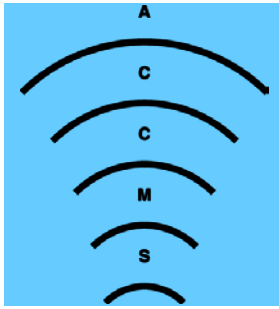
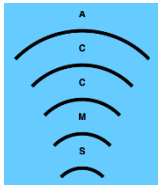
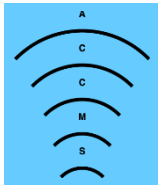


Finding Regularity in Problems with Irregularity

Hiroshi Nakashima
(ACCMS, Kyoto University)



Contents



Players Lineup in HPC Games

- Correctness

  Without it we cannot earn anything.

- Complexity

  $O(N^2)$ cannot defeat $O(N)$ in HPC with $N>>1$.

- Parallelism

  Absence of this veteran makes your supercomputer as slow as a smartphone.

- Locality

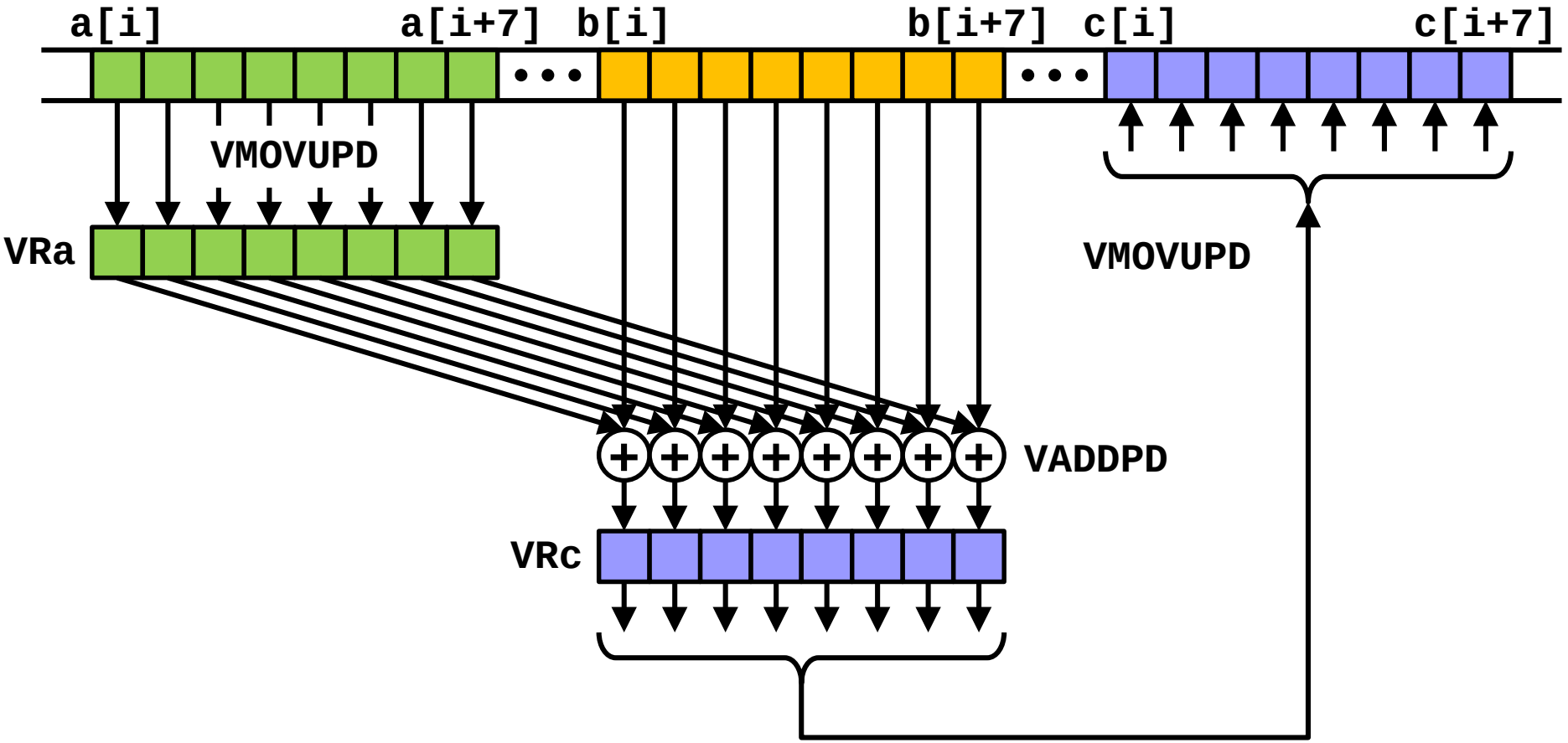
  High memory wall increases its importance in HPC.

- Regularity

  This rookie now controls the game by its ability to up/down the score by a factor of 10x.

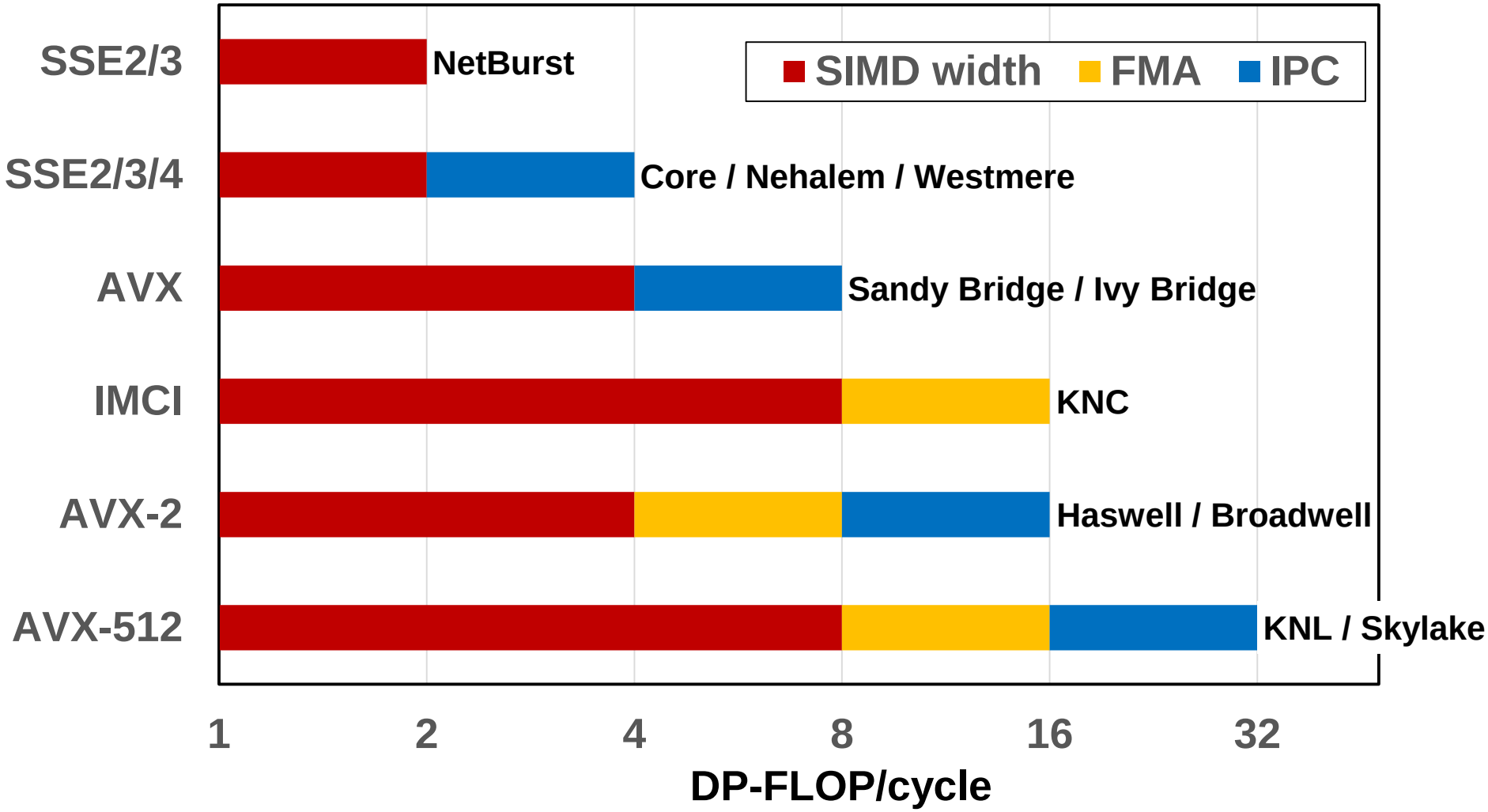
SIMD: Hometown of Regularity

- **double a[],b[],c[];**
  for(i=0;i<n;i++) c[i]=a[i]+b[i];



SIMD: How Widen?

- Intel's SIMD Architecture



SIMD: How Working in Kyoto?

Camphor 2

CRAY XC40

intel Xeon Phi 7250 (KNL)

1.4GHz x 8 x 4 x 68 x 1 x 1800

= 5.48PFlops

DATAWARP 230TB

Camellia

CRAY XC30

intel Xeon Phi 5250D (KNC)

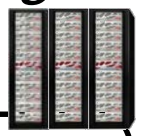
intel Xeon E5-2670v2 (Ivy Bridge)

(1.1GHz x 8 x 2 x 68 x 1 +

2.5GHz x 4 x 2 x 10 x 1) x 482

= 0.58PFlops

Storage



DDN STORAGE ExaScaler

16+8PB

IME 230TB

InfiniBand FDR/EDR (6.8/12.1GB/s•link)

Laurel 2

CRAY CS400 2800XT

intel Xeon E5-2695v4 (Broadwell)

2.1GHz x 4 x 4 x 18 x 2 x 850

= 1.03PFlops

Cinnamon 2

CRAY CS400 4840X

intel Xeon E7-8880v3 (Haswell)

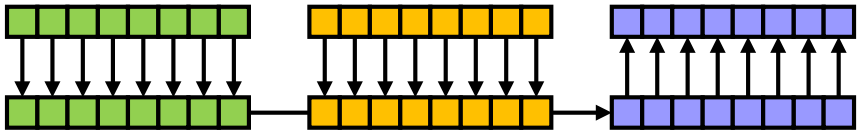
2.3GHz x 4 x 4 x 18 x 4 x 16

= 0.04PFlops

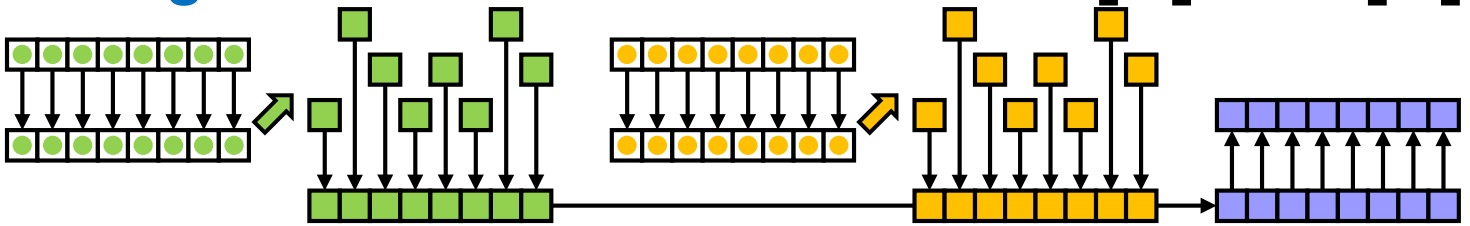
Omni-Path (12.1GB/s•link, BB= 5.15TB/s)

Regularity vs Irregularity
Simple Examples

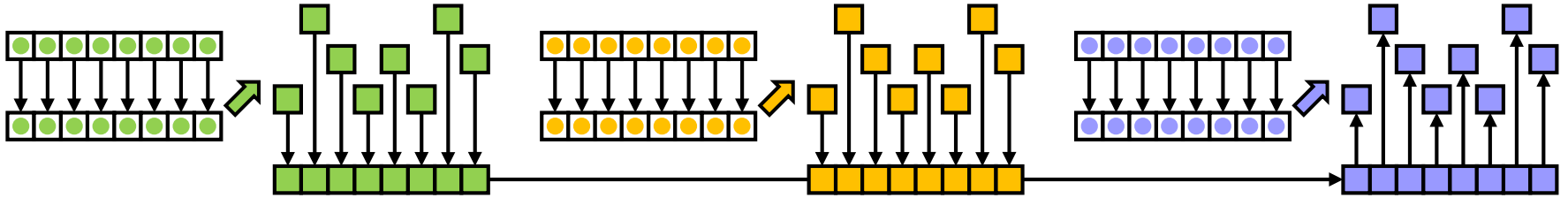
- `for(i=0;i<n;i++) c[i]=a[i]+b[i];`
  regular & fast



- `for(i=0;i<n;i++) c[i]=a[xa[i]]+b[xb[i]];`
  irregular in RHS & slow even if `xa[i]==xb[i]==i`



- `for(i=0;i<n;i++) c[xc[i]]=a[xa[i]]+b[xb[i]];`
  irregular in LHS & very slow even if vectorized



How Vectorized (1/2)

- By Intel 17.0.2 for KNL

`c[i]=a[i]+b[i]`	`c[i]= a[xa[i]]+b[xb[i]]`	`c[xc[i]]= a[xa[i]]+b[xb[i]]`

`vmovups a[i]`
`vaddpd b[i]`
`vmovupd c[i]`
	`vmovdqu xa[i]`
	`vmovdqu xb[i]`
	`vgatherdpd a[]`
	`vgatherdpd b[]`
	`vaddpd`
	`vmovupd c[i]`
		`vmovdqu xa[i]`
		`vmovdqu xb[i]`
		`vmovdqu xc[i]`
		`vgatherdpd a[]`
		`vgatherdpd b[]`
		`vaddpd`
		`vscatterdpd c[i]`

■ Overlap of a/b/c is inspected for vectorization if they are not `restrict`-ed ■ Two-way unrolled further.	■ a/b/c must be `restrict`-ed for vectorization. ■ Has redundant instructions for unnecessary masking (with k*i*=11...1) and zero-clear of destination of gather (& scatter).	

How Vectorized (2/2)

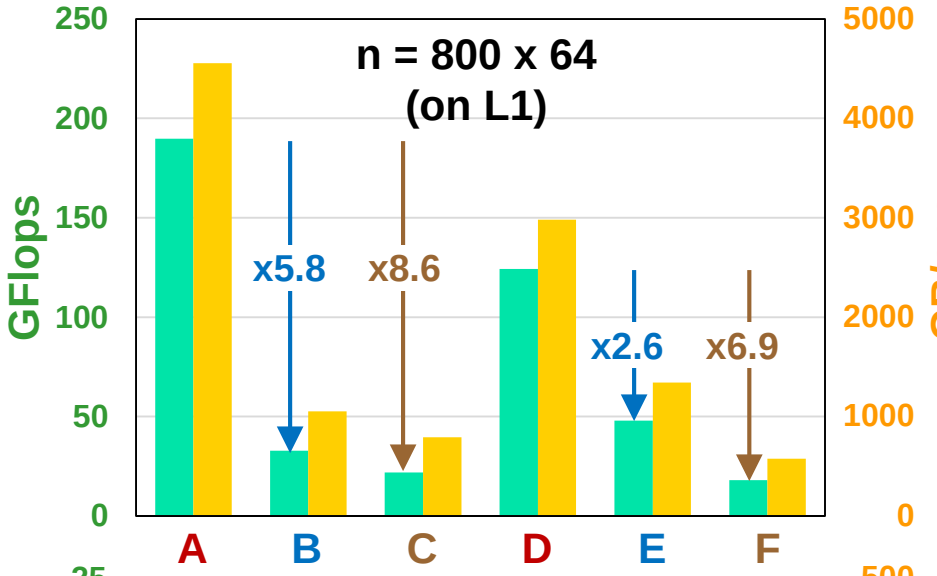
- for **a[]+=b[]**

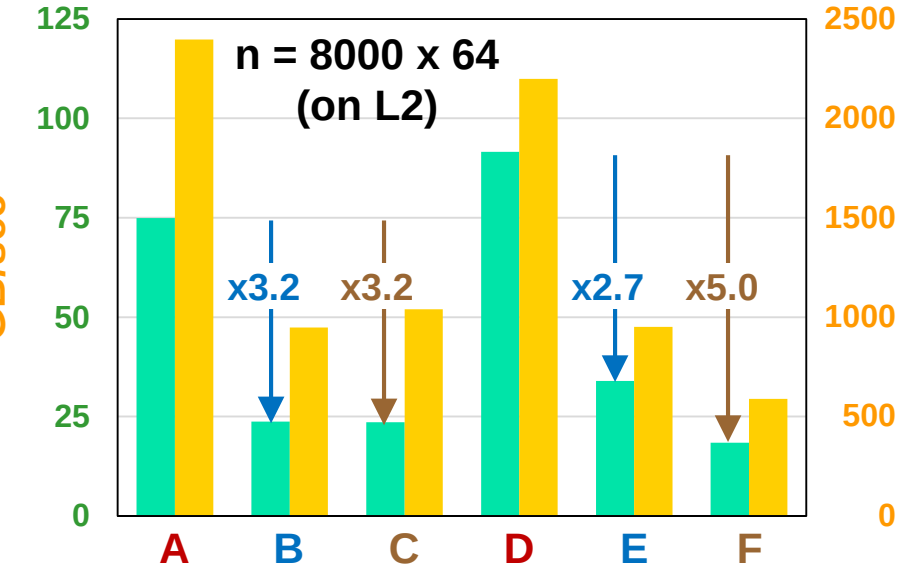
a[i]+=b[i]	a[i]+=b[xb[i]]	a[xa[i]]+=b[xb[i]]
vmovups a[i] vaddpd b[i] vmovupd a[i]	vmovdqu xb[i] vmovups a[i] vgatherdpd b[] vaddpd vmovupd a[i]	vmovdqu xb[i] vgatherdpd b[] vmovdqu xa[i] vpconflictd vgatherdpd a[] vpmovzxdq vptestmq vaddp kmovw testl je *conflict case* vscatterdpd a[i]
▪ ≈ c[]=a[]+b[]	▪ ≈ c[]=a[]+b[]	▪ Needs conflict check in case that xa[] has duplication.

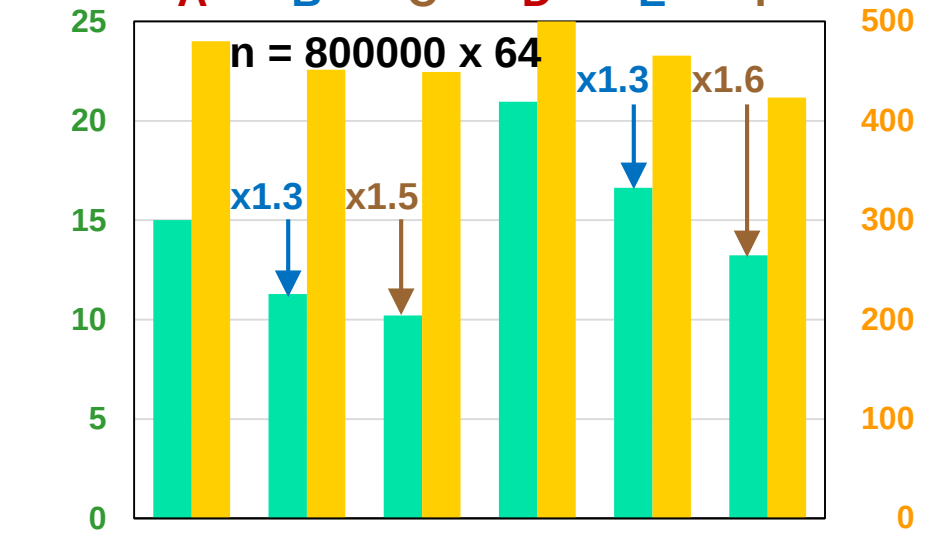
Regularity vs Irregularity
How Fast & Slow

xa[i]=xb[i]=xc[i]=i

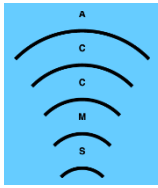
- 64-core (&thread) execution on KNL







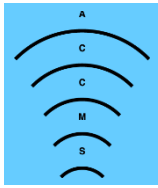
- A: `c[i]=a[i]+b[i]`
  B: `c[i]=a[xa[i]]+b[xb[i]]`
  C: `c[xc[i]]=a[xa[i]]+b[xb[i]]`
  D: `a[i]+=b[i]`
  E: `a[i]+=b[xb[i]]`
  F: `a[xa[i]]+=b[xb[i]]`
- **Severe slowdown in on-cache cases while almost-peak B/W is exerted in off-cache case.**



Regularity vs Irregularity
Why so Slow

- On-Cache Case
  - **SIMD mechanism relies on wide access to a cache line too heavily to perform a set of small-size loads/stores for a gather/scatter efficiently.**
  - **Short latency of L1 access does not allow to coalesce multiple accesses of gather/scatter effectively even when they targets on a single line.**
  - **➔ Gather/scatter severely degrades the effect of our effort of SIMD- and cache-aware implementation.**
    - **SIMD-aware: x3.1 ➔ x2.2-x2.0 (for a[]+=b[])**
    - **cache-aware: x5.9 ➔ x2.9-x1.4**

c.f. SX-ACE's ADB, software-controlled non-coherent cache, is accessible in word-granularity and has load-coalescing mechanism.

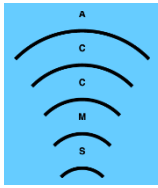


Regularity vs Irregularity
Why not so Slow

- Off-Cache Case
  - **As far as gather/scatter accesses have reasonable spatial locality, last-level cache (L2) effectively coalesces multiple non-temporally-local accesses into a single cache miss.**
  - **Memory controller also effectively coalesces cache-missing accesses from many cores into a series of not-so-random accesses to memory (MCDRAM) to exploit its large bandwidth.**
  - ➔ **Strongly discourage people from improving access locality or eliminating indirection because one single effort is not very effective.**
    - **only improving locality: x1.4 (for a[]+=b[])**
    - **only eliminating indirection: x1.6**
    - **both: x9.4 >> 1.4 x 1.6**

Other Sources of Irregularity

- `for(a=ah,b=bh,c=ch; a&&b&&c;`
  `a=a->n,b=b->n,c=c->n) c->v=a->v+b->v;`
  - Even with a smart complier, you have a scalar pointer chasing followed by a vectorized gather/scatter.
- `for(i=0;i<n;i++)`
  `if (a[i]<0) c[i]=some_func(a[i],b[i]);`
  `else c[i]=a[i]+b[i];`
  - Even with a smart complier, you could have non-vectorized code instead of vectorized `else`-part for the cases of `a[i]>=0` for all lanes even if `a[i]` is usually (or always) non-negative.



Regularity: How to Achieve?

- Fundamentals
  - **Use arrays instead of linked lists.**
  - **Eliminate indexing arrays/functions.**
  - **Eliminate unbalanced conditionals.**
- In Addition
  - **Show the regularity apparently to your compiler.**
  - **Combine cache-awareness to make regularization really effective.**
- However …
  - **How can I regularize my program which operates on irregular data such as sparse matrices and sets of objects?**

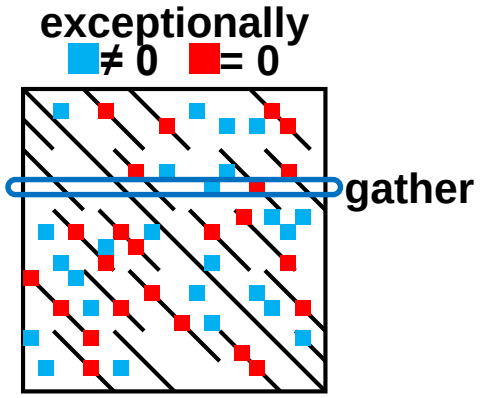
Regularity in Irregular Data: Sparse Matrix in CRS

- CRS Matrix Vector Multiply **y=A*x**

```
for(i=0;i<n;i++){ y[i]=0;
  for(j=A.row[i];j<A.row[i+1];j++)
    y[i]+=A.val[j]*x[A.col[j]];
}
```

  - **Gather on `x[]` is inefficient.**
  - **`A.row[i+1]-A.row[i]` is usually small (up to a few tens) to make the overhead of prologue & epilogue large. (c.f. fixing it with zero-padding may improve performance.)**

- Cannot We Find Regularity?
  - ***If* A has many *k*-diagonal sequences;**
  - **We may represent A as a set of *k*-diagonal sequences (plus exceptional non-zeros).**



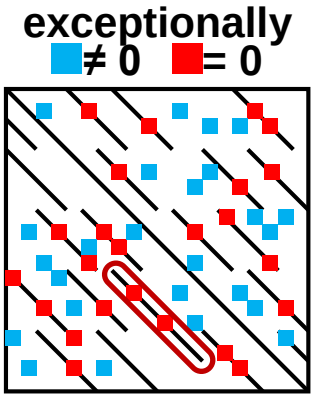
Regularity in Irregular Data: Sparse Matrix in *k*-Diagonal Form

- *k*-Diagonal Matrix Vector Multiply **y=A*x**

```
for(i=0;i<n;i++) y[i]=0;
for(d=0;i<A.ndiag;d++){
  for(i=A.drow[d],j=A.dcol[d],k=A.dval[d];
      k<A.dval[d+1]; i++,j++,k++)
    y[i]+=A.val[k]*x[j];
} // then operate on exceptional non-zeros
```

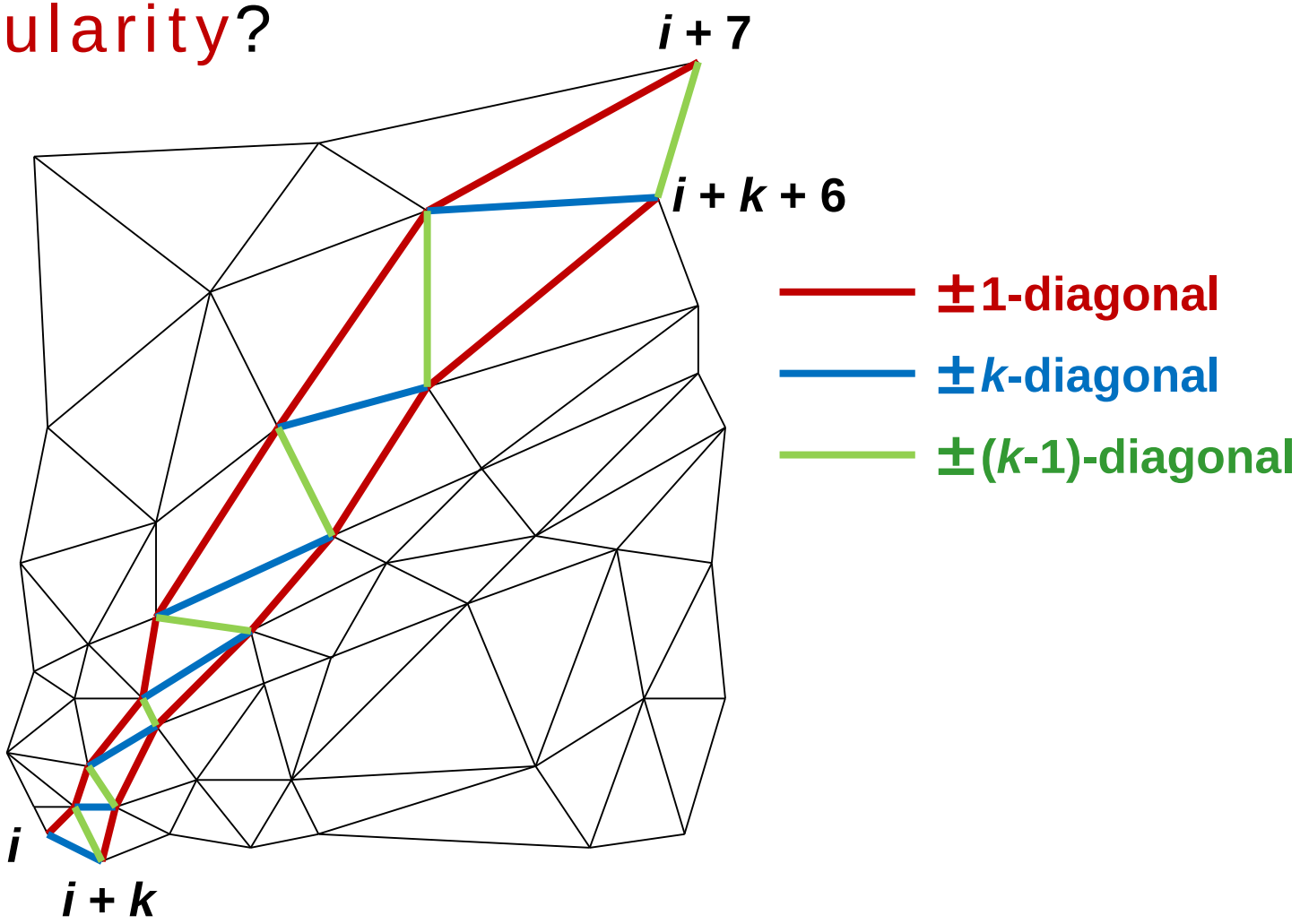
  - **drow[d],dcol[d],dval[d]: row, column and index of val[] of the head of the d-th *k*-diagonal sequence.**

- Can We Find such Sequences?
  - **Easy for cubic structured meshes.**
  - **How can we find them in unstructured meshes of triangles or tetrahedrons?**



Regularity in Irregular Data: Finding *k*-Diagonal Sequences

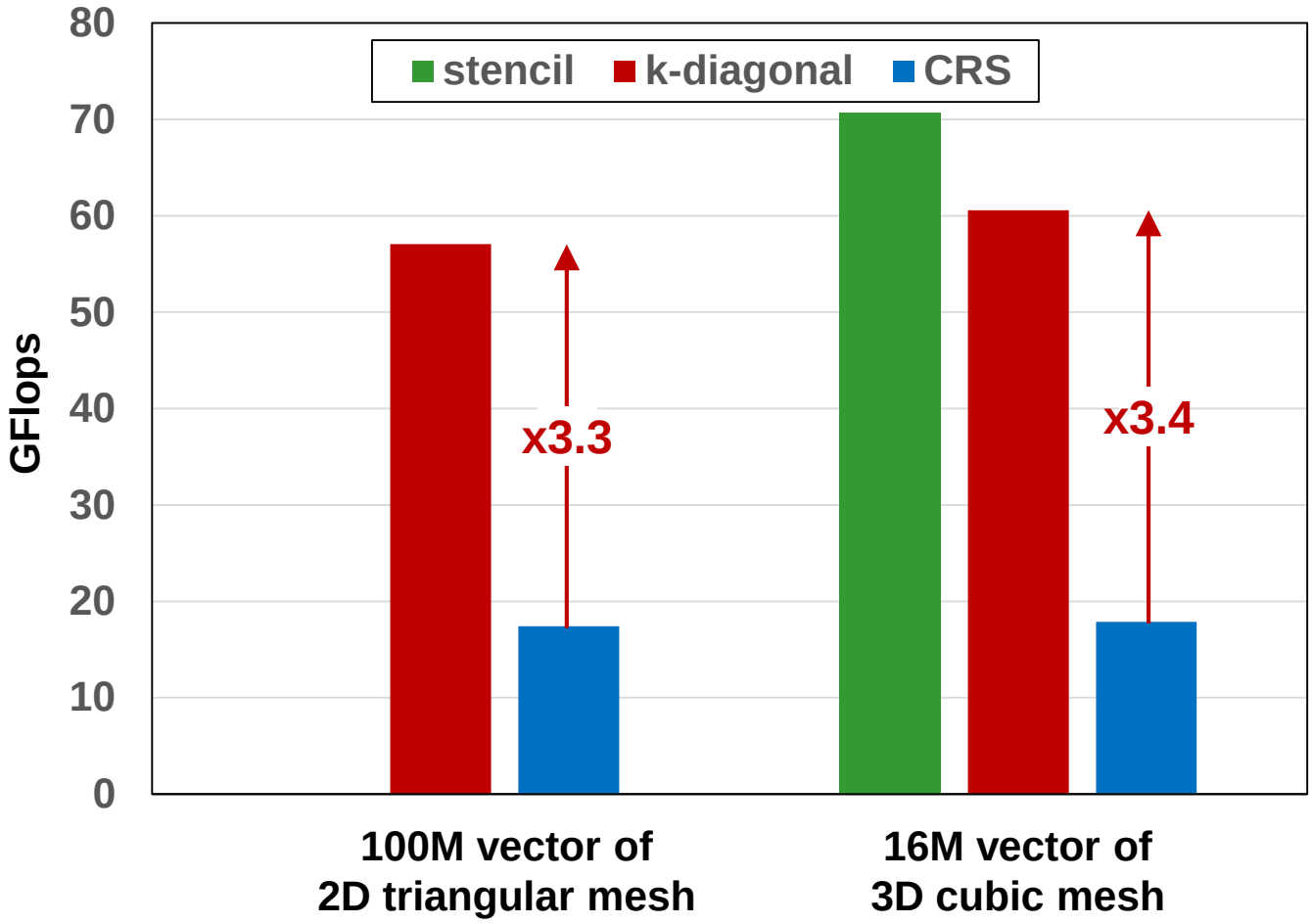
- Regularity?



- Yes!! (though No in general)

Regularity in Irregular Data: How Efficient ?

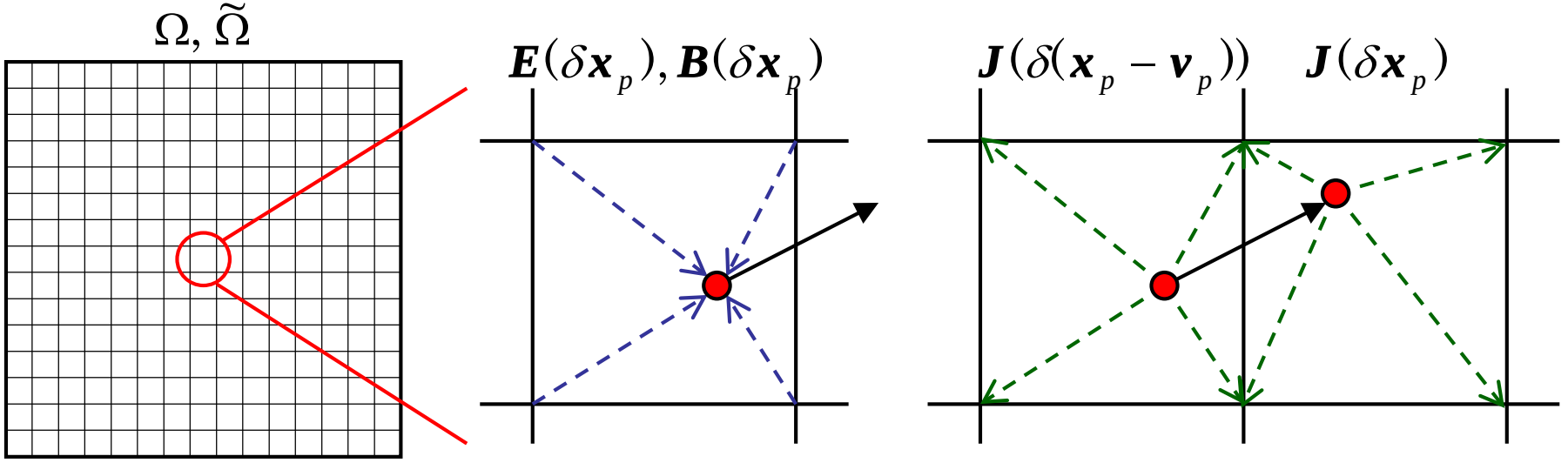
- **y=A*x** on KNL with 64 cores (threads)



Regularity in PIC Simulation: Fundamental Irregularity in PIC

- For each p at x_p in a cell whose vertices are at $\boldsymbol{\delta} x_p$;
  - **Update v_p by Lorentz force determined by E and B at δx_p, and then update x_p by v_p.**
  - **Add the contribution of p's motion to J at δx_p.**

➔ **E[ ][ ][ ], B[ ][ ][ ], J[ ][ ][ ] are accessed by $\lfloor x_p \rfloor + \{0,1\}^3$ with gather/scatter.**



Regularity in PIC Simulation: Regularize by Particle Binning

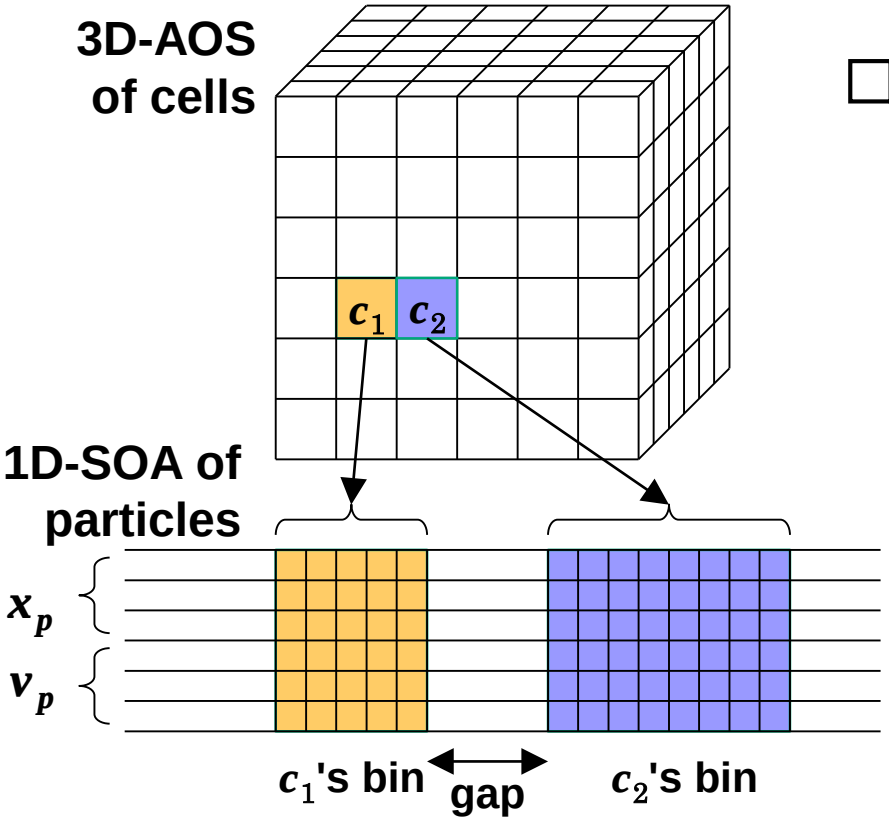
- Let each cell *c* have the set (bin) of all particles in it.
- Scalarize *E/B/J* accessed by all *p* in *c*.

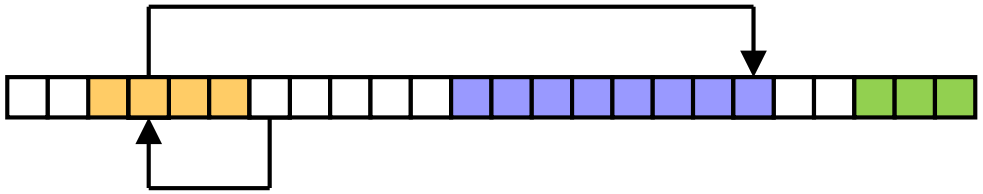
```
for(c in cells){
  {sE}=Earound(c); {sB}=Baround(c);
  for(p in c) v[p]+=lorentz(p,{sE},{sB});
  {sJ}=0;
  for(p in c)
   {{sJ}+=scatter(p); x[p]+=v[p];}
  Jaround(c)+={sJ};
  for(p in c) migrate(p);
}
for(c in cells){
  {sJ}=0;  for(p in c) {sJ}+=scatter(p);
  Jaround(c)+={sJ};
}
```

If x[] and v[] are simple arrays, vectorized well without gather/scatter of E/B/J.

Regularity in PIC Simulation: Regularized Bin Management

- Instead of Irregular Structure/Procedure
  - **Such as linked list or batched radix sort.**
- On-the-Fly Sort on Gapped Arrays





- **Record direction $d_p \in \{-1,0,1\}^3$ of inter-cell migration for all p in c.**
- **Skip p s.t. $d_p = (0,0,0)$.**
- **Migrate p s.t. $d_p \neq (0,0,0)$ to the gap of the destination bin.**
- **Fill the vacancy by the last particle of c.**
- **Migration cannot be vectorized but for a few particles.**
- **Work efficiently unless overflow.**

Regularity in PIC Simulation: Bin Overflow: Cost for Regularity

- If a gap is exhausted
  - **Move overflown particle into a buffer and process all particles in the buffer without binning, until e.g. the cumulative processing cost becomes too large.**

  ← Reduce frequency of bin rearrangement.
- When no longer we can keep particles in the buffer
  - **Resize gaps not only to enlarge them for cells (nearly) overflown but also to keep them as large as possible.**

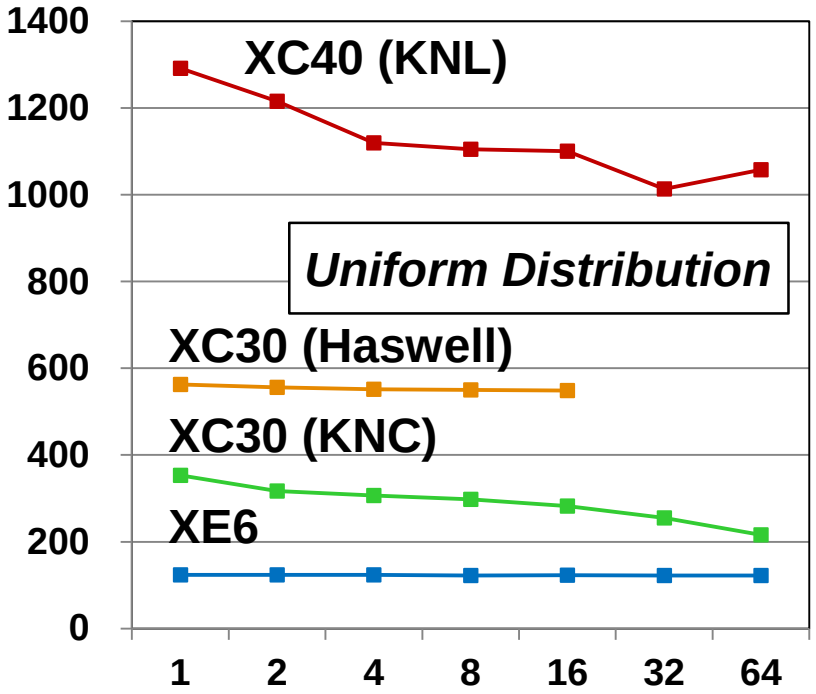
  ← Cope with oscillatory repletion/depletion of bins.
  - **Perform in-place multithreaded SIMD-vectorizable shift of bins according to the new gap sizes.**

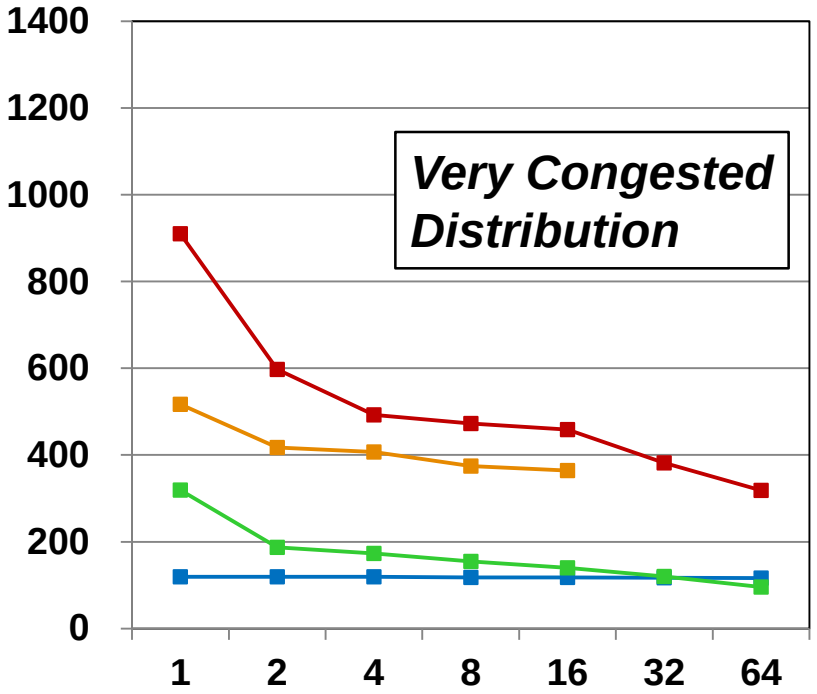
Regularity in PIC Simulation: How Efficient ?

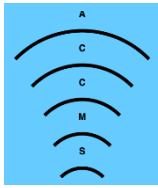
- 1-node
  (10^6 p/s)

	peak TFlops	binning	
		yes	no
XC40 (KNL)	3.05	1179	---
XC30 (Haswell)	1.03	590	291
XC30 (KNC)	1.01	391	57
XE6 (Abu Dhabi)	0.32	---	123

- Multi-node
  (10^6 p/s·node)

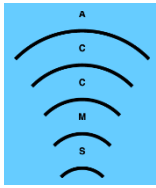




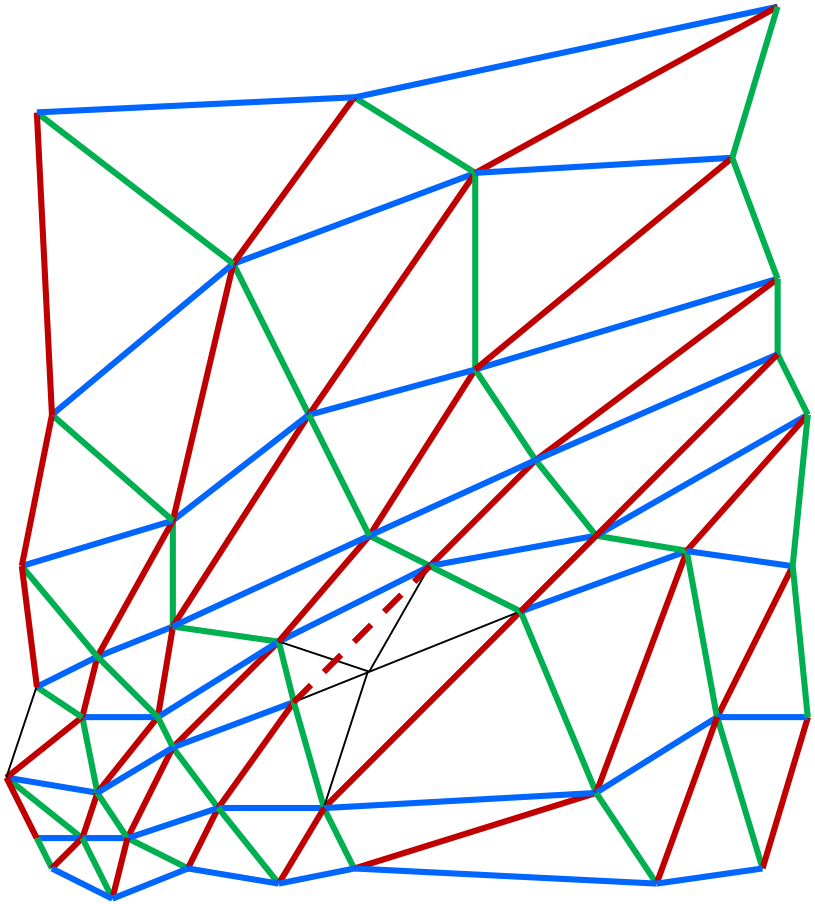


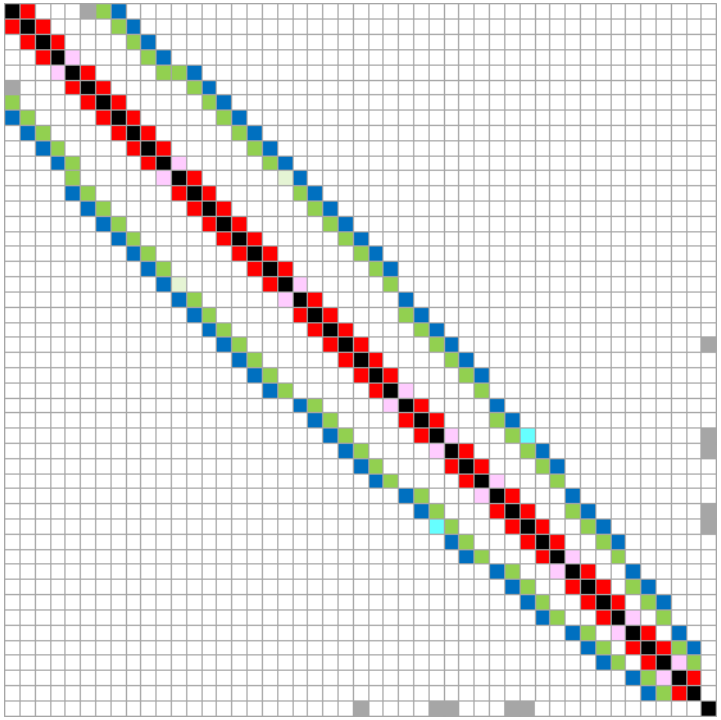
Conclusions

- What I've talked
  - **Importance of regularity for recent processors having wide SIMD mechanisms.**
  - **How to find and exploit in-practice regularity in in-general irregular data/procedures.**
- What I haven't talked
  - **How large effort we have to make for the exploitation of in-practice regularity.**
    - **70+% of my PIC code of 2,622 C lines are for non-kernel operations for regularization (e.g., overflow handling).**
  - **We need regularization libraries to manipulate in-general irregular data (not their structures), i.e.,**
    - **Sparse matrices rather than CRS form of them.**
    - **Sets rather than their linked-list representation.**


A
C
C
M
S

backup





±*k*	0	1	2	3	4	5	6	7	8	except. ≠0
length (except.=0)	47	45(9)	2	5	7	9	5(0) 11(1)	25(1)	14	6