NCCS Brown Bag Series
Vectorization
Efficient SIMD parallelism on NCCS systems

Craig Pelissier* and Kareem Sorathia
craig.s.pelissier@nasa.gov,
kareem.a.sorathia@nasa.gov
SSSO code 610.3

January 28, 2014
Motivation

Exploiting parallelism in programs will increase throughput and more efficiently utilize modern CPUs.

- Compilers guarantee that the results of a program are the same as the results obtained as if the program is executed sequentially! Programs in general are not executed sequentially.

- Compilers try to exploit the inherent parallelism in the programs. Programmers sometimes need to help out the compiler.

- Programs need to be written in a way that exploits the parallel mechanisms available on modern CPUs!
Vector instructions (SIMD)

- **Single Instruction Multiple Data (SIMD)** is a commonly occurring situation!

```
  do i=1,N
      c(i) = a(i) + 5*b(i)
  enddo
```

Assumptions:
(1) Instructions are independent (no dependencies).
(2) Instructions are identical (same operation).
(3) Multiple data (multiple iterations).

Q: How to exploit parallelism?
A: Modern CPUs use vector registers!
Vector processes

- Intel offers vector registers that are essentially large registers (“register banks”) that can fit several data elements.

- Vector registers (Intel) are fully pipelined and execute with 4-cycle latency and single cycle throughput.

<table>
<thead>
<tr>
<th>Size [bits]</th>
<th>Elements/reg (float,double)</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>(4,2)</td>
<td>Westmere</td>
</tr>
<tr>
<td>256</td>
<td>(8,4)</td>
<td>Sandy Bridge</td>
</tr>
<tr>
<td>512</td>
<td>(16,8)</td>
<td>Intel Xeon Phi</td>
</tr>
<tr>
<td>Scalable Unit 1-4, 7</td>
<td>Scalable Unit 8,9</td>
<td>Scalable Unit 8 coprocessors</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>2.8 GHz Intel Xeon Westmere</td>
<td>2.6 GHz Intel Xeon Sandy Bridge</td>
<td>Intel Many Integrated Core (Phi 5110p) (480 cards)</td>
</tr>
<tr>
<td>2 hex-core processors/node</td>
<td>2 oct-core processors/node</td>
<td>60 cores</td>
</tr>
<tr>
<td>24 GB of memory/node</td>
<td>32 GB of memory/node</td>
<td>6 GB memory/card</td>
</tr>
<tr>
<td>Infiniband DDR</td>
<td>Infiniband QDR</td>
<td>PCIexpress</td>
</tr>
<tr>
<td>SSE 128b</td>
<td>AVX 256b</td>
<td>AVX 512b</td>
</tr>
</tbody>
</table>

SSE = Streaming SIMD Extensions (1999)
AVX = Advanced Vector eXtensions (2011)
Guidelines for vectorizable code by example

Disclaimer! This is in no way an exhaustive list of what will and what won’t vectorize. More information can be found at Intel’s guidelines [here](#) and other available resources online.

Use simple for loops.

```fortran
i = 1
while (a(i) >= 0) do ! Not allowed
   a(i) = a(i)+1
   i = i+1
enddo
```

Loop bounds must be known at compile time.

```fortran
do i=1,N
   if (a(i) <= 0) exit
   a(i) = sqrt(a(i))
endo
```

No conditional exit of loops.
Guidelines for vectorizable code by example

Loops with dependencies can’t vectorize.

```fortran
do i=2,N-1
   a(i) = 0.25*a(i-1)+0.5*a(i)+0.25*a(i+1)
enddo
```

✗  Dependences can appear in a single line.

```fortran
do i=1,N
   a(i) = b(i) + c(i)
   d(i) = e(i) - a(i-1) !Allowed
enddo
```

✓  In larger loops dependencies can be more subtle!

```fortran
do i=1,N
   d(i) = e(i) - a(i-1) !Not Allowed
   a(i) = b(i) + c(i)
enddo
```

✗
Guidelines for vectorizable code by example

No control logic inside loop unless it can be masked.

\begin{verbatim}
do i=2,N-1
  !Loop vectorizable because of masking
  if (c(i) .GT. 0) then
    da(i) = a(i)-a(i-1)
  else
    da(i) = a(i+1)-a(i)
  endif
endo
do i=1,N
  rem = mod(i,3) !Not vectorizable
  select case (rem)
    case (0)
      a(i) = sin(a(i))
    case (1)
      a(i) = cos(a(i))
    case (2)
      a(i) = exp(a(i))
  end select
endo
\end{verbatim}

Simple binary logic is allowed because it can be masked!

More complicated branching is not allowed!
No function calls inside the loop … with some exceptions.

```
do i=1,N
    c(i) = sin(a(i)) + 5*cos(b(i)) ✓
enddo
```

Intrinsic math functions are allowed e.g. trigonometric, exponential, logarithmic, and round/floor/ceil …

```
do i=1,N
    !DEC$ INLINE
    c(i) = my_function(a) ?
enddo
```

Functions which can be inlined may vectorize.
## Directing the compiler

### Auto-vectorization directive

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!DIR$ IVDEP</td>
<td>Instructs the compiler to ignore assumed vector dependencies.</td>
</tr>
<tr>
<td>!DIR$ VECTOR [ALWAYS]</td>
<td>Specifies how to vectorize the loop and indicates that efficiency heuristics should be ignored. Using the ASSERT keyword with the VECTOR [ALWAYS] directive generates an error-level assertion message saying that the compiler efficiency heuristics indicate that the loop cannot be vectorize. USE DIR$ IVDEP to ignore the assumed dependencies.</td>
</tr>
<tr>
<td>!DIR$ NOVECTOR</td>
<td>Specifies that the loop should never be vectorized.</td>
</tr>
</tbody>
</table>

### User-mandated directive

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!DIR$ SIMD</td>
<td>Enforces vectorization of the loops.</td>
</tr>
</tbody>
</table>
Some more exotic directives

**Auto-vectorization directive**

<table>
<thead>
<tr>
<th>!DIR$ LOOP COUNT</th>
<th>The value of the loop count affects heuristics used in software pipelining, vectorization, and loop-transformations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>!DIR$ VECTOR [TEMPORAL</td>
<td>NONTEMPORAL]</td>
</tr>
</tbody>
</table>

---

```plaintext
!The next do loop has 1000 iterations
!DEC$ LOOP COUNT (1000)

!The next do loop has either
!100, 200, or 300 iterations
!DEC$ LOOP COUNT (100, 200, 300)

!The next do loop has maximum, minimum, and average iterations of 100/3/20
!Note, a subset of these values may also be used
!DEC$ LOOP COUNT MAX(100), MIN(3), AVG(20)

!I’ll need the LHS again shortly so keep it handy.
!DEC$ VECTOR temporal

!I won’t be using the LHS for a while so don’t waste
!DEC$ VECTOR nontemporal
```
Memory layout

• Choose data layout carefully.
  • Structure of arrays SoA = good for vectorization.
  • Array of Structures = good for encapsulation.

• Use efficient memory access
  • Favor unit stride.
  • Minimize indirect addressing (using pointers to pointers).
  • Alignment of data (next slide).

More efficient for SIMD parallelization

Better for data encapsulation
Memory layout

- Memory is aligned on physical n-byte boundaries.
- SANDY = WESTMERE = 32 byte, Phi = 64 byte.
- load/stores run faster on aligned data.
- Largest penalty paid on the Xeon Phi architecture.
- Directives !DEC$ VECTOR {ALIGNED | UNALIGNED}. Caution: if data is not aligned when accessed incorrect results or exceptions will be thrown (segmentation fault).
- Fortran –align array[n]byte for alignment of all arrays.

Large main loop ⇒ alignment, peel, and remainder amortized.
Small main loop ⇒ alignment and remainder important.
Alignment examples

```fortran
real*8 :: a(6,100), b(6,100), c(6,100)
do j=1,100
   do i=1,6
      !DEC$ vector aligned !This will produce errors
      c(i,j) = a(i,j) + 5*b(i,j)
   enddo
endo
do i=1,6
   !DEC$ vector aligned !This is allowed
   c(i,j) = a(i,j) + 5*b(i,j)
endo
```

Not all iterations in “j” are aligned!

Adding padding we can achieve alignment at the cost of computation and a larger memory footprint. Is it worth it?
Choosing domains that are a multiple of the boundary length can sometimes help with alignment e.g. in finite differencing.

```fortran
!caveat NX%4 = 0
real*8 :: a(NX,NY+1), b(NX,NY+1)
do j=1,NY
  do i=1,NX
    !DEC$ vector aligned! This is allowed
    b(i,j) = 0.5*(a(i,j) - a(i,j+1))
  enddo
enddo
```

```fortran
!caveat (NX+1)%4 = 0
real*8 :: a(NX+1,NY), b(NX+1,NY)
do j=1,NY
  do i=1,NX
    !DEC$ vector aligned! This will produce errors
    b(i,j) = 0.5*(a(i,j) - a(i+1,j))
  enddo
enddo
```
Q: How do I know if loops are vectorized?
A: Use --vec-report[0-6].

Q: How do I prioritize my vectorization efforts?
A: Simple option is to use the loop profiler provided by Intel to determine the most time consuming loops.

2. Running the executable will create an XML file.
3. Run loopprofileviewer.sh to start a GUI.
4. Open XML file with GUI.

Q: How do I know how much vectorization is improving things?
A: Compiling with the flag --no-vec will turn off all vectorization. You can also use more sophisticated software such as VTune amplifier see brown bag.
### Loop profiler output

#### Loop profile viewer: /gafsm/deb02/cpeisell/alice/CRAB/loop_prof_1390323258.xml

<table>
<thead>
<tr>
<th>Function</th>
<th>Function filename</th>
<th>Times</th>
<th>% Time</th>
<th>BfTime</th>
<th>% BfTime</th>
<th>GfTime</th>
<th>% GfTime</th>
<th>Gf count</th>
<th>% Time in loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3189</td>
<td>67,050,617,117</td>
<td>95.39</td>
<td>1.309</td>
<td>28.494</td>
<td>45.282</td>
<td>11.207</td>
<td>1,536,011</td>
<td>3.45</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3190</td>
<td>90,988,854,340</td>
<td>13.36</td>
<td>1.847</td>
<td>4.855</td>
<td>8.360</td>
<td>1.268</td>
<td>1,312,894</td>
<td>0.70</td>
</tr>
<tr>
<td>setup</td>
<td>physiscore/3191</td>
<td>15,451,302</td>
<td>2.14</td>
<td>0.047</td>
<td>0.120</td>
<td>0.073</td>
<td>0.020</td>
<td>1</td>
<td>0.06</td>
</tr>
<tr>
<td>tric</td>
<td>physiscore/3193</td>
<td>2,847,510,202</td>
<td>82.82</td>
<td>0.49</td>
<td>1.50</td>
<td>1.47</td>
<td>0.001</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>main</td>
<td>physiscore/3195</td>
<td>3,250,403,354</td>
<td>93.01</td>
<td>1.55</td>
<td>3.83</td>
<td>3.71</td>
<td>0.001</td>
<td>1</td>
<td>0.01</td>
</tr>
<tr>
<td>init_data</td>
<td>physiscore/3199</td>
<td>687,011,651</td>
<td>98.52</td>
<td>1.27</td>
<td>3.16</td>
<td>3.12</td>
<td>0.001</td>
<td>1</td>
<td>0.01</td>
</tr>
<tr>
<td>init</td>
<td>physiscore/3199</td>
<td>2,271,162,161</td>
<td>78.67</td>
<td>1.73</td>
<td>4.09</td>
<td>3.99</td>
<td>0.001</td>
<td>1</td>
<td>0.01</td>
</tr>
<tr>
<td>loop[32]</td>
<td>physiscore/3199</td>
<td>3,030,471,554</td>
<td>99.09</td>
<td>1.36</td>
<td>3.29</td>
<td>3.26</td>
<td>0.001</td>
<td>1</td>
<td>0.01</td>
</tr>
<tr>
<td>loop</td>
<td>physiscore/3199</td>
<td>5,377,491,201</td>
<td>99.09</td>
<td>1.36</td>
<td>3.29</td>
<td>3.26</td>
<td>0.001</td>
<td>1</td>
<td>0.01</td>
</tr>
</tbody>
</table>

#### Loop summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Function filename</th>
<th>Times</th>
<th>% Time</th>
<th>BfTime</th>
<th>% BfTime</th>
<th>GfTime</th>
<th>% GfTime</th>
<th>Gf count</th>
<th>% Time in loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3430</td>
<td>2,077,752,927</td>
<td>98.96</td>
<td>1.84</td>
<td>3.73</td>
<td>3.66</td>
<td>0.001</td>
<td>1</td>
<td>0.01</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3431</td>
<td>1,307,143,432</td>
<td>99.41</td>
<td>1.32</td>
<td>2.68</td>
<td>2.65</td>
<td>0.001</td>
<td>1</td>
<td>0.01</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3432</td>
<td>7,264,624,031</td>
<td>37.71</td>
<td>0.66</td>
<td>1.33</td>
<td>1.30</td>
<td>0.001</td>
<td>1</td>
<td>0.01</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3433</td>
<td>2,197,308,161</td>
<td>10.46</td>
<td>1.00</td>
<td>1.98</td>
<td>1.95</td>
<td>0.001</td>
<td>1</td>
<td>0.01</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3434</td>
<td>61,955,930</td>
<td>0.31</td>
<td>0.005</td>
<td>0.009</td>
<td>0.009</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3435</td>
<td>28,944,494</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3436</td>
<td>60,858,540</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3437</td>
<td>29,302,959</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3438</td>
<td>29,970,233</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3439</td>
<td>4,801,593</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3440</td>
<td>5,743,276</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3441</td>
<td>2,642,217</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3442</td>
<td>1,341,765</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3443</td>
<td>5,691,935</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3444</td>
<td>5,290,672</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3445</td>
<td>3,387,301</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3446</td>
<td>3,387,301</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3447</td>
<td>3,197,022</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3448</td>
<td>3,197,022</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3449</td>
<td>1,129,693</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3450</td>
<td>999,660</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3451</td>
<td>951,611</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3452</td>
<td>905,088</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>stridedouble, int, double, double, ...</td>
<td>physiscore/3453</td>
<td>2,128,619,484</td>
<td>78.70</td>
<td>1.40</td>
<td>2.89</td>
<td>2.86</td>
<td>0.001</td>
<td>1</td>
<td>0.01</td>
</tr>
</tbody>
</table>

**Average time:**

- Average time: 3.45 ms
- Average Gf time: 3.83 ms
- Average Gf count: 1,312,894
- Average Gf count in loop: 0.70 ms

**Max time:**

- Max time: 999,660 ms
- Max Gf time: 951,611 ms
- Max Gf count: 905,088 ms
- Max Gf count in loop: 2,128,619,484 ms
Regression testing

**Warning!** Reproducibility and optimization are sometimes at odds. Vectorization of loops will alter the bitwise reproducibility.

- A vectorized loop may give slightly different results than the original serial one.

- Variations in memory alignment from one run to the next may cause drifts in answers for different runs.

- Using “-fp-model precise” may disable vectorization.
  - Loops containing transcendental functions will not be vectorized.
Thank You!

Questions?