

Optimizing for Intel Xeon Phi Knights Landing

Steven Warren



CRAY



swarren@cray.com

Legal Disclaimer



Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publicly announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA and YARCDATA. The following are trademarks of Cray Inc.: CHAPEL, CLUSTER CONNECT, CLUSTERSTOR, CRAYDOC, CRAYPAT, CRAYPORT, DATAWARP, ECOPHLEX, LIBSCI, NODEKARE, REVEAL. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used on this website are the property of their respective owners.

Outline



- Make KNL run my code faster!
 - Vectorization
 - Cache blocking
- `--exclusive`

Optimizing for Intel Xeon Phi Knights Landing



- Many KNL-specific optimizations involve MCDRAM in “Flat” mode
 - Since Cori uses “cache” mode, these optimizations generally do not apply
 - If application can strong scale efficiently, can use enough nodes such that the memory footprint/node is less than 16 GB and fit into MCDRAM
- KNL is an x86 processor, thus many of the things you would do for any x86 processor will apply
 - i.e., work done to improve KNL performance will generally improve performance on other modern processors as well

KNL strengths and weaknesses



- Strengths
 - MCDRAM memory bandwidth
 - Effectively a large L3 in “cache” mode (no dedicated L3 on KNL)
 - Larger L2 per core (1 MB / 2-core tile)
 - AVX512 vectors
 - Allows more operations per cycle than previous generations of processors
- Weaknesses
 - Clock GHz
 - Affects scalar operations
- Optimization strategy
 - Vectorize and/or cache block important kernels

But first, a note about affinity...

Process / Thread / Memory Affinity



- **Correct process, thread and memory affinity is the basis for getting optimal performance on KNL. It is also essential for guiding further performance optimizations.**
 - Process Affinity: bind MPI tasks to CPUs
 - Thread Affinity: bind threads to CPUs allocated to its MPI process
 - Memory Affinity: allocate memory from specific NUMA domains
- **Our goal is to promote OpenMP standard settings for portability. For example, OMP_PROC_BIND and OMP_PLACES are preferred to Intel specific KMP_AFFINITY and KMP_PLACE_THREADS settings.**

The following NERSC slides
stolen from Helen. Thanks Helen!

- XTHI is a **very** useful application that will tell you whether or not you are getting the expected placement behavior.
 - <https://github.com/olcf/XC30-Training/blob/master/affinity/Xthi.c>
- Different compilers and MPI stacks have different affinity rules
 - i.e., what works for Intel likely will not work for Cray or GNU
- Replace the call to your application binary to the xthi binary in your srun line to check affinity.
 - Can do this at any scale, but it's best to change the number of PEs to use a single node to avoid confusion of the output.

“numactl -H” displays NUMA info

68-core Quad Cache node:

NUMA Domain 0: all 68 cores (272 logic cores)

```
yunhe@cori01:> salloc -N 1 --qos=interactive -C knl,quad,cache -t 30:00  
salloc: Granted job allocation 5291739
```

```
yunhe@nid02305:> numactl -H
```

```
available: 1 nodes (0)
```

```
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43  
44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87  
88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122  
123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153  
154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184  
185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215  
216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246  
247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271
```

```
node 0 size: 96762 MB
```

```
node 0 free: 93067 MB
```

```
node distances:
```

```
node 0
```

```
0: 10
```

- The quad,cache mode has only 1 NUMA node with all CPUs on the NUMA node 0 (DDR memory)
- The MCDRAM is hidden from the numactl -H command (it is a cache).

Can We Just Do a Naïve Srun?

Example: 16 MPI tasks x 8 OpenMP threads per task on a single 68-core KNL quad,cache node:

```
% export OMP_NUM_THREADS=8
% export OMP_PROC_BIND=spread (other choice are "close","master","true","false")
% export OMP_PLACES=threads (other choices are: cores, sockets, and various
ways to specify explicit lists, etc.)
```

```
% srun -n 16 ./xthi |sort -k4n,6n
```

```
Hello from rank 0, thread 0, on nid02304. (core affinity = 0)
Hello from rank 0, thread 1, on nid02304. (core affinity = 144) (on physical core 8)
Hello from rank 0, thread 2, on nid02304. (core affinity = 17)
Hello from rank 0, thread 3, on nid02304. (core affinity = 161) (on physical core 25)
Hello from rank 0, thread 4, on nid02304. (core affinity = 34)
Hello from rank 0, thread 5, on nid02304. (core affinity = 178) (on physical core 42)
Hello from rank 0, thread 6, on nid02304. (core affinity = 51)
Hello from rank 0, thread 7, on nid02304. (core affinity = 195) (on physical core 59)
Hello from rank 1, thread 0, on nid02304. (core affinity = 0)
Hello from rank 1, thread 1, on nid02304. (core affinity = 144)
```

It is a mess!



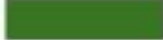
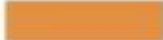



Importance of `-c` and `--cpu_bind` Options



- **The reason: 68 is not divisible by #MPI tasks!**
 - Each MPI task is getting $68 \times 4 / \text{\#MPI tasks}$ of logical cores as the domain size
 - MPI tasks are crossing tile boundaries
- **Set number of logical cores per MPI task (`-c`) manually by wasting extra 4 cores on purpose: `256/\text{\#MPI_tasks_per_node}`.**
 - Meaning to use 64 cores only on the 68-core KNL node, and spread the logical cores allocated to each MPI task evenly among these 64 cores.
 - Now it looks good!
 - `% srun -n 16 -c 16 --cpu_bind=cores ./xthi`
 - Hello from rank 0, thread 0, on nid09244. (core affinity = 0)
 - Hello from rank 0, thread 1, on nid09244. (core affinity = 136)
 - Hello from rank 0, thread 2, on nid09244. (core affinity = 1)
 - Hello from rank 0, thread 3, on nid09244. (core affinity = 137)

Now It Looks Good!

Process/thread affinity are good! (Marked first 6 and last MPI tasks only)

	MPI rank 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
		68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85
	MPI rank 1	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153
		204																220	221
	MPI rank 2	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
		86																102	103
	MPI rank 3	154	155	156	157	158	159	160	161									170	171
		222																238	239
	MPI rank 4	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51		
		104																	
	MPI rank 5	172																	
....		240																	
	MPI rank 15	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67		
		120																	
		188								196	197	198	199						
		256																	

And so on for other MPI tasks and threads

Essential Runtime Settings for Process/Thread Affinity

- **Use `srun -c` and `--cpu_bind` flags to bind tasks to CPUs**
 - `-c <n>` (or `--cpus-per-task=n`) allocates (reserves) **n** CPUs per task (process). It helps to evenly spread MPI tasks, can use up to **n** OpenMP threads per MPI task.
 - Use `--cpu_bind=cores` (no hyperthreads) or `--cpu_bind=threads` (if hyperthreads are used)
- **Use OpenMP envs: `OMP_PROC_BIND`, `OMP_PLACES` to fine pin each thread to a subset of CPUs allocated to the host task**
 - Different compilers may have different implementations
 - The following provide compatible thread affinity among Intel, GNU and Cray compilers:
`OMP_PROC_BIND=true` # Specifying threads may not be moved between CPUs
`OMP_PLACES=threads` # Specifying a thread should be placed on a single CPU
- **Verify with XTHI before running your code!**

But second... Dynamic vs Static linking on KNL

Dynamic vs Static Linking for Qbox on KNL



- Lines 248 – 249 in qb.C require glibc, which is a collection of dynamic libraries in many current operating systems
 - if (getlogin() != 0)
 cout << "<user> " << getlogin() << " </user>" << endl;
- Performance can be greatly increased on KNL for statically linked executables.

Options to link statically

- Can statically link in Cray Libsci libraries (executable remains dynamic) to alleviate some of the performance loss by setting:
 - `LIBS = -Wl,-Bstatic -lsci_cray_mpi_mp -lsci_cray_mpi \`
`-lfftw3f_mpi -lfftw3f_omp -lfftw3f -lfftw3_mpi \`
`-lfftw3_omp -lfftw3 -Wl,-Bdynamic`
- Or compile fully static but add extra compile flags to qb.C:
 - `'-Dmain=stealthy(){return 0;} char* stealth(){return getenv("USER");} int main' -`
`Dgetlogin=stealth`
- Or one could simply modify the code in qb.C to use `getenv()` instead of `getlogin()` and compile fully static.

Dynamic vs Static Linking for Qbox on KNL

- For a 256 node, 880 atom Qbox run using 32 MPI ranks/node and 2 OpenMP threads/rank with nrowmax set to 256 yields the following results:

Link type	Dynamic Linking	Static Linking	Dynamic Linking with Statically Linked Cray Libsci libraries
max time (run time)	330 s	198 s	215 s

Example Analysis and Optimizations:

Vectorization



What is vectorization?

- Vectorization is the practice of converting an algorithm to work on a set of values simultaneously instead of a single value one-by-one.

What prevents vectorization?

- Complexity in loops which the compiler can not interpret
 - Indirect memory accesses
 - Logical statements
 - Recurrences on variables

How To Know If Your Loops Are Vectorizing

- CCE can provide “listing” files with compilation which will give an easily interpreted and detailed description of every line in your source
 - **-hlist=a**
- Intel and GNU compiler provide similar capabilities.
- Use the listing file to determine if your changes allow the compiler to apply better optimizations
 - You do NOT need to execute the code to check if the compiler applies optimizations

%%%		L o o p m a r k	L e g e n d	%%%
		Primary Loop Type	Modifiers	
		----- ---- ----	-----	
		A - Pattern matched	a - atomic memory operation	
			b - blocked	
		C - Collapsed	c - conditional and/or computed	
		D - Deleted		
		E - Cloned		
		F - Flat - No calls	f - fused	
		G - Accelerated	g - partitioned	
		I - Inlined	i - interchanged	
		M - Multithreaded	m - partitioned	
			n - non-blocking remote transfer	
			p - partial	
		R - Rerolling	r - unrolled	
			s - shortloop	
		V - Vectorized	w - unwound	
		+ - More messages listed at end of listing		

Example Loop



```
67.      1 2          PF = 0.0
68.    + 1 2 3--<    DO 44030 I = 2, N
69.      1 2 3        AV  = B(I) * RV
70.      1 2 3        PB  = PF
71.      1 2 3        PF  = C(I)
72.      1 2 3        IF ((D(I) + D(I+1)) .LT. 0.) PF = -C(I+1)
73.      1 2 3        AA  = E(I) - E(I-1) + F(I) - F(I-1)
74.      1 2 3        1    + G(I) + G(I-1) - H(I) - H(I-1)
75.      1 2 3        BB  = R(I) + S(I-1) + T(I) + T(I-1)
76.      1 2 3        1    - U(I) - U(I-1) + V(I) + V(I-1)
77.      1 2 3        2    - W(I) + W(I-1) - X(I) + X(I-1)
78.      1 2 3        A(I) = AV * (AA + BB + PF - PB + Y(I) - Z(I)) + A(I)
79.      1 2 3--> 44030 CONTINUE
```

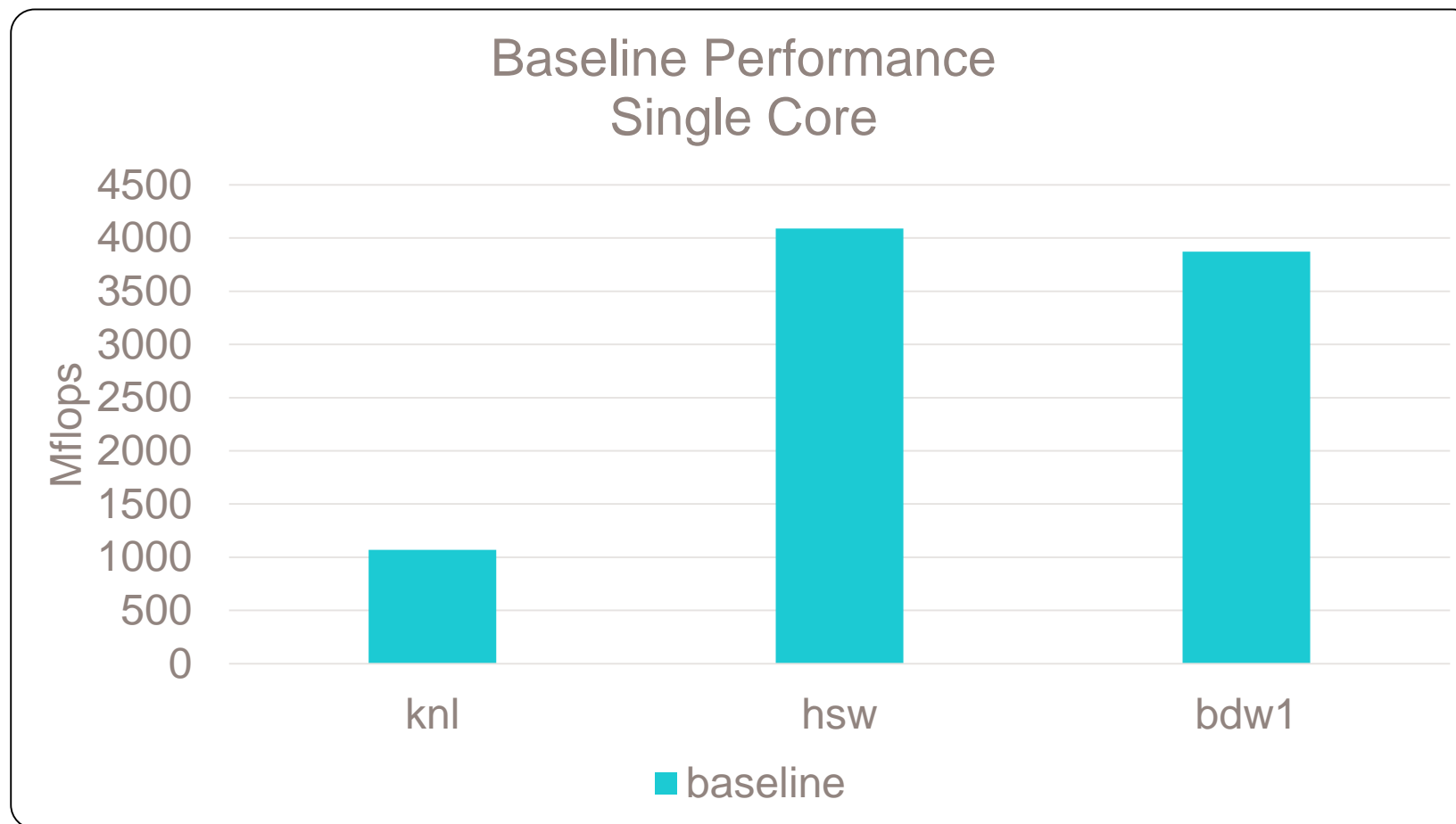
ftn-6254 ftn: VECTOR LP44030, File = lp44030.f, Line = 68

A loop starting at line 68 was not vectorized because a recurrence was found on "pf" at line 71.

- There is a recurrence on the scalar 'PF'
- Use the 'explain' tool to learn more about what a recurrence is

```
> explain ftn-6254
```

Example



What's Preventing Vectorization?

- Let's do a vector dependency analysis assuming VL=2

68.	+ 1 2 3--<	DO 44030 I = 2, N	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> Compiler would promote scalars to vectors </div>
...			
70.	1 2 3	PB = PF	
71.	1 2 3	PF = C(I)	
72.	1 2 3	IF ((D(I) + D(I+1)) .LT. 0.) PF = -C(I+1)	
...			
78.	1 2 3	A(I) = AV * (AA + BB + PF - PB + Y(I) - Z(I)) + A(I)	

Compiler will not promote
PF to a 3 element vector

$$PB \begin{pmatrix} 2 \\ 3 \end{pmatrix} \propto PF \begin{pmatrix} 1 \\ 2 \end{pmatrix} \qquad PF \begin{pmatrix} 2 \\ 3 \end{pmatrix} \propto C \begin{pmatrix} 2 \\ 3 \\ 4 \end{pmatrix} \qquad A \begin{pmatrix} 2 \\ 3 \end{pmatrix} \propto PB, PF \rightarrow PF \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

- Vectorization may be possible with modification, but loop is not concurrent safe

What can we do to vectorize this loop?



- Convert PF from a scalar to a vector (1-D array)
 - Warning! Be cognizant of how changing this variable may affect other regions of the code
 - Is PF a global or local variable? Is the final result of PF used elsewhere?
 - May need to use a temporary variable array for the loop and store back into PF if needed
- Eliminates the need for the PB scalar variable in the loop

Optimization changes



- What optimizations did the compiler apply to our new version?

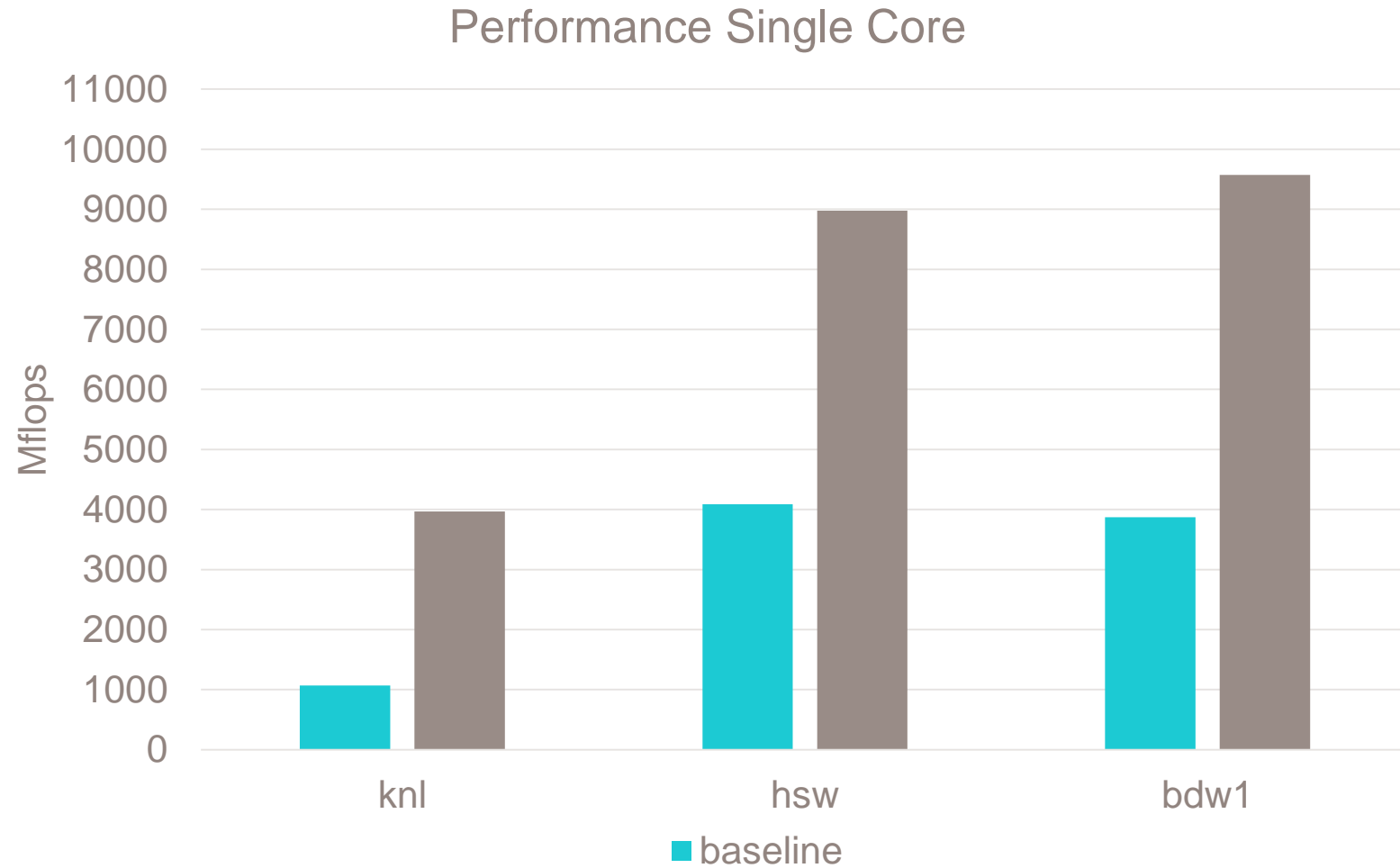
```
66.      1 2                VPF(1) = 0.0
67.      1 2 Vr2--<        DO 44031 I = 2, N
68.      1 2 Vr2            AV      = B(I) * RV
69.      1 2 Vr2            VPF(I) = C(I)
70.      1 2 Vr2            IF ((D(I) + D(I+1)) .LT. 0.) VPF(I) = -C(I+1)
71.      1 2 Vr2            AA      = E(I) - E(I-1) + F(I) - F(I-1)
72.      1 2 Vr2            1      + G(I) + G(I-1) - H(I) - H(I-1)
73.      1 2 Vr2            BB      = R(I) + S(I-1) + T(I) + T(I-1)
74.      1 2 Vr2            1      - U(I) - U(I-1) + V(I) + V(I-1)
75.      1 2 Vr2            2      - W(I) + W(I-1) - X(I) + X(I-1)
76.      1 2 Vr2            A(I) = AV * (AA + BB + VPF(I) - VPF(I-1) + Y(I) - Z(I)) +
A(I)
77.      1 2 Vr2--> 44031 CONTINUE
```

```
ftn-6005 ftn: SCALAR LP44030, File = lp44030.f, Line = 67
A loop starting at line 67 was unrolled 2 times.
```

```
ftn-6204 ftn: VECTOR LP44030, File = lp44030.f, Line = 67
A loop starting at line 67 was vectorized.
```

- How does the performance of this version compare with the original?

Original vs Vectorized performance



Example Analysis and Optimizations:

Cache blocking

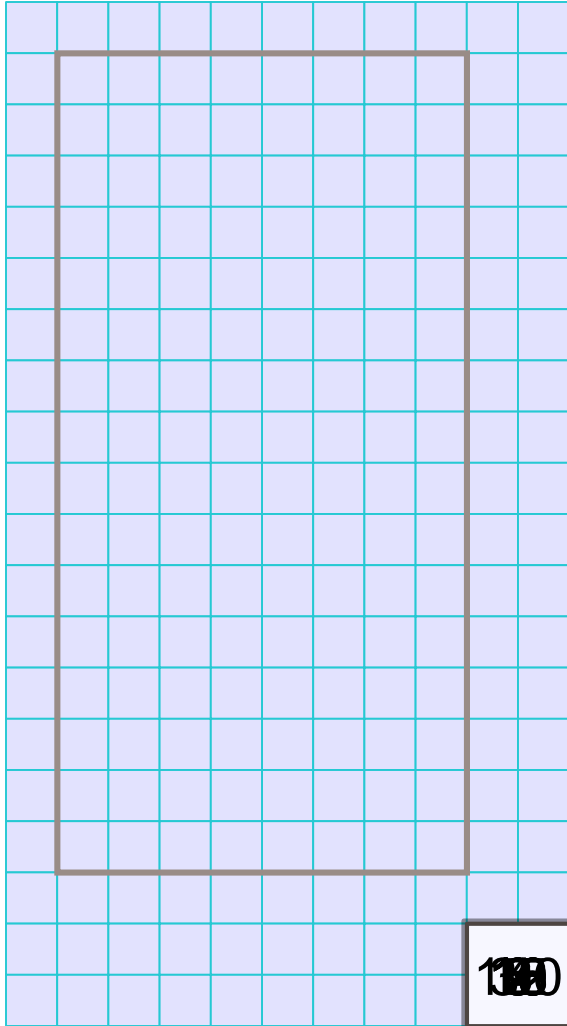


Data Reuse will be important



- Data reuse will be critical to performance
- Reuse out of MCDRAM will reduce requirements on main memory
- Reuse out of lower levels of cache will lower requirements on MCDRAM
- In order to know how to cache block properly we need to know the trip counts of loops and the sizes of various arrays as accurately as possible

A SIMPLE EXAMPLE



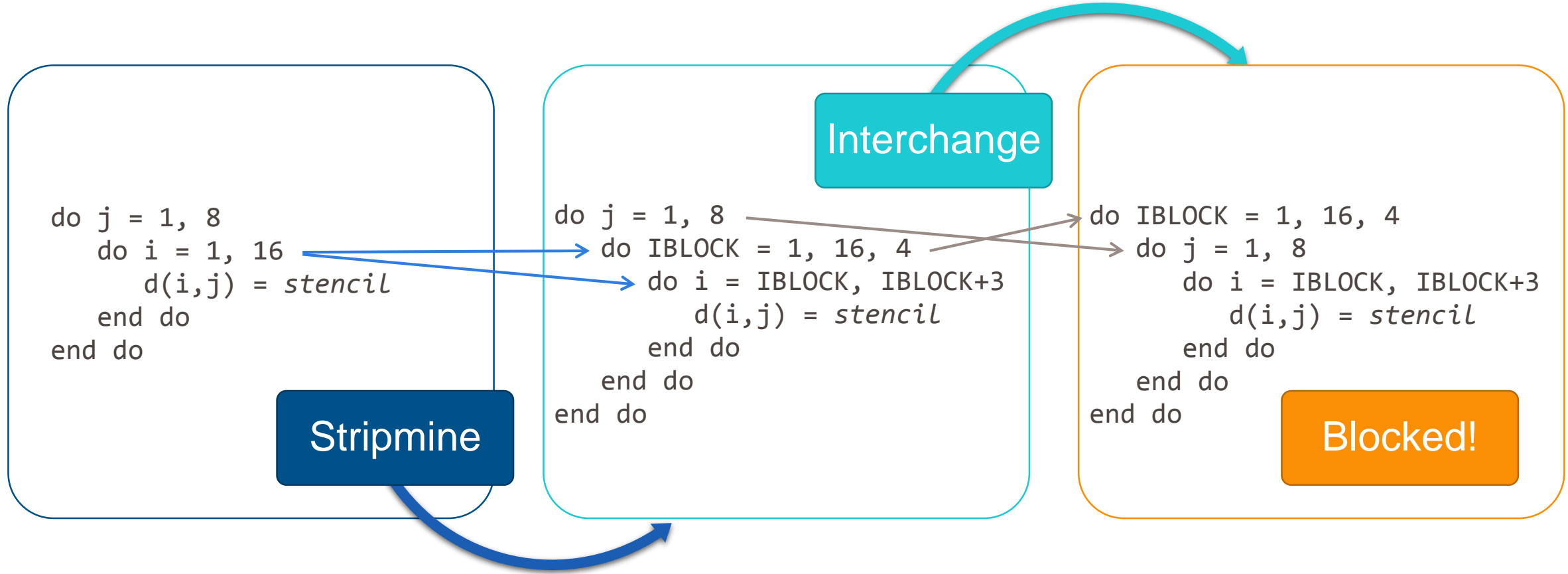
- 2D 5-point Laplacian

```
do j = 1, 8
  do i = 1, 16
    d(i,j) = u(i-1,j) + u(i+1,j) &
              - 4*u(i,j)          &
              + u(i,j-1) + u(i,j+1)

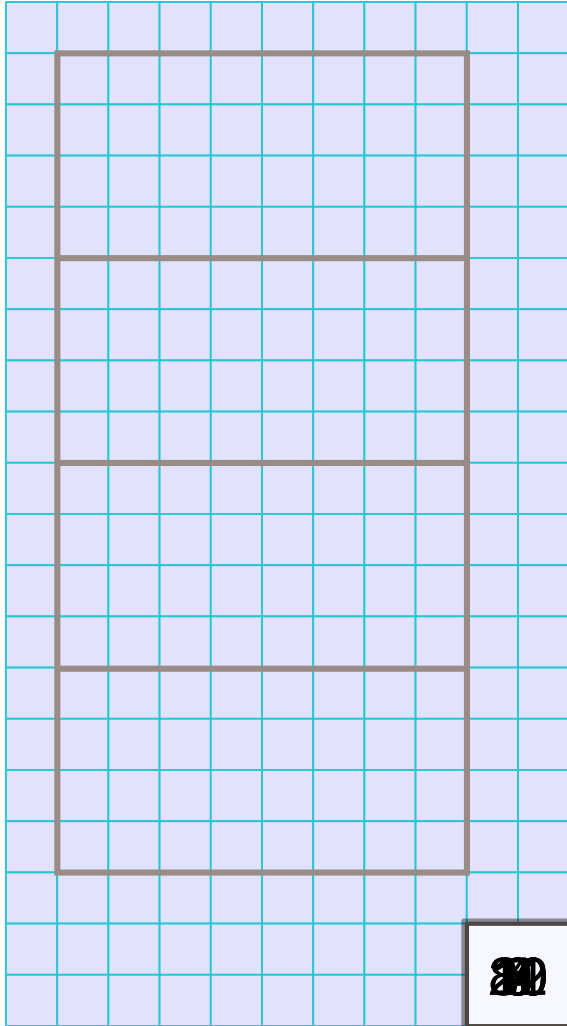
  end do
end do
```

- Simple cache structure for this example:
 - Assume each cache line holds 4 array elements
 - And cache can hold 12 lines of u data
- No cache reuse between outer loop iterations

BLOCKING = STRIPMINE + INTERCHANGE



BLOCKING TO INCREASE REUSE

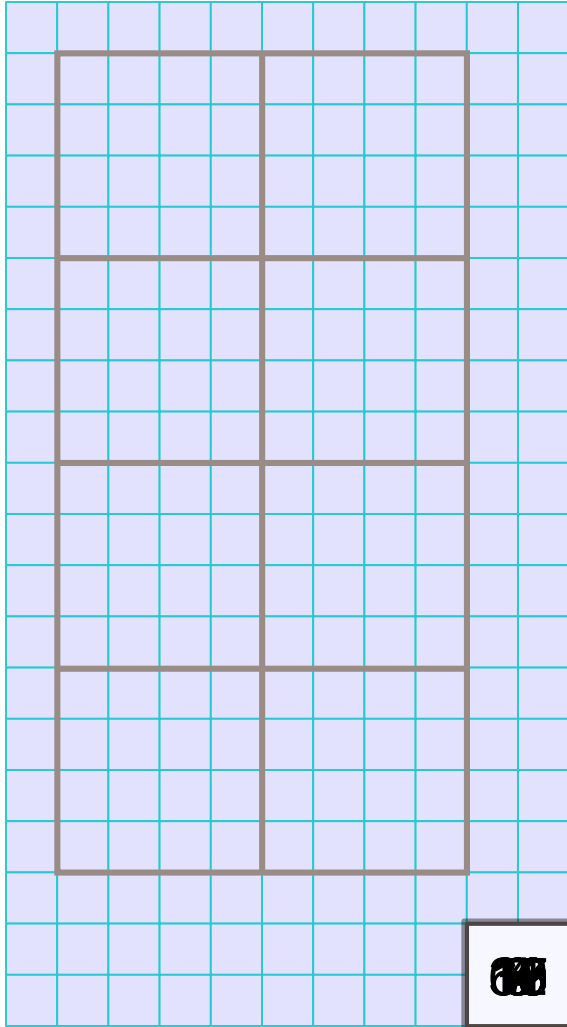


- Block the inner loop

```
do IBLOCK = 1, 16, 4
  do j = 1, 8
    do i = IBLOCK, IBLOCK + 3
      d(i,j) = u(i-1,j) + u(i+1,j) &
               - 4*u(i,j)          &
               + u(i,j-1) + u(i,j+1)
    end do
  end do
end do
```

- Now we have reuse of the j+1 data

EVEN BETTER!



- Iterate over 4×4 blocks for better spatial locality

```
do JBLOCK = 1, 8, 4
  do IBLOCK = 1, 16, 4
    do j = JBLOCK, JBLOCK + 3
      do i = IBLOCK, IBLOCK + 3
        d(i,j) = u(i-1,j) + u(i+1,j) &
                  - 4*u(i,j)          &
                  + u(i,j-1) + u(i,j+1)
      end do
    end do
  end do
end do
```

- CCE has directives for this
 - !dir\$ blockable(i,j)
 - !dir\$ blockingsize(4)

Example Analysis and Optimization:

miniGhost



Example app: miniGhost

- “mini-app” from the NERSC8 procurement.
- 27-point 3-D stencil application
- Simulates diffusion
- Like most stencil codes, it is main memory bandwidth bound
 - Data reuse will lessen contention for memory accesses

Main compute loop

- Craypat suggests the following loop is about ~50% of the run time

```
288. + b-----<      DO K = 1, NZ
289. + b b-----<      DO J = 1, NY
290.   b b Vb-----<      DO I = 1, NX
291.   b b Vb
292.   b b Vb          SLICE_BACK = GRID(I-1,J-1,K-1) + GRID(I-1,J,K-1) + GRID(I-1,J+1,K-1) + &
293.   b b Vb          GRID(I  ,J-1,K-1) + GRID(I  ,J,K-1) + GRID(I  ,J+1,K-1) + &
294.   b b Vb          GRID(I+1,J-1,K-1) + GRID(I+1,J,K-1) + GRID(I+1,J+1,K-1)
295.   b b Vb
296.   b b Vb          SLICE_MINE = GRID(I-1,J-1,K)   + GRID(I-1,J,K)   + GRID(I-1,J+1,K) + &
297.   b b Vb          GRID(I  ,J-1,K)   + GRID(I  ,J,K)   + GRID(I  ,J+1,K) + &
298.   b b Vb          GRID(I+1,J-1,K)   + GRID(I+1,J,K)   + GRID(I+1,J+1,K)
299.   b b Vb
300.   b b Vb          SLICE_FRONT = GRID(I-1,J-1,K+1) + GRID(I-1,J,K+1) + GRID(I-1,J+1,K+1) + &
301.   b b Vb          GRID(I  ,J-1,K+1) + GRID(I  ,J,K+1) + GRID(I  ,J+1,K+1) + &
302.   b b Vb          GRID(I+1,J-1,K+1) + GRID(I+1,J,K+1) + GRID(I+1,J+1,K+1)
303.   b b Vb
304.   b b Vb          WORK(I,J,K) = ( SLICE_BACK + SLICE_MINE + SLICE_FRONT ) / 27.0
305.   b b Vb
306.   b b Vb----->      END DO
307.   b b----->      END DO
308.   b----->      END DO
```

- CCE does vectorize and also attempts to cache block the inner loop, but can we do better?

Listing file explanations

- CCE may attempt to cache block for L2 based upon the targeted architecture.
- Generally, L1 is too small and L3 is too “slow”

```
ftn-6294 ftn: VECTOR MG_STENCIL_3D27PT, File = MG_STENCIL_COMPS.F, Line = 287
  A loop starting at line 287 was not vectorized because a better candidate was found at line 289.
```

```
ftn-6049 ftn: SCALAR MG_STENCIL_3D27PT, File = MG_STENCIL_COMPS.F, Line = 287
  A loop starting at line 287 was blocked with block size 8.
```

```
ftn-6294 ftn: VECTOR MG_STENCIL_3D27PT, File = MG_STENCIL_COMPS.F, Line = 288
  A loop starting at line 288 was not vectorized because a better candidate was found at line 289.
```

```
ftn-6049 ftn: SCALAR MG_STENCIL_3D27PT, File = MG_STENCIL_COMPS.F, Line = 288
  A loop starting at line 288 was blocked with block size 8.
```

```
ftn-6049 ftn: SCALAR MG_STENCIL_3D27PT, File = MG_STENCIL_COMPS.F, Line = 289
  A loop starting at line 289 was blocked with block size 256.
```

```
ftn-6204 ftn: VECTOR MG_STENCIL_3D27PT, File = MG_STENCIL_COMPS.F, Line = 289
  A loop starting at line 289 was vectorized.
```

Blocking = Stripmine + Interchange

```

287.  + 1-----<      DO KK = 1, NZ, block_k
288.  + 1 2-----<      DO JJ = 1, NY, block_j
289.  + 1 2 3-----<      DO II = 1, NX, block_i
290.  + 1 2 3 4-----<      DO K = KK, KK+(block_k-1)
291.  + 1 2 3 4 5-----<      DO J = JJ, JJ+(block_j-1)
292.    1 2 3 4 5 V--<      DO I = II, II+(block_i-1)
293.    1 2 3 4 5 V
294.    1 2 3 4 5 V      SLICE_BACK =  GRID(I-1,J-1,K-1) + GRID(I-1,J,K-1) + GRID(I-1,J+1,K-1) + &
295.    1 2 3 4 5 V      GRID(I  ,J-1,K-1) + GRID(I  ,J,K-1) + GRID(I  ,J+1,K-1) + &
296.    1 2 3 4 5 V      GRID(I+1,J-1,K-1) + GRID(I+1,J,K-1) + GRID(I+1,J+1,K-1)
297.    1 2 3 4 5 V
298.    1 2 3 4 5 V      SLICE_MINE =  GRID(I-1,J-1,K)   + GRID(I-1,J,K)   + GRID(I-1,J+1,K) + &
299.    1 2 3 4 5 V      GRID(I  ,J-1,K)   + GRID(I  ,J,K)   + GRID(I  ,J+1,K) + &
300.    1 2 3 4 5 V      GRID(I+1,J-1,K)   + GRID(I+1,J,K)   + GRID(I+1,J+1,K)
301.    1 2 3 4 5 V
302.    1 2 3 4 5 V      SLICE_FRONT = GRID(I-1,J-1,K+1) + GRID(I-1,J,K+1) + GRID(I-1,J+1,K+1) + &
303.    1 2 3 4 5 V      GRID(I  ,J-1,K+1) + GRID(I  ,J,K+1) + GRID(I  ,J+1,K+1) + &
304.    1 2 3 4 5 V      GRID(I+1,J-1,K+1) + GRID(I+1,J,K+1) + GRID(I+1,J+1,K+1)
305.    1 2 3 4 5 V
306.    1 2 3 4 5 V      WORK(I,J,K) = ( SLICE_BACK + SLICE_MINE + SLICE_FRONT ) / 27.0
307.    1 2 3 4 5 V
308.    1 2 3 4 5 V-->      END DO
309.    1 2 3 4 5---->      END DO
310.    1 2 3 4----->      END DO
311.    1 2 3----->      END DO
312.    1 2----->      END DO
313.    1----->      END DO

```

Listing file explanations

ftn-6306 ftn: VECTOR MG_STENCIL_3D27PT, File = MG_STENCIL_COMPS.F, Line = 287

A loop starting at line 287 was not vectorized because the iteration space is too irregular.

ftn-6306 ftn: VECTOR MG_STENCIL_3D27PT, File = MG_STENCIL_COMPS.F, Line = 288

A loop starting at line 288 was not vectorized because the iteration space is too irregular.

ftn-6303 ftn: VECTOR MG_STENCIL_3D27PT, File = MG_STENCIL_COMPS.F, Line = 289

A loop starting at line 289 was not vectorized because an inter-loop dependence relation is too complicated.

ftn-6303 ftn: VECTOR MG_STENCIL_3D27PT, File = MG_STENCIL_COMPS.F, Line = 290

A loop starting at line 290 was not vectorized because an inter-loop dependence relation is too complicated.

ftn-6303 ftn: VECTOR MG_STENCIL_3D27PT, File = MG_STENCIL_COMPS.F, Line = 291

A loop starting at line 291 was not vectorized because an inter-loop dependence relation is too complicated.

ftn-6204 ftn: VECTOR MG_STENCIL_3D27PT, File = MG_STENCIL_COMPS.F, Line = 292

A loop starting at line 292 was vectorized.

How to set the correct block sizes

- Typically, you want a larger amount of the inner iteration with smaller amounts in the other loops
 - Depends on the loop characterization and what data should be / could be / need to be reused
 - Powers of 2 generally are best if full index can not be held in cache
- Depending on the particular problem size, a proper cache blocking can provide a 50% speed-up for this particular loop on KNL
 - May see smaller impact on earlier Xeon processors since L2 misses are supported by an L3 cache.

Summary

- Code Characterization is an important first step in preparing for KNL
 - Target Science
 - Target Scaling
 - Hotspot identification
- Process affinity is critical for run performance
- Statically linked binaries likely to perform better than dynamically linked binaries.
- KNL node is different from XEON node
 - Single node optimizations will be an early focus
 - A properly designed kernel will help with optimization efforts
 - Vectorization is important and will become even more so with future processors
- Data reuse is important, but how important will depend on memory footprints and access patterns

THANK YOU

QUESTIONS?



swarren@cray.com

cray.com



@cray_inc



linkedin.com/company/cray-inc-/

