Cache-aware Scheduling and Performance Modeling with LLVM-Polly and Kerncraft

Julian Hammer [RRZE] <julian.hammer@fau.de>, Johannes Doerfert [UdS] <doerfert@cs.uni-saarland.de>,
Georg Hager [RRZE], Gerhard Wellein [RRZE] and Sebastian Hack [UdS]

[RRZE] Regional Computing Center Erlangen
[UdS] Saarland University

Outline

1. Motivation
2. Background
   - Memory Hierarchy
   - Cache Blocking
   - Layer Conditions (and example)
   - Performance Modelling & Kerncraft
   - Polyhedral Representation
3. Implementation
   - Polly Layer Conditions
   - Kerncraft Export
4. Evaluation
5. Outlook & Conclusion

Motivation

Analytical models and compiler infrastructure a great match.

- Numeric kernels–in particular–stencils may profit from reduced memory and inter-cache traffic through spatial blocking
- Tedious implementation work for developer
- Block size selection requires insight into computer architecture and access pattern OR exhausting parameter studies

This is **work-in-progress**.

We show the theory, approach, unadorned results and current problems.

Background

Memory Hierarchy

Loads cause misses along all caches until they “hit” the required data.

Each level keeps all data of the next (smaller) cache and replaces least-recently-used (LRU) data.

HW prefetcher loads from Main Memory (Mem) to L3.

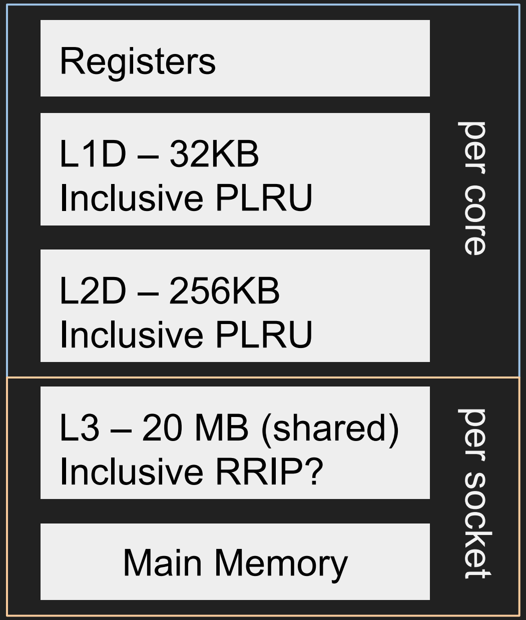


Illustration of Ivy Bridge Memory Hierarchy

Stencil Example

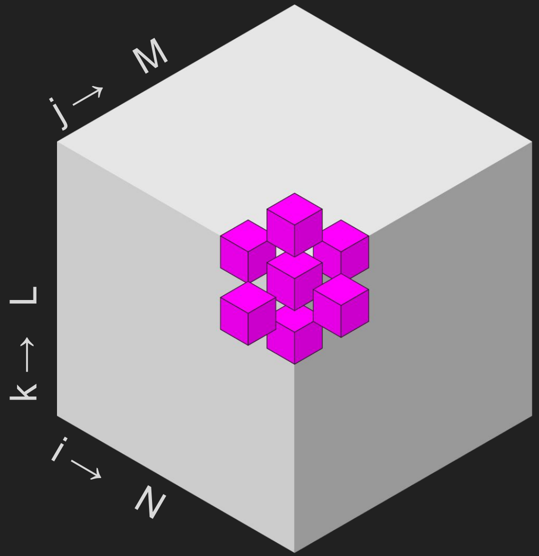
Offset access pattern, typically in 2D or 3D

3D 7-Point Stencil example:

- N*M*L*2 * 8 byte memory requirement (dp)
- 7 load and 1 store stream total

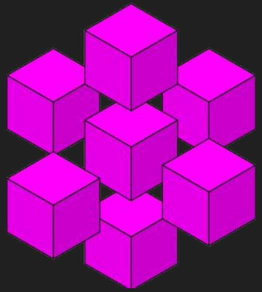
How many misses?

```
for(int k=1; k<L-1; k++)
 for(int j=1; j<M-1; j++)
  for(int i=1; i < N-1; i++)
   b[k*N*M+j*N+i] = (
     a[k*N*M+(j-1)*N+i] + a[k*N*M+(j+1)*N+i] +
     a[k*N*M+j*N+(i-1)] + a[k*N*M+j*N+i] +
     a[k*N*M+j*N+(i+1)] + a[(k-1)*N*M+j*N+i] +
     a[(k+1)*N*M+j*N+i]) * s;
```

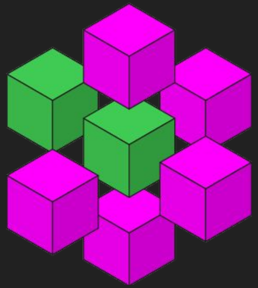


Layer Conditions[0] – Idea

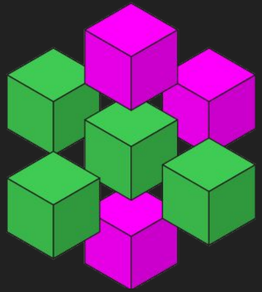
Model assumes inclusive LRU caches.



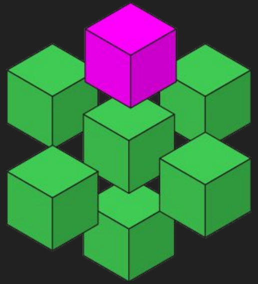
No cache
0 hits
(theoretical)



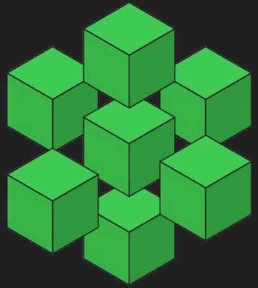
Reuse in 1D
2 hits



Reuse in 2D
4 hits



Reuse in 3D
6 hits

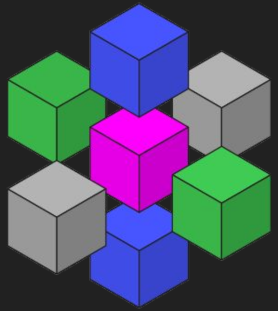


Full caching
7+1 hits

[0] Hammer et al, Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels

Layer Conditions

Analytically derived conditions for cache hit and misse from access offsets.



1. Compile list of access offsets:

   L = {1, 1, N-1, N-1, (M-1)*N, (M-1)*N, ∞, ∞}

1	from green to pink offsets
N-1	from green to grey offsets
(M-1)*N	from blue to grey offsets
∞	from last access to a[] and b[]

2. For each tail t in L, we get:

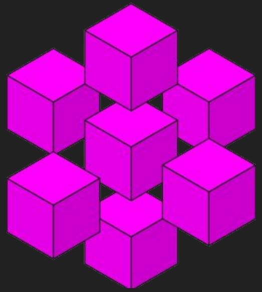
   If cache > $(\sum \{ e \mid e \in L, e <= t \} + |\{ e \mid e \in L, e > t \}| * t)*s$, then we expect

   $|\{ e \mid e \in L, e <= t \}|$ hits

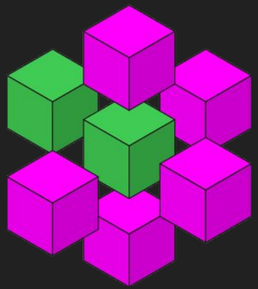
   $|\{ e \mid e \in L, e > t \}|$ misses

Layer Conditions

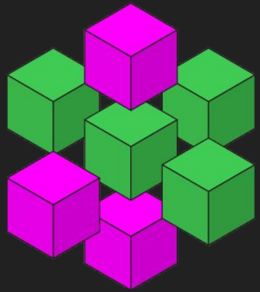
Model assumes inclusive LRU caches



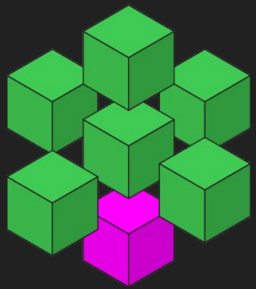
No cache
0 hits
(theoretical)



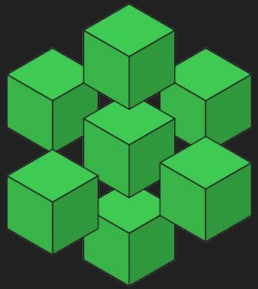
Reuse in 1D
2 hits
cache > 7*2*8 B
with tail = 1



Reuse in 2D
4 hits
cache > (6N-4)*8 B
with tail = N-1



Reuse in 3D
6 hits
cache > (4NM-2N)*8 B
with tail = (M-1)*N



Full caching
7+1 hits
cache > 2NML*8 B

Layer Conditions – Setup

1. Collect (symbolic) accesses in loop nest (A)
2. Sort A
3. Compute access offsets (L)
4. For each array add one infinity (oo) to L
5. Sort L

```
# ordered accesses from 3D-7pt
A = sorted([
    a+(k-1)*N*M+j*N+i,
    a+k*N*M+(j-1)*N+i, a+k*N*M+j*N+i-1,
    b+k*N*M+j*N+i, a+k*N*M+j*N+i+1,
    a+k*N*M+(j+1)*N+i, a+(k+1)*N*M+j*N+i ])

L = [oo]  # begin with one infty in list
for acs1, acs2 in zip(A[:-1], A[1:]):
    # offsets between “consecutive” accesses
    diff = acs2 - acs1
    if a in diff and b in diff:
        diff = oo
    L.append(diff)
L.sort()

L = [oo, oo, (N-1)*M, (N-1)*M, N-1, N-1, 1, 1]
```

Layer Conditions – Evaluation

A different cache hit/miss situation is expected for each non-infinity tail in L:

- If cache is larger then *'sum over all l in L with l <= tail plus tail times the number of l > tail'*, than we expect to observe
- *'number of l <= tail'* cache hits
- *'number of l > tail'* cache misses

```
layer_conditions = []
for tail in set(L):
    if tail == oo: continue
    lc = {
        'cache_requirement': (
            # cached elements / hits
            sum([ l for l in L if l <= tail ]) +
            # uncached elements / misses
            len([ l for l in L if l > tail ])*tail
        ) * element_size,
        'cache_hits': len([ l for l in L if l <= tail ])
        'cache_misses': len([ l for l in L if l > tail ])}
    print("For caches >= {cache_requirement} bytes,
           expect {cache_hits} hits and
           {cache_misses} misses".format(**lc))
    layer_conditions.append(lc)
```

https://rrze-hpc.github.io/layer-condition/

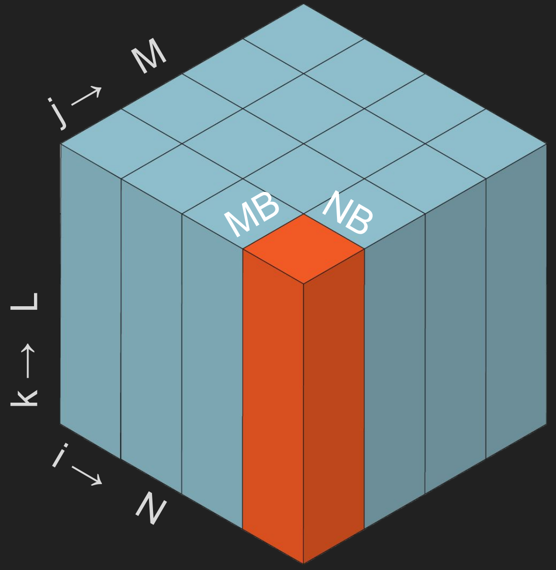
Cache Blocking

Strategy to reduce memory and inter-cache traffic, by traversing the data in blocks (or tiles), reuse is increased.

From layer conditions:
3D: 2 misses if 32*N*M - 16*N < cache
2D: 4 misses if 48*N - 32 < cache

Choose NB and MB accordingly, while maximizing N (to avoid short inner-loop overheads).

3d7pt: 4 misses in 32KB L1, 2 misses in 20MB L3
NB < 682 && NB*MB < 655360



Performance Modelling

Prediction of the actual performance requires more than predictions of data transfers. Performance models combine memory models (e.g., layer conditions) with execution models (e.g., peak flops or IACA analysis) to an overall runtime.

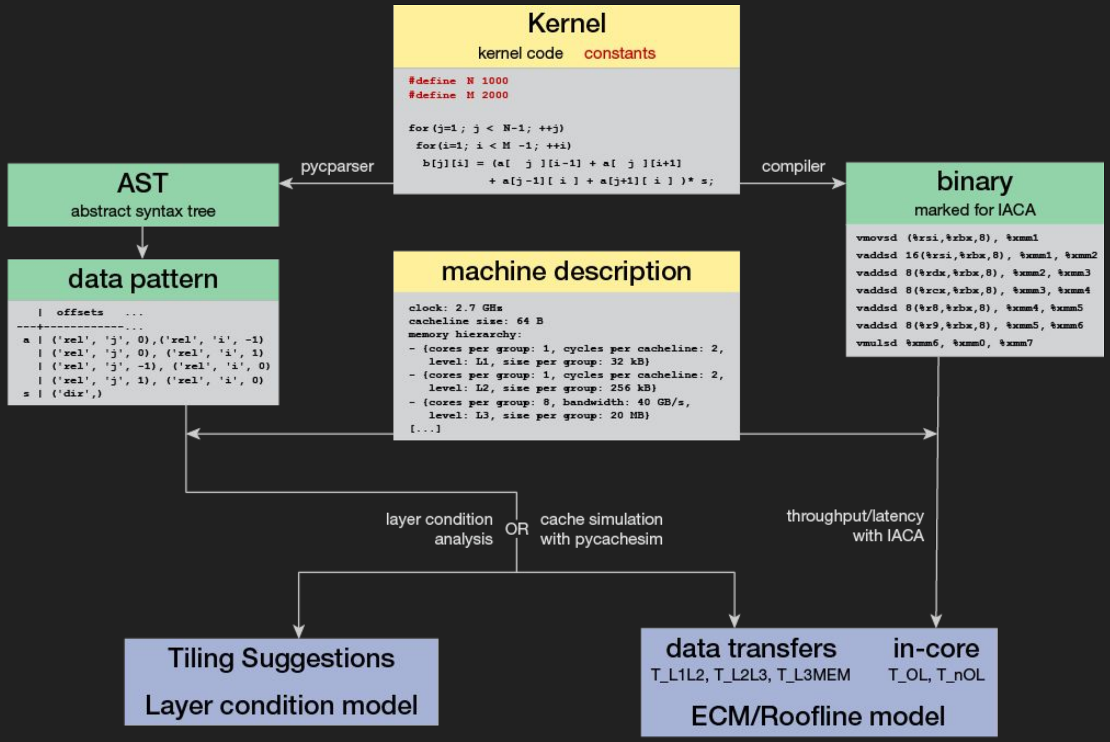
Execution-Cache-Memory and **Roofline** models allows classification into memory and compute bound, to avoid tiling overheads.

-> Future work / to be implemented

Kerncraft[1]

Automatic performance model toolkit, based on static analysis and cache simulation.

Predicts loop runtime based on Roofline and ECM model.



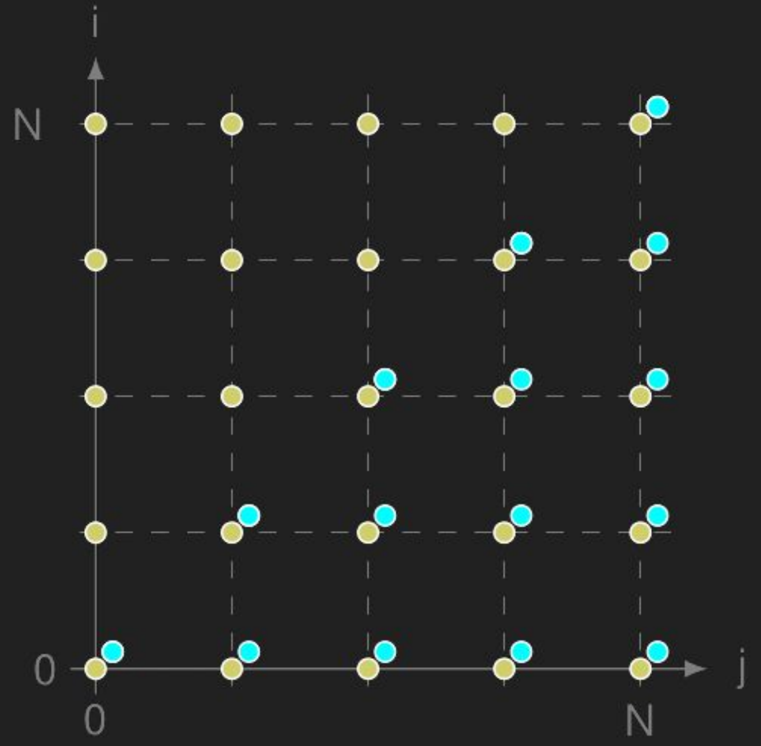
[1] https://github.com/RRZE-HPC/kerncraft

Polyhedral Representation

```
for (int i = 0; i <= N; i++)
  for (int j = 0; j <= N; j++) {
S:  A[i][j] = /* ... */;
    if (j <= i)
P:    A[i][j]+= A[j][i];
  }
```

$$\mathcal{I}_S = \{(S, (i, j)) \mid 0 \leq i \leq N \wedge 0 \leq j \leq N\}$$

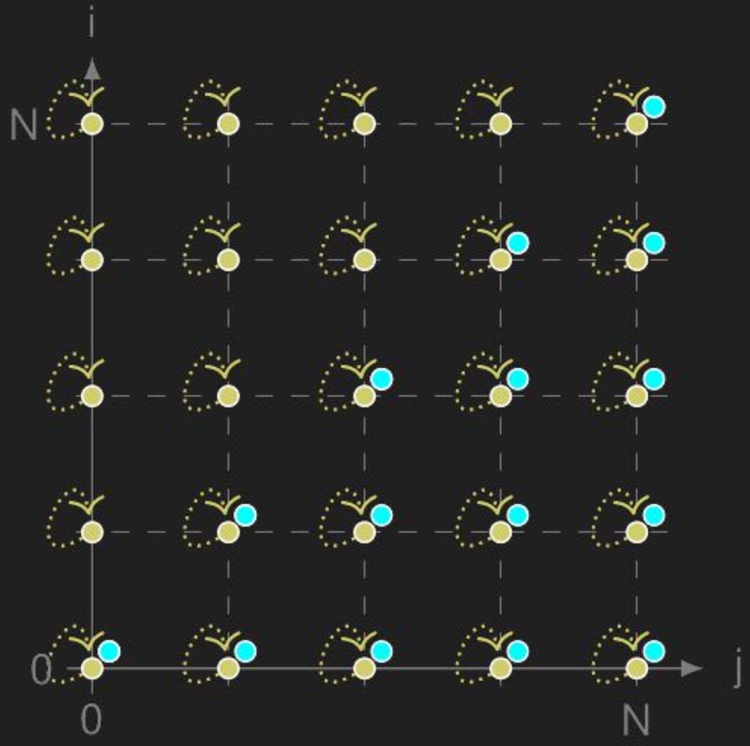
$$\mathcal{I}_P = \{(P, (i, j)) \mid 0 \leq i \leq N \wedge 0 \leq j \leq i\}$$



Polyhedral Representation

```
for (int i = 0; i <= N; i++)
  for (int j = 0; j <= N; j++) {
S:  A[i][j] = /* ... */;
    if (j <= i)
P:    A[i][j]+= A[j][i];
  }
```

$$\mathcal{F}_{S} = \{(S,(i,j)) \rightarrow (i,j)\}$$

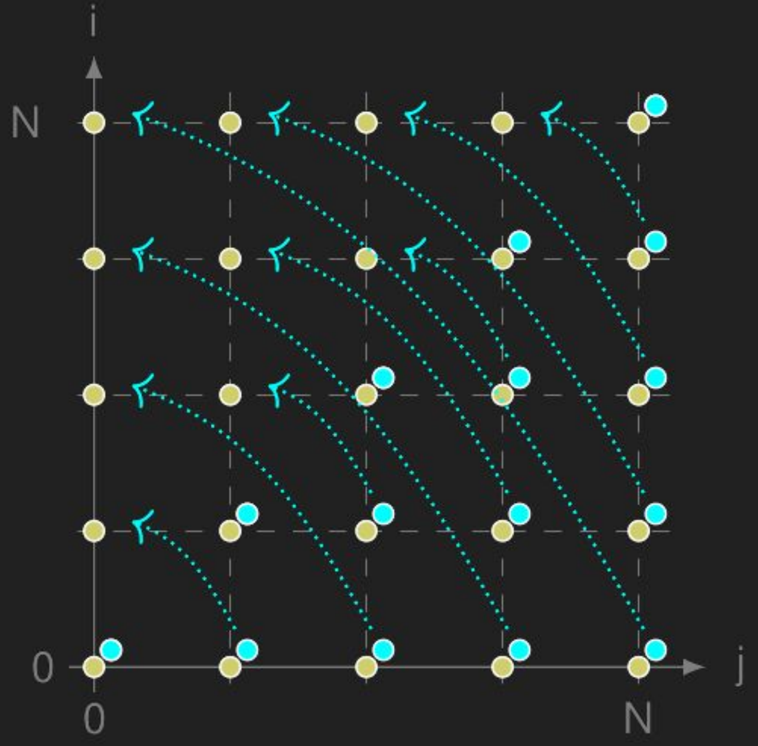


Polyhedral Representation

```
for (int i = 0; i <= N; i++)
  for (int j = 0; j <= N; j++) {
S:  A[i][j] = /* ... */;
    if (j <= i)
P:    A[i][j]+= A[j][i];
  }
```

$$\mathcal{F}_{P_1} = \{(P, (i, j)) \rightarrow (i, j)\}$$

$$\mathcal{F}_{P_2} = \{(P, (i, j)) \rightarrow (j, i)\}$$

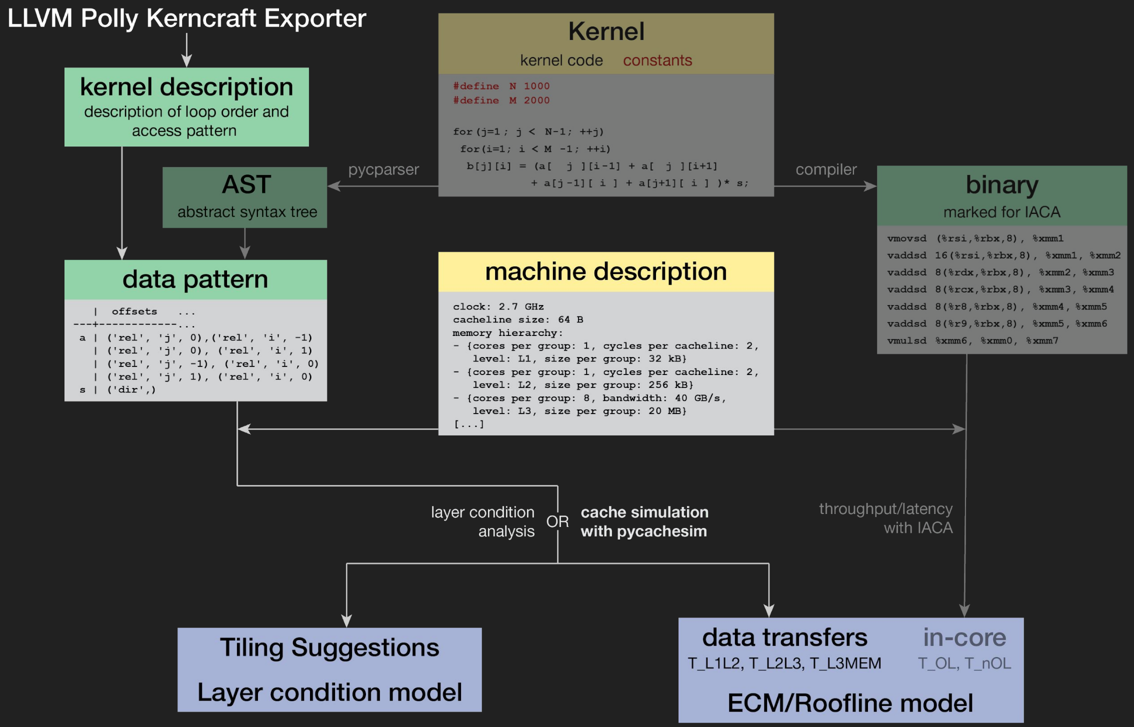


Implementation

Polly Kerncraft Exporter

Use Polly to automatically detect and extract kernel descriptions in large source bases.

Starting point for manual analysis and modelling.



Polly Layer Conditions

- Replacement for Polly's “fixed tiling strategy”
  - 32 is not always the best option
- Tiling can improve but also regress performance
  - Versioning for in-cache and in-memory tile size selection
- “Delinearization” severely limits polyhedral recognition
  - manual inspection tedious and hard

Tile Size Selection Algorithm – In-Cache

Goal: **Minimize misses** in fastest cache and **maximize** inner loop **iterations**

For each *cache* evaluate layer conditions with maximum *tail*, until LC and a minimum-iterations-requirement is fulfilled.

Minimum iterations are defined as 100 for inner loop and 10 for all other.

Tile Size Selection Algorithm – In-Cache (Example)

3D LC:	2 misses	if 32*N*M - 16*N	< cache_size
2D LC:	4 misses	if 48*N - 32	< cache_size
1D LC	6 misses	if 112	< cache_size

~~NB = 681~~

~~MB = 2~~

~~NB = 100~~

~~MB = 9~~

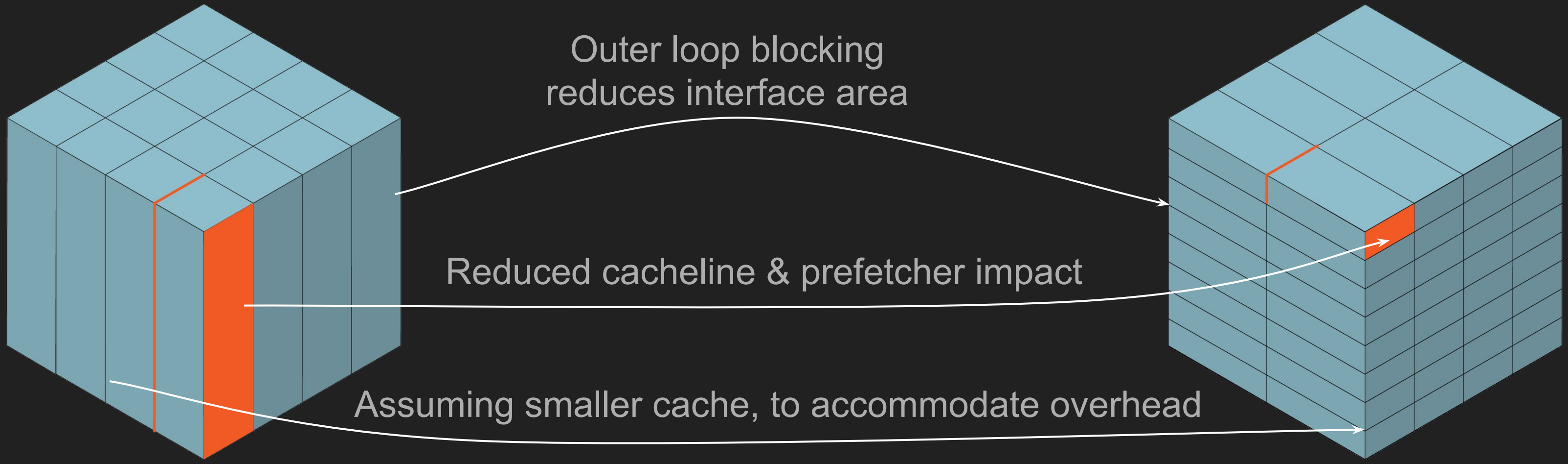
MB = 11

	3D LC	2D LC	1D LC
32 KB L1	~~2*N*M-N < 2048~~	N < 682	fulfilled
256 KB L2	2*N*M-N < 16384	N < 5460	fulfilled
20MB L3	2*N*M-N < 1311360	N < 436906	fulfilled

Tile Size Selection Algorithm – In-Memory

Minimize cache misses for half of L3 and maximize inner blocking factor

Add outer loop blocking with constant factor of 16



Evaluation

Used Benchmarks and System

- 3D 7pt and 3D “well conditioned”
- polybench[2] stencils v2.4.1
- OptEWE[3]
- Harris [PolyMage benchmarks][4]
- 172.mgrid [SPEC CPU2000]

Environment:
Intel Xeon CPU E5-2660 v2 @ 2.20GHz (fixed, no turbo)
(patched) LLVM 6.0, clang, flang, (patched) Polly
LIKWID instrumentation for L2, L3 and Memory volumes
Pinned all processes

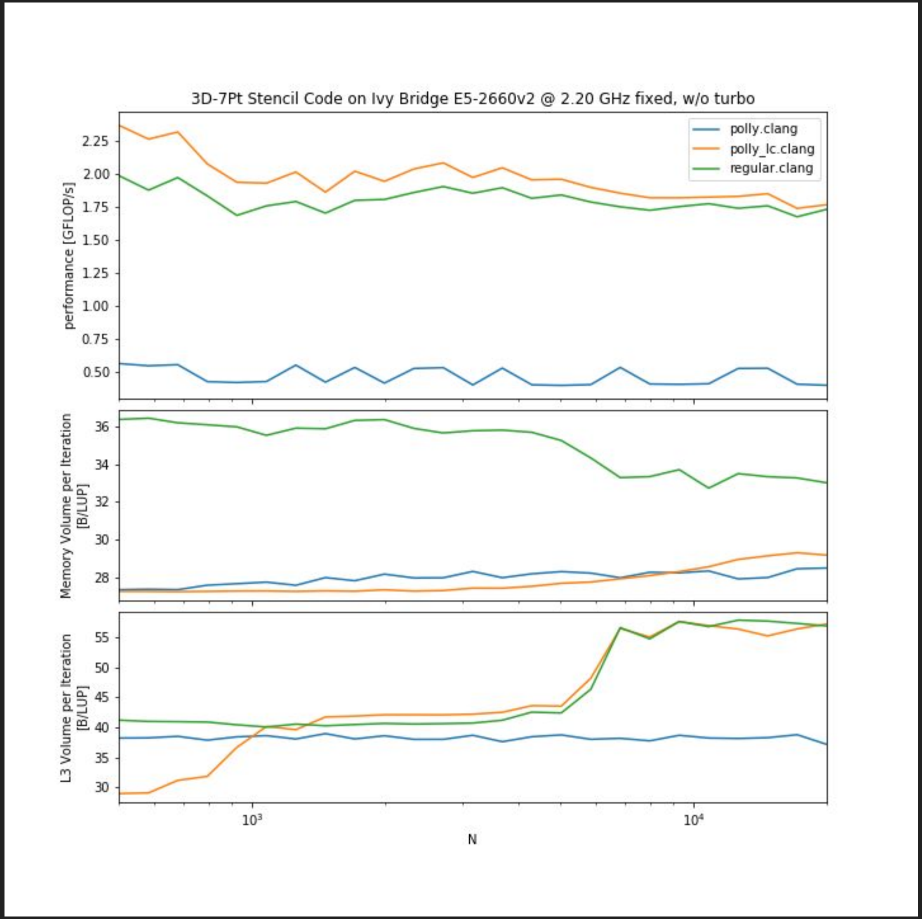
[2] http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/
[3] https://github.com/mohamso/optewe
[4] Mullapudi et al., PolyMage: Automatic Optimization for Image Processing Pipelines
[5] http://accc.riken.jp/en/supercom/himenobmt/

3D 7pt

Performance gain for large N

Reduced data volume in cache and memory

Data volume is not everything...

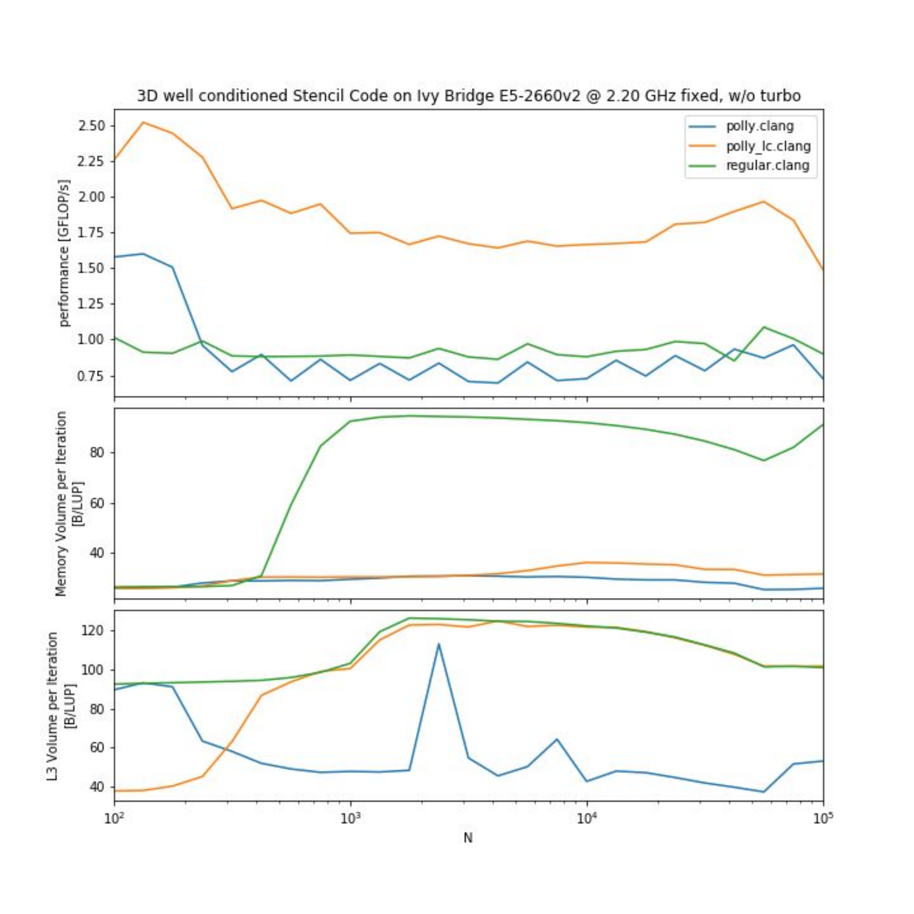


3D “well conditioned”

Performance gains overall measured N

Slightly reduced L3 volume

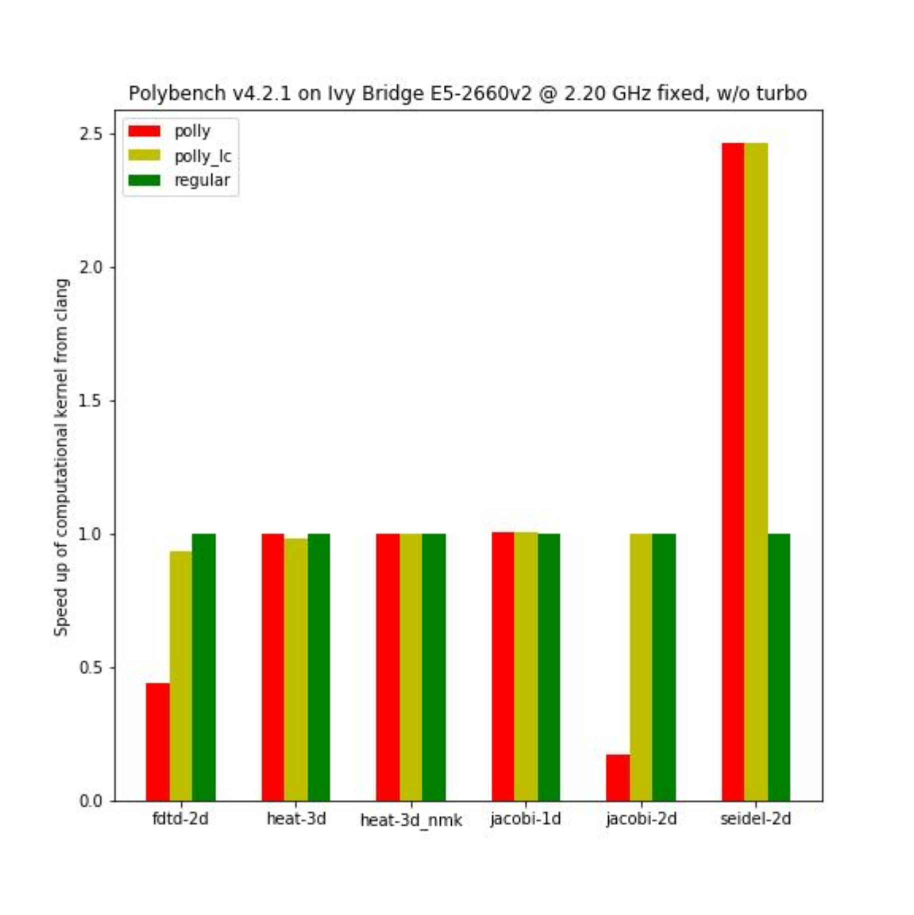
Speedup comes also from polly-enabled vectorization, but plain polly kills it again with tiny blocks



Polybench Stencils

- heat-3d
- heat-3d_nmk
- fdtd-2d
- jacobi-1d
- jacobi-2d
- seidel-2d

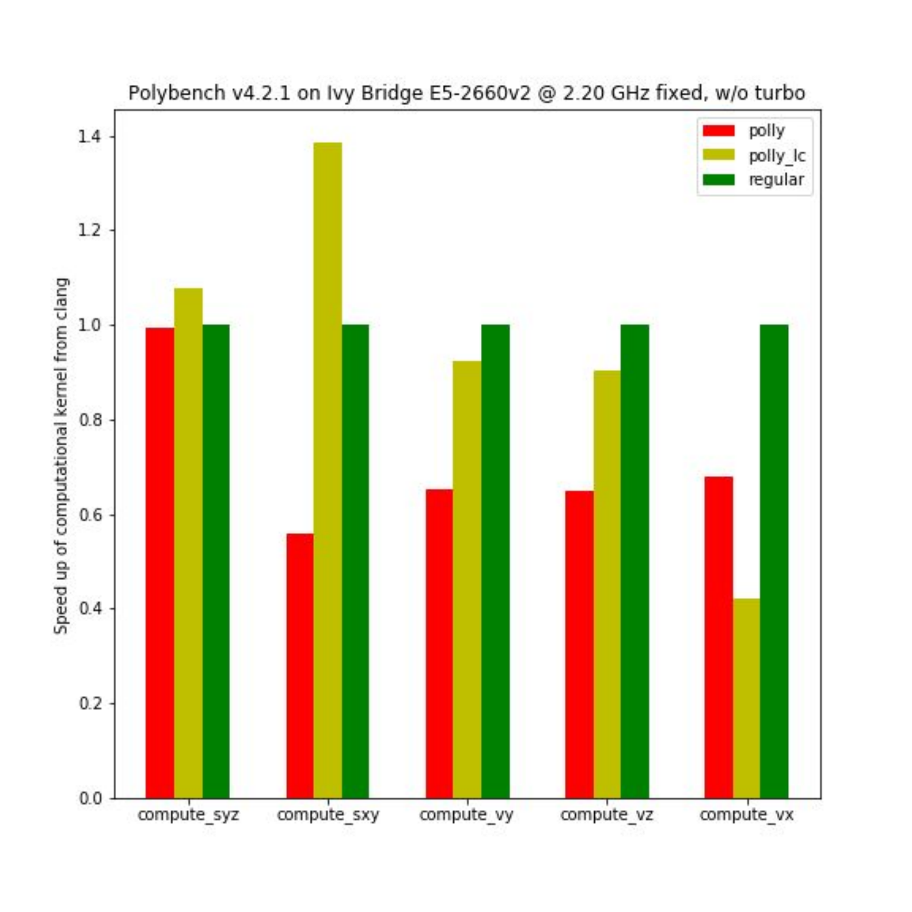
Speed up, without regression!



OptEWE

Only few kernels have reuse and could benefit from tiling.

Speed downs, in particular `compute_vx`, need to be investigated.



Himeno

As described in [6], spatial block will not yield performance gains.

[6] https://blogs.fau.de/hager/archives/7850

PolyMage Image Processing Pipelines

Harris corner detection

- 12 arrays, 11 loop nests (each 2D), 65 memory accesses

	Sequential (arith. avg/median)	Parallel (arith. avg/median)
Regular (no tiling)	168.7ms / 170.5ms	77.6ms / 76.8 ms
Polly tiling	249.8ms / 252.7ms	94.6ms / 92.9ms
Polly-LC tiling	167.6ms / 165.3ms	78.0ms / 77.2ms
Polly-LC (in-memory)	169.9ms / 170.8ms	82.1ms / 80.5ms
Polly-LC (in-cache)	169.3ms / 168.9ms	118.0ms / 116.3ms

51 runs, default input, Intel(R) Core(TM) i7-4800MQ

172.mgrid [SPEC CPU2000]

20% reduced L3 volume and slightly reduced main memory volume, but no performance increase. Possibly computation bound.

	Runtime	Mem. volume	L3 volume	L2 volume
Regular (no tiling)	61 s	252 GB	418 GB	446 GB
Polly tiling	73 s	257 GB	690 GB	632 GB
Polly-LC tiling	61 s	248 GB	346 GB	472 GB

reference input

Outlook & Conclusion

Outlook

- OpenMP shared cache support
- Tweak heuristics parameters
- Support for strided accesses (cache lines!)
- Runtime tile size variation
- Predict if kernel is memory/cache or compute bound

Conclusion

- Approached trade-off between minimal loop length and cache usage
- For suited codes, speedups over regular LLVM and Polly are significant
- Generally, fewer and less regressions compared to Polly
- Basis for further analytical model-driven optimationzations

Thanks

Questions?
Discussion!