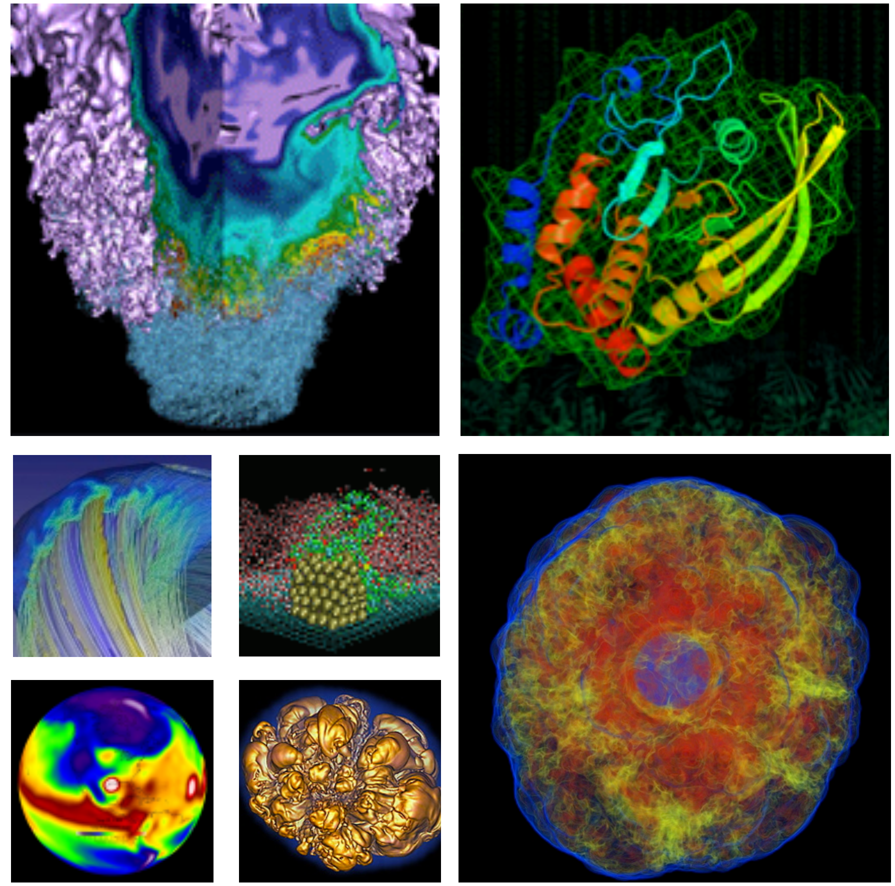
OpenMP and Vectorization Training Introduction



Jack Deslippe, Helen He, Harvey Wasserman, Woo-Sun Yang

NERSC App Readiness Team

Reason for These Tutorials

- **Preparation for NERSC's Cori System (2016)**
- **Energy-efficient *manycore* processor**
  – Multicore (heavyweight): slow evolution from 2–12 cores per chip; core performance matters more than core count
  – Manycore (lightweight): jump to 30–60+ cores per chip; core count essentially matters more than individual core performance
  – Manycore: Relatively simplified, lower-frequency computational cores; less instruction-level parallelism (ILP); engineered for lower power consumption and heat dissipation

Manycore Programming Challenges

- **More difficult to keep manycore processors busy with useful work**
  – programs must overcome comparatively larger memory latency

- **Programs must expose increased levels of parallelism and express that parallelism differently**

Optimization Choices for Cori

Scalability Dimension	Critical Dependence?
MPI / Interconnect Scaling	Current: required Cori: depends
Thread Scaling in Shared Address Space (e.g., OpenMP)	Current: nice, may help Cori: Essential
Vectorization	Current: nice, may help Cori: Essential
Cache Utilization	Current: nice, helps Cori: Essential
On-Package Memory	Current: N/A Cori: Very helpful

Based on "Intel® Xeon Phi™ Coprocessor NWP Application Experiences Programming," Experiences Programming Weather, Climate, and Earth-System Models on Multi-Core Platforms, Mike Greenfield (Intel)

Today's Tutorials

- **Part of NERSC's effort to help users transition to Cori**
  - A way to get users to start thinking about these issues in a productive way

- **More vendor and NERSC training coming soon**

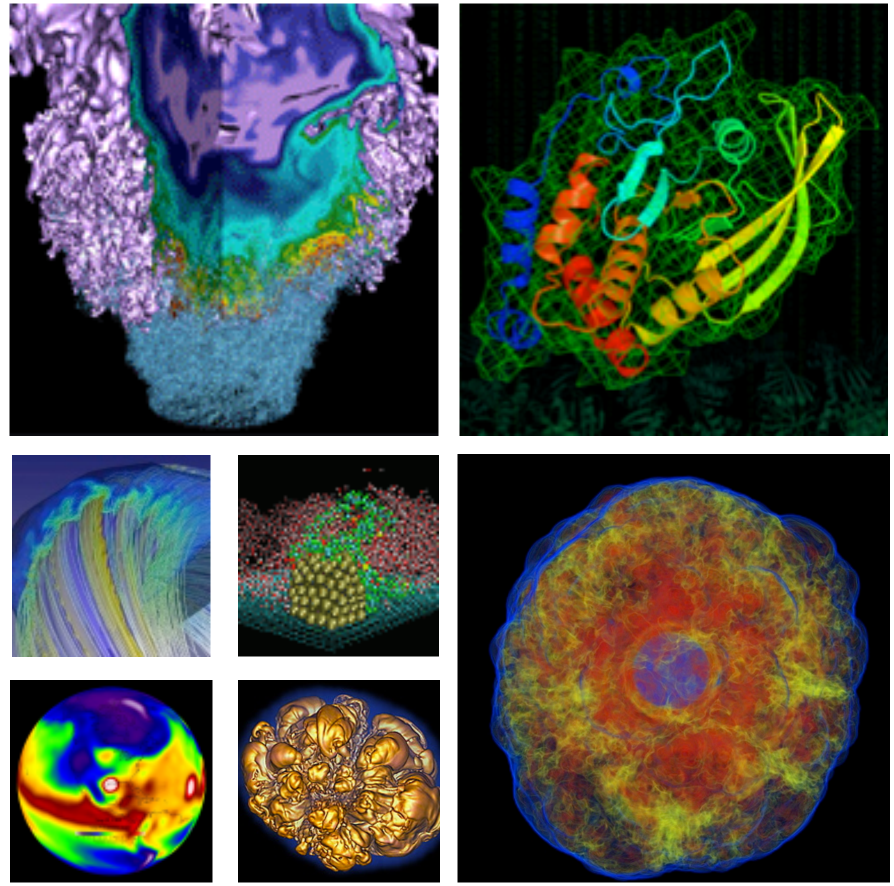
- **More case studies here:**

http://www.nersc.gov/users/computational-systems/cori/app-case-studies/

https://www.nersc.gov/users/computational-systems/edison/programming/vectorization/

https://www.nersc.gov/users/software/debugging-and-profiling/vtune/

Vectorization



NERSC 40 YEARS at the FOREFRONT 1974-2014

Harvey Wasserman

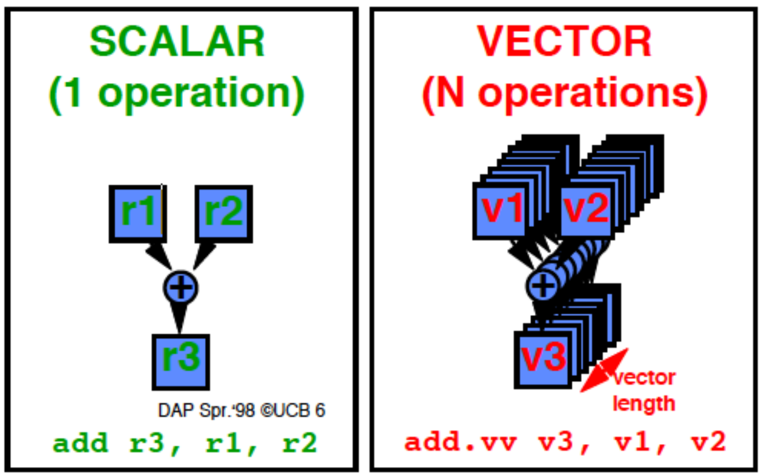
U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

What's All This About Vectorization?

- **Vectorization is an on-node, in-core way of exploiting data level parallelism in programs by applying the same operation to multiple data items in parallel.**

```
DO I= 1, N
     Z(I) = X(I) + Y(I)
ENDDO
```



- Requires *transforming* a program so that a single instruction can launch many operations on different data
- Applies most commonly to array operations in loops

What is Required for Vectorization?

- **Code transformation**

```
DO I = 1, N
   Z(I) = X(I) + Y(I)
ENDDO
```

Compiler →

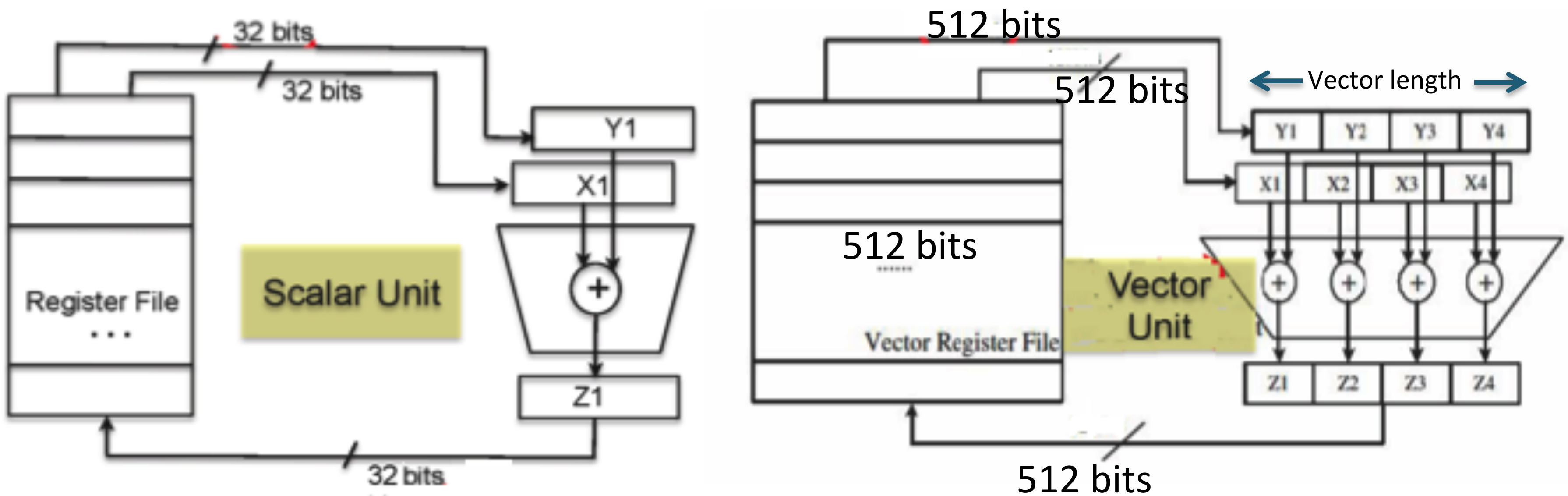
```
DO I = 1, N, 4
  Z(I)    =   X(I) +   Y(I)
  Z(I+1) = X(I+1) + Y(I+1)
  Z(I+2) = X(I+2) + Y(I+2)
  Z(I+3) = X(I+3) + Y(I+3)
ENDDO
```

- **Compiler generates vector instructions:**

```
VLOAD  X(I), X(I+1), X(I+2), X(I+3)
VLOAD  Y(I), Y(I+1), Y(I+2), Y(I+3)
VADD   Z(I, ..., I+3)   X+Y(I, ..., I+3)
VSTORE Z(I), Z(I+1), Z(I+2), Z(I+3)
```

What is Required for Vectorization?

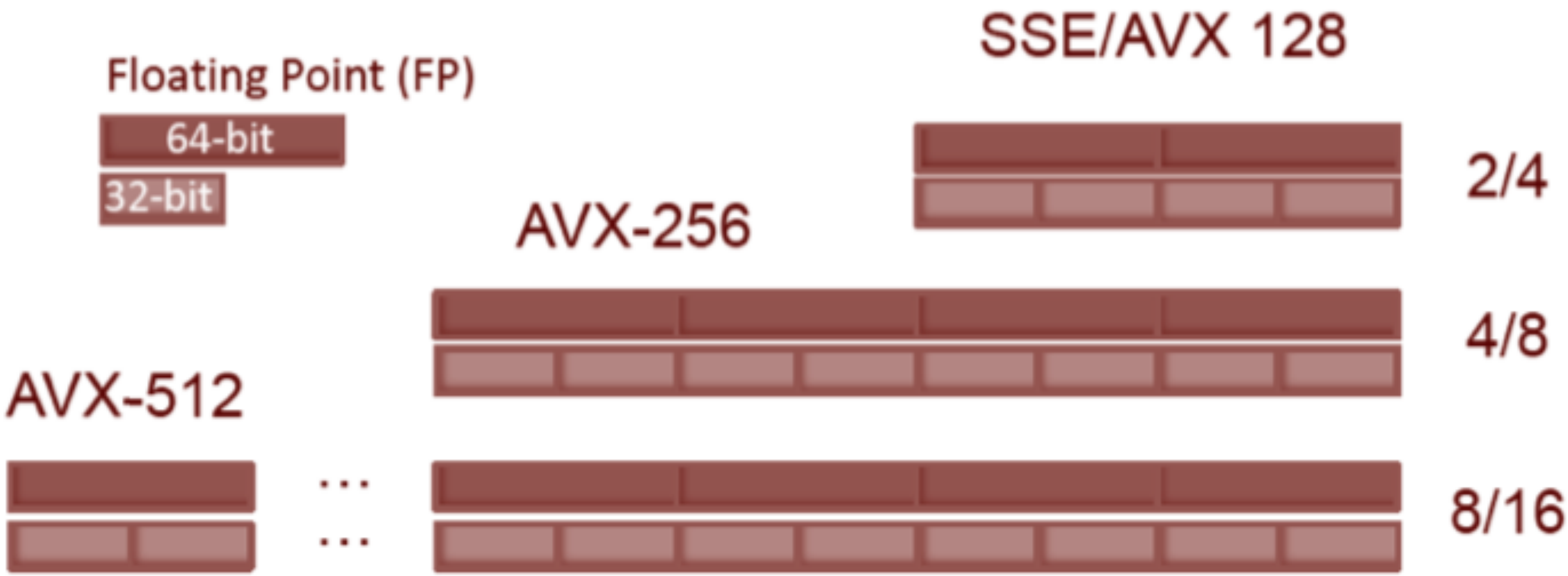
- **Vector Hardware: vector registers and vector functional units**



SIMD and Vector Hardware

- **Single-Instruction, Multiple Data: a single instruction applies the same operation, executed by multiple processing elements, to multiple data concurrently**
- **Intel KNL incorporates Intel Advanced Vector Extensions 512 (Intel AVX-512) instructions.**
  - includes 32 vector registers, each 512 bits wide, eight dedicated mask registers, 512-bit data operations
  - will also be supported by some future Xeon processors
- **Evolution of instruction set architecture from**
  - shorter to longer vector lengths
  - more general/flexible vector instructions

Evolution of Vector Hardware



- **Translates to (peak) speed: cores per processor X vector length X CPU Speed X 2 arith. ops per vector**

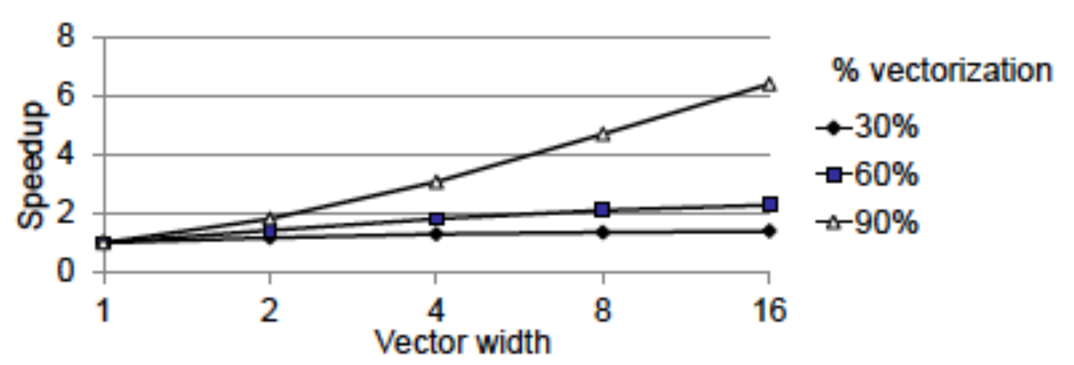
Why Vectorization?

- **Compiler ensures no dependencies => long pipeline, high rate**
- **Vector instructions access memory with known pattern; amortize memory latency**
- **Fewer instructions, wider execution units, lots of work per instruction – more energy efficient (~10X fewer instructions)**
- **Simpler circuit design – high performance without energy and design complexity costs of highly out-of-order processing;**
  – Important, since CPU speeds have reached a plateau due to power limitations
- **Scalable: higher performance as more HW resources available**
- Vector performance contribution is likely to increase in the future as vector lengths increase – hardware vector length may double every four years.

Vector Performance

- **Application performance will be very sensitive to:**
  1. **Hardware vector length**
  2. **Data must be in cache; must be properly aligned (see Jack/Woo-Sun)**
  3. **Percentage of code that can use vectors (time in vectorizable loops)**
  4. **Length of loops that vectorize**
  5. **Computational intensity and types of arithmetic operations**
  6. **Memory access pattern within loops; constant, unit stride is best**
  7. **Conditional characteristics**

Near theoretical peak speed when all of the above are met!



Amdahl's Law: overall improvement by using a faster mode of computing is limited by the fraction of time that can be used by the faster mode.

From Aaron Birkland, Cornell CAC

How to Vectorize Your Code?

- **Auto-Vectorization analysis by the compiler**
- **Auto-Vectorization analysis by the compiler enhanced with directives – code annotations that suggest what can be vectorized**
- **Code explicitly for vectorization using OpenMP 4.0 SIMD pragmas or SIMD intrinsics (not portable)**
- **Use assembly language**
- **Use vendor-supplied optimized libraries**

Compiler Vectorization of Loops

- **Enabled with default optimization levels for Intel and Cray compilers on Edison/Hopper (and Intel on Babbage)**
- **Compiler informs you of failure.**
  Use -vec-report=# **flag**
  Will change soon to
  -qopt-report -qopt-report-phase=vec

- **gcc/gfortran: use**
  –ftree-vectorize **flag, combined with optimization > –O2**
  –ftree-vectorizer-verbose **for a vectorization report**
- **Cray compiler: automatic**

How Do I Know if My Code Vectorized?

Line numbers

```
% ftn -o v.out -vec-report=3 yax.f
yax.f(12): (col. 10) remark: LOOP WAS VECTORIZED
yax.f(13): (col. 22) remark: loop was not vectorized: not inner loop
yax.f(18): (col. 10) remark: LOOP WAS VECTORIZED
yax.f(26): (col. 13) remark: LOOP WAS VECTORIZED
yax.f(25): (col. 10) remark: loop was not vectorized: not inner loop
yax.f(24): (col. 7) remark: loop was not vectorized: not inner loop
```

```
edison11 h/hjw> cc -vec-report=6 -c mm.c
mm.c(7): (col. 2) remark: loop was not vectorized: loop was transformed to memset or memcpy
mm.c(10): (col. 2) remark: vectorization support: reference c has aligned access
mm.c(10): (col. 2) remark: vectorization support: reference c has aligned access
mm.c(10): (col. 2) remark: vectorization support: reference a has aligned access
mm.c(6): (col. 2) remark: vectorization support: unroll factor set to 4
mm.c(6): (col. 2) remark: PERMUTED LOOP WAS VECTORIZED
mm.c(9): (col. 2) remark: loop was not vectorized: not inner loop
mm.c(7): (col. 2) remark: loop was not vectorized: not inner loop
```

Useful Tools from the Cray Compiler

- **Cray "loopmark" option: -rm**

```
57.  + 1------------<        do it=1,itmax
58.  + 1 br4--------<           do j=1,n
59.  + 1 br4 b------<              do k=1,n
60.    1 br4 b Vr2--<                 do i=1,nr
61.    1 br4 b Vr2                       c(i,j) = c(i,j) + a(i,k) * b(k,j)
62.    1 br4 b Vr2-->                 end do
63.    1 br4 b------>              end do
64.    1 br4-------->           end do
65.    1------------>        end do
```

```
b - blocked
r - unrolled
V - Vectorized
```

```
ftn-6254 ftn: VECTOR File = matmat.F, Line = 57
  A loop starting at line 57 was not vectorized because a recurrence was found on "c" at line 61
ftn-6294 ftn: VECTOR File = matmat.F, Line = 58
  A loop starting at line 58 was not vectorized because a better candidate was found at line 60.
ftn-6049 ftn: SCALAR File = matmat.F, Line = 58
  A loop starting at line 58 was blocked with block size 8.
ftn-6005 ftn: SCALAR File = matmat.F, Line = 58
  A loop starting at line 58 was unrolled 4 times.
```

Vector Analysis Problem

- Algorithms and/or programming styles can obscure or inhibit vectorization. Requirements:
- Loop trip count known at entry to the loop at runtime
- Single entry and single exit
- Innermost loop of a nest**
- No function calls or I/O
- No data dependencies in the loop
- Uniform control flow (although conditional computation can be implemented using “masked” assignment)

Will not vectorize:

```
DO I = 1, N
    IF (A(I) < X) CYCLE
ENDDO
```

Compiler Vectorization of Loops

```
      DO jkm = 1, ndiag
         jk = jkm
         do mi = 1, mmi-1
            if (jk .le. mdiag(mi)) go to 100
            jk = jk - mdiag(mi)
         end do
         mi = mmi
100      continue
```

```
edison10 CODES/SWEEP> ftn -c -vec-report=2 sweep.f
edison10 CODES/SWEEP> cat list
edison10 CODES/SWEEP> ftn -c -vec-report=6 sweep.f
sweep.f(162): (col. 10) remark: PARTIAL LOOP WAS VECTORIZED
sweep.f(190): (col. 19) remark: LOOP WAS VECTORIZED
sweep.f(189): (col. 19) remark: loop was not vectorized: not inner loop
sweep.f(273): (col. 18) remark: REMAINDER LOOP WAS VECTORIZED
sweep.f(332): (col. 13) remark: loop was not vectorized: vectorization possible but seems inefficient
sweep.f(359): (col. 13) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate
```

Data Dependencies

- **Vectorization results in changes in the order of operations within a loop.**
- **Yet, program semantics must be maintained. Each result must be independent of previous results.**
- **Compiler will err on the conservative side to produce correct code; you need to go back and verify that there really is a dependency.**
- **In C, pointers can hide data dependencies**
  - Memory regions they point to may overlap

Code Without Dependences

- **Code transformation**

```
DO I = 1, N
   Z(I) = X(I) + Y(I)
ENDDO
```

Compiler →

```
DO I = 1, N, 4
  Z(I)   =   X(I) +   Y(I)
  Z(I+1) = X(I+1) + Y(I+1)
  Z(I+2) = X(I+2) + Y(I+2)
  Z(I+3) = X(I+3) + Y(I+3)
ENDDO
```

```
VLOAD  X(I), X(I+1), X(I+2), X(I+3)
VLOAD  Y(I), Y(I+1), Y(I+2), Y(I+3)
VADD   Z(I, ..., I+3)   X+Y(I, ..., I+3)
VSTORE Z(I), Z(I+1), Z(I+2), Z(I+3)
```

Code Without Dependences

```
DO I = 1, 4
  A(I) = B(I) + C(I)
  D(I) = E(I) + F(I)
END DO
```

Non-Vector

```
A(1) = B(1) + C(1)
 D(1) = E(1) + F(1)
A(2) = B(2) + C(2)
 D(2) = E(2) + F(2)
A(3) = B(3) + C(3)
 D(3) = E(3) + F(3)
A(4) = B(4) + C(4)
 D(4) = E(4) + F(4)
```

Vector

Order of execution

```
A(1) = B(1) + C(1)
A(2) = B(2) + C(2)
A(3) = B(3) + C(3)
A(4) = B(4) + C(4)
 D(1) = E(1) + F(1)
 D(2) = E(2) + F(2)
 D(3) = E(3) + F(3)
 D(4) = E(4) + F(4)
```

Vectorization changes the order of computation compared to sequential case – Compiler determines when it's safe to do so.

Code With Dependences

```
DIMENSION A(5)
DATA (A(I),I=1,5 /1.,2.,3.,1.,2./)
X = 6.0
DO I = 1, 3
   A(I+2) = A(I) + X
END DO
```

OLD (NOT UPDATED) VALUE

SCALAR

```
A(3) = A(1) + X    7=1+6
A(4) = A(2) + X    8=2+6
A(5) = A(3) + X   13=7+6
```

VECTOR

```
A(3) = A(1) + X    7=1+6
A(4) = A(2) + X    8=2+6
A(5) = A(3) + X    9=3+6
```

NOT VECTORIZABLE

Data Dependencies

- **Examples:**

```
DO I=2,N-1
  A(I) = A(I-1) + B(I)
END DO
```

Compiler detects backward reference on `A(I-1)`

Read-after-write (also known as "flow dependency")

```
DO I=2,N-1
  A(I-1) = X(I) + DX
  A(I)   = 2.*DY
END DO
```

Compiler detects same location being written

Write-after-write (also known as ”output dependency")

```
ftn -vec-report=2 -c mms.f90
mms.f90(190): (col. 5) remark: loop was not vectorized: existence of vector dependence
ftn -vec-report=6 -c mms.f90
mms.f90(191): (col. 7) remark: vector dependence: assumed FLOW dependence between
                    mms_module_mp_ib_ line 191 and mms_module_mp_ib_ line 191
```

Loops with Conditionals

- **This loop will vectorize:**

```
DO I=1,N
  IF (A(I).NE.0) A(I) = D(I)*C(I)
END DO
```

Compiler generates vector mask instructions that will cause the result of the computation to be stored only where the condition is TRUE

Can vectorize block IF constructs but performance may depend on "truth" ratio

```
DO I=1,N
  IF (D(I).NE.0) THEN
   A(I) = D(I)*C(I)+S*B(I)
  ELSE
   A(I) = 0.0
END DO
```

AutoVectorizing Tips

- Use countable, single-entry/single-exit DO or “for” loops; loop bounds must be invariant within the loop; change IF sequences to DO loops
- Avoid branches switch, GOTO or CYCLE statements and most function calls
- Use array notation instead of pointers
- Avoid dependences; split loop if possible
- Use loop index directly in array subscripts
- Constant, unit-stride memory access
- Use built-in functions (e.g., `sqrt` instead of `A**5`)
- Use Intel Math Kernel Library (MKL) whenever possible.
  - Specialized/optimized versions of many math functions (e.g., blas, lapack).
- Split an unvectorizable loop into two separate loops: one vectorizable and one non-vectorizable.
- Use only one data type within a loop.
- Join two loops together where possible to lower loop overhead and memory access, and improve scheduling and pipelining.

Additional Vectorization of Loops

- When the compiler does not vectorize automatically due to dependences
  - The programmer can inform the compiler that it is safe to vectorize:
    - #pragma ivdep
    - #pragma ibm independent_loop
  - The programmer can sometimes manually transform the code to remove data dependences:
    - Loop Distribution or loop fission
    - Reordering Statements
    - Node Splitting
    - Scalar expansion
    - Loop Peeling
    - Loop Fusion
    - Loop Unrolling
    - Loop Interchanging

Some Loop Transformations

- Fusion: merge adjacent loops with identical control into one loop
- Peeling: Remove (generally) the first or last iteration of the loop into separate code outside the loop
- Fission: Split a single loop into more than one, generally to remove a dependency
- Interchange (permutation): reverse the order in a loop nest
- Combinations: unroll & jam = unroll an outer loop in a nest and fuse with an inner loop

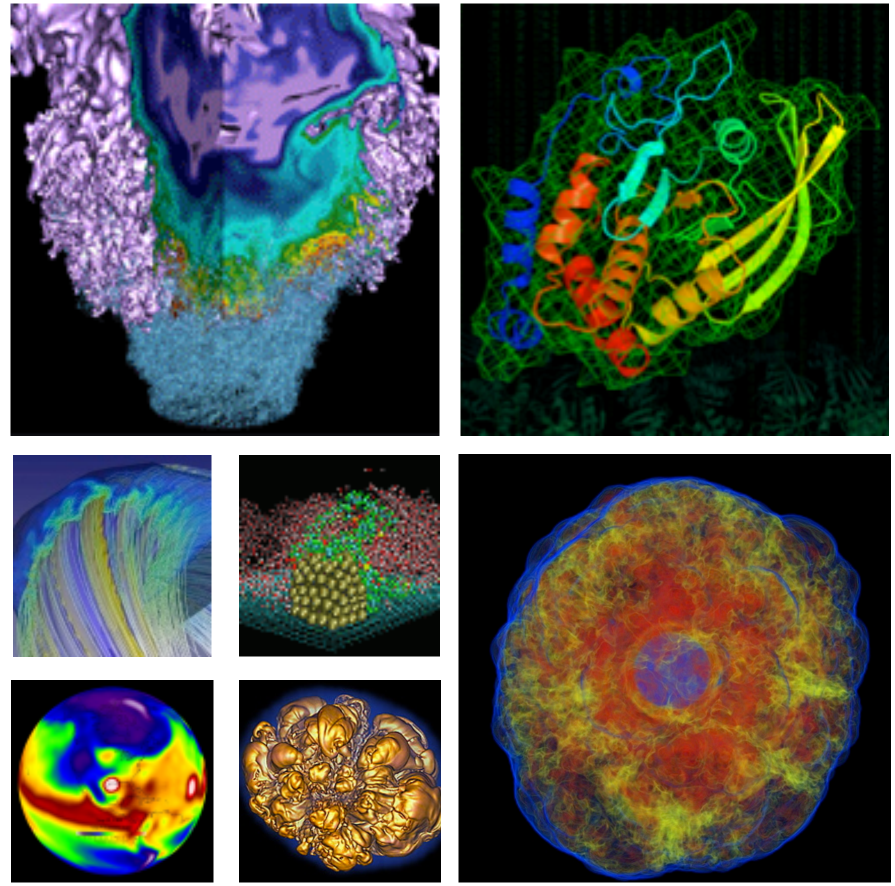
Get Started Now

- We urge you to begin exploring vectorization and OpenMP using Edison and the Intel Knights Corner Xeon Phi MIC Babbage testbed
- Edison: Vectorization analysis can be done but small vectorization performance effect
- Babbage: Other performance (e.g., MPI scaling) may be poor but if code vectorizes well, it probably will on Cori, too.
- You can assess overall vectorization effect by compiling with vectorization turned off
  (-no-vec -no-simd)

Next Steps

- **Woo-Sun Yang**
  - Performance effects associated with memory bandwidth
  - C pointer dependencies
  - Other important transformations
- **Jack Deslippe**
  - Setting Expectations
  - Directives/Pragmas SIMD
  - Using VTUne for vectorization improvement
  - BerkeleyGW Case Study: Then and Now

Vectorization Performance



Woo-Sun Yang

NERSC User Services Group

October 28, 2014

NERSC 40 YEARS at the FOREFRONT 1974-2014

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

Vectorization performance

- **Factors that affect vectorization performance**
  - Efficient loads and stores with vector registers
    - Data in caches
      - Will be discussed in Jack's talk
      - Cache blocking, prefetching
    - Data aligned to a certain byte boundary in memory
    - Unit stride access
  - Efficient vector operations
    - Certain arithmetic operations not at full speed
- **Near theoretical speed-up with vectorization when all the conditions are met**
- **Examine some aspects (and others)**
- **Examples from https://www.nersc.gov/users/computational-systems/edison/programming/vectorization/**

Memory alignment

- **More instructions are needed to collect and organize in registers if data is not optimally laid out in memory**
- **Data movement is optimal if the address of data starts at certain byte boundaries**
  – SSE: 16 bytes (128 bits)
  – AVX: 32 bytes (256 bits)
  – KNC: 64 bytes (512 bits)

Memory alignment to assist vectorization

- **From** https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization
- **Alignment of data (Intel)**
  - Fortran compiler flag -align
    - '-align array<n>bytes', where n=8,16,32,664,128,256, as in '–align array64byte'
    - Entities of COMMON blocks: '-align commons' (4-byte); '-align dcommons' (8-byte); '-align qcommons' (16-byte); '-align zcommons' (32-byte); none for 64-byte
    - '-align rec<n>byte', where n=1,2,4,8,16,32,64: for derived-data-type components
  - Alignment directive/pragmas in source code
    - Fortran
      - !dir$ attributes align: 64::A – when A is declared
      - !dir$ assume_aligned A:64 – informs that A has been aligned
      - !dir$ vector aligned – vectorize a loop using aligned loads for all arrays
    - C or C++
      - 'float A[1000] __attribute__((align(64));' or '__declspec(align(64)) float A[1000];' when declaring a static array
      - _alligned_malloc()/_aligned_free() or _mm_malloc()/_mm_free() to allocate heap memory
      - __assume_aligned(A,64)
      - #pragma vector aligned – vectorize a loop using aligned loads for all arrays

Example code

```
void myfunc(float p[]) {
  __assume_aligned(p,32);
  int i;
  for (i=0; i<N; i++)
    p[i]++;
}
...
int main() {
  ...
  float A[N], B[N] __attribute__((aligned(32)));
  float *C, *D;
  ...
  C = _mm_malloc(N*sizeof(float),32);
  ...
  myfunc(A);

#pragma vector aligned
  for (i=0;i<N; i++)
    A[i] = B[i] * C[i] + D[i];
  ...
}
$ icc –vec-report=3 atest1.c
...
atest1.c(29): (col. 3) remark: LOOP WAS VECTORIZED
atest1.c(9): (col. 3) remark: LOOP WAS VECTORIZED
```

Informs the compiler that the variable is aligned by 32 bytes, to vectorize the following loop

Arrays start on 32-byte boundaries

Arrays start on 32-byte boundaries

Vectorize the loop because all data in the loop is aligned

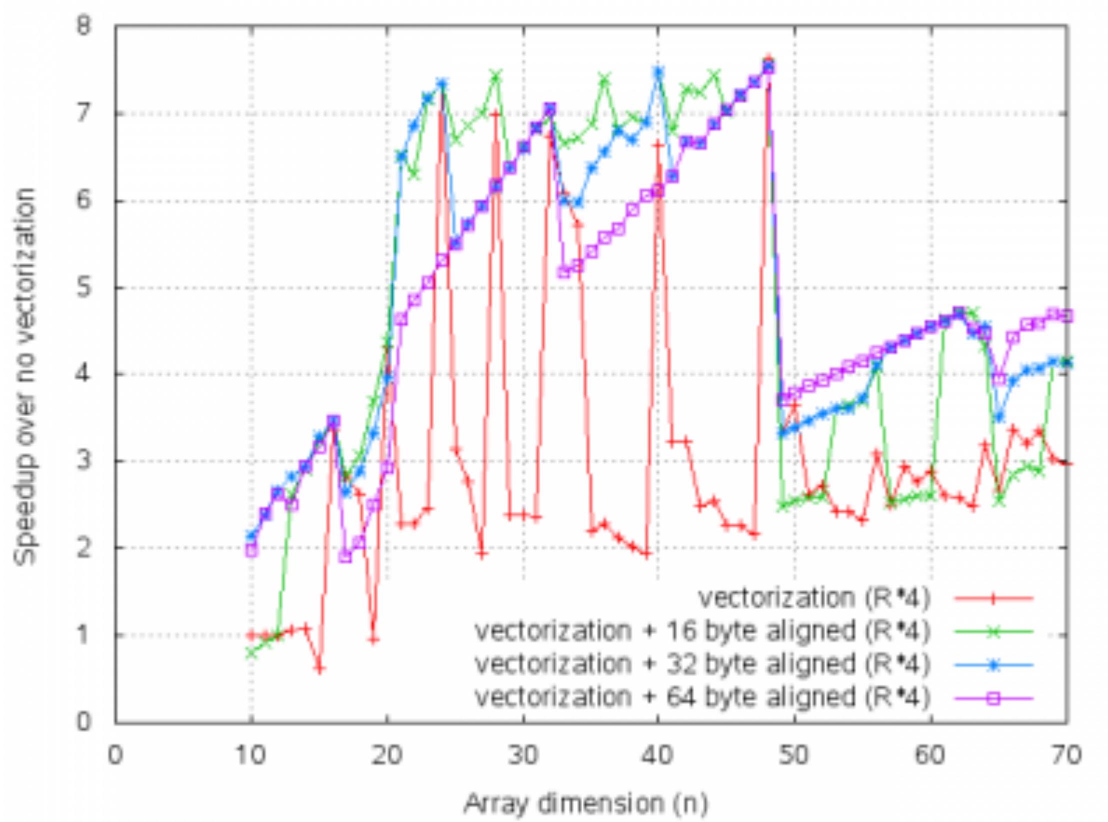
Memory alignment for multidimensional arrays

- **Multi-dimensional arrays need to be padded in the fastest-moving dimension, to ensure all the subarrays to align to the desired byte boundaries**
  - Fortran: first array dimension
  - C/C++: last array dimension
- **npadded = ((n + veclen – 1) / veclen) * veclen**
  - No alignment requested: veclen = 1
  - 16-byte alignment (SSE): veclen = 4 (sp) or 2 (dp)
  - 32-byte alignment (AVX): veclen = 8 (sp) or 4 (dp)
  - 64-byte alignment (KNC): veclen = 16 (sp) or 8 (dp)

Memory alignment example

- **Naïve matrix-matrix multiplication on Edison**

```
real, allocatable :: a(:,:), b(:,:), c(:,:)
!dir$ attributes align : 32 :: a,b,c
...
allocate (a(npadded,n))
allocate (b(npadded,n))
allocate (c(npadded,n))
...
do j=1,n
   do k=1,n
!dir$ vector aligned
      do i=1,npadded
         c(i,j) = c(i,j) &
            + a(i,k) * b(k,j)
      end do
   end do
end do

!... Ignore c(n+1:npadded,:)
```



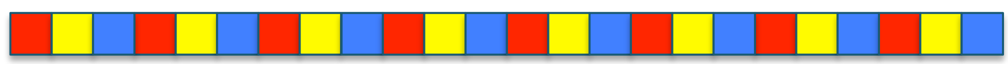
Memory alignment example

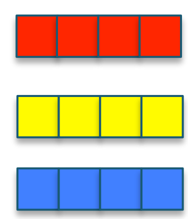
```
% cat ${INTEL_PATH}Samples/en_US/C++/vec_samples

% cat Driver.c
...
#define COLBUF 1
...
#define COLWIDTH COL+COLBUF
...
        FTYPE a[ROW][COLWIDTH]  __attribute__((aligned(16)));
        FTYPE b[ROW]                    __attribute__((aligned(16)));
        FTYPE x[COLWIDTH]               __attribute__((aligned(16)));
...

% cat Multiply.c
...
#pragma vector aligned
...
                for (j = 0;j < size2; j++) {
                        b[i] += a[i][j] * x[j];
                }
...
```

AoS vs. SoA

- **Data objects with component elements or attributes**
- **Array of a structure (AoS)**
  - The natural order in arranging such objects
  - But it gives non-unit strided access when loading into vector registers

```
type coords
  real :: x, y, z
end type
type (coords) :: p(100)
real dsquared(100)

do i=1,100
  dsquared(i) = p(i)%x**2 + p(i)%y**2 + p(i)%z**2
end do
```

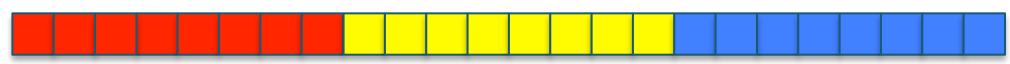


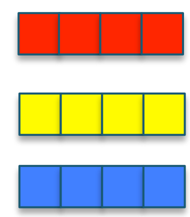


AoS vs. SoA

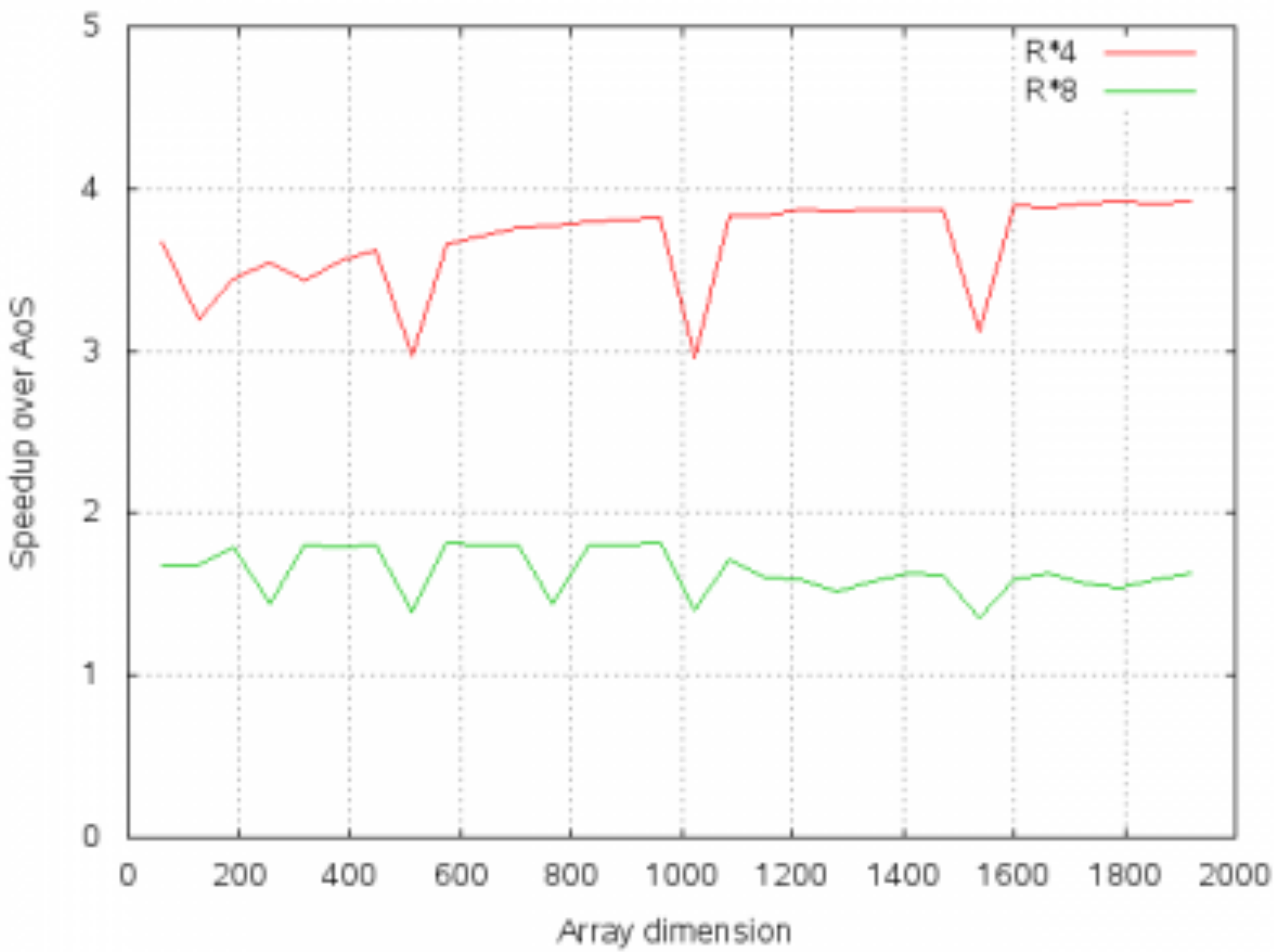
- **Structure of arrays (SoA)**
  - Unit strided access when loading into vector registers
  - More efficient with loading into vector registers

```
type coords
  real :: x(100), y(100), z(100)
end type
type (coords) :: p
real dsquared(100)

do i=1,100
  dsquared(i) = p%x(i)**2 + p%y(i)**2 + p%z(i)**2
end do
```





AoS-SoA example on Edison



Elemental function

- **An elemental function operates element-wise and returns an array with the same shape as the input parameter**
  – Widely used in Fortran intrinsic functions (but not in a vectorization sense)
- **When declared, the Intel compiler generates a vector version and a scalar version of the function**
- **A function call within a loop generally inhibits vectorization. But a loop containing a call to an elemental function can be vectorized. In that case, the vector version is used**
- **In vector mode, the function is called with multiple data packed in a vector register and returns packed results**

Elemental function example

```
module fofx
contains
  function f(x)
!dir$ attributes vector :: f
    real, intent(in) :: x
    real f
    f = cos(x * x + 1.) / (x * x + 1.)
  end function
end module

program main
  use fofx
  real a(100), x(100)
  ...
  do i=1,100
    a(i) = f(x(i))
  end do
  ...
end program
```

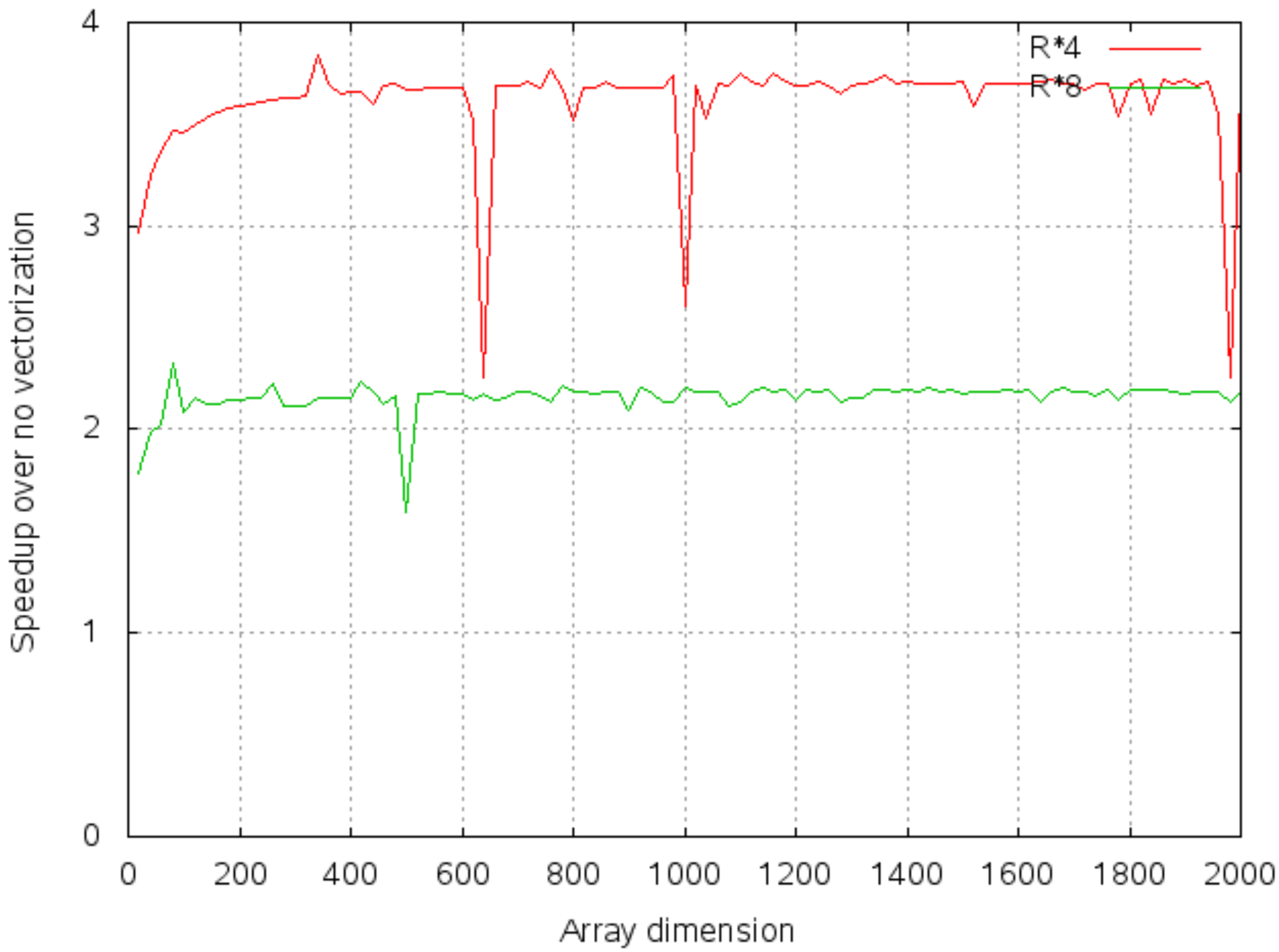
Line 7

Elemental function in vectorization sense

Line 67

```
$ ifort –vec-report=3 elemental.f90
...
elemental.f90(67): (col. 11) remark: LOOP WAS
VECTORIZED
...
elemental.f90(7): (col. 18) remark: FUNCTION WAS
VECTORIZED
elemental.f90(7): (col. 18) remark: FUNCTION WAS
VECTORIZED
```

Elemental function example on Edison



Elemental function example 2

- **This seems to vectorize, too, and generate similar results**

```
module fofx
contains
  elemental function f(x)
!dir$ attributes vector :: f
    real, intent(in) :: x
    real f
    f = cos(x * x + 1.) / (x * x + 1.)
  end function
end module

program main
  use fofx
  real a(100), x(100)
  ...
  a = f(x)
  ...
end program
```

This Fortran 'elemental' clause nothing to do with vect.

Elemental function in vectorization sense

Line 67

```
$ ifort –vec-report=3 elemental.f90
...
elemental.f90(67): (col. 11) remark: LOOP WAS
VECTORIZED
...
elemental.f90(7): (col. 28) remark: FUNCTION WAS
VECTORIZED
elemental.f90(7): (col. 28) remark: FUNCTION WAS
VECTORIZED
```

Aliasing

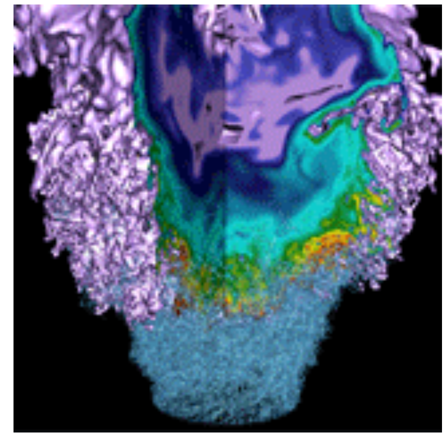
- Use the 'restrict' key to indicate that the memory regions referenced by pointers don't overlap, thus, it's safe to vectorize
- Use with the '-restrict' compiler option

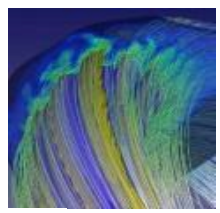
```
$ cat atest.c
...
void add(float * restrict a, float* restrict b, float* restrict c) {
  for (int i=0; i<SIZE; i++) {
    c[i] += a[i] + b[i];
  }
}
$ icc -c -restrict -vec-report=3 atest.c
atest.c(5): (col. 3) remark: LOOP WAS VECTORIZED
atest.c(5): (col. 3) remark: REMAINDER LOOP WAS VECTORIZED
```

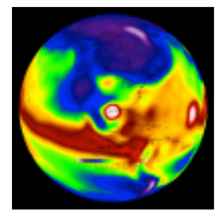
No overlapping region among a, b, c; thus no data dependency

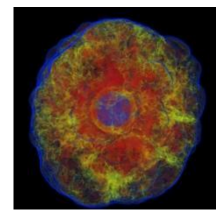
- More info in https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf

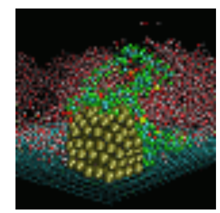
Will Vectorization Help My Code?











NERSC 40 YEARS at the FOREFRONT

Jack Deslippe

U.S. DEPARTMENT OF ENERGY | Office of Science

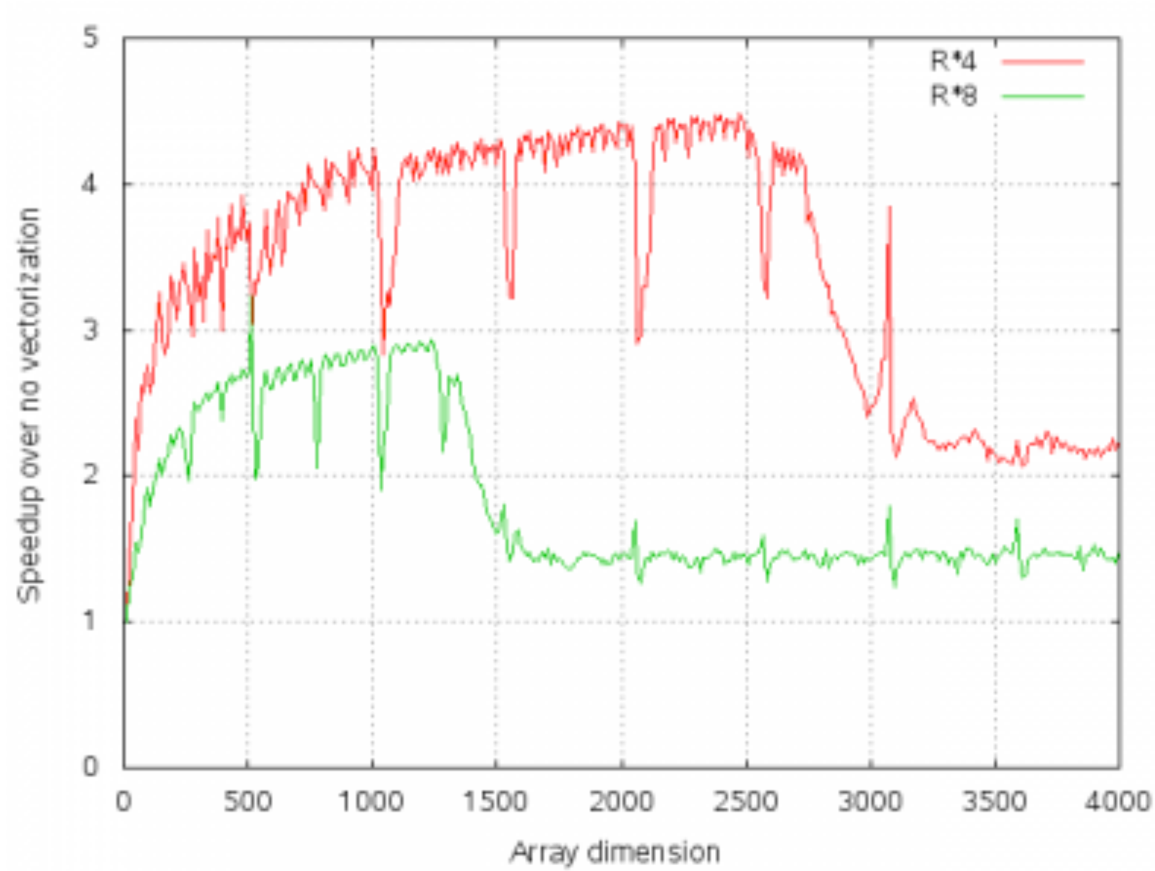
BERKELEY LAB
Lawrence Berkeley National Laboratory

It Depends

- Are you memory bandwidth bound or…
- Are you compute bound?

Vector Add Example:

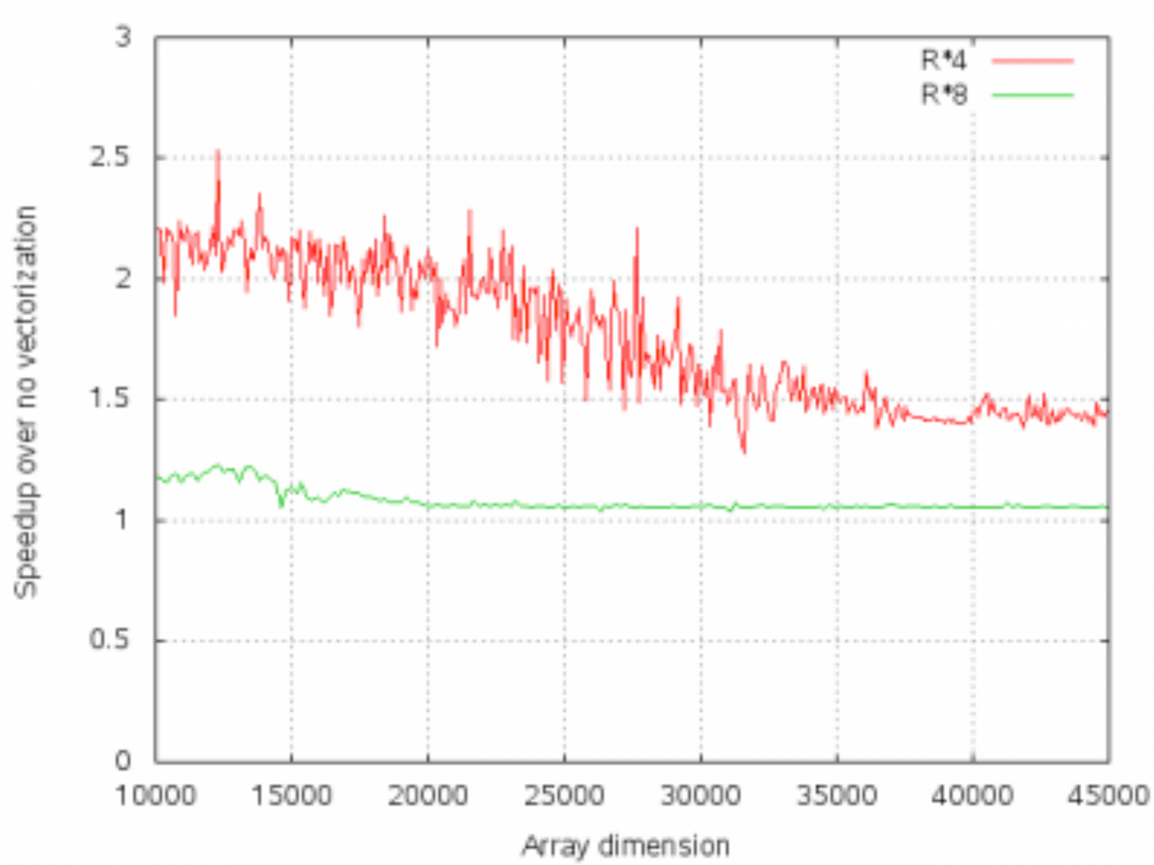
```
do i=1,n
    c(i) = a(i) + b(i)
end do
```



Speedup close the theoritical max below L1 Cache. Worse as array size passes L1 size.

Vector Add Example:

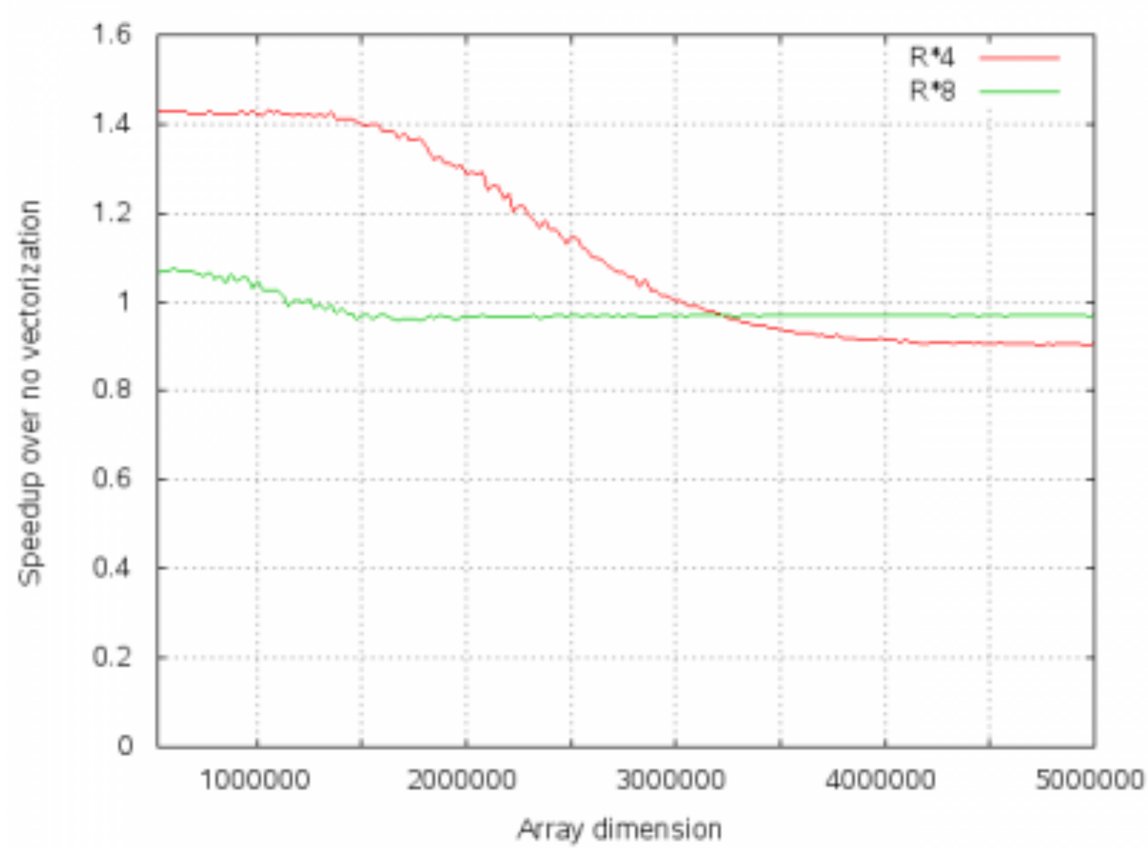
```
do i=1,n
    c(i) = a(i) + b(i)
end do
```



Speedup drops again as pass L2 cache size.

Vector Add Example:

```
do i=1,n
    c(i) = a(i) + b(i)
end do
```



Speedup once again drops above L3 boundary.

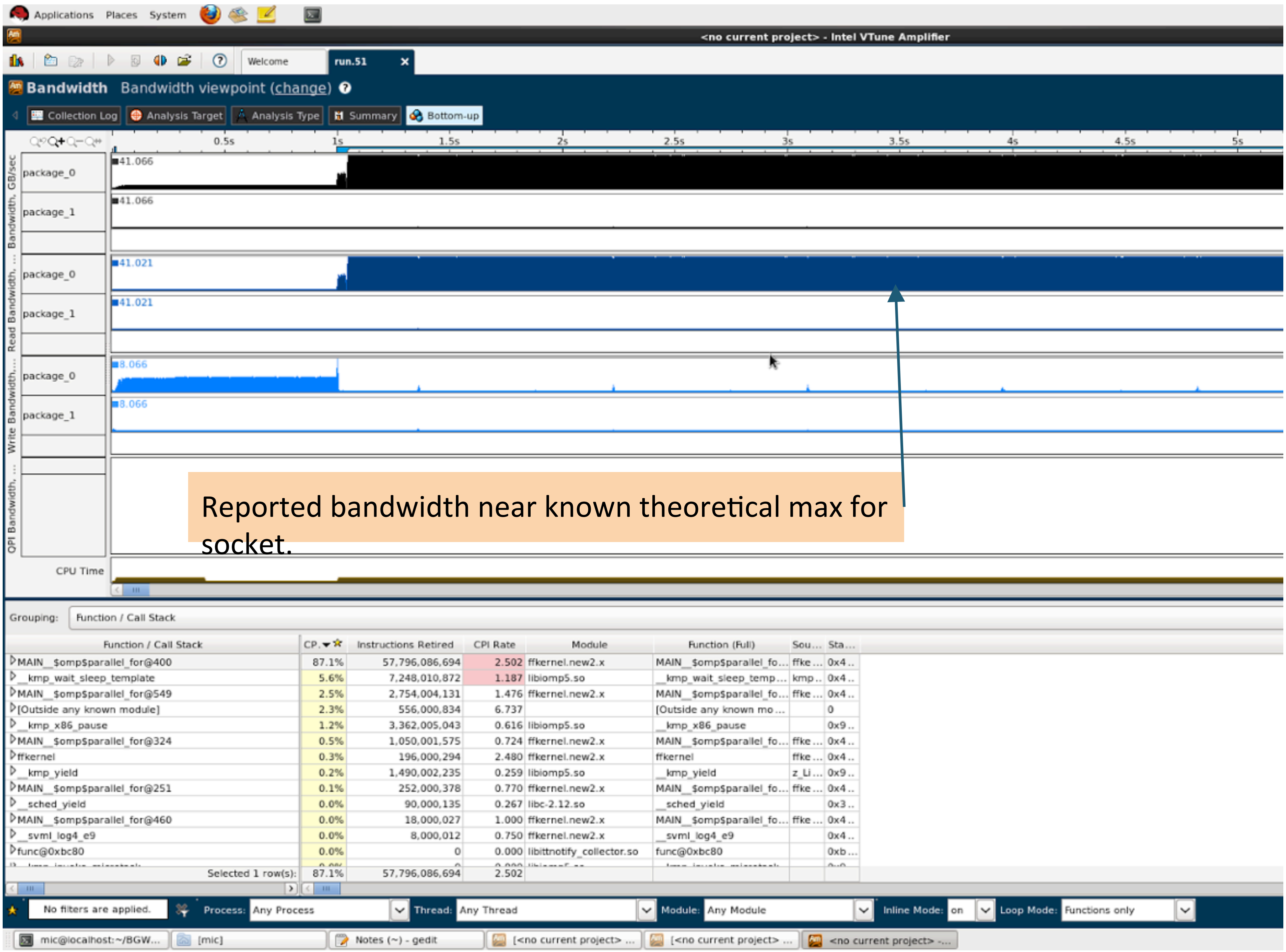
How to know if you are bandwidth bound

Use VTune Bandwidth Analysis on Babbage:

```
% amplxe-cl -collect bandwidth -r run.name ./code.x <code arguments>

% amplxe-gui
```

Compare bandwidth usage to reported max: (164GB/s read; 76 GB/s write for Babbage)

https://www.nersc.gov/users/computational-systems/edison/configuration/compute-nodes/ (Edison)



Reported bandwidth near known theoretical max for socket.

How Reduce Memory Bandwidth Requirements

"Cache Blocking":

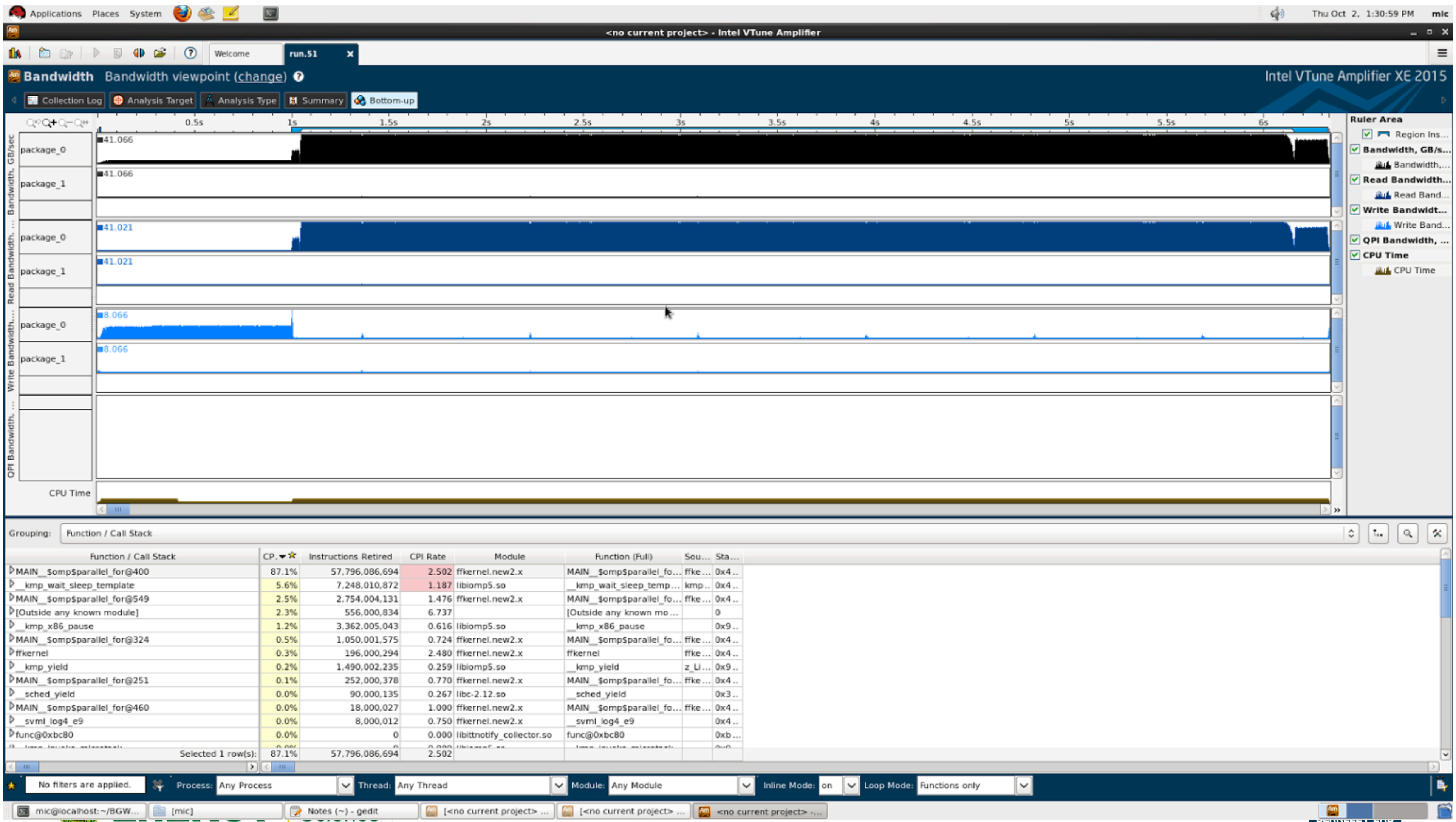
```
do i = 1, n
    do j = 1, m
        c += a(i) * b(j)
    enddo
enddo
```

```
do jout = 1, m, block
    do i = 1, n
        do j = jout, jout+block
            c += a(i) * b(j)
        enddo
    enddo
enddo
```

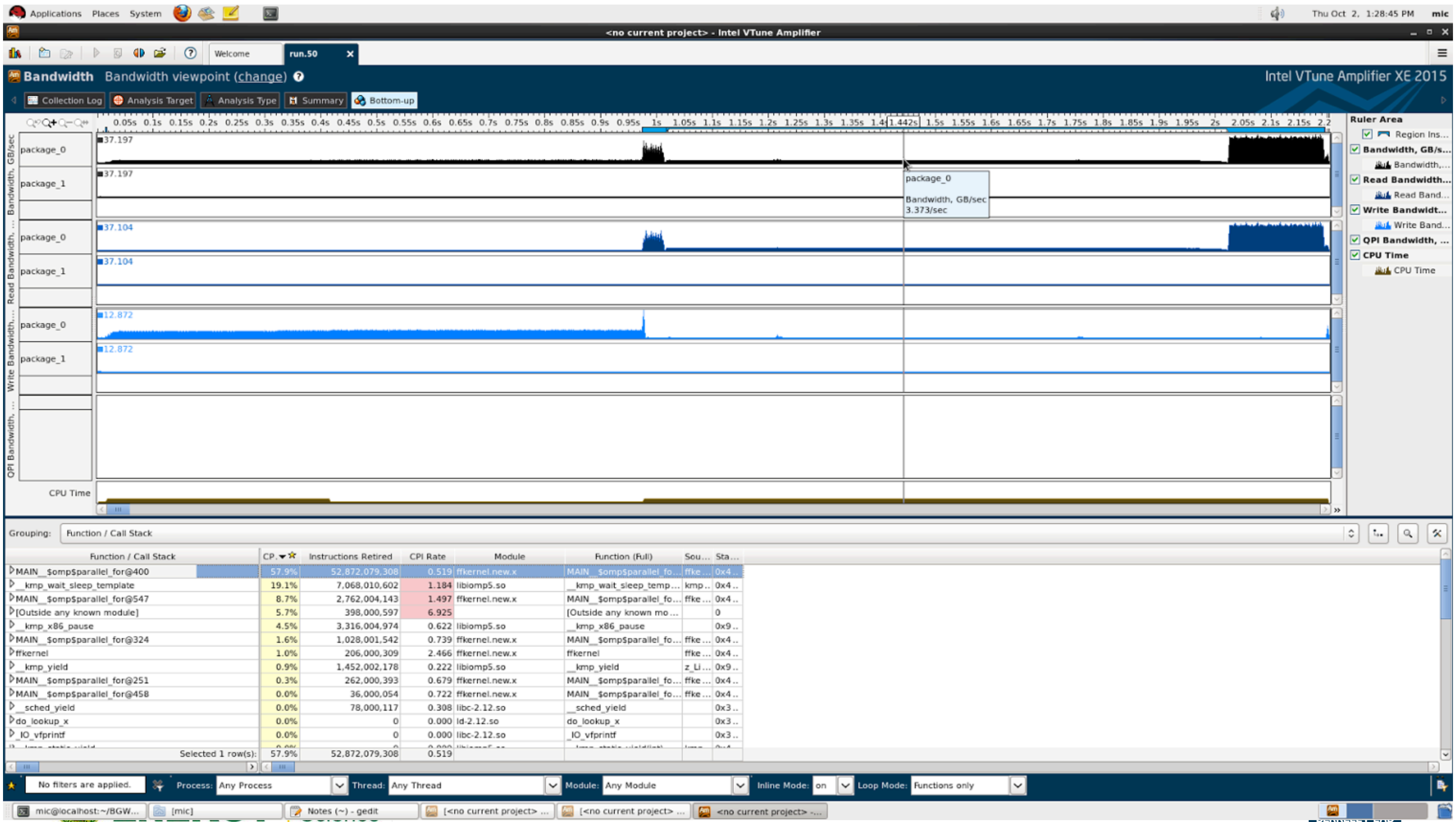
Loads From DRAM:
n*m + n

Loads From DRAM:
m/block * (n+block)
= **m*n/block** + m

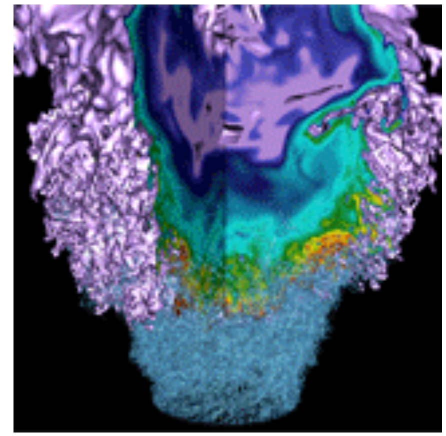
BerkeleyGW Example Before Tuning

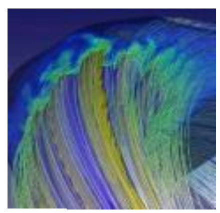


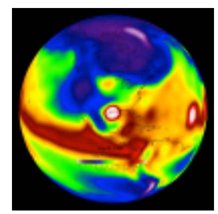
BerkeleyGW Example After Tuning

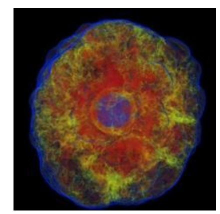


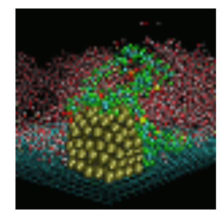
Using OpenMP SIMD











NERSC 40 YEARS at the FOREFRONT

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

OpenMP 4 SIMD

A year ago:
To get vector code, you had to use intrinsics, pray the compiler chose to vectorize a loop, or use compiler specific directives.

Today:

```
#pragma omp simd reduction(+:sum) aligned(a : 64)
for(i = 0; i < num; i++) {
 a[i] = b[i] * c[i];
 sum = sum + a[i];
}
```

Warning

- Using OpenMP SIMD bypasses the compiler analysis;
- use with caution!
- Incorrect results possible!
- Poor performance possible!
- Memory errors possible!

Aligned Memory

Fortran:

`-align array64byte , -align rec64byte` compiler flags to get aligned heap memory allocation

```
real var(100)
!dir$ attributes align:64::var
```

C:

```
_mm_malloc(8*sizeof(float), 64);
```

OpenMP 4 SIMD

OpenMP Parallel + SIMD on Same Loop

```
#pragma omp parallel for simd
for(i = 0; i < num; i++) {
 sum = sum + a[i];
}
```

OpenMP 4 SIMD

OpenMP SIMD…. Functions

```
#pragma omp declare simd
float myfunction(float a, float b, float c )
{
 return a * b + c ;
}
```

```
#pragma omp simd
for(i = 0; i < num; i++) {
 OUT[i] = myfunction(arraya[i], arrayb[i], arrayc[i]);
}
```

BGW Last Year Lessons

```
!$OMP DO reduction(+:achtemp)
  do my_igp = 1, ngpown

    ...

    do iw=1,3

      scht=0D0
      wxt = wx_array(iw)

      do ig = 1, ncouls

        !if (abs(wtilde_array(ig,my_igp) * eps(ig,my_igp)) .lt. TOL) cycle

        wdiff = wxt - wtilde_array(ig,my_igp)
        delw = wtilde_array(ig,my_igp) / wdiff

        ...

        scha(ig) = mygpvar1 * aqsntemp(ig) * delw * eps(ig,my_igp)

        scht = scht + scha(ig)

      enddo ! loop over g

      sch_array(iw) = sch_array(iw) + 0.5D0*scht

    enddo

    achtemp(:) = achtemp(:) + sch_array(:) * vcoul(my_igp)

  enddo
```

ngpown typically in 100's to 1000s. Good for many threads.

Original inner loop. Too small to vectorize!

ncouls typically in 1000s - 10,000s. Good for vectorization. Don't have to worry much about memory. alignment.

Attempt to save work breaks vectorization and makes code slower.

BGW New Lessons

```
!$OMP DO SIMD reduction(+:achtemp)
  do my_igp = 1, ngpown

    ...

    do iw=1,3

      scht=0D0
      wxt = wx_array(iw)

      do ig = 1, ncouls

        !if (abs(wtilde_array(ig,my_igp) * eps(ig,my_igp)) .lt. TOL) cycle

        wdiff = wxt - wtilde_array(ig,my_igp)
        delw = wtilde_array(ig,my_igp) / wdiff

        ...

        scha(ig) = mygpvar1 * aqsntemp(ig) * delw * eps(ig,my_igp)

        scht = scht + scha(ig)

      enddo ! loop over g

      sch_array(iw) = sch_array(iw) + 0.5D0*scht

    enddo

    achtemp(:) = achtemp(:) + sch_array(:) * vcoul(my_igp)

  enddo
```

ADD SIMD AT THE SAME LEVEL OF OpenMP DO

Thank you