-vectorized code
- Compare the performance of vectorized vs non-vectorized code
- Take an initial look at compiler vectorization reports

## Lab: Simple Vectorization

- Though contrived, observed vector performance increase was almost close to ideal – almost 100% code in tight vectorizable loop
- Results for Sandy Bridge (Laptop):

| Compile Options | Time |
|---|---|
| -no-vec –O3 | .937s |
| -O3 | .242s |
| -O3 -axAVX | .125s |

## 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] → 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 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:

$$\begin{tikzcd}
a[1] = a[0] + b[1] \arrow[r] & a[1] = 0 + 6 \arrow[r] & a[1] = 6 \arrow[dl] \\
a[2] = a[1] + b[2] \arrow[r] & a[2] = 6 + 7 \arrow[r] & a[2] = 13 \arrow[dl] \\
a[3] = a[2] + b[3] \arrow[r] & a[3] = 13 + 8 \arrow[r] & a[3] = 21 \arrow[dl] \\
a[4] = a[3] + b[4] \arrow[r] & a[4] = 21 + 9 \arrow[r] & a[4] = 30
\end{tikzcd}$$

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 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

- 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

- 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 safe?:

```
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, add bounds checking tests (overhead)

## 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
  - 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`

## Lab: Vector hinting and reports

- In this lab, we will use the Intel compiler to compile code that has a vector dependency
- By analyzing the reports and adding #pragma statements, we will see if we can get around the compiler’s dependency analysis checks, and what the effects are.

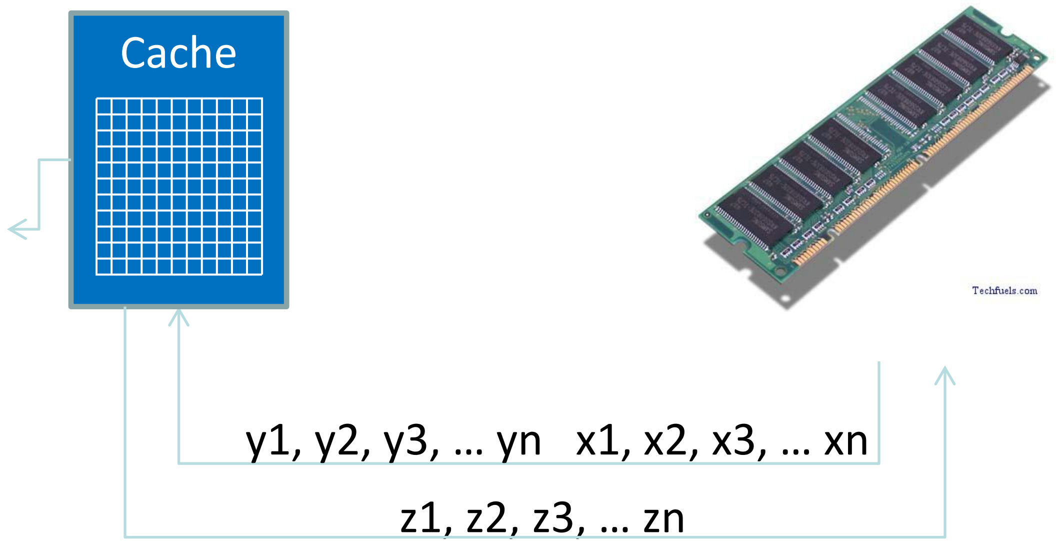
## Lab: Vector Hinting and Reports

- Multiple levels of vector reports can help diagnose potential issues
- Compilers (Intel) must be conservative when vectorizing loops. User markup (e.g #pragma)
- Sometimes this conservatism is warranted.
  - Can lead to incorrect results if we're not careful when we override!
- Domain of incorrect results can be influenced by vector width.

## Cache and Alignment

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

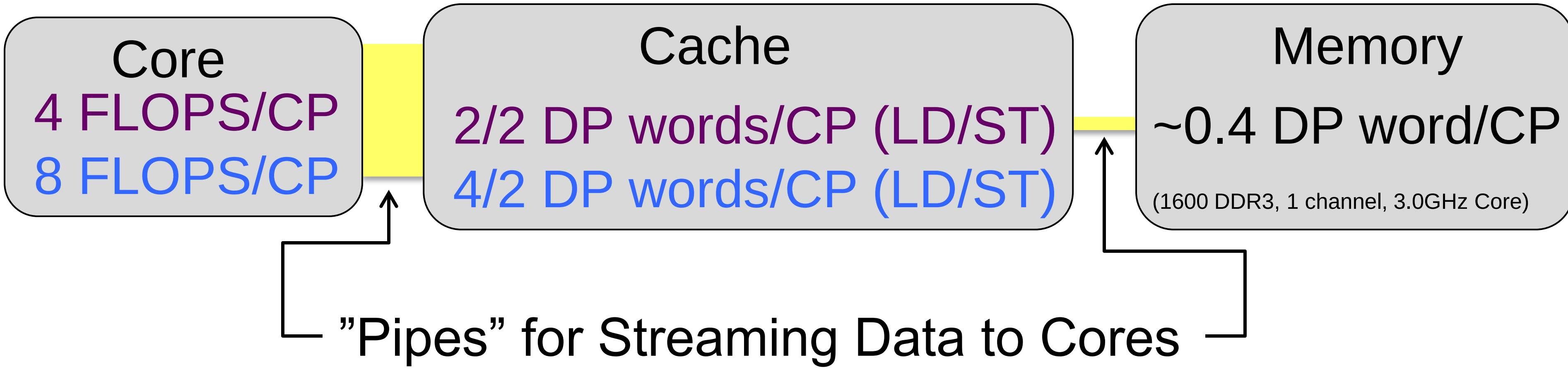
ymm2 ymm0 ymm1



- Optimal vectorization requires concerns beyond SIMD unit!
  - Registers: Alignment of data on 128, 256 bit boundaries
  - Cache: Cache is fast, memory is slow
  - Memory: Sequential access much faster than random/strided

## Cache Utilization

- Loads/stores to L1 cache are fastest
- System memory is very slow in comparison
- If vector units are starved for data, effectiveness is reduced significantly!



## Strided access

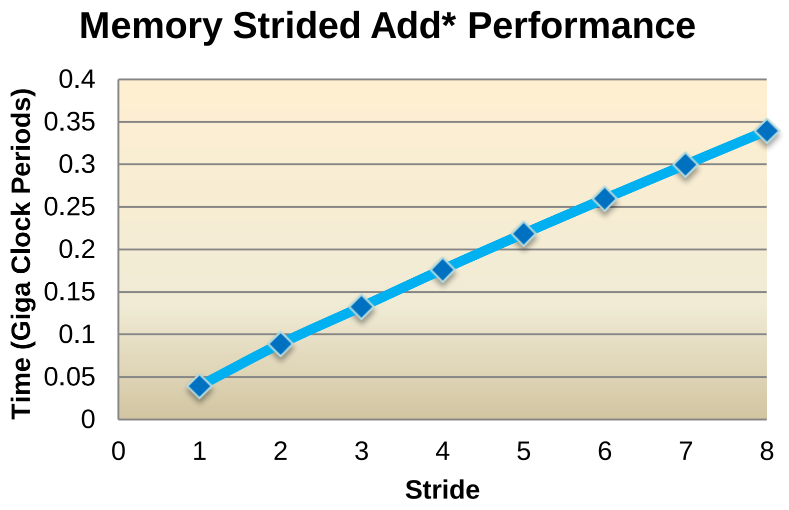
- Fastest usage pattern is “stride 1”: perfectly sequential
- Best performance when CPU can load L1 cache from memory in bulk, sequential manner
- Stride 1 constructs:
  - Iterating Structs of arrays vs arrays of structs
  - Multi dimensional array:
    - Fortran: stride 1 on “inner” dimension
    -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

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

```
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



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

## Cache and Alignment

- Consider our simple unrolling example
  - Unroll N interactions of loop
  - Convert to load/operate/store vector instructions

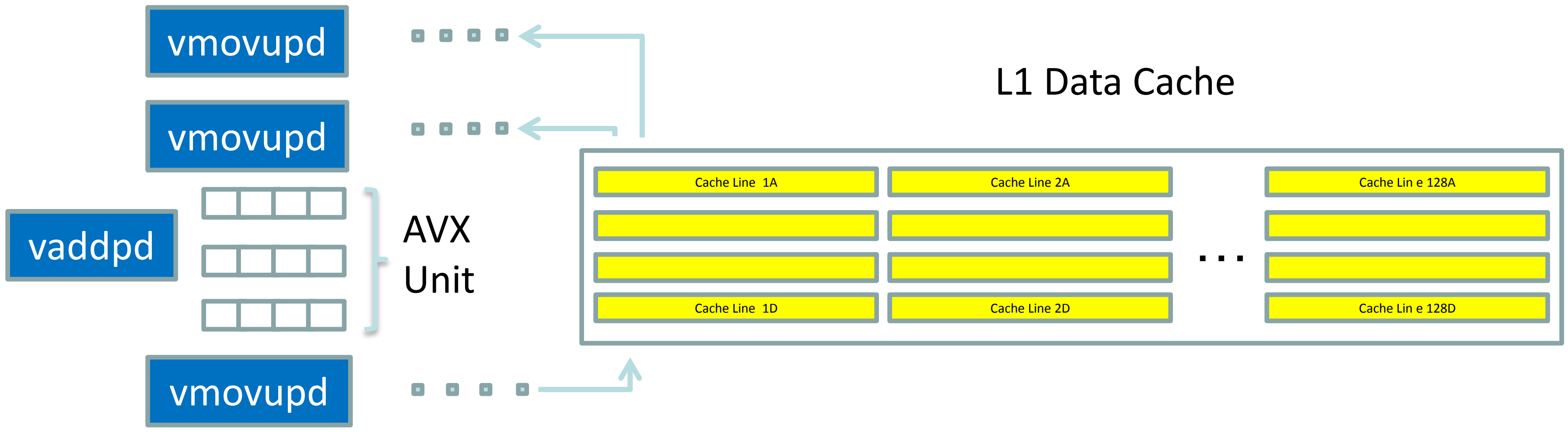
```
for (i=0; i<N;i++) {
      a[i]=b[i]+c[i];
}
```

V = AVX
U = unaligned
P = packed (vector)
D = double

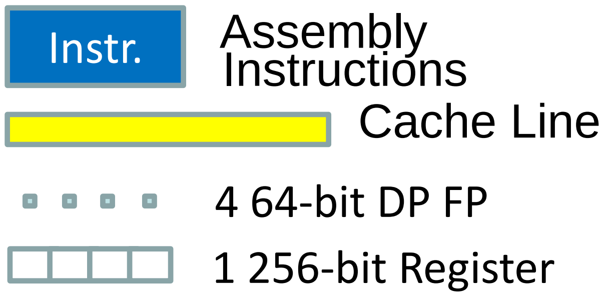
```
vmovapd [next a bytes] xmm0
vmovapd [next c bytes] xmm1
vaddpd xmm0, xmm1
vmovapd xmmm1 [next a bytes]
```

```
vmovupd [next a bytes] ymm0
vmovupd [next c bytes] ymm1
vaddpd ymm0, ymm1
vmovupd ymmm1 [next a bytes]
```

## Cache and Alignment

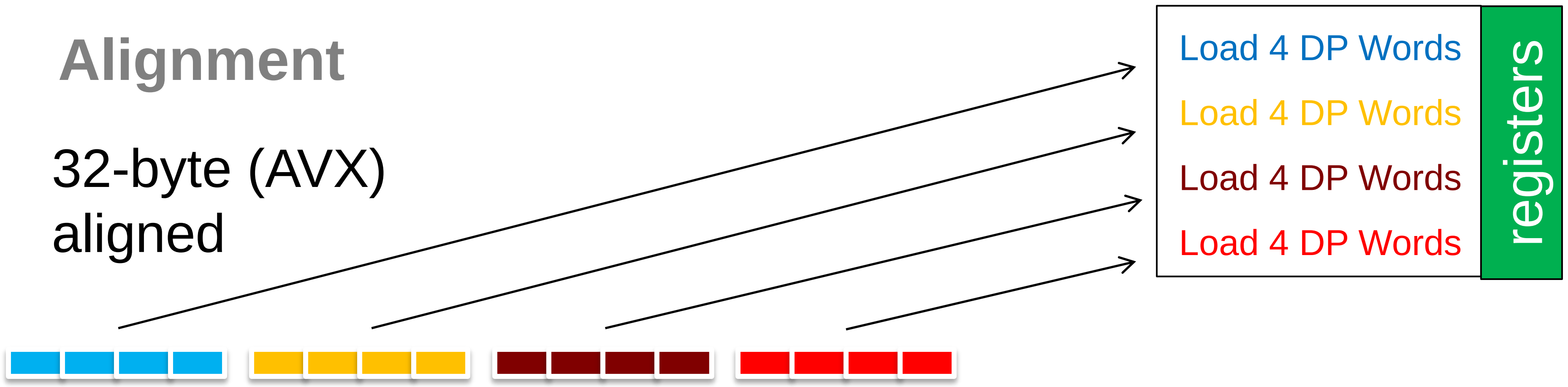


- Vector load instructions move multiple values from cache into registers simultaneously.
- Fastest when entire cache line moved as one unit, i.e. aligned

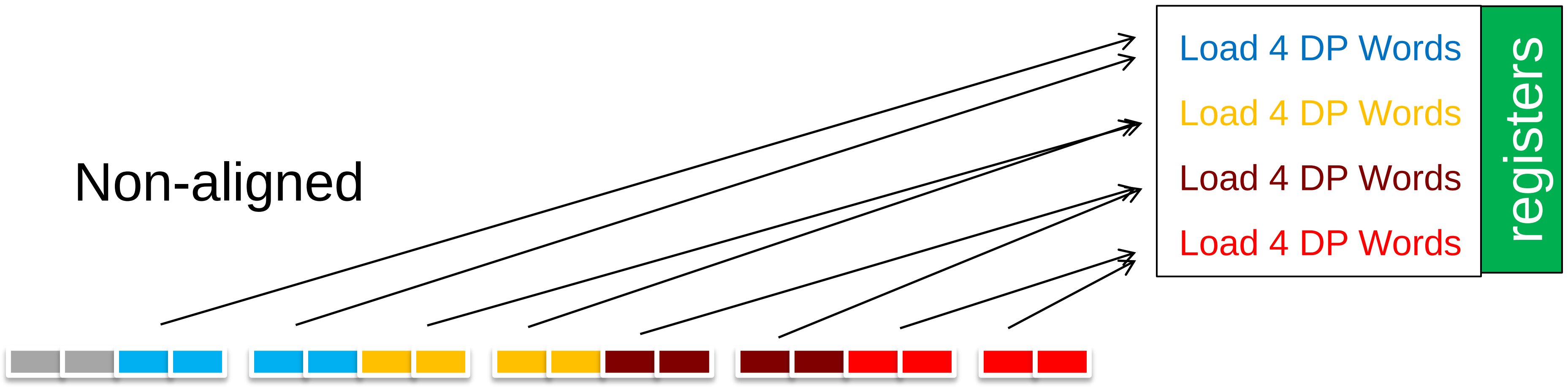


## Alignment

32-byte (AVX) aligned



Non-aligned



## Alignment

- Applies especially to arrays, structs
  - Iterating through multi-dimensional arrays may affect alignment if colums/rows are not a multiple of cache line length.
    - Solution: use padding and adapt your algorithm
- Alignment boundary depends on processor architecture
  - Westmere, Opteron (Lonestar, Ranger): 16 byte
  - Sandy Bridge (Stampede): 32 byte
  - MIC (Stampede): 64 byte
- Compilers are great at automatically handling alignment
  - Harder to determine if they're successful, though
  - May notice alignment issues through decreased performance
  - Glance at assembly, look for unaligned instructions in tight loops (e.g. mov**u**.., vmov**u**..

## Manual Alignment

- For Intel, compiler directives can force compiler to assume correct alignment
  - #pragma vector align asserts that data in the loop is aligned to the appropriate boundary
  - Be careful with SSE – can segfault if you're wrong!
- Can add alignment attributes when declaring variables to guarantee they're aligned
  - Usually the compiler already accounts for this if all references are in the same file, or multiple files are compiled with -ipo
  - _declspec(align(16, 8)) for Intel, __attribute__((aligned(16))) for gcc
- Can force dynamic memory allocation to be aligned
  - With Intel compiler, use _mm_malloc or _mm_free

## 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
  - PerfExpert (available at TACC)
  - Visualize CPU cycle waste spent in data access (L1 cache miss, TLB misses, etc)

## Lab: Using Profilers and analyzing instructions

- Quick introduction to PerfExpert profiling tool to analyze data access patterns
- Look at assembly code to determine if vectorized and/or aligned.

## Conclusion

- OpenMP and Vectorization are synergistic.
  - Need to use all cores, keep vector units on each core busy to achieve peak FLOPs on CPUs or MIC coprocessors.
- 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.