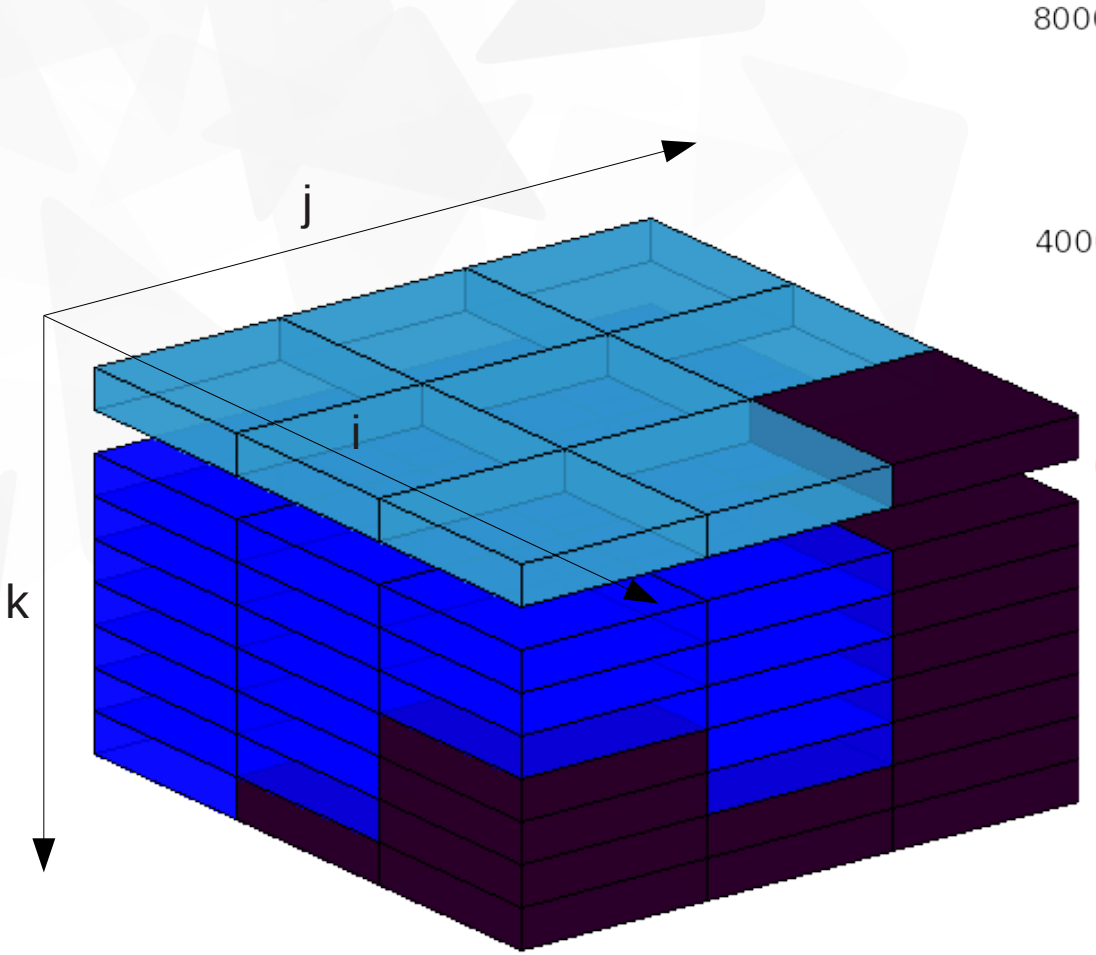
Refactoring for Xeon Phi

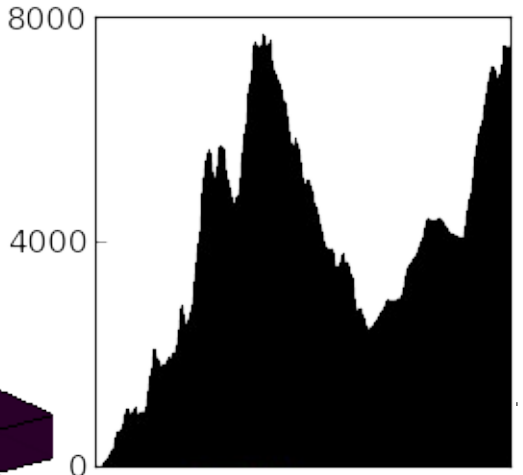
Jacob Weismann Poulsen, DMI

Outline

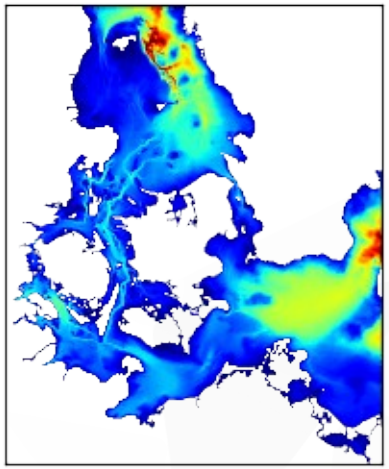
- NWP versus ocean modeling
- Node performance
  - Thread parallelization
  - SIMD vectorization
- Performance results

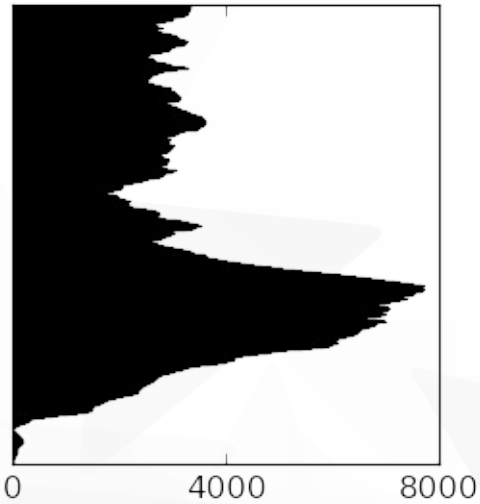
The data is sparse (fractal structures)



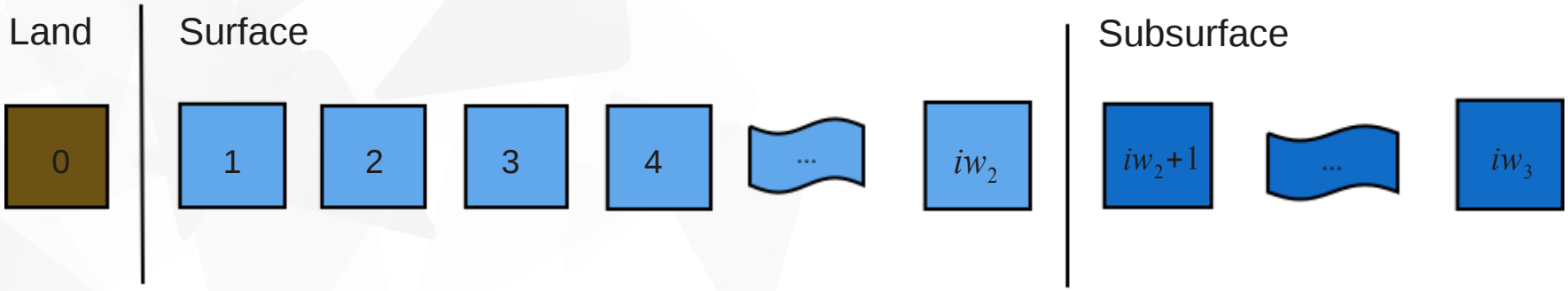


11.1%





Data layout (serial, indirect addressing)



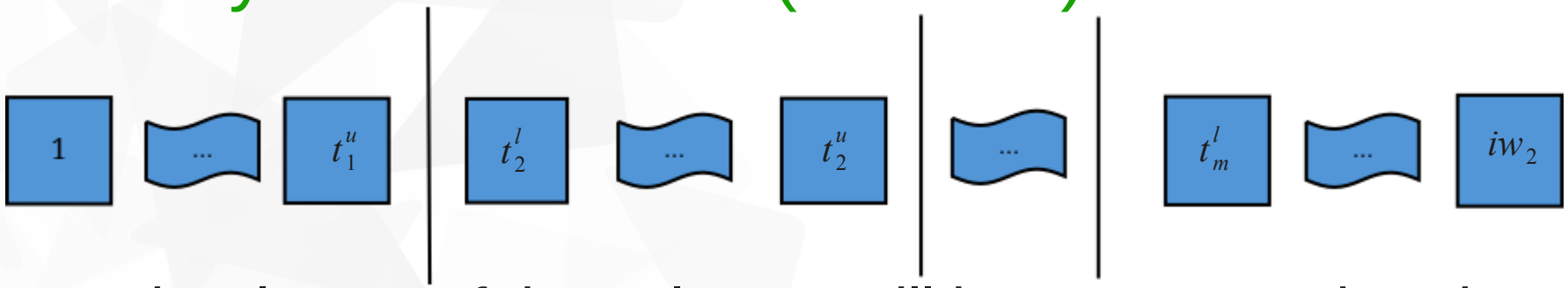

```
do iw = 1,iw2
  i = ind(1,iw)
  j = ind(2,iw)
  ! all surface wet-points (i,j) reached with stride-1
      ... u(iw) …
enddo
do iw = 1,iw2
  kb = kh(iw)
  if (kb < 2) cycle
  i = ind(1,iw)
  j = ind(2,iw)
  mi0 = mmk(2,i,j)-2
  do k = 2, kb
    ! all subsurface wet-points (k,i,j) are reached with stride-1
    mi = mi0 + k
    ... u(mi) ...
  enddo
enddo
```

Data layout (serial)

- Data layout revisited:
  - Horizontally (unstructured) columnar meshes
  - Indirect addressing in the horizontal:

    mmk(1:kmax,:,:) ⇒ mmk(1:2,:,:) ⇒ mmk1(:,:) ; mmk2(:,:)
  - Direct addressing in the vertical
  - (GungHo paper 2013 – similar conclusion for NWP)
- Observation
  - Any enumeration of the surface points and any enumeration of the subsurface points imposes a unique cache pattern (D1, L2, TLB) and some are obviously better than others. Finding the infimum is NP-hard but SFCs could lead to reasonable heuristics. A true challenge to formulate a well-posed problem, though.

Data layout for threads (or tasks)



- Another layout of the columns will impose a another threaded layout of data.

```
...
!$OMP PARALLEL DEFAULT(SHARED)
call foo( ... );call bar(...); ...
!$OMP BARRIER
call halo_update(...)
!$OMP BARRIER
call baz( ... );call quux(...); ...
!$OMP END PARALLEL
...
subroutine foo(...)
  ...
  call domp_get_domain(kh, 1, iw2, nl, nu, idx)
  do iw=nl,nu
    i = ind(1,iw)
    j = ind(2,iw)
    ! all threadlocal wet-points (:,:,:) are reached here
    ...
  enddo
end subroutine foo
```

Thread (and task) load balancing

- Formal definition:

Let $I = \{1, \ldots, m\}$ be the column index set and let $\{w_1, \ldots, w_m\}$ be the weights associated with the individual columns. Let n denote the number of threads/tasks. A disjoint subinterval $I_i = \{[l_i; u_i]\}_{i=1,\ldots,n}$ covering of I induces a cost vector $(c_1, \ldots, c_n)$ with $c_i = \sum_{j=l_i}^{u_i} w_j$. The cost c of the covering is defined as $\max_i c_i$. The balance problem is to find a covering that minimizes c.

- Observation: The NP-hard problem is reduced to the integer partition problem with exact solution with time complexity: $O(m^2n)$.
- Heuristics: Greedy approach or alternating greedy approach with runtime complexity: O(n).
- The weights can be a sum of sub weights while retaining problem complexity!

Thread parallelism

- Must be SPMD based (like MPI) and *not* loop based to minimize synchronization, barriers surrounding MPI halo swaps only.
- On NUMA architectures proper NUMA-layout for all variables is important.
- Consistent loop structures and consistent data layout and usage throughout the whole code.
- Proper balancing is very important at scale (Amdahl). It can be done either offline (exact) or online (heuristic).
- Tuning options for balancing: Linear regression based on profiles.

Thread parallelism (load balancing strategies)

- Greedy heuristic versus exact solution:
  - 48 threads:
    - Largest relative difference 3D: 0.126%
    - Largest relative difference 2D: 0.09%
  - 240 threads:
    - Largest relative difference 3D: 1.39%
    - Largest relative difference 2D: 2.39%
- Conclusion: The heuristic works fine at these thread counts (may be testcase dependent, though)

Refactoring for SIMD

Actually not as simple as it may sound....

SIMD target loops

- All loops are structured like this:

```
do iw=       ! horizontal - mpi/openmp parallelization
  do k=      ! vertical   - vectorization
    do ic=   ! innermost loop (in advection) with number of tracers
      …
    enddo
  enddo
enddo
```

- Could vectorize at the iw-level but hardware is not ready.Thus, the aim is to vectorize all the k-loops
- Trivial obstacles to vectorization
  - Indirections
  - Assumed-shape (F2008 contiguous attribute)
  - Branches (min/max/sign)

SIMD target loops

- Obstacles to proper SIMD code generation:
  - Design choice – should we do the columns one-by-one using work arrays or should we do the subset that we can do to together and manually deal with the emerging remainder loops
- Refactor strategy using computational intensity (CI) and D1 pressure as the guide lines. High CI is good
- … but not too high; use blocking to reduce pressure on D1, L2,...
- Back-of-the-envelope estimates of how many tripcounts it takes to flush D1 and L2.

Premature abstraction is the root of all evil

(a hands on experience)

Premature abstraction is the root of all evil

- This topic may not coincide with your expectations:
  - I will not talk about how one can loose a leg with OOD (google it, e.g. Mike Acton).
  - I will not talk about how one looses performance by using the HW abstraction that cores within a node has distributed memory (do the math on a piece of paper).
  - …
- Instead I will describe how the most simple HW abstraction (a 2D-array) will result in more than 2x performance loss on Xeon Phi and this should serve as a warning against using even the most simple abstractions without a prior analysis of consequences.

Premature abstraction is the root of all evil

- The design idea was to hold all tracers in one 2D-array and treat all tracers in a similar fashion in one go like this (simplified illustration of the obstacle):

```
1 do k=2,kmax
2   k1 = k+off1
3   k2 = k+off2
4   t(1:nc,k) = t(1:nc,k) + A(k)*(B(1:nc,k1)-B(1:nc,k2))
5 enddo
```

- With dynamic nc the compiler vectorizes nc-loop:

  (4): (col. 7) remark: LOOP WAS VECTORIZED

- With static nc, the compiler vectorizes the k-loop:

  (1): (col 7) remark: LOOP WAS VECTORIZED

Premature abstraction is the root of all evil

- Alas, this is the code generated:
  - AVX (essentially a software gather operation):

```
...
vmovsd  (%r10,%rcx,2), %xmm6
vmovhpd 16(%r10,%rcx,2), %xmm6, %xmm6
vmovsd  32(%r10,%rcx,2), %xmm7
vmovhpd 48(%r10,%rcx,2), %xmm7, %xmm7
vinsertf128 $1, %xmm7, %ymm6, %ymm7
...
```

  - MIC (a hardware gather):

```
...
vgatherdpd (%r13,%zmm2,8), %zmm6{%k5}
...
```

- Especially for the MIC target not what we aimed at (issues on SNB/IVB with 256-bit unaligned load/store so a software gather may not be as bad as it looks to me)

Premature abstraction is the root of all evil

- The obstacle in a nutshell: A static nc (2) implies unrolling:

```
do k=1,kmax
  k1 = k+off1
  k2 = k+off2
  t(1,k) = t(1,k) + A(k)*(B(1,k1)-B(1,k2))
  t(2,k) = t(2,k) + A(k)*(B(2,k1)-B(2,k2))
enddo
```

- And the unrolling implies that the optimizer sees the loop as a stride-2 loop but we know better so let's state what the compiler should have done (next slide)
- And no.... interchanging loops is not the solution since it implies a 2x cost on BW and VL is reduced by 1/nc :

The compiler transformation that we hoped for:

- Proper handling of a mix of 2D and 1D (load):

zmm1←	t(1,1)	t(2,1)	t(1,2)	t(2,2)	t(1,3)	t(2,3)	t(1,4)	t(2,4)
zmm2←	t(1,5)	t(2,5)	t(1,6)	t(2,6)	t(1,7)	t(2,7)	t(1,8)	t(2,8)
zmm3←	B(1,1+k1)	B(2,1+k1)	B(1,2+k1)	B(2,2+k1)	B(1,3+k1)	B(2,3+k1)	B(1,4+k1)	B(2,4+k1)
zmm4←	B(1,5+k1)	B(2,5+k1)	B(1,6+k1)	B(2,6+k1)	B(1,7+k1)	B(2,7+k1)	B(1,8+k1)	B(2,8+k1)
zmm5←	B(1,1+k2)	B(2,1+k2)	B(1,2+k2)	B(2,2+k2)	B(1,3+k2)	B(2,3+k2)	B(1,4+k2)	B(2,4+k2)
zmm6←	B(1,5+k2)	B(2,5+k2)	B(1,6+k2)	B(2,6+k2)	B(1,7+k2)	B(2,7+k2)	B(1,8+k2)	B(2,8+k2)
zmm7←	A(1)	A(1)	A(2)	A(2)	A(3)	A(3)	A(4)	A(4)
zmm8←	A(5)	A(5)	A(6)	A(6)	A(7)	A(7)	A(8)	A(8)

Trick

- Proper handling of a mix of 2D and 1D (arithmetic):

```
zmm9  = zmm1 + zmm7*(zmm3-zmm5)!k=1,4;nc=1,2
zmm10 = zmm2 + zmm8*(zmm4-zmm6)!k=5,8;nc=1,2
```

- But did not get so we need to drop the 2D abstraction if performance matters to us.

Performance numbers

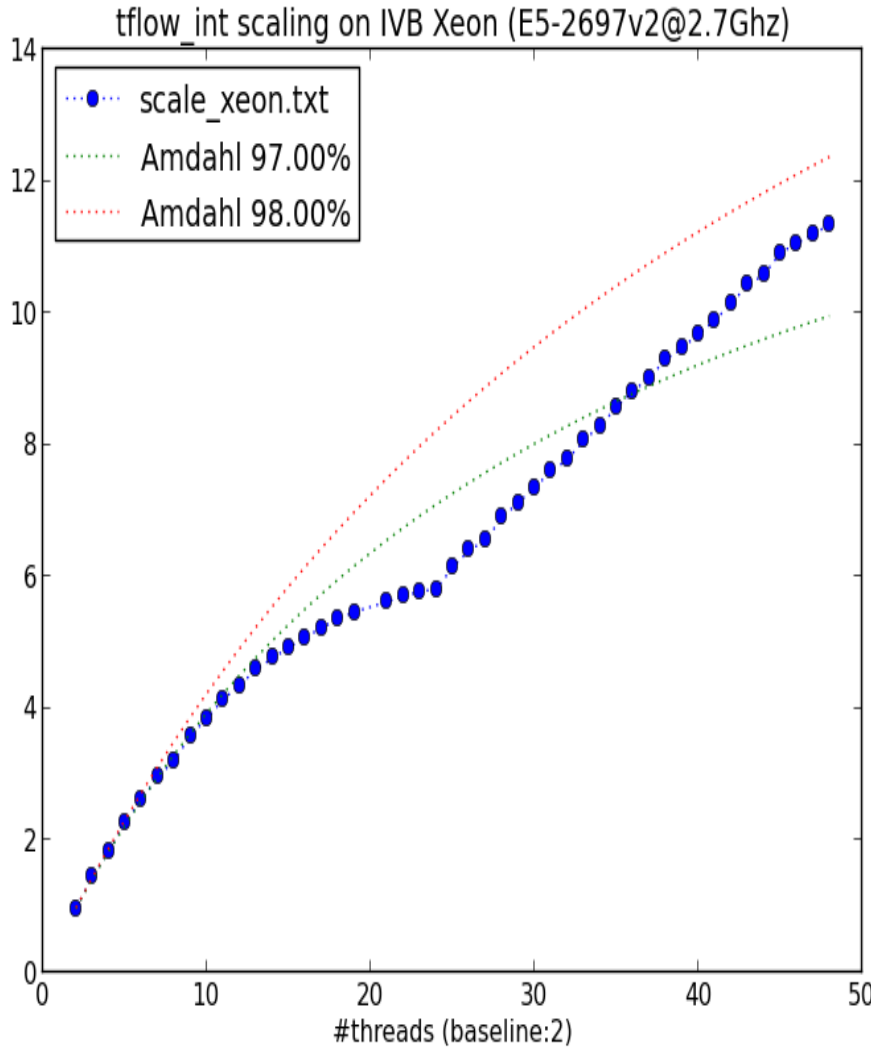
The module for the advection was chosen as a candidate for tunings. A single node run on both IVB and KNC showed that ≈41% of the time was spent here. The time spent on KNC was 3x the time on IVB when we started to investigate this.

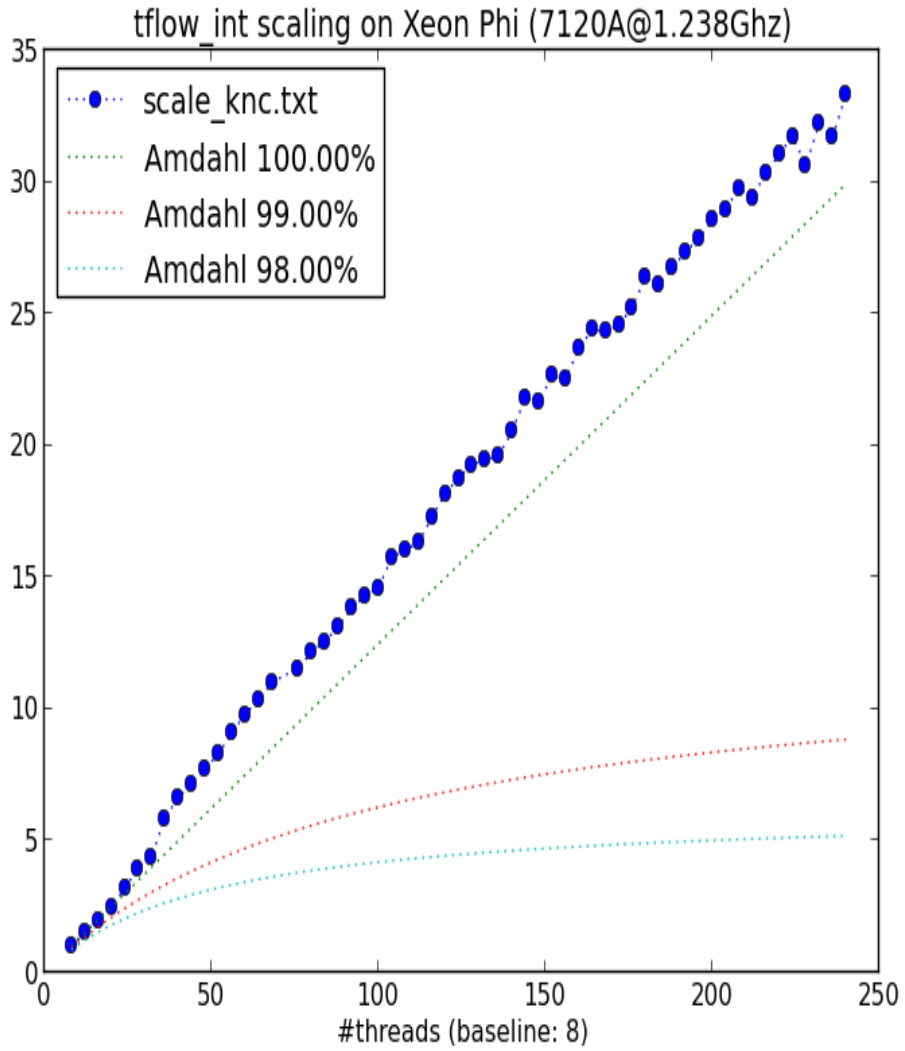
Benchmark systems

- Intel Xeon E5-2697 v2 (30Mb cache, 2.70 GHz)
  - Launched Q3, 2013
  - Number of cores/threads on 2 sockets: 24/48
  - DDR3-1600 MHz, 8*8 GB
  - Peak flops (HPL: 543 GF/s, 450 Watt)
  - Peak BW (Stream: 84 GB/s, 408 Watt)

- Intel Xeon Phi 7120A (30.5Mb cache, 1.238 GHz)
  - Launched Q2, 2013
  - Number of cores/threads: 60/240
  - GDDR5, 5.5 GT/s, 16 GB
  - Peak flops (HPL: 999 GF/s, 313 Watt)
  - Peak BW (Stream: 181 GB/s, 283 Watt)

Speedup on Xeon versus Xeon Phi





Benchmark summary (work in progress)

Subr.	2S-IVB 2697v2 (sec)	KNC C0 7120A (sec)	IVB BW (GB/s)	KNC BW (GB/s)	IVB BW (%)	KNC BW (%)	VI	ER
delta	15.047	8.643	85.03	137.80	100.00	76.56	7.93	2.52
cx	18.514	11.251	84.80	138.99	100.00	77.22	7.28	2.39
advection	10.065	7.127	89.33	141.77	100.00	78.76	8.05	2.05
rin	25.561	19.707	85.32	129.54	100.00	71.97	5.72	1.88
c_tu	6.844	7.656	85.49	115.45	100.00	64.14	6.05	1.30
c_tv	11.311	9.68	85.91	122.02	100.00	67.79	6.04	1.69
c_tw	10.112	14.084	87.53	118.60	100.00	65.89	6.14	1.04
c_tt	20.55	17.523	86.97	131.47	100.00	73.04	7.47	1.70
c_dtx	16.048	16.553	86.12	120.42	100.00	66.90	7.18	1.41
c_dty	16.344	17.551	87.87	119.09	100.00	66.16	7.16	1.35
c_dtz	15.622	15.462	87.30	117.34	100.00	65.19	6.94	1.47
tflow_int	**84.502**	**71.668**	**89.158**	**137.9643**	**100.00**	**76.65**	**6.73**	**1.71**

Acknowledgement:

Karthik Raman, Michael Greenfield, Larry Meadows, Intel
Bill Long, Cray
Per Berg, Lars Johnasen, Rong Fu, DMI

Backup/reference slides

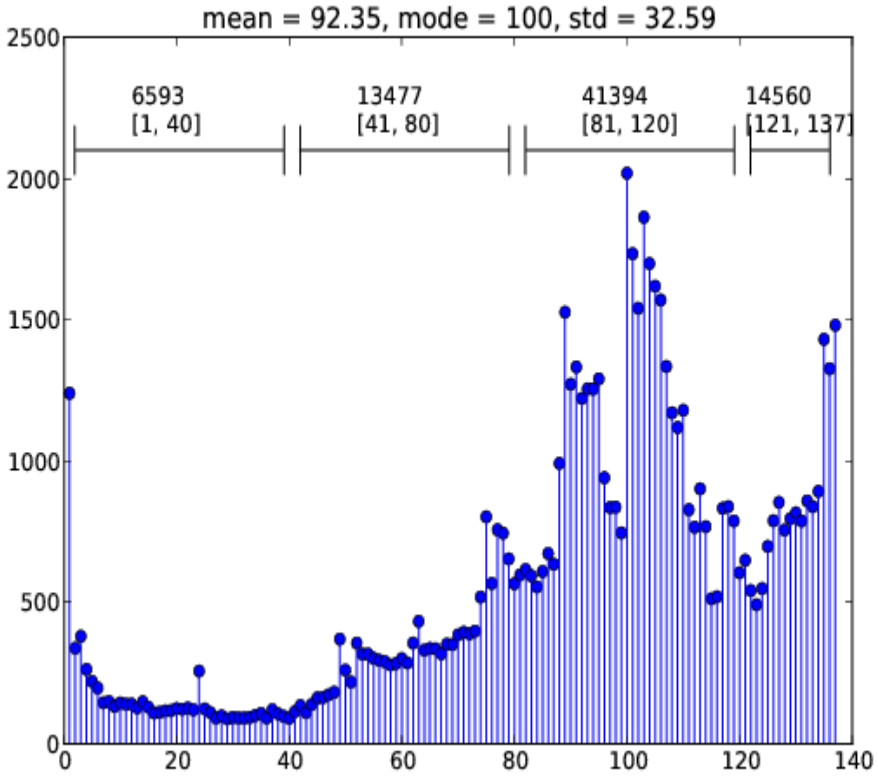
Benchmark systems

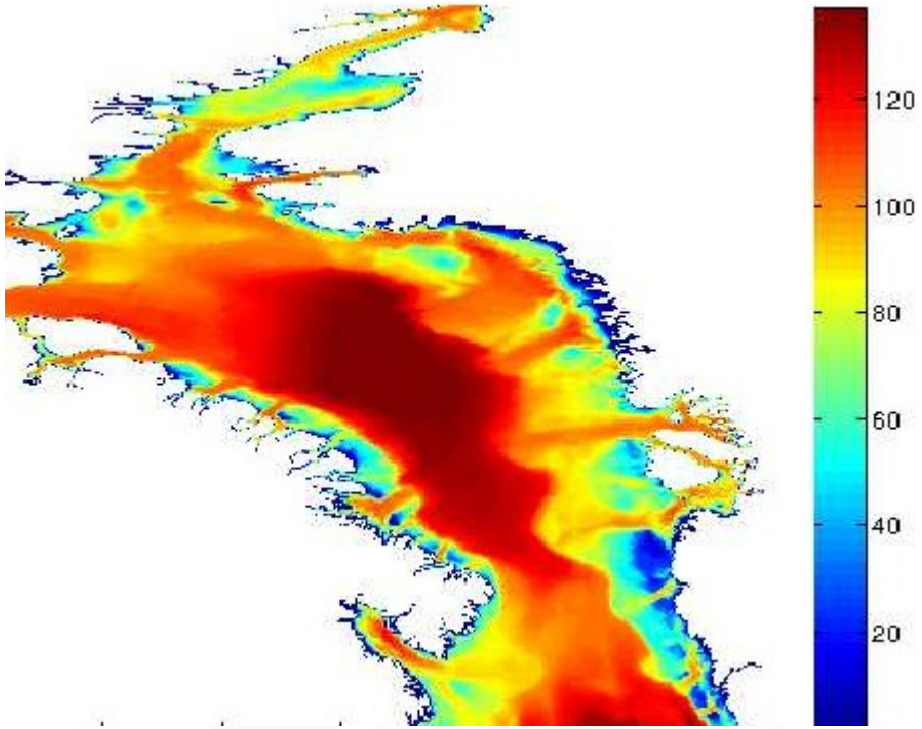
- Compiler: Intel Version 14.0.1.106
  - Flags on Xeon: -O3 -fno-alias -ipo -fimf-precision=low -fimf-domain-exclusion=15 -align array64byte -xAVX
  - Flags on Phi: -O3 -fno-alias -ipo -fimf-precision=low -fimf-domain-exclusion=15 -align array64byte -mmic -opt-streaming-stores always -opt-streaming-cache-evict=0
- Thread placement: compact

- Xeon versus Xeon Phi
  - HPL: Xeon Phi is 1.84 times faster than Xeon and Xeon uses 1.44 times more watt to attain peak
  - Stream triad: Xeon Phi is 2.15 times faster than Xeon and Xeon uses 1.44 times more watt to attain peak

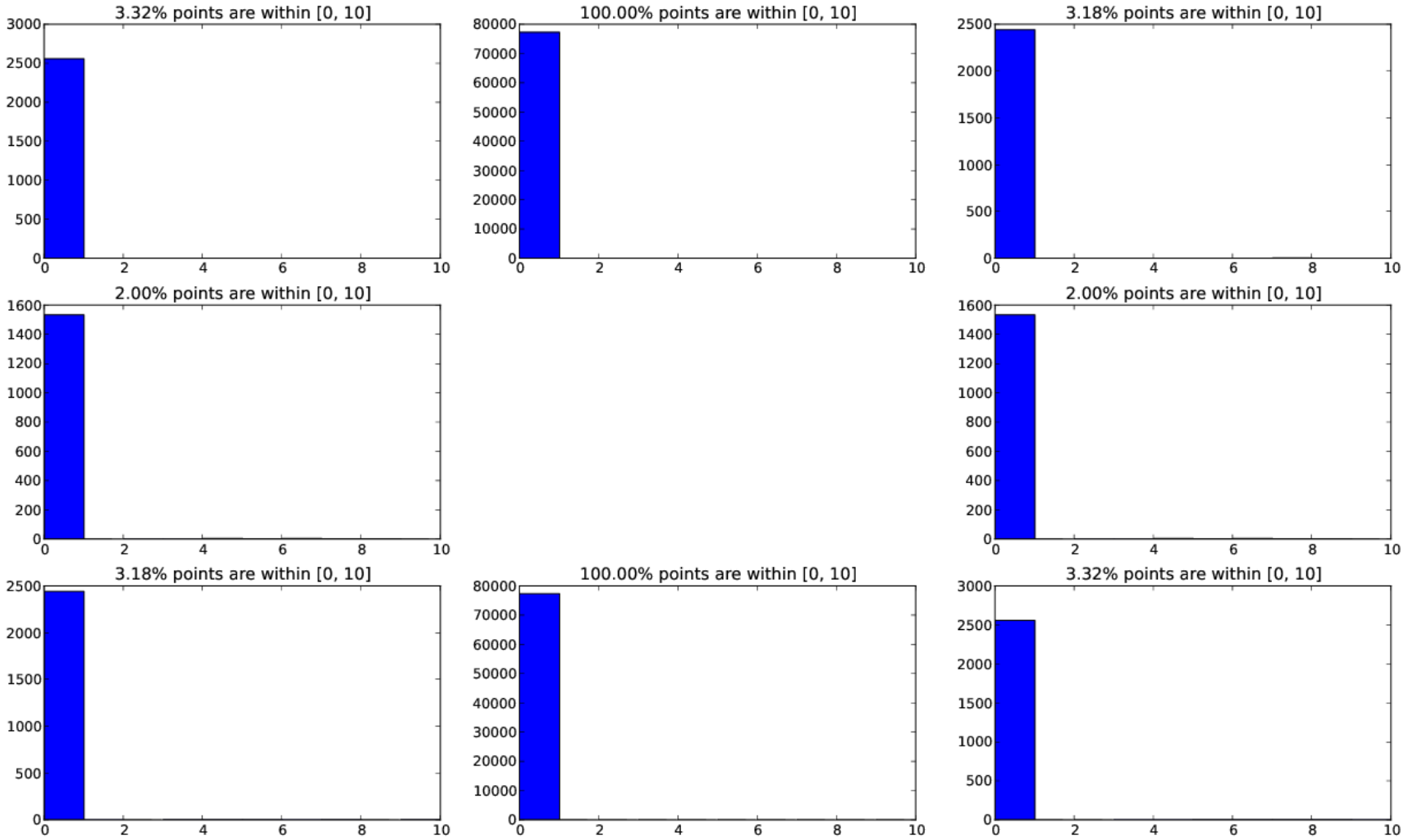
Benchmark testcase – Baffin Bay (2nm)

- Dimension: 565X331x137; iw2=77285; iw3=7136949





Proximity study (current layout)



Space Filling curves – The H-curve