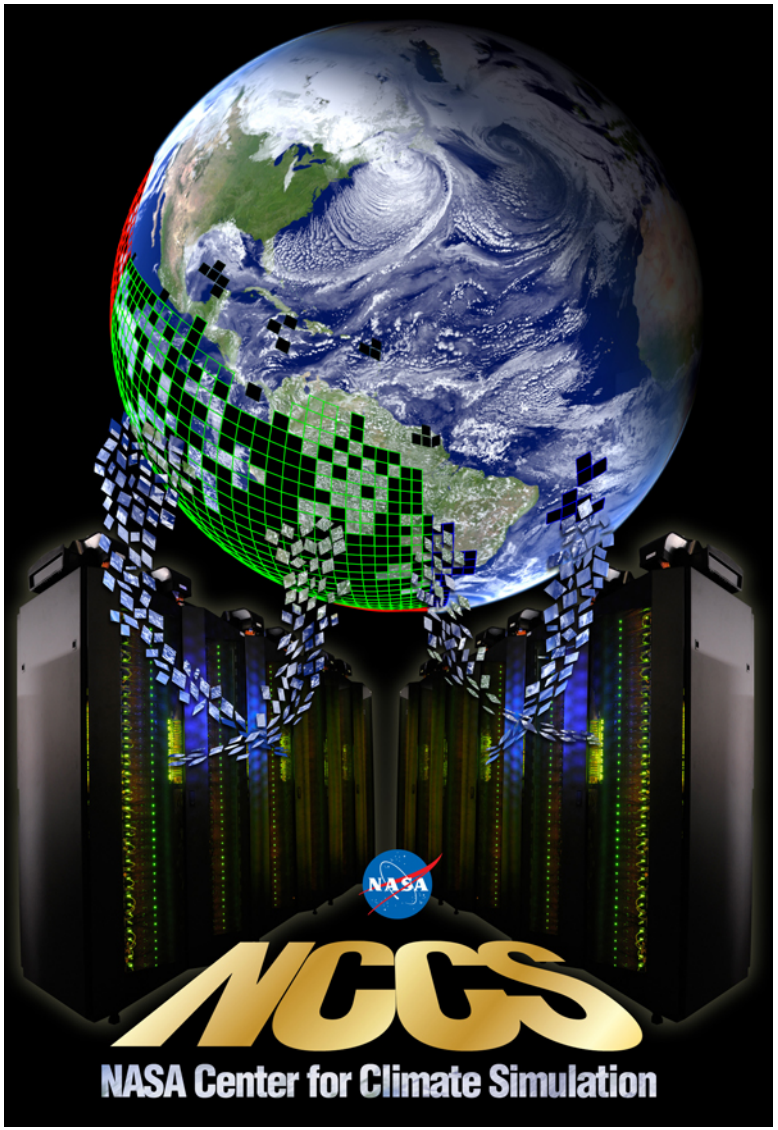
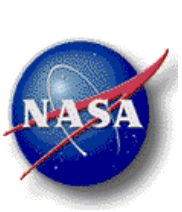
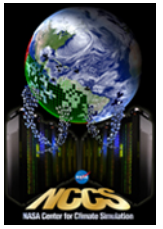


NCCS Brown Bag Series





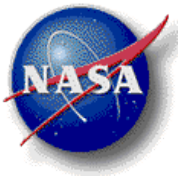
Vectorization

Efficient SIMD parallelism on NCCS systems

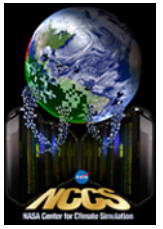
Craig Pelissier* and Kareem Sorathia

craig.s.pelissier@nasa.gov,
kareem.a.sorathia@nasa.gov
SSSO code 610.3

January 28, 2014

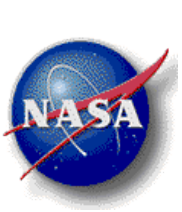


Motivation

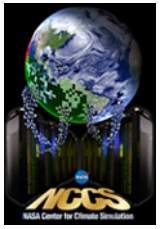


Exploiting parallelism in programs will increase throughput and more efficiently utilize modern CPUs.

- 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)



- **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD) is a commonly occurring situation!

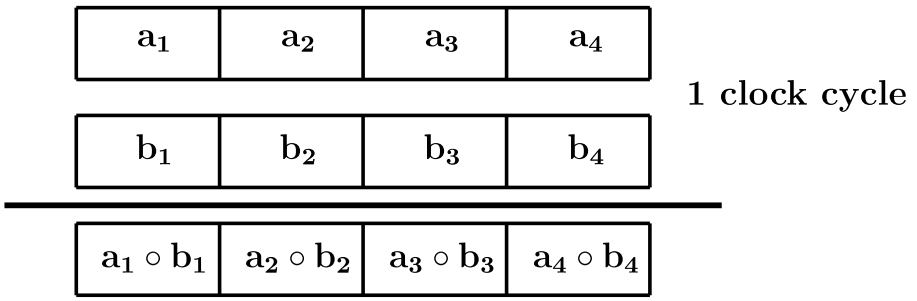
```
do  i=1,N
     c(i) = a(i) + 5*b(i)
enddo
```

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.



Size [bits]	Elements/reg (float,double)	Resource
128	(4,2)	Westmere
256	(8,4)	Sandy Bridge
512	(16,8)	Intel Xeon Phi

NCCS systems overview

Scalable Unit 1-4, 7	Scalable Unit 8,9	Scalable Unit 8 coprocessors
2.8 GHz Intel Xeon Westmere	2.6 GHz Intel Xeon Sandy Bridge	Intel Many Integrated Core (Phi 5110p) (480 cards)
2 hex-core processors/node	2 oct-core processors/node	60 cores
24 GB of memory/node	32 GB of memory/node	6 GB memory/card
Infiniband DDR	Infiniband QDR	PCIexpress
SSE 128b	AVX 256b	AVX 512b

SSE = **S**treaming **S**IMD **E**xtensions (1999)
AVX = **A**dvanced **V**ector e**X**tensions (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.

```
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.

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

✗

No conditional exit of loops.

Guidelines for vectorizable code by example

Loops with dependencies can't vectorize.

```
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.

```
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!

```
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.

```
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
enddo
```

Simple binary logic is allowed because it can be masked!

```
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
enddo
```

More complicated branching is not allowed!

Guidelines for vectorizable code by example

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 …

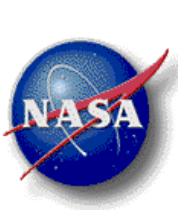
```
do i=1,N
   !DEC$ INLINE
   c(i) = my_function(a)
enddo
```

?

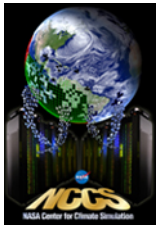
Functions which can be inlined may vectorize.

Directing the compiler

Auto-vectorization directive	
!DIR$ IVDEP	Instructs the compiler to ignore assumed vector dependencies.
!DIR$ VECTOR [ALWAYS]	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.
!DIR$ NOVECTOR	Specifies that the loop should never be vectorized.
User-mandated directive	
!DIR$ SIMD	Enforces vectorization of the loops.



Some more exotic directives



Auto-vectorization directive	
!DIR$ LOOP COUNT	The value of the loop count affects heuristices used in software pipelining, vectoriaztion, and loop-transformations.
!DIR$ VECTOR [TEMPORAL | NONTEMPORAL]	Specifies whether streaming stores should (non-temporal) or shouldn't be used (temporal).

```
!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 wont be using the LHS for a while so dont 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

Structure of Arrays (SoA)

a_1	a_2	a_3	b_1	b_2	b_3

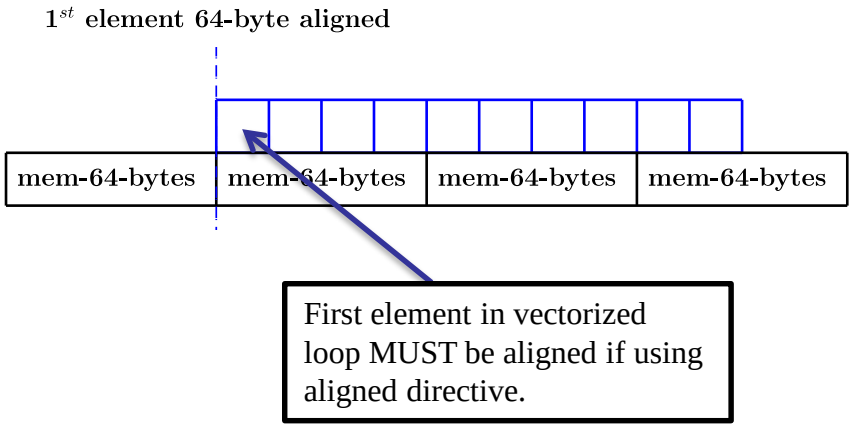
Better for data encapsulation

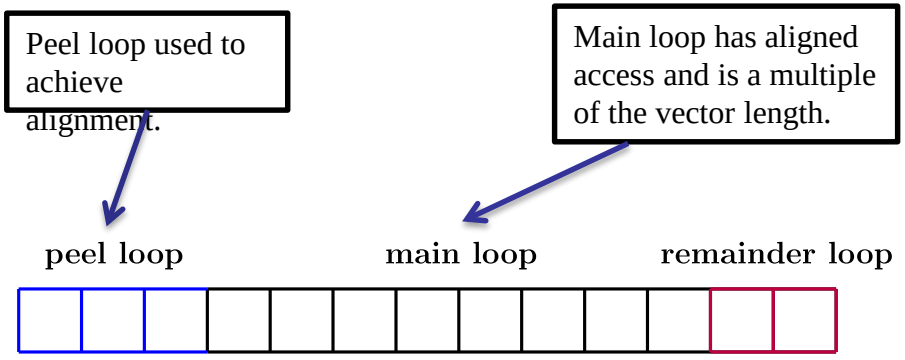
Array of Structures (AoS)

a_1	b_1	a_2	b_2	a_3	b_3

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.





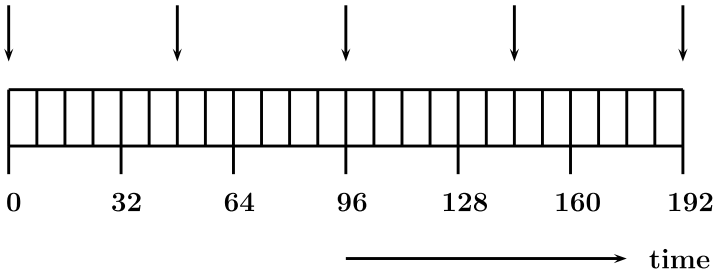
Large main loop ➔ alignment, peel, and remainder amortized.
Small main loop ➔ alignment and remainder important.

Alignment examples

```
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
enddo
```

✗

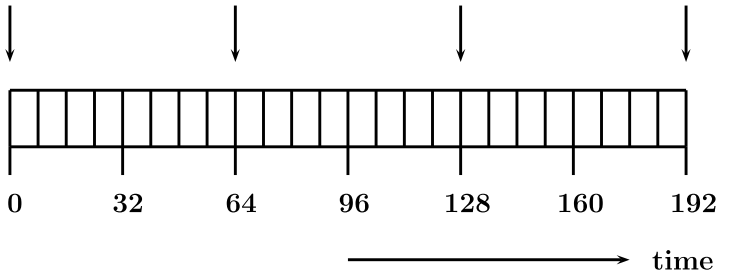
Not all iterations in “j” are aligned!




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

✓

Adding padding we can achieve alignment at the cost of computation and a larger memory footprint. Is it worth it?

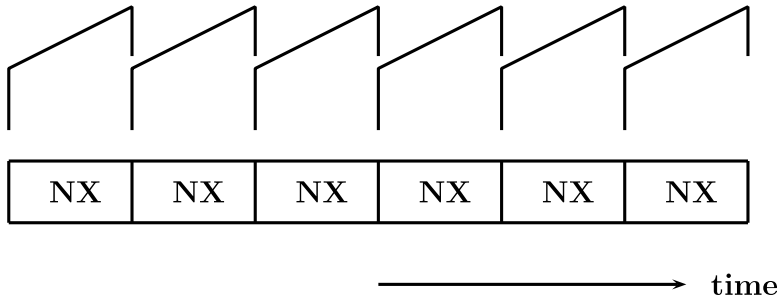


Alignment examples

Choosing domains that are a multiple of the boundary length can sometimes help with alignment e.g. in finite differencing.

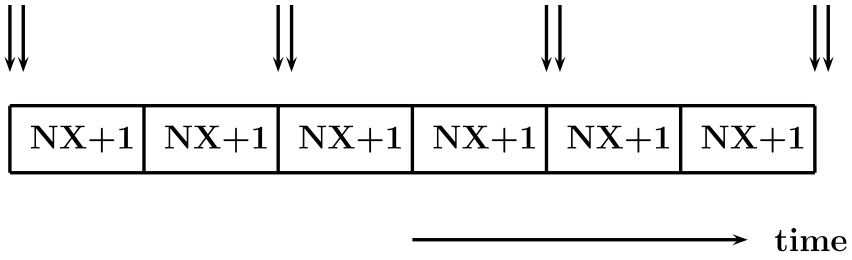
```
!caveat NX%4 = 0
real*8 :: a(NX,NY+1), b(NX,NY+1)
do j=1,NY
  do i=1,NX
    !DEC$ vector aligned !This is allowed
        b(i,j) = 0.5*(a(i,j) - a(i,j+1))
  enddo
enddo
```

✓




```
!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
```

✗



Vectorization analysis

Q: How do I know if loops are vectorized?
A: Use –vec-report[0-6].

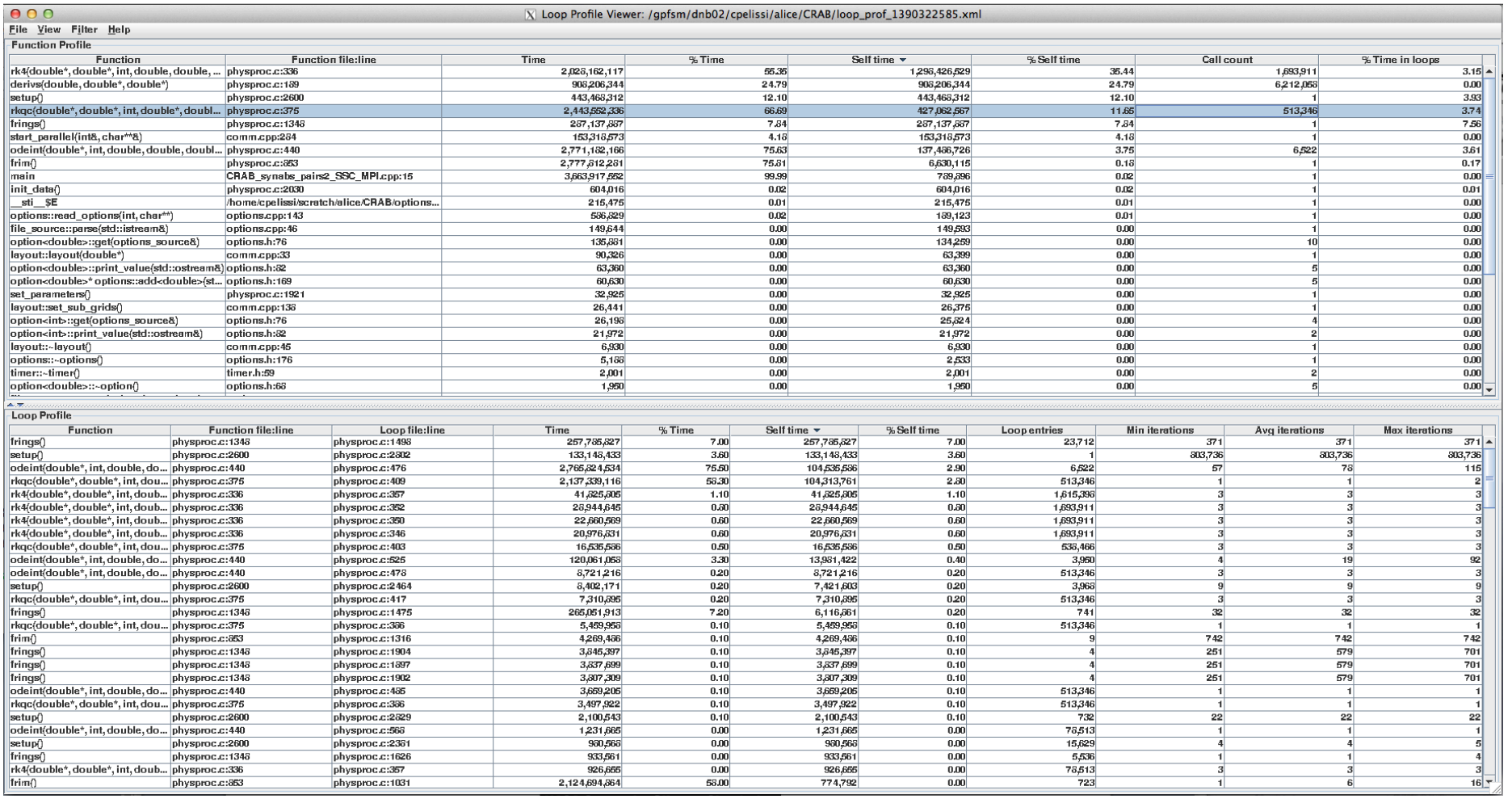
```
cpelissi@discover27:~/scratch/programs> icpc -vec-report3 -DSTRIDE=1 vector_stride_test.cpp
vector_stride_test.cpp(14): (col. 3) remark: loop was not vectorized: existence of vector dependence
vector_stride_test.cpp(17): (col. 12) remark: vector dependence: assumed OUTPUT dependence between  line 17 and  line 16
vector_stride_test.cpp(16): (col. 12) remark: vector dependence: assumed OUTPUT dependence between  line 16 and  line 17
vector_stride_test.cpp(16): (col. 12) remark: vector dependence: assumed OUTPUT dependence between  line 16 and  line 17
vector_stride_test.cpp(17): (col. 12) remark: vector dependence: assumed OUTPUT dependence between  line 17 and  line 16
vector_stride_test.cpp(29): (col. 5) remark: loop was not vectorized: #pragma novector used
vector_stride_test.cpp(22): (col. 3) remark: loop was not vectorized: not inner loop
```

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.

1.Compile with the proper flags: -profile-functions -profile-loops=all -profile-loops-report=2.
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: /gpfsm/dnb02/cpelissi/alice/CRAB/loop_prof_1390322585.xml

File View Filter Help

Function Profile

Function	Function file:line	Time	% Time	Self time	% Self time	Call count	% Time in loops
rk4(double*, double*, int, double, double, ...	physproc.c:336	2,028,162,117	55.35	1,296,426,529	35.44	1,693,911	3.15
derivs(double, double*, double*)	physproc.c:189	908,206,344	24.79	908,206,344	24.79	6,212,055	0.00
setup()	physproc.c:2600	443,468,312	12.10	443,468,312	12.10	1	3.93
rkqc(double*, double*, int, double*, doubl...	physproc.c:375	2,443,552,336	66.69	427,062,567	11.65	513,346	3.74
frings()	physproc.c:1348	287,137,887	7.84	287,137,887	7.84	1	7.56
start_parallel(int&, char**&)	comm.cpp:284	153,318,573	4.18	153,318,573	4.18	1	0.00
odeint(double*, int, double, double, doubl...	physproc.c:440	2,771,182,166	75.63	137,486,726	3.75	6,522	3.61
frim()	physproc.c:853	2,777,812,281	75.81	6,630,115	0.18	1	0.17
main	CRAB_synabs_pairs2_SSC_MPI.cpp:15	3,663,917,552	99.99	769,896	0.02	1	0.00
init_data()	physproc.c:2030	604,016	0.02	604,016	0.02	1	0.01
__sti__$E	/home/cpelissi/scratch/alice/CRAB/options...	215,475	0.01	215,475	0.01	1	0.00
options::read_options(int, char**)	options.cpp:143	586,829	0.02	189,123	0.01	1	0.00
file_source::parse(std::istream&)	options.cpp:46	149,644	0.00	149,593	0.00	1	0.00
option<double>::get(options_source&)	options.h:76	135,861	0.00	134,259	0.00	10	0.00
layout::layout(double*)	comm.cpp:33	90,326	0.00	63,399	0.00	1	0.00
option<double>::print_value(std::ostream&)	options.h:82	63,360	0.00	63,360	0.00	5	0.00
option<double>* options::add<double>(st...	options.h:169	60,630	0.00	60,630	0.00	5	0.00
set_parameters()	physproc.c:1921	32,925	0.00	32,925	0.00	1	0.00
layout::set_sub_grids()	comm.cpp:138	26,441	0.00	26,375	0.00	1	0.00
option<int>::get(options_source&)	options.h:76	26,193	0.00	25,824	0.00	4	0.00
option<int>::print_value(std::ostream&)	options.h:82	21,972	0.00	21,972	0.00	2	0.00
layout::~layout()	comm.cpp:45	6,930	0.00	6,930	0.00	1	0.00
options::~options()	options.h:176	5,166	0.00	2,533	0.00	1	0.00
timer::~timer()	timer.h:59	2,001	0.00	2,001	0.00	2	0.00
option<double>::~option()	options.h:66	1,950	0.00	1,950	0.00	5	0.00

Loop Profile

Function	Function file:line	Loop file:line	Time	% Time	Self time	% Self time	Loop entries	Min iterations	Avg iterations	Max iterations
frings()	physproc.c:1348	physproc.c:1498	257,785,827	7.00	257,785,827	7.00	23,712	371	371	371
setup()	physproc.c:2600	physproc.c:2802	133,146,433	3.60	133,146,433	3.60	1	803,736	803,736	803,736
odeint(double*, int, double, do...	physproc.c:440	physproc.c:476	2,765,824,534	75.50	104,535,536	2.90	6,522	57	76	115
rkqc(double*, double*, int, dou...	physproc.c:375	physproc.c:409	2,137,339,116	58.30	104,313,761	2.80	513,346	1	1	2
rk4(double*, double*, int, doub...	physproc.c:336	physproc.c:357	41,825,805	1.10	41,825,805	1.10	1,615,396	3	3	3
rk4(double*, double*, int, doub...	physproc.c:336	physproc.c:352	28,944,645	0.80	28,944,645	0.80	1,693,911	3	3	3
rk4(double*, double*, int, doub...	physproc.c:336	physproc.c:350	22,660,569	0.60	22,660,569	0.60	1,693,911	3	3	3
rk4(double*, double*, int, doub...	physproc.c:336	physproc.c:346	20,976,831	0.60	20,976,831	0.60	1,693,911	3	3	3
rkqc(double*, double*, int, dou...	physproc.c:375	physproc.c:403	16,535,536	0.50	16,535,536	0.50	538,466	3	3	3
odeint(double*, int, double, do...	physproc.c:440	physproc.c:525	120,061,053	3.30	13,931,422	0.40	3,950	4	19	92
odeint(double*, int, double, do...	physproc.c:440	physproc.c:478	8,721,216	0.20	8,721,216	0.20	513,346	3	3	3
setup()	physproc.c:2600	physproc.c:2464	8,402,171	0.20	7,421,603	0.20	3,966	9	9	9
rkqc(double*, double*, int, dou...	physproc.c:375	physproc.c:417	7,310,895	0.20	7,310,895	0.20	513,346	3	3	3
frings()	physproc.c:1348	physproc.c:1475	265,051,913	7.20	6,116,861	0.20	741	32	32	32
rkqc(double*, double*, int, dou...	physproc.c:375	physproc.c:386	5,459,953	0.10	5,459,953	0.10	513,346	1	1	1
frim()	physproc.c:853	physproc.c:1316	4,269,486	0.10	4,269,486	0.10	9	742	742	742
frings()	physproc.c:1348	physproc.c:1904	3,845,397	0.10	3,845,397	0.10	4	251	579	701
frings()	physproc.c:1348	physproc.c:1897	3,837,699	0.10	3,837,699	0.10	4	251	579	701
frings()	physproc.c:1348	physproc.c:1902	3,807,309	0.10	3,807,309	0.10	4	251	579	701
odeint(double*, int, double, do...	physproc.c:440	physproc.c:485	3,659,205	0.10	3,659,205	0.10	513,346	1	1	1
rkqc(double*, double*, int, dou...	physproc.c:375	physproc.c:396	3,497,922	0.10	3,497,922	0.10	513,346	1	1	1
setup()	physproc.c:2600	physproc.c:2829	2,100,543	0.10	2,100,543	0.10	732	22	22	22
odeint(double*, int, double, do...	physproc.c:440	physproc.c:566	1,231,665	0.00	1,231,665	0.00	78,513	1	1	1
setup()	physproc.c:2600	physproc.c:2361	930,563	0.00	930,563	0.00	15,629	4	4	5
frings()	physproc.c:1348	physproc.c:1626	933,561	0.00	933,561	0.00	5,536	1	1	4
rk4(double*, double*, int, doub...	physproc.c:336	physproc.c:357	926,655	0.00	926,655	0.00	78,513	3	3	3
frim()	physproc.c:853	physproc.c:1031	2,124,694,364	58.00	774,792	0.00	723	1	6	16

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?