Vectorization

Dan Stanzione, Lars Koesterke, Bill Barth, Kent Milfeld
dan/lars/bbarth/milfeld@tacc.utexas.edu

XSEDE 12
July 16, 2012
# Vectorization

<table>
<thead>
<tr>
<th>Single Instruction Multiple Data (SIMD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The SIMD Hardware, SIMD Processing</td>
</tr>
<tr>
<td>Evolution of SIMD Hardware</td>
</tr>
<tr>
<td>Data Registers</td>
</tr>
<tr>
<td>Instruction Set Overview, AVX</td>
</tr>
<tr>
<td>Vectorization</td>
</tr>
<tr>
<td>Vector Compiler Options/Reports</td>
</tr>
<tr>
<td>Vector Addition– What’s involved ?</td>
</tr>
<tr>
<td>Cross Product</td>
</tr>
<tr>
<td>Intrinsics</td>
</tr>
<tr>
<td>Vector Programming</td>
</tr>
<tr>
<td>Register Fill</td>
</tr>
<tr>
<td>Stream from Cache and Memory</td>
</tr>
<tr>
<td>Strided Access</td>
</tr>
<tr>
<td>Data Alignment</td>
</tr>
<tr>
<td>Inlining</td>
</tr>
<tr>
<td>Aliasing</td>
</tr>
<tr>
<td>Compiler Directives</td>
</tr>
<tr>
<td>Alignment</td>
</tr>
<tr>
<td>Vectorization</td>
</tr>
<tr>
<td>Cache Bypass -- NonTemporal</td>
</tr>
<tr>
<td>Summary</td>
</tr>
</tbody>
</table>
**SIMD Hardware (for Vectorization)**

**SAXPY Operation**

\[
\begin{bmatrix}
Z_1 \\
Z_2 \\
Z_3 \\
\vdots \\
Z_n
\end{bmatrix} = a^* \begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
\vdots \\
x_n
\end{bmatrix} + \begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
\vdots \\
y_n
\end{bmatrix}
\]

- **Registers:**
  - Alignment of data on 64, 128, or 256 bit boundaries might be important

- **Cache:**
  - Access to elements in caches is fast, access from memory is much slower

- **Memory:**
  - Store vector elements sequentially for fastest aggregate retrieval

- **Optimal Vectorization requires concerns beyond the SIMD Unit!**
  - Operations: Requires elemental (independent) operations (SIMD operations)
  - Registers: Alignment of data on 64, 128, or 256 bit boundaries might be important
  - Cache: Access to elements in caches is fast, access from memory is much slower
  - Memory: Store vector elements sequentially for fastest aggregate retrieval
Vectorization, or SIMD* processing, allows a simultaneous, independent instruction on multiple data operands with a single instruction. (Loops over array elements often provide a constant stream of data.)

Note: Streams provide Vectors of length 2-16 for execution in the SIMD unit.

*SIMD = Single Instruction Multiple Data
### Evolution of SIMD Hardware

<table>
<thead>
<tr>
<th>Year</th>
<th>Registers</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~1997</td>
<td>80-bit</td>
<td>MMX</td>
<td>Integer SIMD (in x87) reg</td>
</tr>
<tr>
<td>~1999</td>
<td>128-bit</td>
<td>SSE1</td>
<td>SP FP SIMD (xMM0-8)</td>
</tr>
<tr>
<td>~2001</td>
<td>128-bit</td>
<td>SSE2</td>
<td>DP FP SIMD (xMM0-8)</td>
</tr>
<tr>
<td></td>
<td>128-bit</td>
<td>SSEx</td>
<td>DP FP SIMD</td>
</tr>
<tr>
<td>~2010</td>
<td>256-bit</td>
<td>AVX</td>
<td>DP FP SIMD (yMM0-16)</td>
</tr>
<tr>
<td>~2012</td>
<td>512-bit</td>
<td>ABRni</td>
<td></td>
</tr>
<tr>
<td>~2014</td>
<td>512-1024-bit</td>
<td>(Haswell)</td>
<td></td>
</tr>
</tbody>
</table>

32-bit = Single Precision (SP) Floating Point (FP)<br>64-bit = Double Precision (DP) Floating Point (FP)

For 10 years DP Vectors have had a length of 2 !
In 4 years the DP Vector length will increase by a factor of 8 !
Data Registers: Intel SSE/AVX/MIC

**Floating Point (FP)**
- 64-bit Double Precision (DP)
- 32-bit Single Precision (SP)

**SSE-128 & AVX-128**
- 2x DP FP
- 4x SP FP
- 1x 128-bit dqword
- 2x 64-bit quadword
- 4x 32-bit doubleword
- 8x 16-bit word
- 16x 8-bit (byte)

**AVX-256**
- 4x DP FP
- 8x SP FP
- 8x DP FP
- 16x SP FP

**MIC-512**
- 8x DP FP
- 16x SP FP
Instruction Set Overview (Design)

• Optimal for 64-bit compilation/operation
• Uses Vex prefix (v)
  • Extendable to 512-/1024-bit SIMD
  • Can reference 3 (and 4) registers (non-destructive destination encoding)
  • New Instructions, broadcast to registers, mask, permute, etc.
• Alignment is important for optimal performance
• FMA (Fused Multiply Add) available in 2nd generation (hw/sw): Haswell/AVX2
### Instruction Set Overview (AVX)

<table>
<thead>
<tr>
<th></th>
<th>AVX 128-bit VEX Prefix</th>
<th>AVX 256-bit Vex Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Legacy SIMD</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Vector Data Movement</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Vector FP</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Vector Int</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Int</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>New Functionality</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Permute (v)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mask (v)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Broadcast (v)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Insert/Extract/Zero</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

But, let’s step back and look at the bigger picture!
Vectorization

- **REGISTER SIZE DETERMINES** the number of Simultaneous Operands in a SIMD operation.
- **Compilers are good at vectorizing** inner loops
  --gathering Multiple Data & executing a Single Instruction
- Each iteration must be independent
- **Inlined functions and intrinsic SVML Library functions** can provide vectorization opportunities.
Vector Compiler Options

• Compiler will look for vectorization opportunities at optimization
  – O2 level.

• Use architecture option:
  –x<simd_instr_set> to ensure latest vectorization
    hardware/instructions set is used.

• Confirm with vector report:
  – vec-report=<n>, n=“verboseness”

• To get assembly code, myprog.s:
  – S

• Rough Vectorization estimate: run w./w.o. vectorization
  -no-vec
Vectorization Report

• Intel Vector Reporting is now OFF by default.
• USE vector reporting to report on loops NOT vectorized, reports 4,5.

% ifort -xHOST -vec-report=4 prog.f90 –c

prog.f90(31): (col. 11) remark:
   loop was not vectorized: existence of vector dependence.

... -vec-report=5

prog.f90(31): (col. 4) ...assumed ANTI dependence between z line 31 and z line 31.
prog.f90(31): (col. 4) ...assumed FLOW dependence between z line 31 and z line 31.
• Each iteration can be executed independently: loop will vectorize. (Use –vec-report=6 for details.)

• Compiler is aware of data size: but may be unaware of data alignment and cache.
Vector Add

- **Vectorization**
  - **Load** 2/4 DP PFs (SSE/AVX)
  - **Add** 2/4 DP PFs (SSE/AVX)
  - **Store** 2/4 DP PFs (SSE/AVX)

**C**

```c
double a[N], b[N], c[N];
...
for(j=0;j<N;j++) { 
    a[j]=b[j]+c[j];
}
```

**F90**

```fortran
real*8 :: a(N), b(N), c(N)
...
do i = 1,N;
    a(j)=b(j)+c(j)
enddo
```

**Instruction Register**

- **movupd** XMM/YMM
- **addpd** XMM/YMM
- **movupd** XMM/YMM

p (packed) implies a vector operation.
u (unaligned), d(double)
Vector Add -- AVX

- Only vector code will load multiple sets of data into registers simultaneously.
- Non-aligned sets do consume more Clock Periods (CPs).

VMOVUPD
VMOVUPD
VADDPD
VMOVUPD

L1 Data Cache

AVX Unit

Cache Line 1A
Cache Line 2A
Cache Line 128A
Cache Line 1D
Cache Line 2D
Cache Line 128D

Instr.
Assembly Instructions

Cache Line
4 64-bit DP FP
1 256-bit Register
Vectorization (on KNC)

```c
void mult(double *a, double *b, double *c, int n) {
    for (int i = 0; i < n; i++)
        a[i] = b[i] + c[i];
}

subroutine mult(a, b, c, n); real*8 :: a(n), b(n), c(n)
    do i = 1, n;
        a(i) = b(i) + c(i);
    enddo
end subroutine
```

Scalar Instructions
8 instructions, 8 element pairs

Vector Instruction
1 instruction, 8 element pairs
Beyond Arithmetic/Load/Store

• Predicates (more) for comparison
• Shuffle
• Broadcast
• Masked-move for conditional load/store
• Insert/extract
• Permute

• Future: FMA
Cross Product: \( C = A \times B \)

Beyond normal vectorization—Shuffling Elements (vector instructions)

- MOVAPS XMM0, [A]
- MOVAPS XMM2, [B]
- MOVAPS XMM1, XMM0
- SHUFPS XMM0, XMM0, 0xc9
- SHUFPS XMM2, XMM2, 0xd2
- MULPS XMM0, XMM2
- SHUFPS XMM1, XMM1, 0xd2
- SHUFPS XMM2, XMM2, 0xd2
- MULPS XMM1, XMM2
- SUBPS XMM0, XMM1
- MOVUPS [C], XMM0

\[
\begin{align*}
C_z &= Ax \times By - Ay \times Bx \\
C_y &= Az \times Bx - Ax \times Bz \\
C_x &= Ay \times Bz - Az \times By
\end{align*}
\]
Intel Intrinsics

- Available with AVX
- No need to write in-line assembly
- Not available for Fortran
Assembly vs Intrinsics -- functions

- **Assembly**
  - Actually, more complex
  ```c
  void inadd(double *a, double *b, double *c) {
    __asm
    { mov rax, a
      mov rbx, b
      mov rcx, c

      movupd xmm0, [rax]
      movupd xmm1, [rbx]
      addpd xmm0, xmm1

      movupd [rcx], xmm0
    }
  }
  ```

- **Intrinsics**
  - Use C bindings to call from Fortran
  ```c
  void inadd(double *a, double *b, double *c) {
    __m128d m0,m1;

    m0 = _mm_load_pd(a);
    m1 = _mm_load_pd(b);
    m0 = _mm_add_pd(m0,m1);
    _mm_store_pd(c,m0);
  }
  ```

- \( a(i) = b(i) + c(i) \)
Vector Programming

- **Optimal Performance Considerations**
  - REGISTER Fill (Vectorize)
    - Vector Loops
    - Dependencies
  - STREAM from Cache and Memory
    - Memory and Cache Bandwidths
    - Strided Access
  - Use Libraries (not covered)

DP= Double Precision Word, 8-Byte storage
Vector Loops

• Write loops with independent iterations
• Loops will “Vectorize” →
  – Will be unrolled or “pipelined” to allow simultaneous transfers of multiple data elements from L1 Cache to vector register. (2 DP words for Westmere and 4 DP words for Sandy Bridge)

• Loops need to be/have:
  – Countable; single entry and single exit; straight-line code (no switches; but if’s may be masked), “inner loop”; no function calls (e.g. io).
Vector Loops

• There are many functions for which the compiler has a vectorizable version:
  
  \begin{align*}
  \sin, \acos, \cos, \acos, \text{etc.}; \exp, \pow, \log, \text{etc.}; \\
  \erf, \erfc; \ceil, \floor, \fmin, \fmax, \text{etc.}
  \end{align*}

• Use the --opt-report-phase ipo_inl option for an inlining report.
Unrolling allows compiler to re-construct loop for Vector operations:

```
do i=1,N, 4
  a(i ) = b( i )*c( i )
  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)
end do
```

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

Unrolling allows

```
do i=1,N
  a( i ) = b( i )*c( i )
end do
```

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

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

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

Dependency Analysis—

RAW: Read after Write not vectorizable
  - variable is written first, and read subsequently.

WAR: Write after Read vectorizable
  - Variable is read first, and written subsequently.

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

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

Dependency Analysis—

RAR: Read after Read vectorizable
  • Not really a dependency, since there is no write.

WAW: Write after Write NOT vectorizable
  • Same variable is written to in more than 1 iteration.

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

for( i=0; i<N; i++)
a[i%M] = b[i] + c[i];
Loop Dependencies

- RAW example:

```
for( i=2; i<=N; i++) a[i] = a[i-1] + b[i];
do i=2, N ;   a(i) = a(i-1) + b(i); enddo
```

Check for vectorizability: Do vector loads, operations, and stores across multiple iterations produce the same results as executing each iteration sequentially?

<table>
<thead>
<tr>
<th>RAW</th>
<th>iteration 1</th>
<th>Iteration 2</th>
<th>Iteration 3</th>
<th>Iteration 4</th>
<th>...N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A(2) = A(1)</td>
<td>A(3) = A(2)</td>
<td>A(4) = A(3)</td>
<td>A(5) = A(4)</td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>B(2) *</td>
<td>B(3) *</td>
<td>B(4) *</td>
<td>B(5) *</td>
<td>:</td>
</tr>
</tbody>
</table>

NOT VECTORIZABLE

Load A’s, Load B’s, **Vector Operate**, Store into A’s -- DOES NOT produce the same results as 4 sequential iterations.
### Loop Dependencies

**WAR example:**

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

Check for vectorizability: Do vector loads, operations, and stores across multiple iterations produce the same results as executing each iteration sequentially?

<table>
<thead>
<tr>
<th>WAR</th>
<th>iteration 1</th>
<th>A(1)</th>
<th>=</th>
<th>A(2)</th>
<th>*</th>
<th>B(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 2</td>
<td>A(2)</td>
<td>=</td>
<td>A(3)</td>
<td>*</td>
<td>*</td>
<td>B(2)</td>
</tr>
<tr>
<td>Iteration 3</td>
<td>A(3)</td>
<td>=</td>
<td>A(4)</td>
<td>*</td>
<td>*</td>
<td>B(3)</td>
</tr>
<tr>
<td>Iteration 4</td>
<td>A(4)</td>
<td>=</td>
<td>A(5)</td>
<td>*</td>
<td>*</td>
<td>B(4)</td>
</tr>
<tr>
<td>...N</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

**VECTORIZABLE**

Load A’s, Load B’s, **Vector Operate**, Store into A’s -- DOES produce the same results as 4 sequential iterations.
Many applications access elements of Arrays.

These elements may reside in the caches (L1 | L2 | L3) or memory when the load/store instructions are executed.

Performance is dependent on data location.
Memory and Cache Bandwidths

- If a stream of data (series of accesses) is from memory (BW limited), vectorization has little or no effect on performance – the vector units are starved.
- If the majority of accesses is from memory the application is called memory-bandwidth limited.

**Core**
- 4 FLOPS/CP
- 8 FLOPS/CP

**Cache**
- 2/2 DP words/CP (LD/ST)
- 4/2 DP words/CP (LD/ST)

**Memory**
- ~0.4 DP word/CP
  - (1600 DDR3, 1 channel, 3.0GHz Core)

"Pipes" for Streaming Data to Cores
Strided Access

• Striding decreases performance.
  The effect is greatest for accesses from memory, and can be minimal for the closest (L1) cache.

For L2/L3 and memory, moving unused data degrades effective bandwidth (for useful data).

For L1 Cache, don’t split vectors across cache lines.
Strided Access

• 1-D Array
  – Stride 1 access is best!

• Multi-D Array
  Fortran: Stride 1 access on “inner” dimension is best.
  C/C++: Stride 1 access on “outer” dimension is best.

F90

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

C

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

- Striding through Memory cache reduces effective bandwidth to vector units—~(1-stride/8) for DP arrays. (upper right)
- Effective L2 BW degradation is less (bottom right).

Vec. Programming

Strided Access

Strided Access

Memory Strided Add* Performance

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

Lower is better.

L2 Strided Add* Performance

```
*do i = 1, 2048*istride, istride
    a(i) = b(i) + c(i) * sfactor
enddo
```

Lower is better.
Alignment

• Alignment of data and data structures can affect performance. For AVX, alignment to 32-byte boundaries (4 Double Precision words) allows a single reference to a cache line for moving 4DP words into the registers (SIMD support). For MIC, alignment is 64 bytes.

• Compilers are great at detecting alignment and peeling off a few iterations before working on a sustained alignment within a loop body.

(Aligned data can use the more efficient movdq a instruction, rather than the less efficient movdq u instruction.)
Alignment

32-byte Aligned

Cache Line 0
Cache Line 1
Cache Line 2
Cache Line 3

Single Cache access for 4 DP Words

Non-Aligned

Cache Line 0
Cache Line 1
Cache Line 2
Cache Line 3
Cache Line 4

Across Cache Line access for 4 DP Words
In multi-dimensional arrays, resizing (padding) lower dimensions for alignment can be beneficial.

E.g. \( a(15,16,16) \rightarrow a(16,16,16) \)

The \( i \) dimension will determine the alignment in subsequent accesses for \( j > 1 \).

* Be careful about cache thrashing.
Inlining

- Functions within a loop prevent vectorization.
  - Inlining can often overcome this problem.

\[\text{for}(i=0; \ i<nx; \ i++)\{\]
\[\quad x = x0 + i*h;\]
\[\quad \text{sum} = \text{sum} + \text{do}_r2(x, y, xp, yp);\]
\[\}\]

\[\text{double do}_r2(\text{double } x, \text{ double } y, \text{ double } xp, \text{ double } yp)\{\]
\[\quad \text{double } r2;\]
\[\quad r2 = (x-xp)*(x-xp) + (y-yp)*(y-yp);\]
\[\quad \text{return } r2;\]
\[\}\]

- Since the call and function are in different files, inlining and vectorization don’t occur. Use interprocedural optimization option (-ipo) to inline & vectorize.
- If call and function are within the same unit (file), inlining and vectorization are performed at -O2 optimization and higher.
### Inlining

#### file main.c

```c
... 
for(i=0; i<nx; i++){
    x = x0 + i*h;
    sum = sum + do_r2(x, y, xp, yp);
}
... 
```

#### file funs.c

```c
double do_r2(double x, double y, double xp, double yp){
    double r2;
    r2 = (x-xp)*(x-xp) + (y-yp)*(y-yp);
    return r2;
}
```

### Inlining - Vectorization - Time (ms)

<table>
<thead>
<tr>
<th>Inlining</th>
<th>Vectorization</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>not inlined</td>
<td>not vectorized</td>
<td>1.55</td>
</tr>
<tr>
<td>inlined</td>
<td>not vectorized</td>
<td>0.44</td>
</tr>
<tr>
<td>inlined</td>
<td>vectorized</td>
<td>0.056</td>
</tr>
</tbody>
</table>
Aliasing

• In general, two different pointers can have the same target (can point to the same memory location). Hence, loops in routines that use pointer references may not vectorize.

• In routines that pass pointers, make sure the compiler can be certain that references don’t overlap.
void my_cp(int nx, double *a, double *b){
    for(int i=0; i<nx; i++) a[i]=b[i];
}

This will generally vectorize, because the compiler can check for overlap at runtime.

void my_combine(int *ioff, int nx,
        double * a, double * b, double * c){
    for(int i=0; i<nx; i++) a[i]=b[i]+c[i+*ioff];
}

The compiler cannot determine that there is no aliasing—so it won’t vectorize the loop.
Aliasing

• In general, two different pointers can have the same target (can point to the same memory location). Hence, loops in routines that use pointer references may not vectorize:

```c
void my_combine(int *ioff, int nx,
    double * a, double * b, double * c){
    for(int i=0; i<nx; i++) { a[i]=b[i]+c[i+*ioff]; }
}
```

if a and c have been “equivalenced”, there may be dependencies. (Strict aliasing means that two objects of different types cannot refer to the same location in memory.)
void my_combine(int * ioff, int nx, 
    double * a, double * b, double * c){
    for(int i=0; i<nx; i++){ a[i]=b[i]+c[i+*ioff]; }
}

$ icpc -xhost -c -O2 func.cpp -vec-report=2
func.cpp(3): (col. 4) remark: loop was not vectorized: \ 
existence of vector dependence.
void my_combine(int * ioff, int nx,
    double * a, double * b, double * c){
    for(int i=0; i<nx; i++) { a[i]=b[i]+c[i+ioff]; }
}

Vectorization allowed with ansi aliasing rules:

$ icpc -xhost -c -O2 func.cpp -ansi-alias -vec-report=2
func.cpp(3): (col. 4) remark: LOOP WAS VECTORIZED.
func.cpp(3): (col. 4) remark: loop skipped: multiversioned.

But option applies to whole unit; be careful, use with caution:
void my_combine(int * restrict ioff, int nx, 
    double * restrict a, double * restrict b, double * restrict c){
    for(int i=0; i<nx; i++){ a[i]=b[i]+c[i+*ioff]; }
}

Vectorization allowed with “restrict” declaration/option – specific to routine:

$ icpc -xhost -c -O2 func.cpp -restrict -vec-report=2 
func.cpp(3): (col. 4) remark: LOOP WAS VECTORIZED.

Can use __restrict alone with Intel compiler.
Compiler Directives: Hints and Coercion

alloc_section
distribute_point
inline, noinline, and forceinline
ivdep
loop_count
memref_control
novector
optimize
optimization_level
prefetch/noprefetch
simd
unroll/nounroll
unroll_and_jam/nounroll_and_jam
vector
• Unaligned accesses are slower.
  – Non-sequential across “bus”.
  – Cross cache line boundary.
• \#pragma vector aligned or !DEC$ vector aligned

Alignment can be forced
C: memalign(XXbyte,size)
F90: Use compiler option
  -align arrayXXbyte

for(i=0; i<loops; i++)
  #pragma vector aligned
  for(j=0; j<N-i; j++) a[j]=b[j]+c[j];

0.75 CP/Op w.o. pragma*
0.50 CP/OP with pragma*

*When executed without –xSSE4.1 on Westmere.
Vectorization, Hints

- An unknown index offset forces the compiler to assume a dependency – loop not vectorized.
- Use IVDEP to hint (strongly suggest) vectorization.

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

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

-vec-report2 info

• remark: loop was not vectorized: existence of vector dependence.

• remark: SIMD LOOP WAS VECTORIZED.

```c
#pragma ivdep
#pragma vector always
for (int i=0 ; i<end ; i++)
    a[i][0] = (b[i][0] - b[i+1][0]);
    a[i][1] = (b[i][1] - b[i+1][1]);
```

```c
#pragma simd
for (int i=0 ; i<end ; i++)
    a[i][0] = (b[i][0] - b[i+1][0]);
    a[i][1] = (b[i][1] - b[i+1][1]);
```
Non-temporal (Streaming) Stores

Directive Example

```c
!DEC$ VECTOR NONTEMPORAL
do i = 1,N
    c(i) = a(i)
end do
```

```c
#pragma vector nontemporal
for(i=0;i<N;i++){
    c[i] = a[i];
}
```

Non-temporal

- Load a(i)
- Updated c(i) collected in write-combining buffer—and sent to memory when full (64B).
- Memory Traffic = 1 read + 1 write
Vectorization Summary

• AVX is here.

• Register widths are leaping forward→ be aware of SIMD power.

• Vector Instruction Sets are getting larger→ more opportunity for handling the data in SIMD.

• Just because register sizes double (quadruple) that doesn’t mean your performance will increase by the same amount.
Questions Welcome