Considerations for computational performance of algorithms for hydrocodes on advanced* architectures

J. Fung

Los Alamos National Laboratory

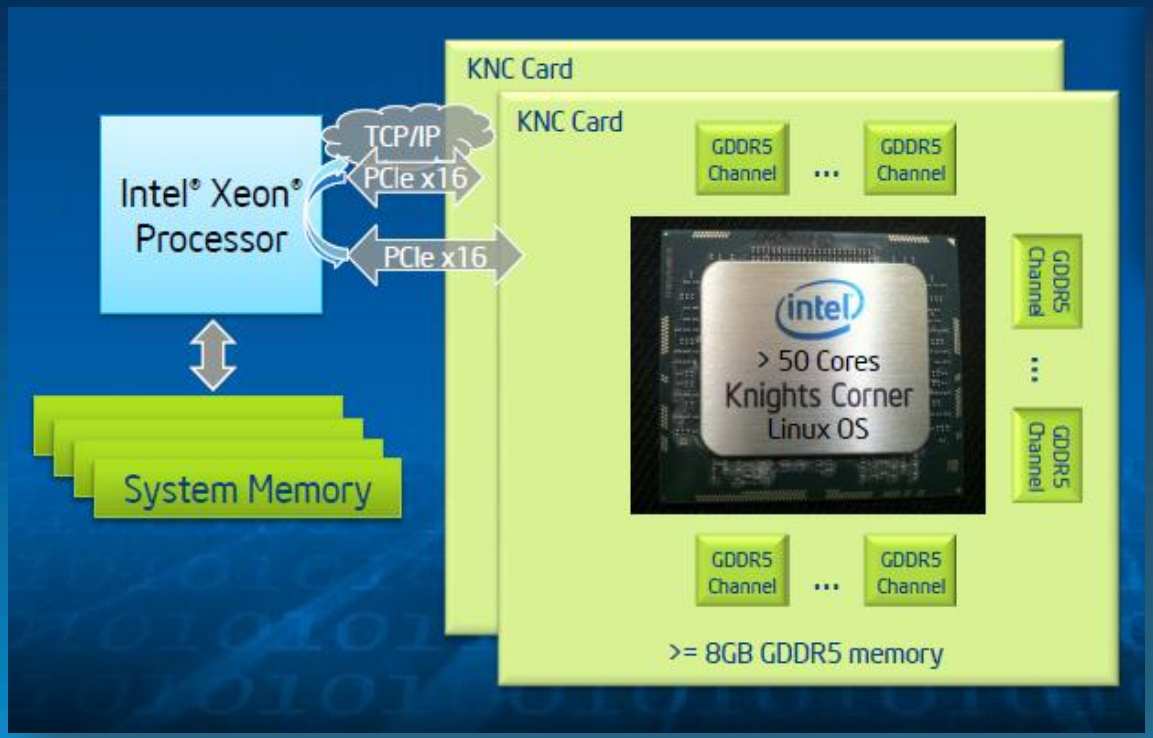
with R. T. Aulwes, M. T. Bement, C. R. Ferenbaugh, T. M. Kelley,

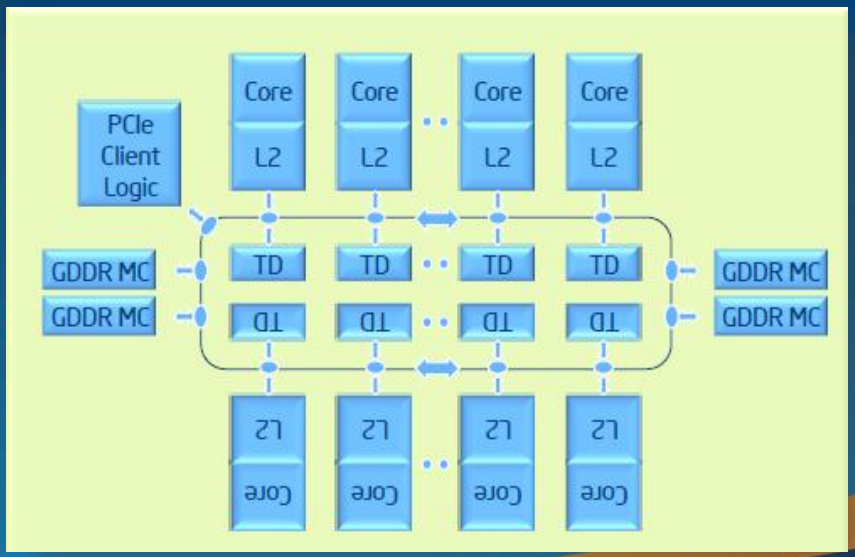
M. A. Kenamond, B. R. Lally, E. G. Lovegrove, E. M. Nelson, and D. M. Powell

MULTIMAT 2013: San Francisco
2-6 September 2013

Motivation

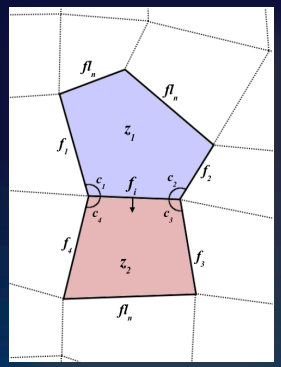
- Emerging computer architectures pose memory constraints that may affect performance of existing algorithms and codes.
- What are design techniques that we may employ for performance?
- What can we do with existing algorithms (on existing architectures)?



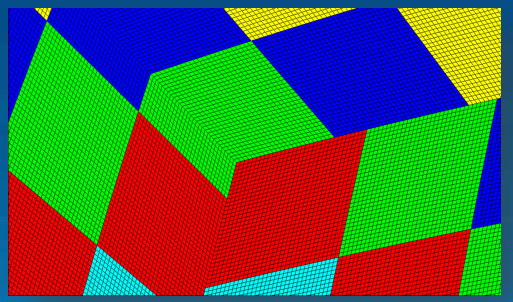


Outline

- Threading
  - Diffusion matrix assembly

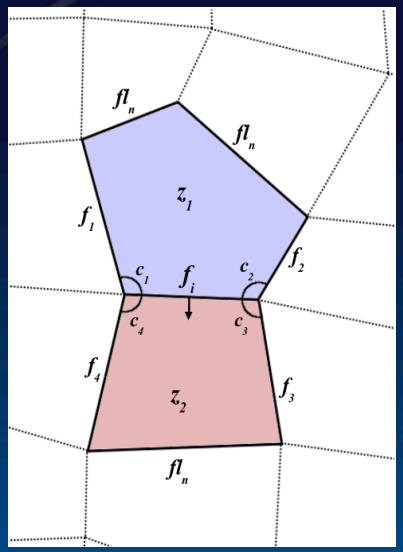


- Vectorization – arcane or unfamiliar?
  - Reaction rates
  - Gradient kernel



- Cache blocking
  - Hydro subkernels
  - Diffusion kernels

Consider a mimetic diffusion discretization for unstructured meshes.



Solve for fluxes:

$$(\mathscr{S} + \mathscr{D}\dagger\mathscr{M}\Omega^{-1}\mathscr{D})\mathbf{W}^{n+1} = \mathscr{D}\dagger\mathscr{M}\Omega^{-1}F^{n+1}$$

Update scalars:

$$U^{n+1} = \Omega^{-1}(F^{n+1} - \mathscr{D}\mathbf{W}^{n+1})$$

- Enforces an adjoint relationship between the *discrete* operators **DIV** and **GRAD**
- Works with convex, non-intersecting meshes
- Second-order accurate in space for quad meshes
- Symmetric, positive-definite matrix

$$(S_{11}A\xi)_{(i,j)} = \left(\sum_{k,l=0}^{1} \frac{1}{k_{(i-k,j)}} \cdot \frac{V^{(i-k,j)}_{(i,j+l)}}{\sin^2(\varphi^{(i-k,j)}_{(i,j+l)})}\right) A\xi_{(i,j)},$$

$$(S_{12}A\eta)_{(i,j)} = \sum_{k,l=0}^{1} \left(\frac{(-1)^{k+l}}{k_{(i-k,j)}} \cdot \frac{V^{(i-k,j)}_{(i,j+l)}}{\sin^2(\varphi^{(i-k,j)}_{(i,j+l)})} \cos(\varphi^{(i-k,j)}_{(i,j+l)})\right) \times A\eta_{(i-k,j+l)},$$

$$(S_{21}A\xi)_{(i,j)} = \sum_{k,l=0}^{1} \left(\frac{(-1)^{k+l}}{k_{(i,j-l)}} \cdot \frac{V^{(i,j-l)}_{(i+k,j)}}{\sin^2(\varphi^{(i,j-l)}_{(i+k,j)})} \cos(\varphi^{(i,j-l)}_{(i+k,j)})\right) \times A\xi(i+k, j-l),$$

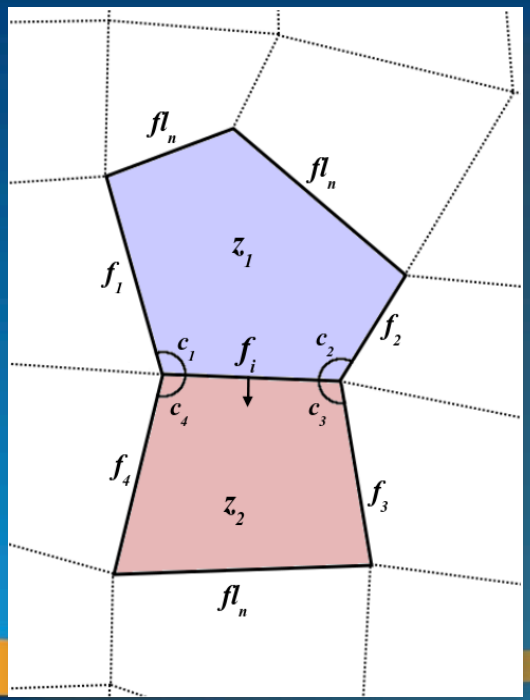
$$(S_{22}A\eta)_{(i,j)} = \left(\sum_{k,l=0}^{1} \frac{1}{k_{(i,j-l)}} \cdot \frac{V^{(i,j-l)}_{(i+k,j)}}{\sin^2(\varphi^{(i,j-l)}_{(i+k,j)})}\right) A\eta_{(i,j)}.$$

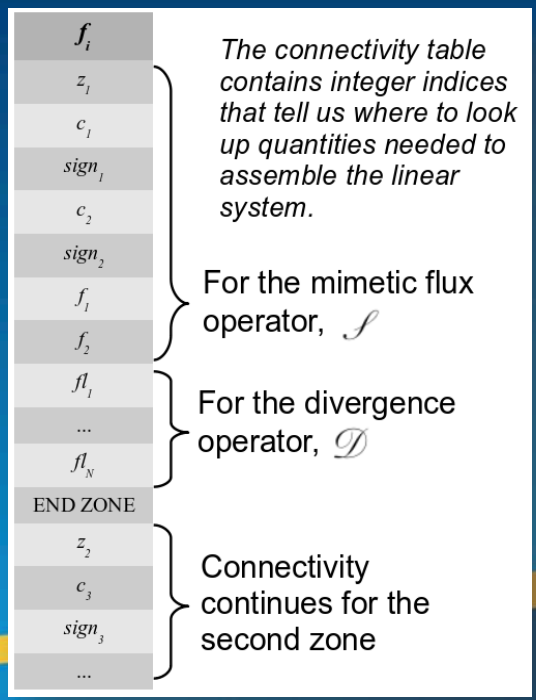
Derived from inner product

Shashkov and Steinberg, 1996

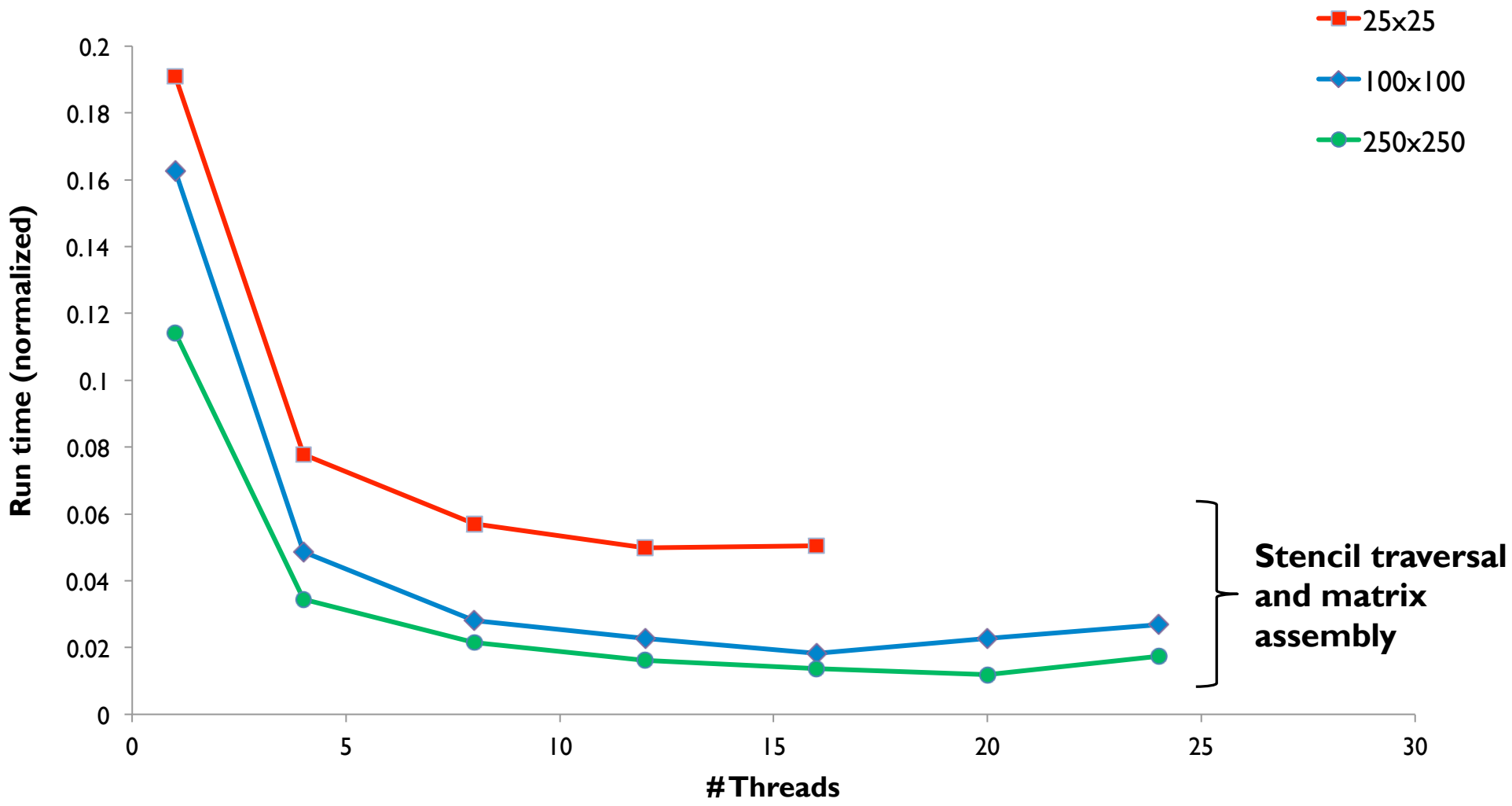
Traverse the stencil and store connectivity ahead of time. Then attempt to thread over face loops.

- Simple retrieval and insertion of matrix values
  - However, no appreciable speedup over on-the-fly value retrieval
- Determine row sizes for compressed sparse row (CSR) matrix storage format
- Allow modularization of boundary conditions
  - Stay tuned – this can be used for kernel and subkernel design
- If the mesh connectivity is static, this only needs to be done once

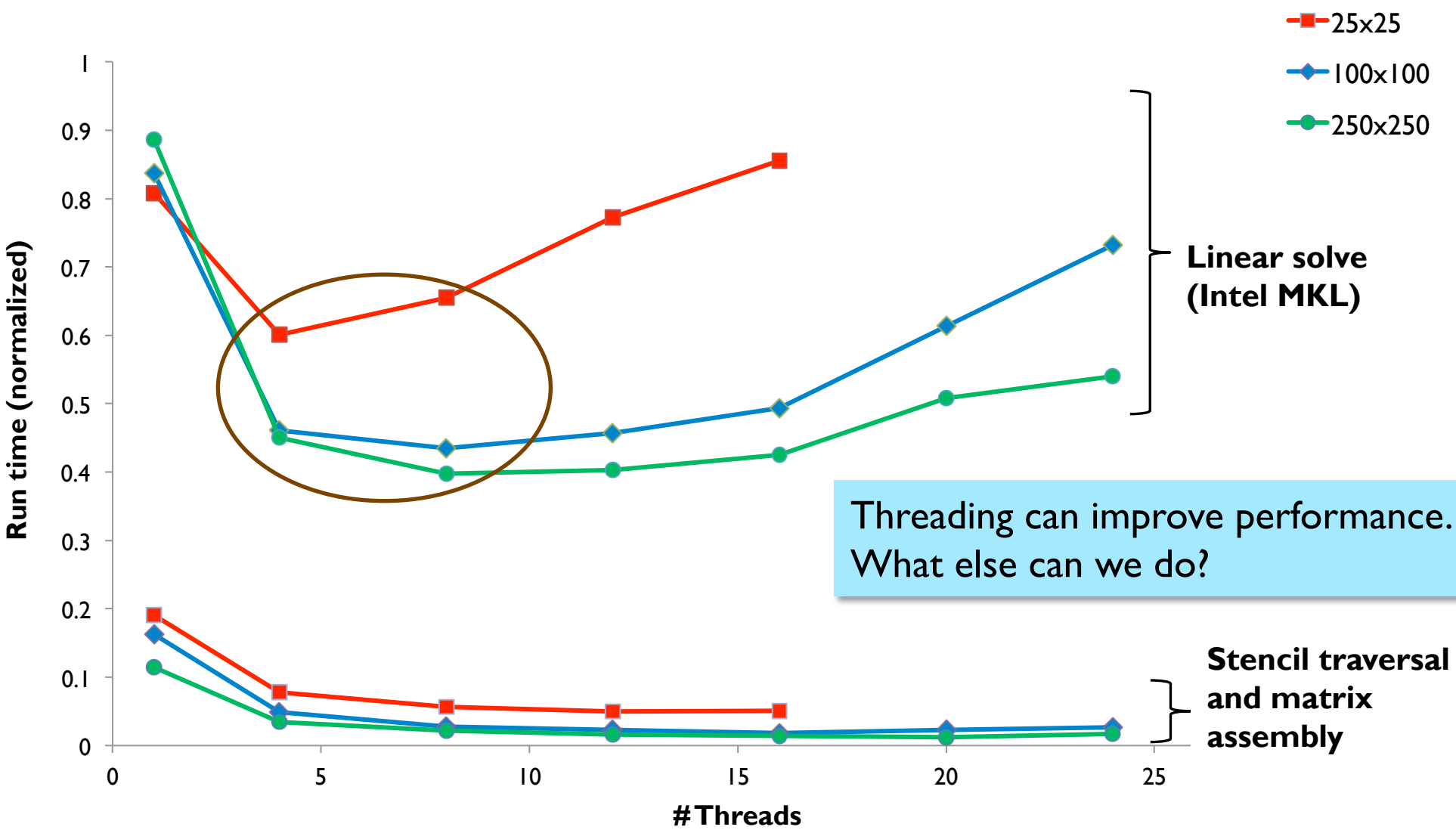




OpenMP threading leads to about 6-8X speedup.

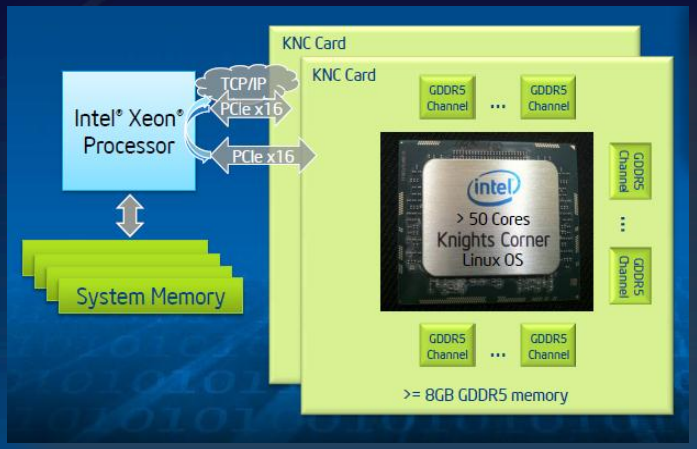


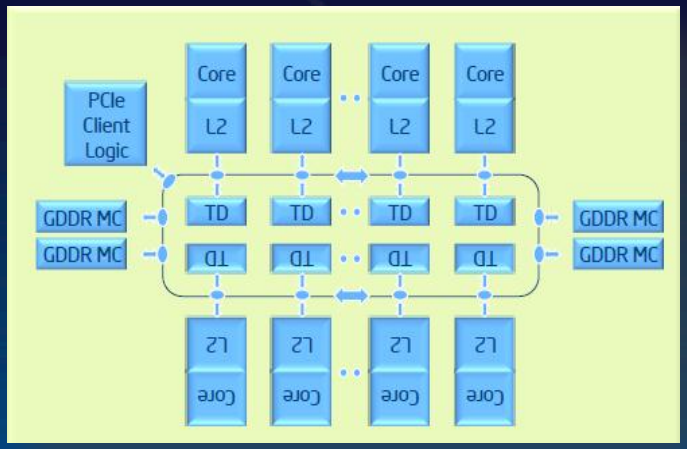
However, the performance bottleneck is the linear solve, not the matrix assembly!

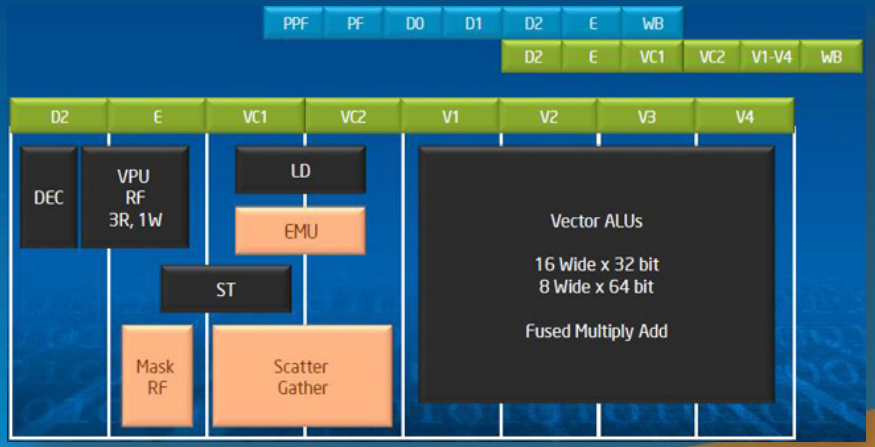


Threading can improve performance. What else can we do?

The Intel MIC is an example of emerging computer architecture.



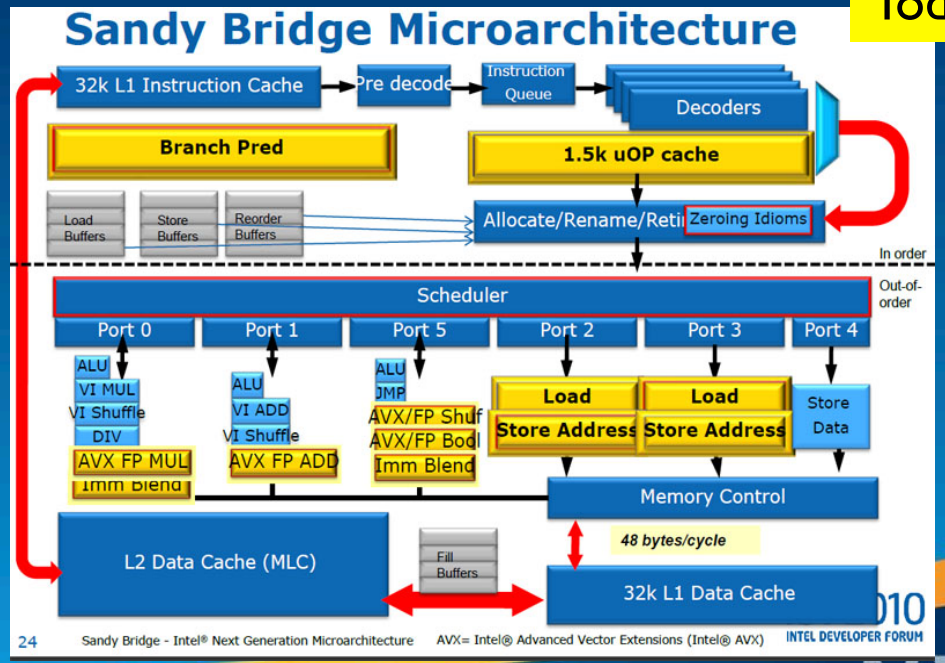




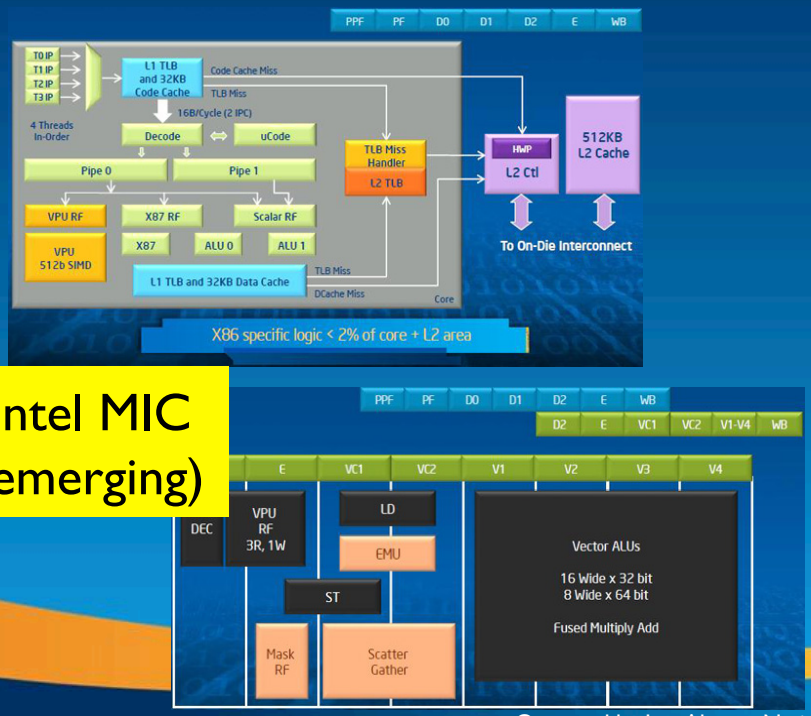
Intel MIC presents a prototype for emerging architectures that we can study today.

- Current parallelism methodologies (MPI, threading) will help with emerging architectures.
- Note: the Intel L1, L2 cache sizes for the Intel MIC are 32KB and 512 KB, respectively. This is structurally similar to what we have today (i.e. Sandy Bridge)
  - What about nontraditional (or forgotten/ignored) methodologies such as cache utilization and vectorization?

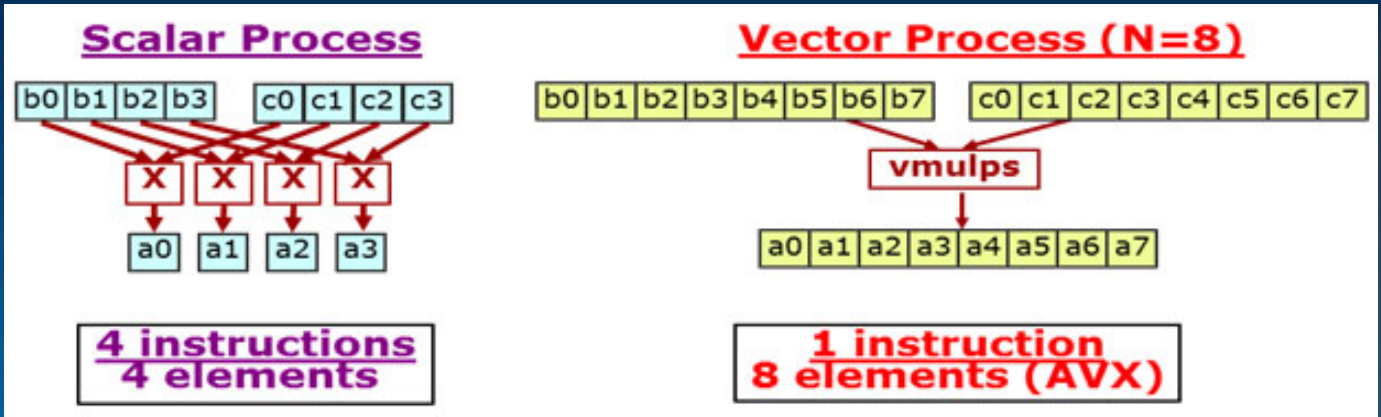
Today



Intel MIC (emerging)



What is vectorization?

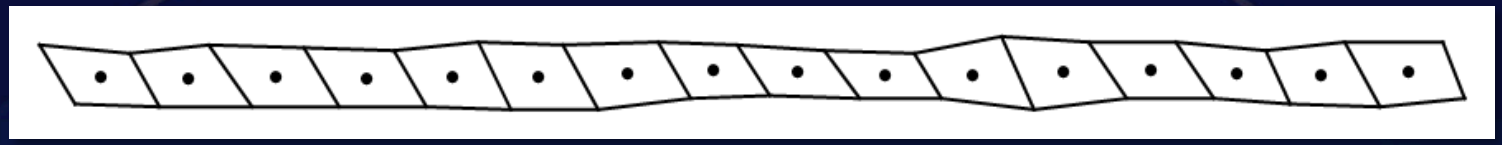


Issues confronting vectorization.

1. Independent calculations amenable to SIMD
   - vector math, vector operations
2. Vector loads and stores
3. Latency of operations, memory transfers
   - dependencies between operations
   - pipelines, independent execution units
   - out of order vs in order
4. Loop and subroutine overhead
5. Alignment of data
   - relative to memory address space and to cache lines
   - relative to other data in order to avoid unnecessary shuffles, cache spills and register spills
6. Locality of data
   - cache, registers

Wow. We now scale the algorithmic complexity way down to study these concepts.

Zone center calculation is a simple example that raises vectorization questions and challenges.



```
void ZoneCenters21A(const int nz1, const int vs1,
  const double *xv, double *xz){
  for(int i=0;i<nz1;++i)
    xz[i]=(xv[i]+xv[i+1]+xv[vs1+i]+xv[vs1+i+1])*0.25;
}
```

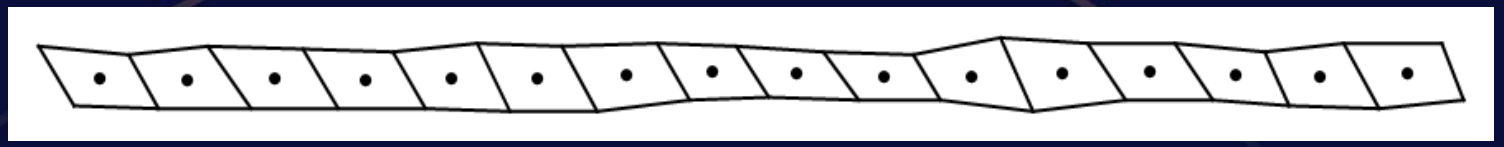
3 additions, 1 multiply per zone, vectorizes

Eliminating a common subexpression reduces number of additions but introduces a loop carried dependency.

```
void ZoneCenters21B(const int nz1, const int vs1,
  const double *xv, double *xz){
  double xl=xv[0]+xv[vs1];
  for(int i=0;i<nz1;++i){
    double xr=xv[i+1]+xv[vs1+i+1];
    xz[i]=(xr+xl)*0.25;
    xl=xr; // carry
  }
}
```

2 additions, 1 multiply per zone, compiler does NOT vectorize

One can still vectorize this loop with the loop carried dependency.



```
VECTOR xl=xv[0:3]+xv[vs1:vs1+3];
for(int i=0;i<nz1-7;i+=4){
  VECTOR xl_next=xv[i+4:i+7]+xv[vs1+i+4:vs1+i+7];
  VECTOR xr=VECTOR(xl[1:3],xl_next[0]);
  xz[i+0:i+3]=(xr+xl)*0.25;
  xl=xl_next; // carry
}
```

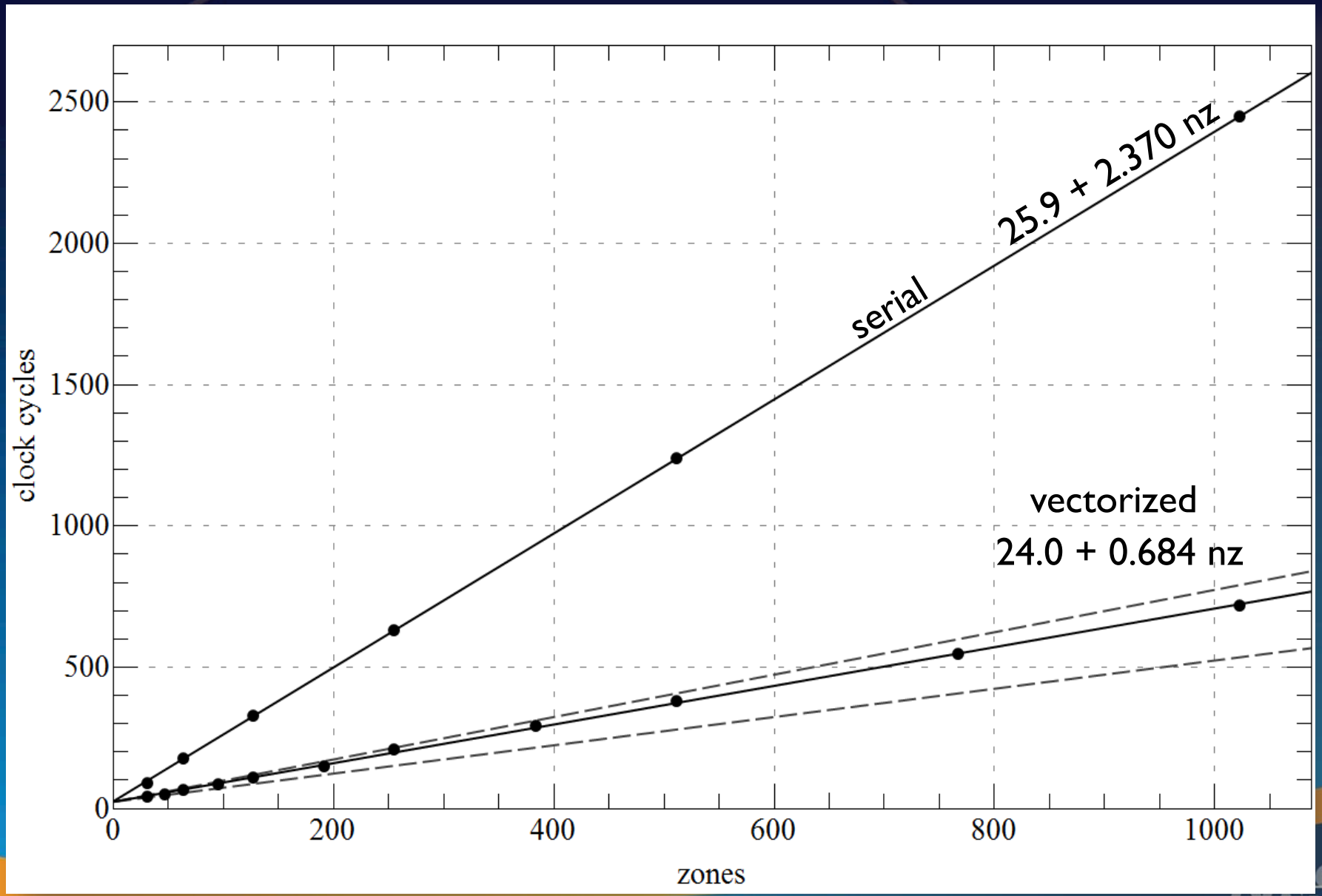
Pseudo code with vectors of 4 ignoring cleanup at end of loop.

How can we express this algorithm in code so the compiler gets it and generates optimal machine code?

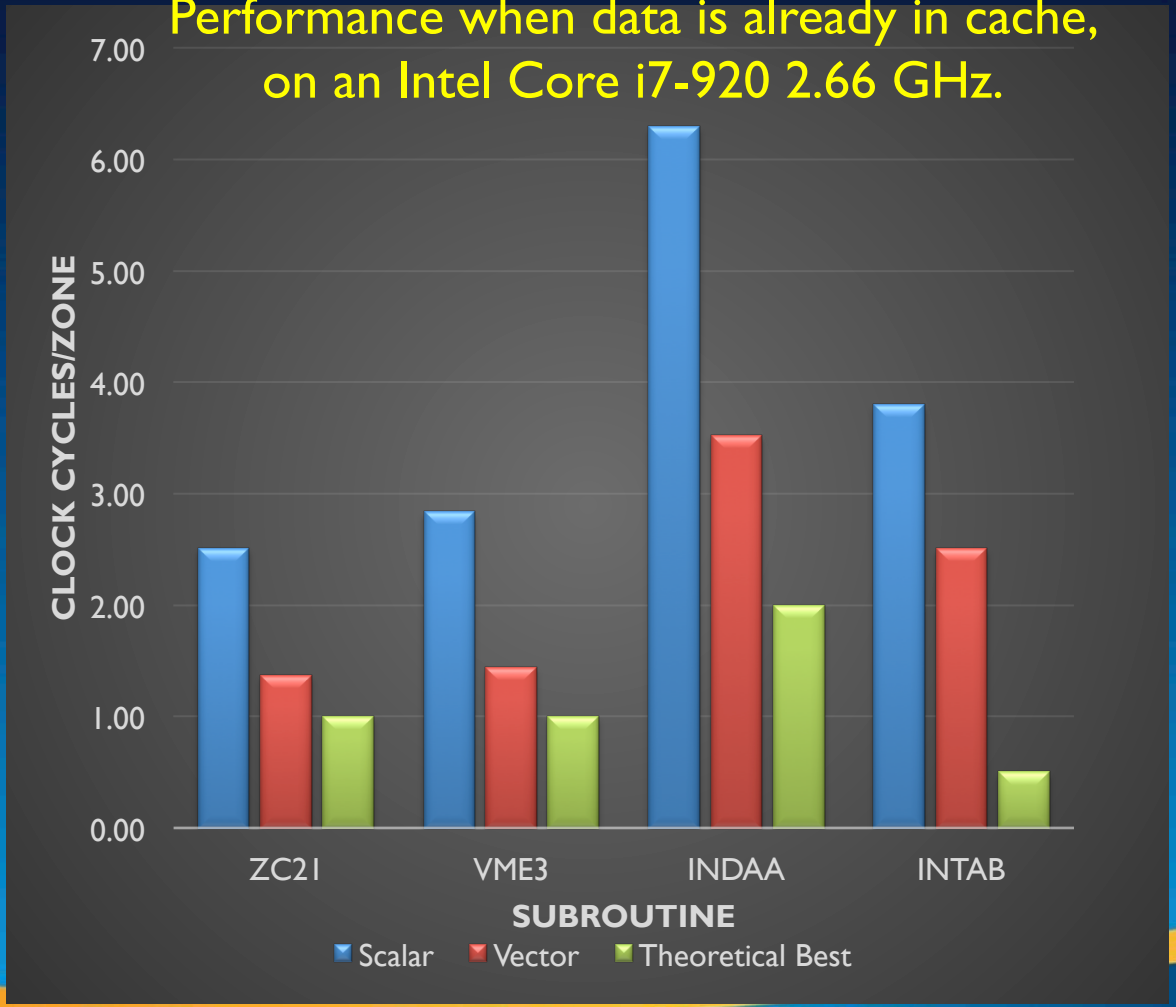
So we expect the zone center calculation to take 2 clock cycles per vector of 4 zones. This is my estimate of peak performance.

- Recall pipelines with 1 result per clock cycle per unit.
- Recall the 1 multiplication should be concurrent with an addition.
- Assuming 2 vector reads and 1 vector write per 2 clock cycles is okay.

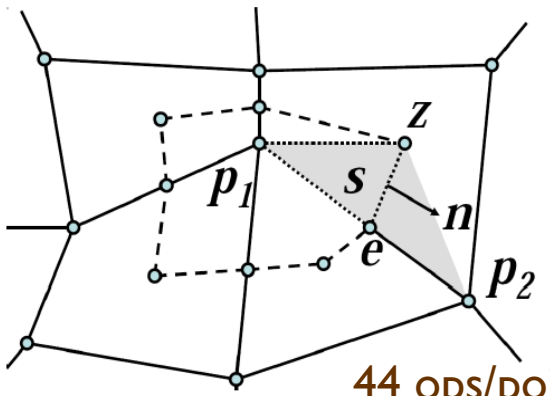
The vectorized zone-center calculation works.



Vectorization results on an Intel x86/64 architecture with SSE2.



A simple gradient kernel "vectorizes".

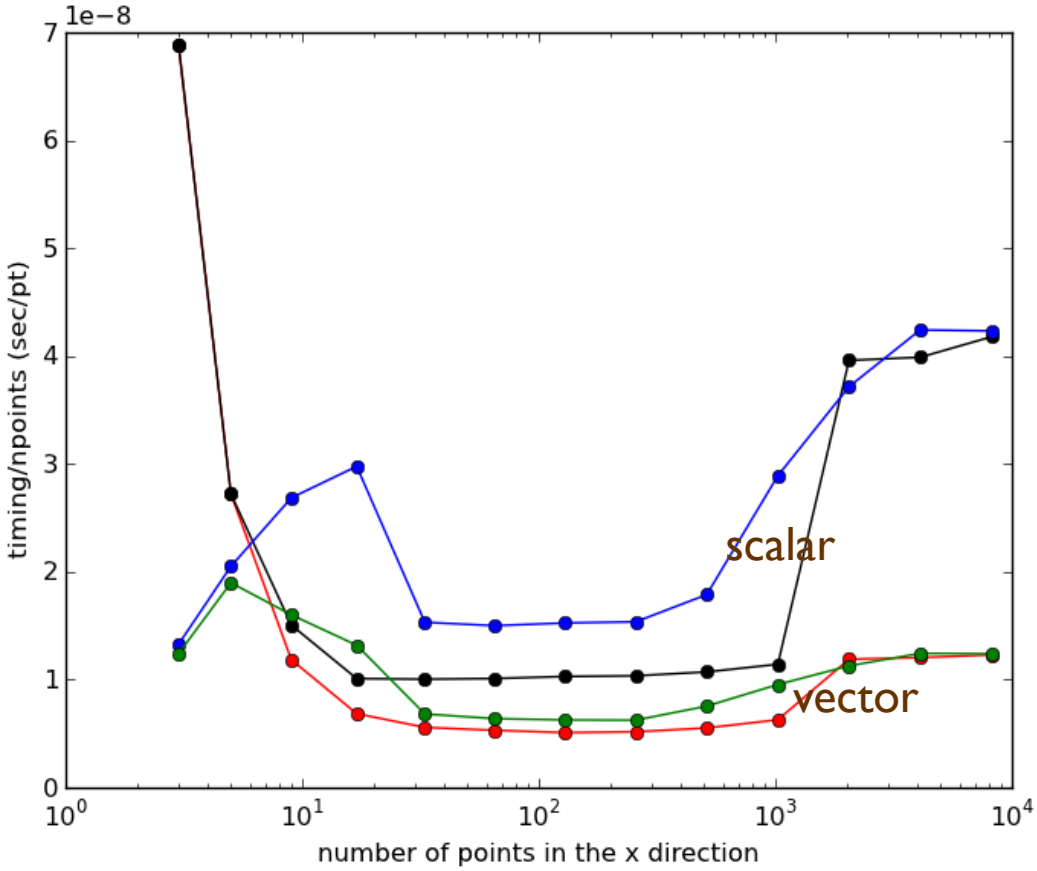


44 ops/point.

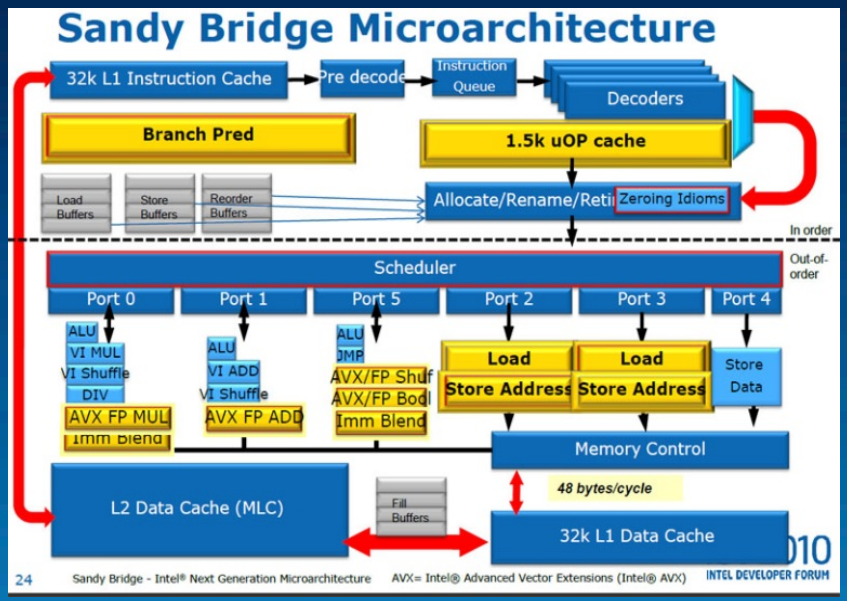
$$(\vec{\nabla}_h f)_p = \frac{1}{A_p} \sum_{s \in \mathcal{S}(p)} (\alpha_{s,p} f_z \hat{n}_s l_s)$$

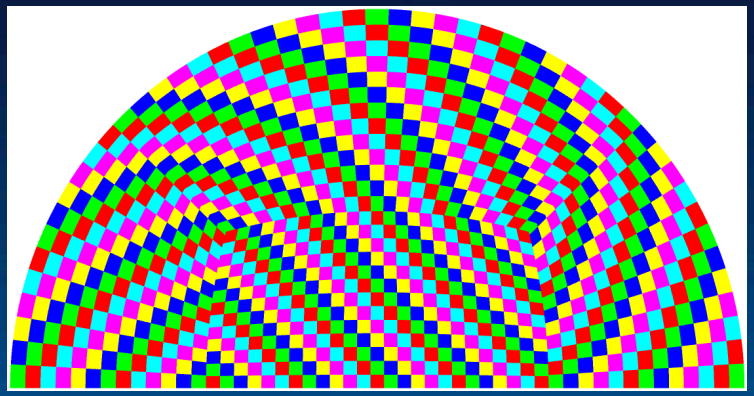
$$A_p = \sum_{s \in \mathcal{S}(p)} \frac{\|(\vec{x}_{p_2} - \vec{x}_{p_1}) \times (\vec{x}_z - \vec{x}_{p_1})\|}{4}$$

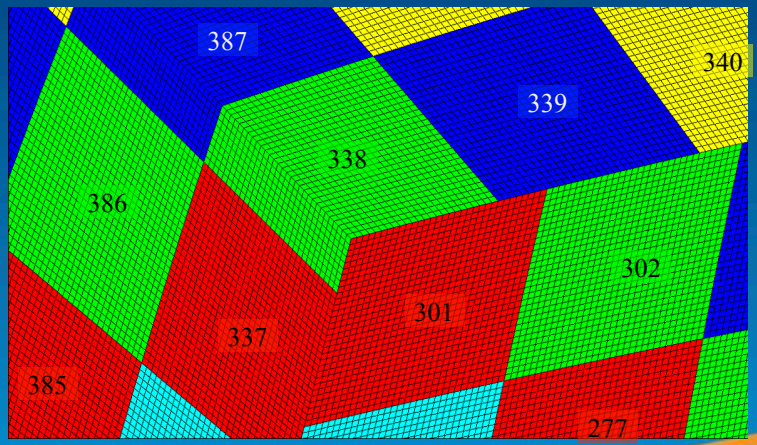
Timing for gradient calculation over square or rectangular meshes



Cache blocking, or tiling, is a memory management technique for performance.



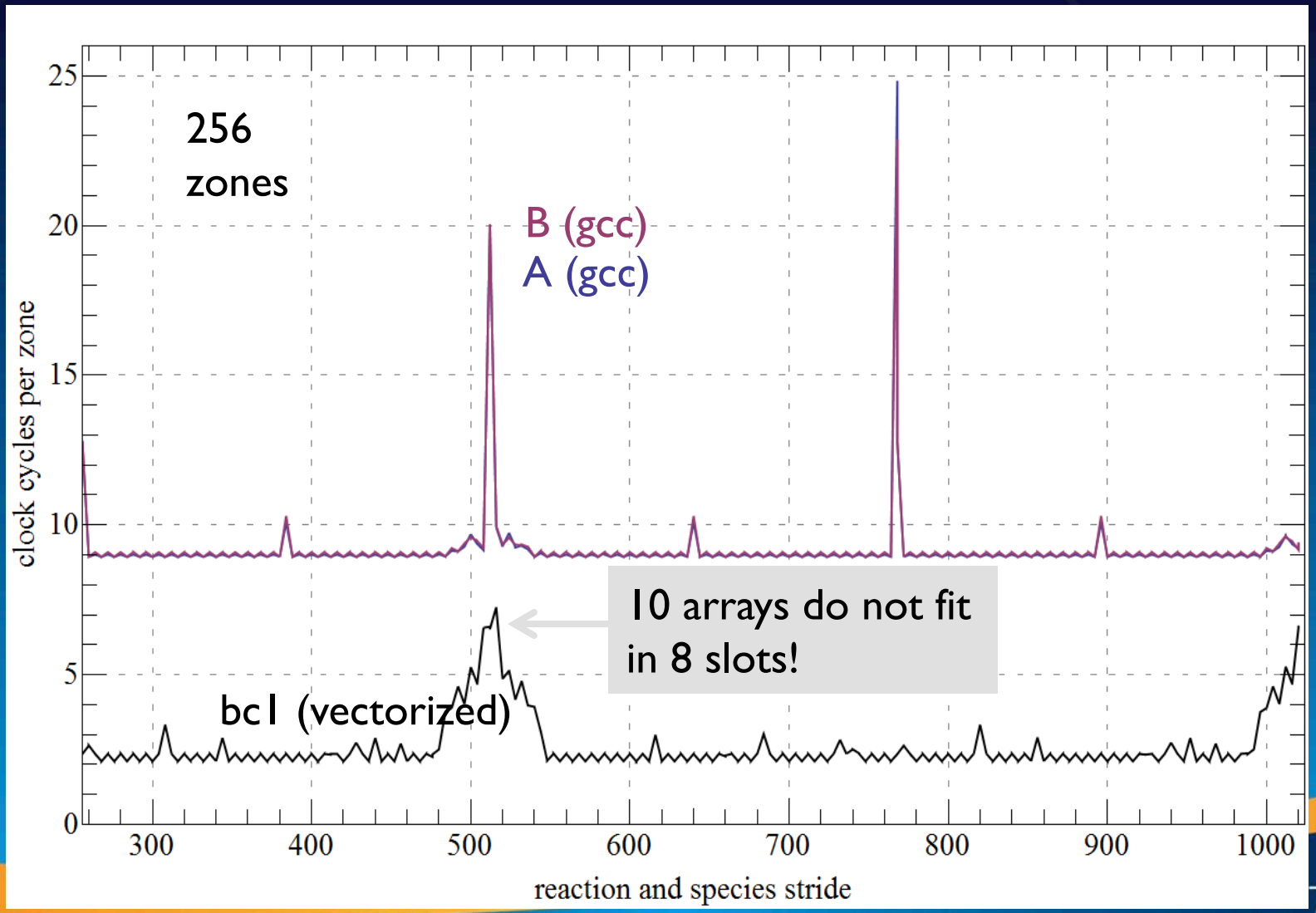




Profile your code!

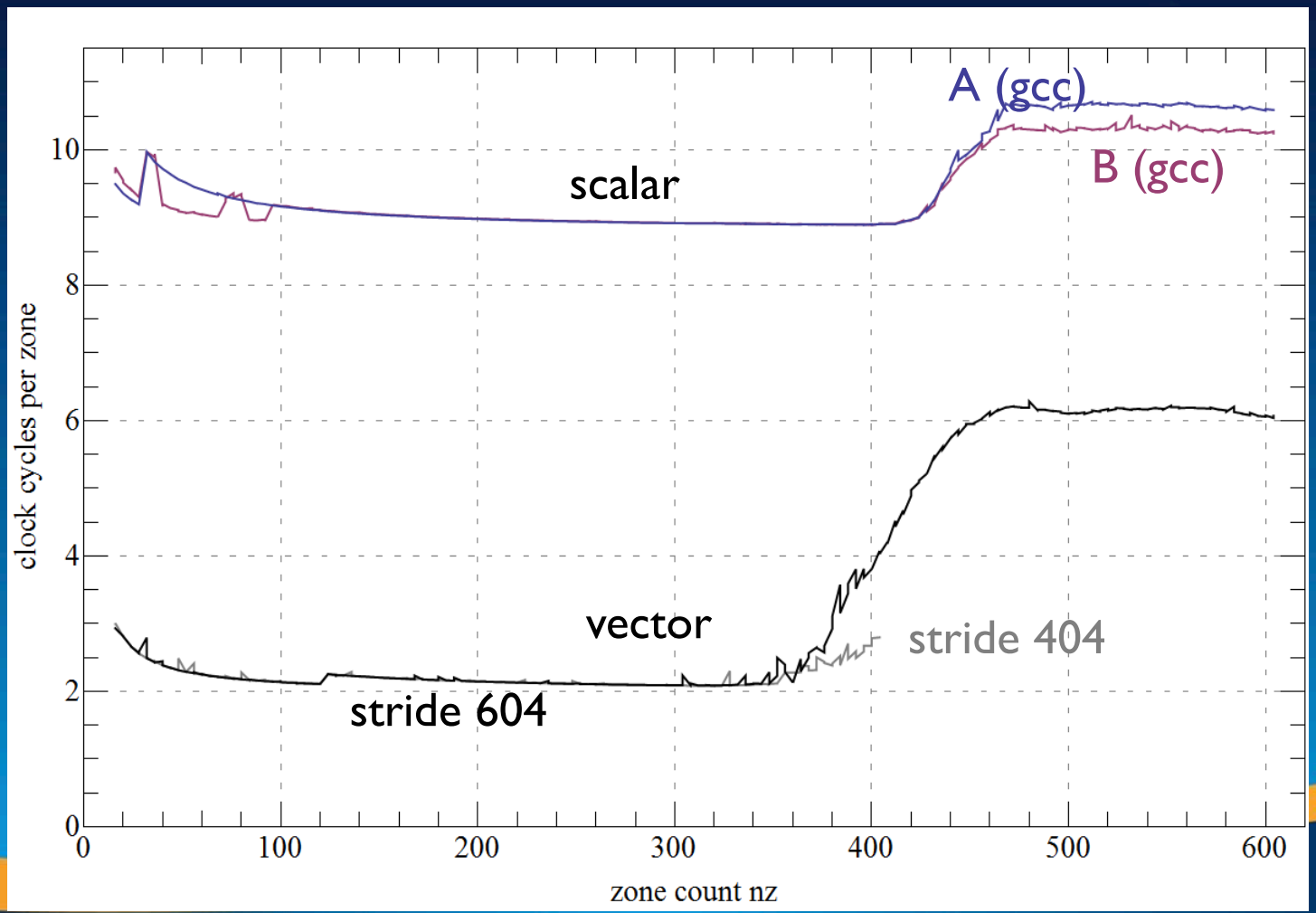
Array layout in memory matters.
Check mapping into 32 kB 8-way set associative cache.

Reaction rate calculations



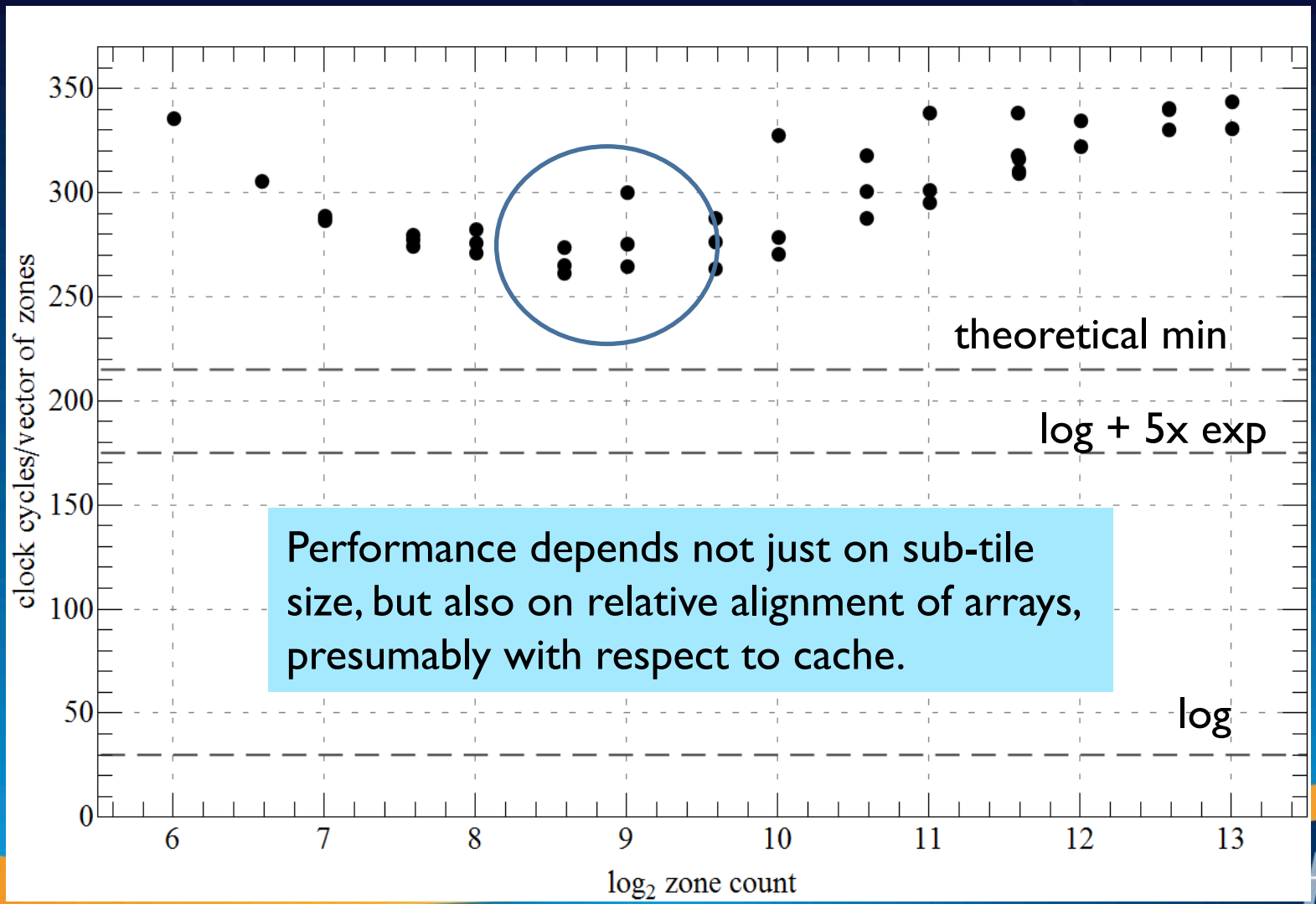
Performance is better with data in L1 cache. Can we extend good performance to data in L2 and beyond?

Reaction rate calculations



Another challenge is getting the data into L1 cache as needed.

Reaction rate calculation on 8192 zones, using vectorized subroutines, subtiled



Let's examine one step in cell-centered hydro to explore cache blocking.

- Move data to cache, and keep it there. Do as much physics as you can before moving to the next set of data.

sideAreaNormals

cornerVelocities

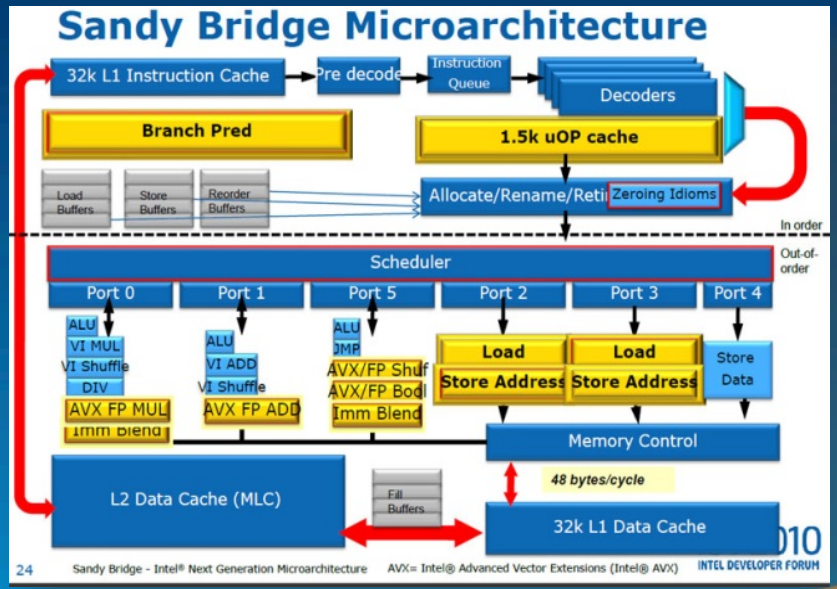
cornerUnitNormals

cornerDissipationVectors

impedances

cornerPressures

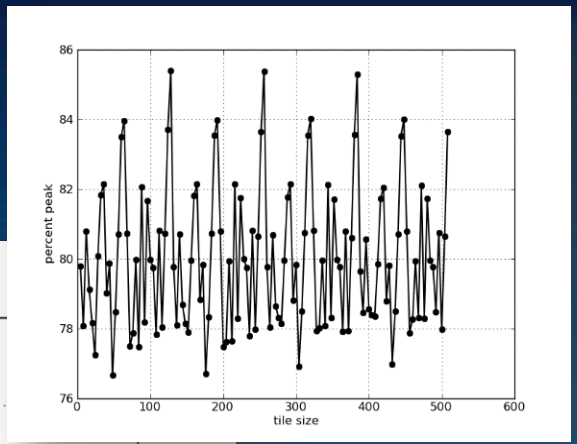
sideAdotN

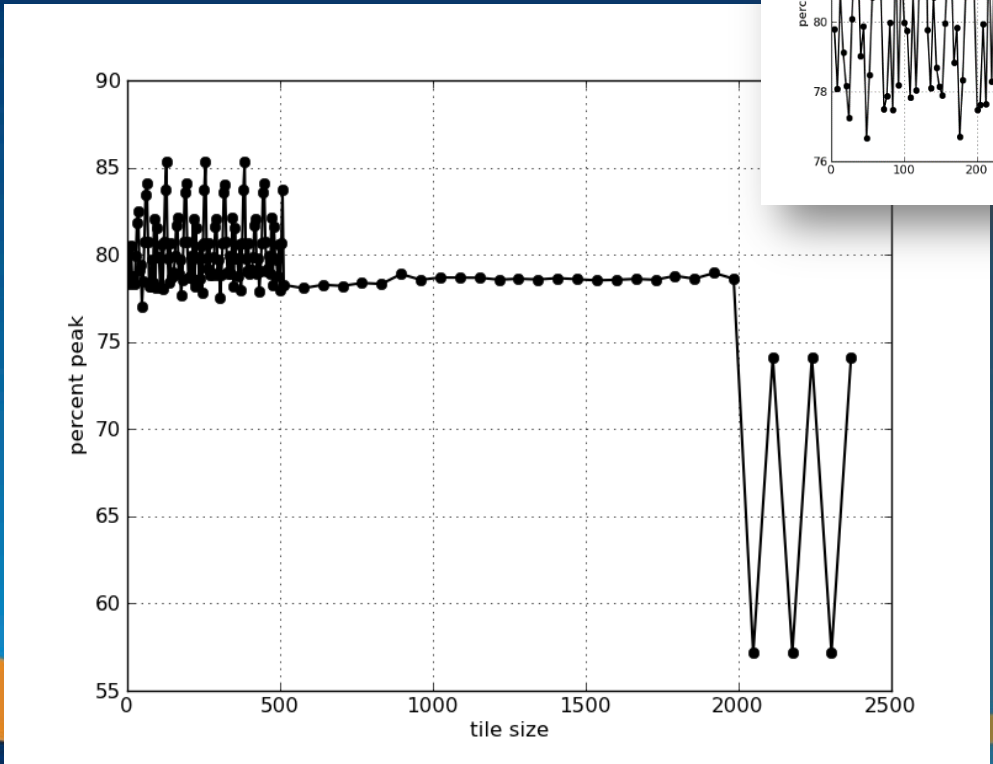


Example: a prototypical kernel from CCH exhibits a performance tradeoff with tile size.

- Cell-centered hydro: corner velocity calculation **cornerVelocities**

Percent peak versus tile size





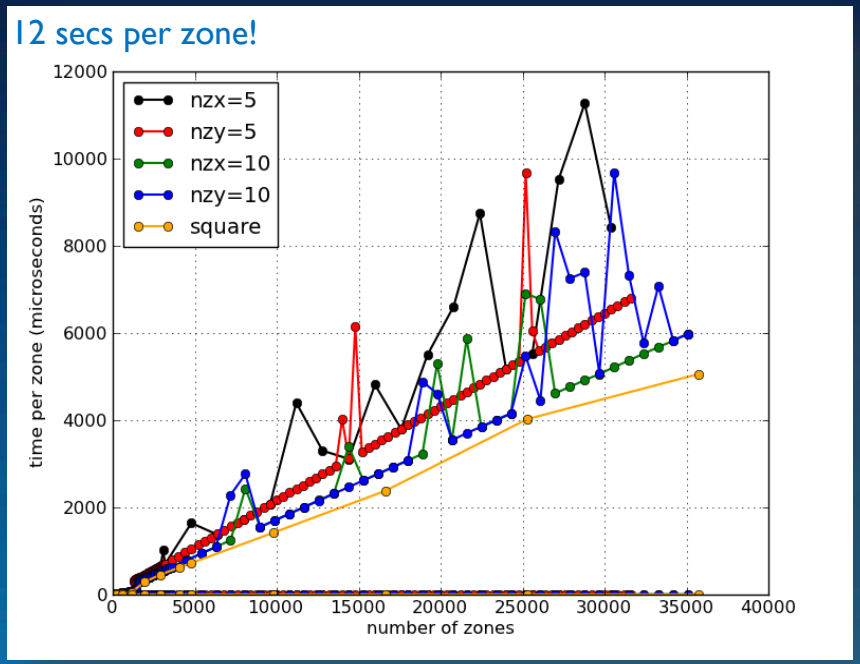
- Peak cycles per iteration = cpi = 4 (2 cycles/store, 2 stores/iteration)
- Iterations per zone = ipz = 8 (x&y, 4 corners)
- Number of zone rows = nzr = 1024
- Number of zone columns (sub-tile size) = nzc = variable
- Number of zones = nz = nzc*nzr
- Number of subtile repeat iterations nsri = 1000
- Peak number of cycles nc = cpi*ipz*nsri*nzr*nzc = 4*8*1000*1024*nzc = 32768000*nzc
- Peak runtime = pt = nc/10.4e9
- Peak cycles/zone w/ AVX pcpz = cpi*ipz*nsri = 2*8*1000 = 16000
- Actual cycles/zone w/ AVX = acpz = 2.6e9/ runtime/(cpi*ipz*nsri*nzr*nzc)

Back to our diffusion discretization, matrix assembly memory layout and tile size contribute to performance.

With full matrix storage, performance gets worse with the zone count.

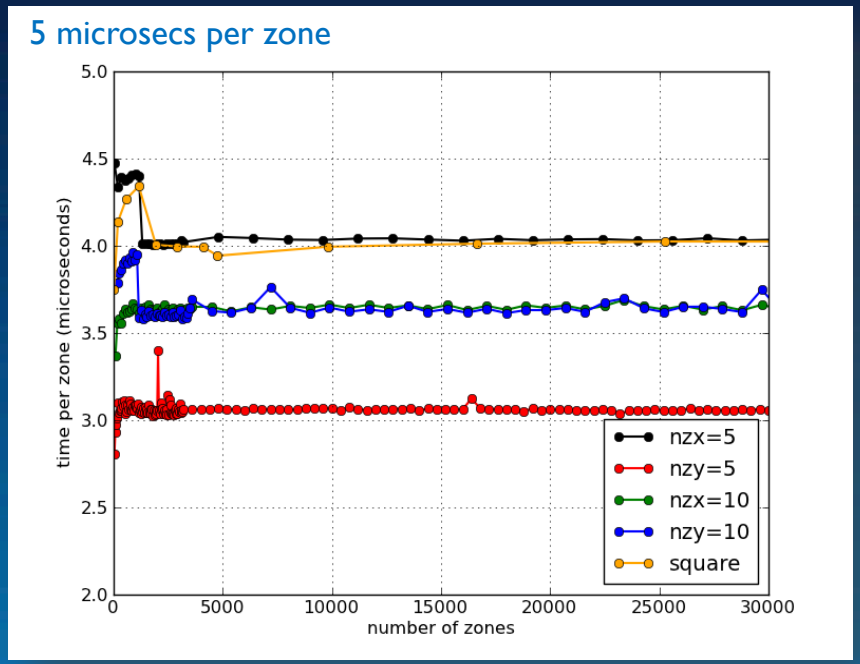
With sparse matrix storage, performance is much better, but the tile geometry becomes important.

12 secs per zone!



number of zones

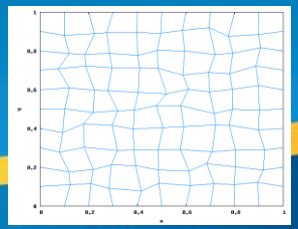
5 microsecs per zone



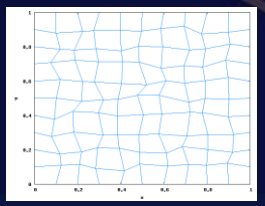
number of zones

matrix assembly for:

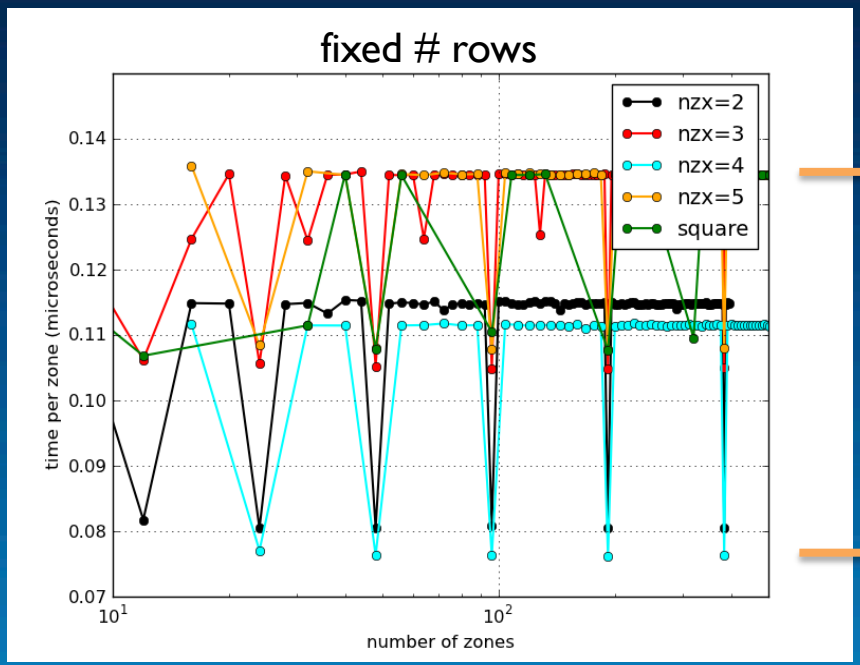
$$\mathscr{A}U = (\Omega + \mathscr{D}\mathscr{G})\,U = F$$

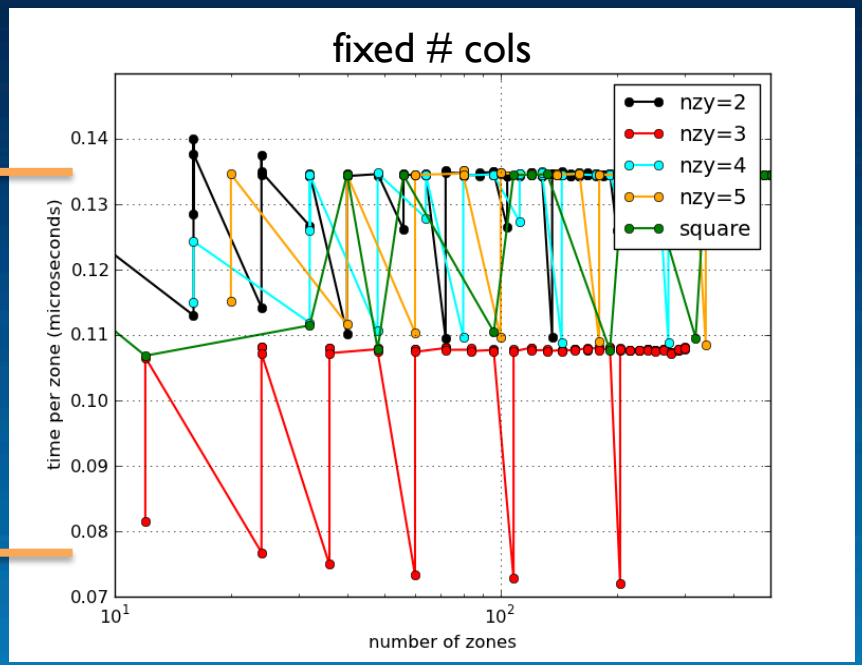


In the flux formulation, a subkernel of the matrix assembly demonstrates a different tradeoff with tile size.



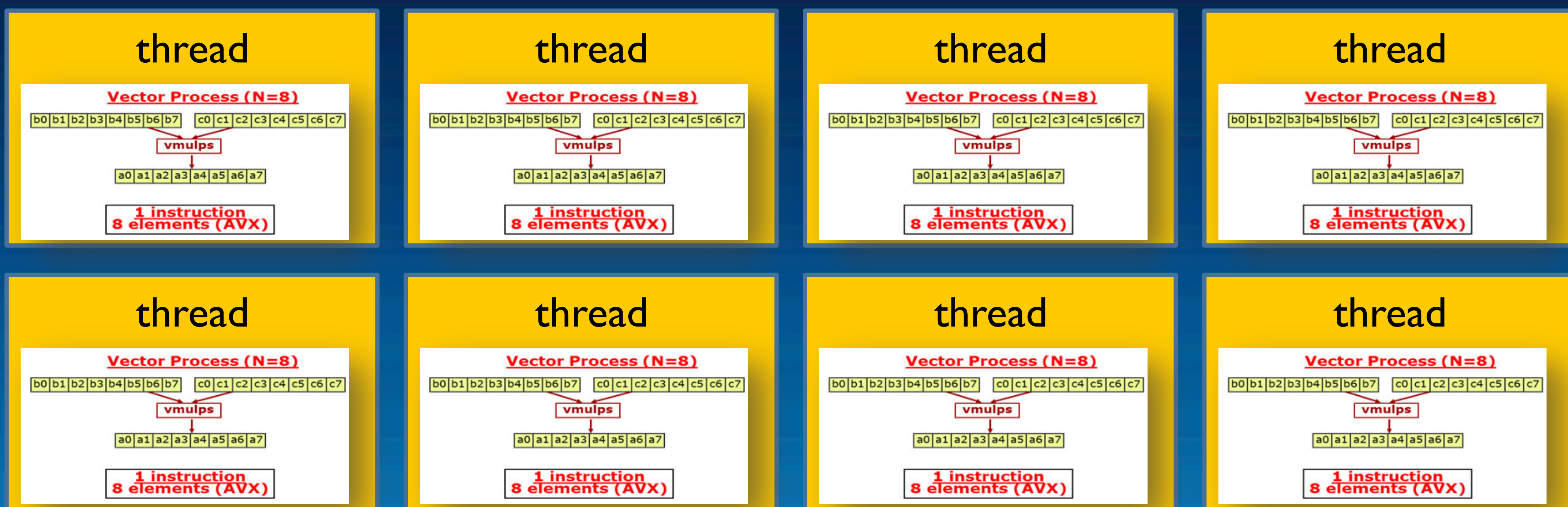
The subkernel displays a sensitivity to mesh geometry
and memory alignment that can affect performance by about 40%.





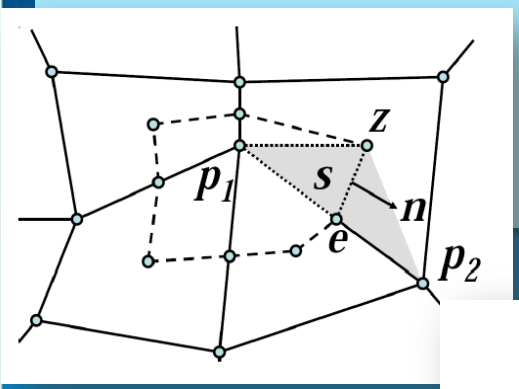
$$(\mathscr{S} + \mathscr{D}\dagger\mathscr{M}\Omega^{-1}\mathscr{D})\mathbf{W}^{n+1} = \mathscr{D}\dagger\mathscr{M}\Omega^{-1}F^{n+1}$$

Can we combine threading and vectorization?



Consider our point-centered gradient of a zone-centered field.

```
for(int i=0; i<tmp; ++i){          // Omit the first and last point in row j.
          pareai = 0.5/((0.5*(px[i+ip2]+px[i+ip1])-px[i+ip0])*(py[i+ip2] - py[i+ip1]) +
                        (0.5*(px[i+ip3]+px[i+ip2])-px[i+ip0])*(py[i+ip3] - py[i+ip2]) +
                        (0.5*(px[i+ip4]+px[i+ip2])-px[i+ip0])*(py[i+ip4] - py[i+ip3]) +
                        (0.5*(px[i+ip1]+px[i+ip3])-px[i+ip0])*(py[i+ip1] - py[i+ip4]));
          pgradx[i+ip0] = ((py[i+ip2] - py[i+ip1]) * zfield[i+iz1] +
                           (py[i+ip3] - py[i+ip2]) * zfield[i+iz2] +
                           (py[i+ip4] - py[i+ip3]) * zfield[i+iz3] +
                           (py[i+ip1] - py[i+ip4]) * zfield[i+iz4]   ) * pareai;
          pgrady[i+ip0] = ((px[i+ip1] - px[i+ip2]) * zfield[i+iz1] +
                           (px[i+ip2] - px[i+ip3]) * zfield[i+iz2] +
                           (px[i+ip3] - px[i+ip4]) * zfield[i+iz3] +
                           (px[i+ip4] - px[i+ip1]) * zfield[i+iz4]   ) * pareai;
```

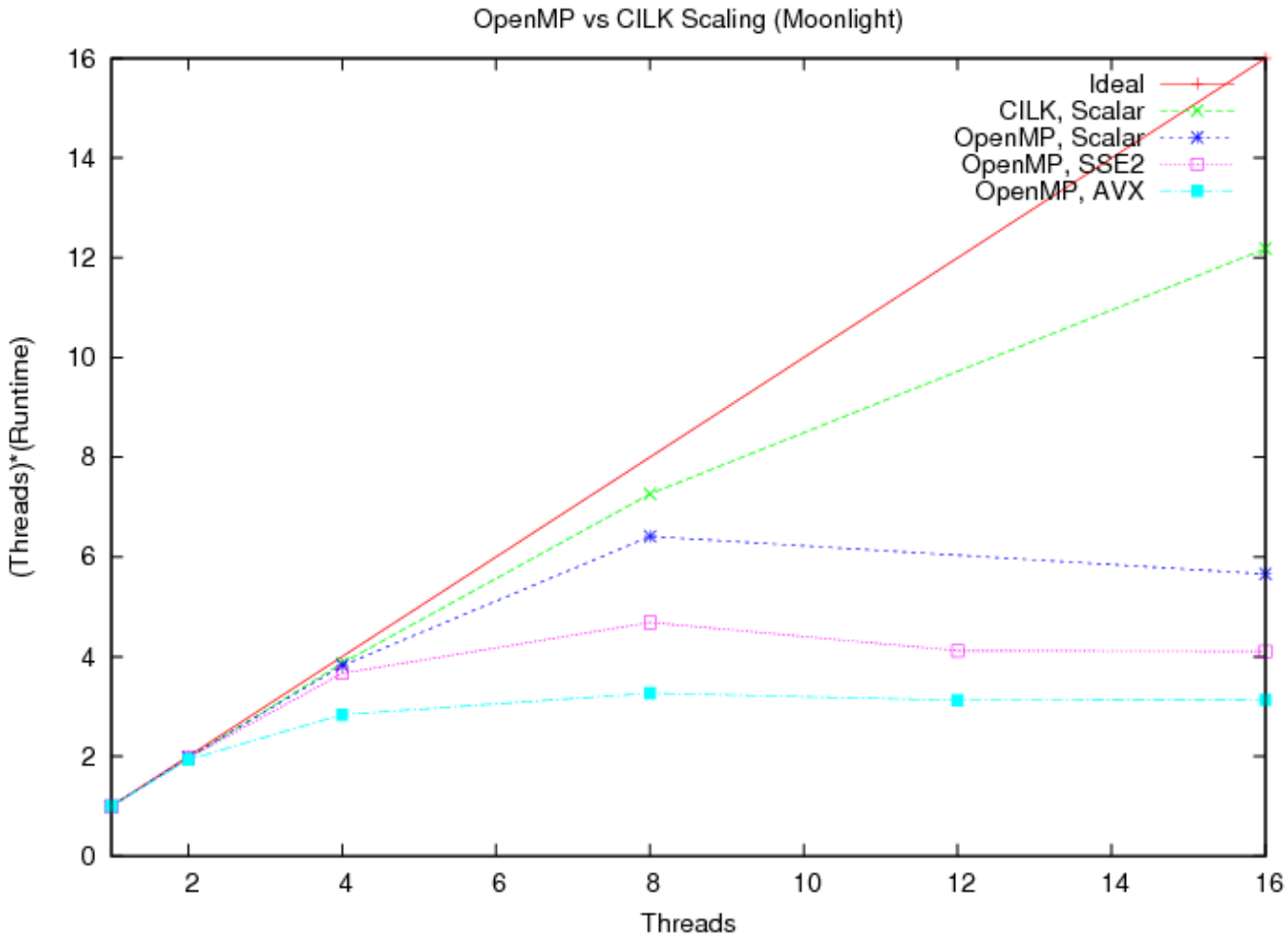


$$(\vec{\nabla}_h f)_p = \frac{1}{A_p} \sum_{s \in \mathcal{S}(p)} (\alpha_{s,p} f_z \hat{n}_s l_s)$$

$$A_p = \sum_{s \in \mathcal{S}(p)} \frac{\|(\vec{x}_{p_2} - \vec{x}_{p_1}) \times (\vec{x}_z - \vec{x}_{p_1})\|}{4}$$

- Total raw operations per vector of points:
  - Multiplies: 18
  - Divides: 1
  - Adds/Subtracts: 29
  - Loads / Stores: 14 / 2

Vectorization and threading are in conflict for this test algorithm.



- Vectorizes well only in serial or with 2 or 4 threads. However, our threading domains are specified in direct competition with vector domains.

Stay tuned!

What is the ideal situation?

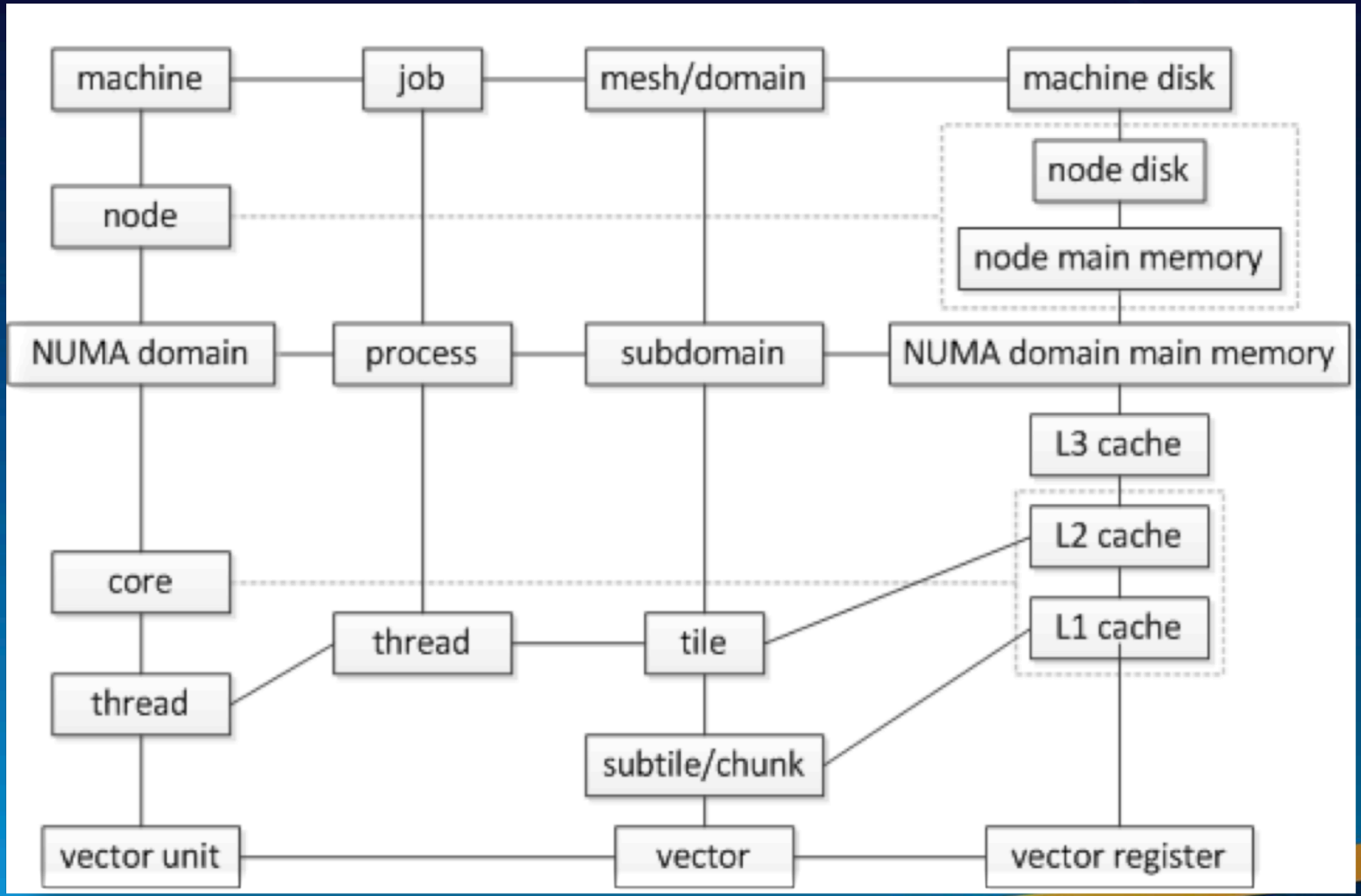
Code developers decompose their algorithms into a flexible subroutines that

- work in parallel across cores and nodes (threads and MPI),
- vectorize,
- use distinct floating point units concurrently,
- fill the floating point pipelines,
- to compute their results nearly as fast as theoretically possible
- on blocks of data that fit in L1 or L2 cache.

Flexible means the subroutines can be rearranged, replaced, called with different arguments (counts, strides or data layout, data pointers), and/or extended for memory management.

Software-hardware performance optimization can take place across the "landscape".

processing software task task construct storage



Concluding remarks

- We have a lot of work to do!
- We have explored the performance of prototypical algorithms in light of threading, vectorization, and cache blocking.
  - Unstructured mesh algorithms accelerated through threading.
  - Simple algorithms, gradient methods, and diffusion routines highlight benefits of vectorization and cache blocking, while presenting new (or old) challenges in memory layout and management.
- Some algorithms trivially admit to *some* acceleration through these techniques, especially techniques in isolation.
  - Other algorithms require care or redesign in their memory layout.
  - Combining techniques also requires care.
  - Subdivide algorithms into subkernels and profile!
- Performance profiling may help characterize memory dependencies on acceleration through cache blocking, enabling acceleration through careful organization of memory (with minimal change to code), even for complex algorithms.

References:

- M. Shashkov and S. Steinberg, Solving Diffusion Equations with Rough Coefficients in Rough Grids, Journal of Computational Physics, 129, pp. 383-405, (1996).
- J. Hyman, M. Shashkov and S. Steinberg, The Numerical Solution of Diffusion Problems in Strongly Heterogenous Non-Isotropic Materials, Journal of Computational Physics, 132, pp. 130-148, (1997).
- http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner
- http://software.intel.com/en-us/articles/cache-blocking-techniques
- http://software.intel.com/en-us/articles/vectorization-essential
- http://www.hardwaresecrets.com/article/Inside-the-Intel-Sandy-Bridge-Microarchitecture/1161/1

Backup slide 2