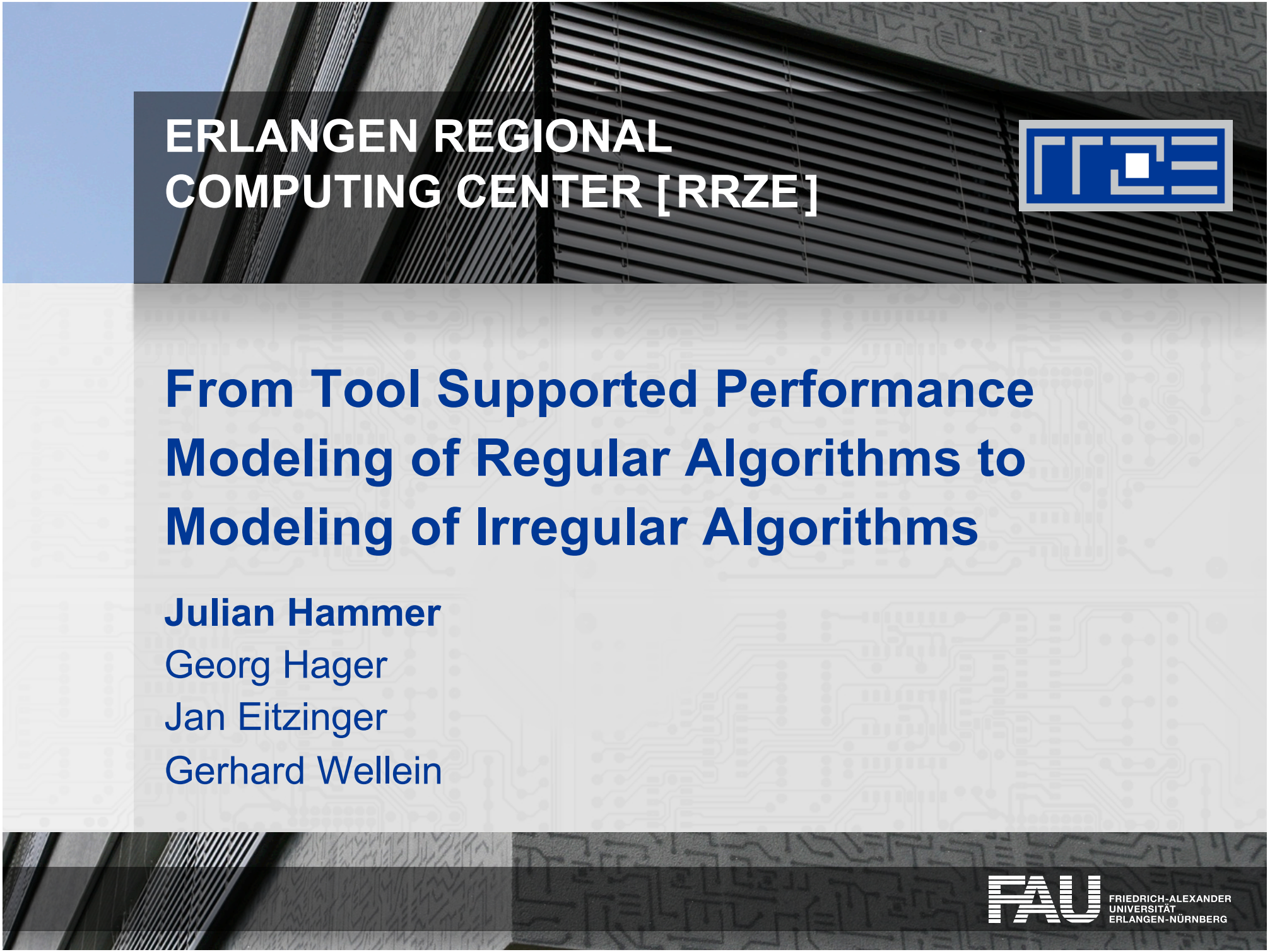


ERLANGEN REGIONAL
COMPUTING CENTER [RRZE]

From Tool Supported Performance Modeling of Regular Algorithms to Modeling of Irregular Algorithms

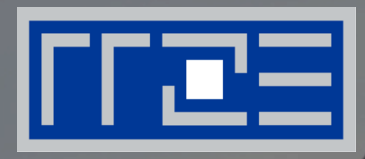
Julian Hammer
Georg Hager
Jan Eitzinger
Gerhard Wellein

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

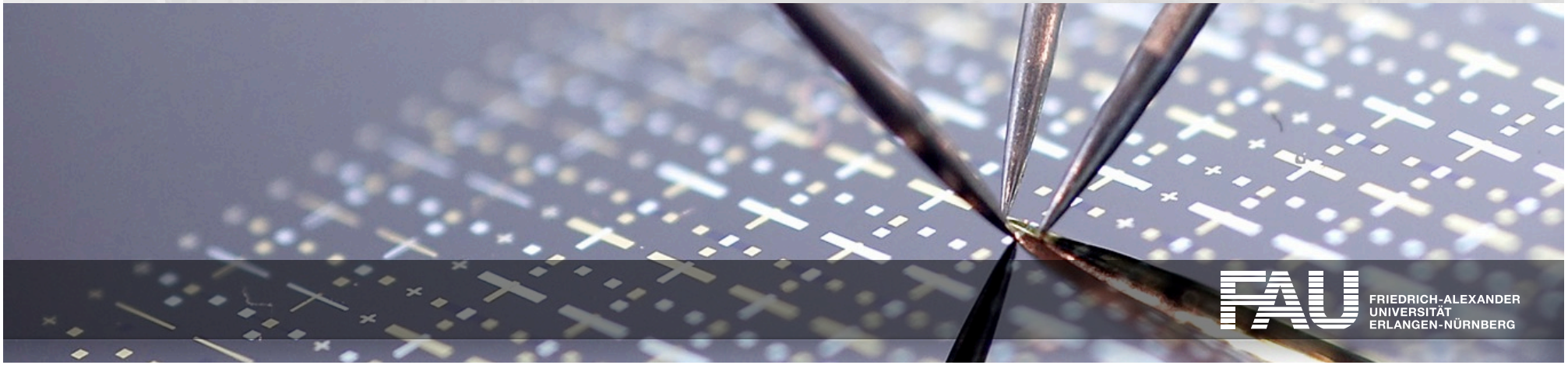
Overview

1. Loop Kernels
2. Roofline and ECM
3. Kerncraft
    1. Overview and Structure
    2. Output and Results
4. 3D-long-range Example
5. Outlook
    1. Irregular Algorithms

LOOP KERNELS



Automatic **Loop Kernel** Analysis and
Performance Modeling with Kerncraft



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Loop Kernels

```
double a[5000], b[5000];
double s;

for(i=0; i<5000; ++i)
    a[i] = s * b[i];
```

```
double a[5000][5000];
double b[5000][5000];
double s;

for(j=1; j<5000-1; ++j)
    for(i=1; i<5000-1; ++i)
        b[j][i] = ( a[j][i-1] + a[j][i+1]
                  + a[j-1][i] + a[j+1][i] )
                  * s;
```

- Many inner-loop iterations
- No branching
- Access fully determined by loop counters (i.e., no irregularities)

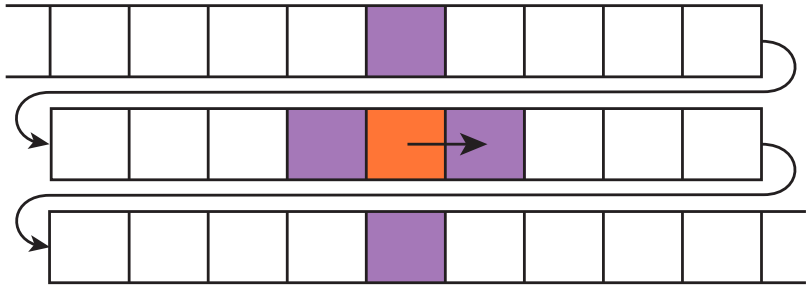
Loop Kernels

```
double a[5000], b[5000];
double s;

for(i=0; i<5000; ++i)
    a[i] = s * b[i];
```

Streaming Kernel

- Simple structure
- No data-reuse

```
double a[5000][5000];
double b[5000][5000];
double s;

for(j=1; j<5000-1; ++j)
    for(i=1; i<5000-1; ++i)
        b[j][i] = ( a[j][i-1] + a[j][i+1]
                  + a[j-1][i] + a[j+1][i] )
                  * s;
```



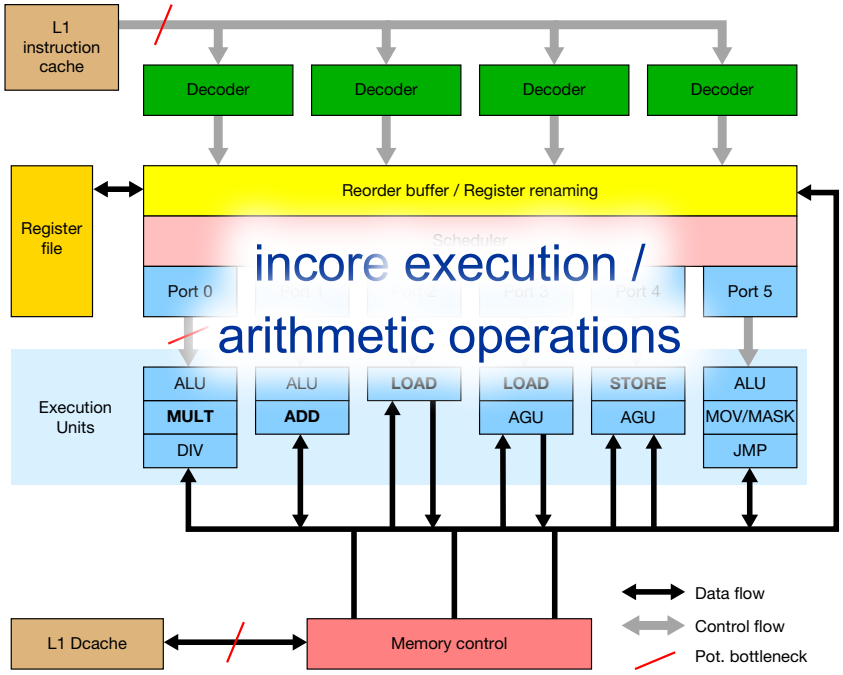
Stencil Code

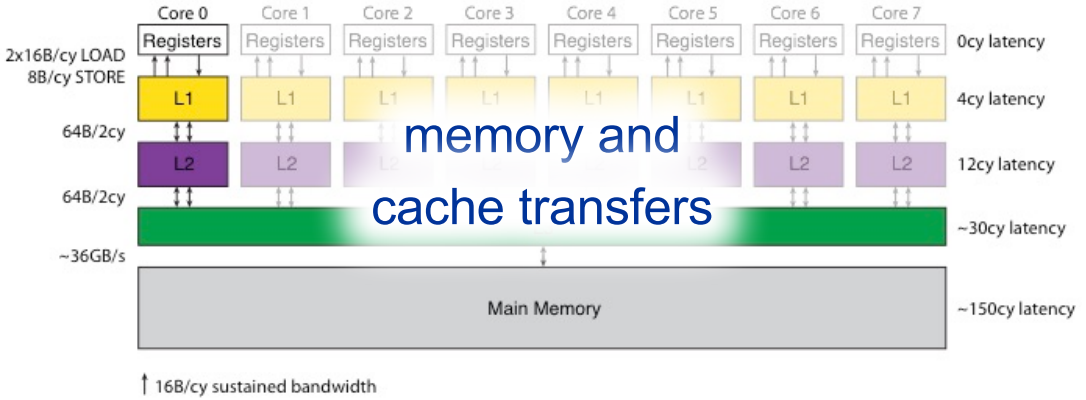
- Complex Structure
- Heavy data-reuse

Loop Kernels

How to predict performance on complex architectures?

Two major contributions/bottlenecks:



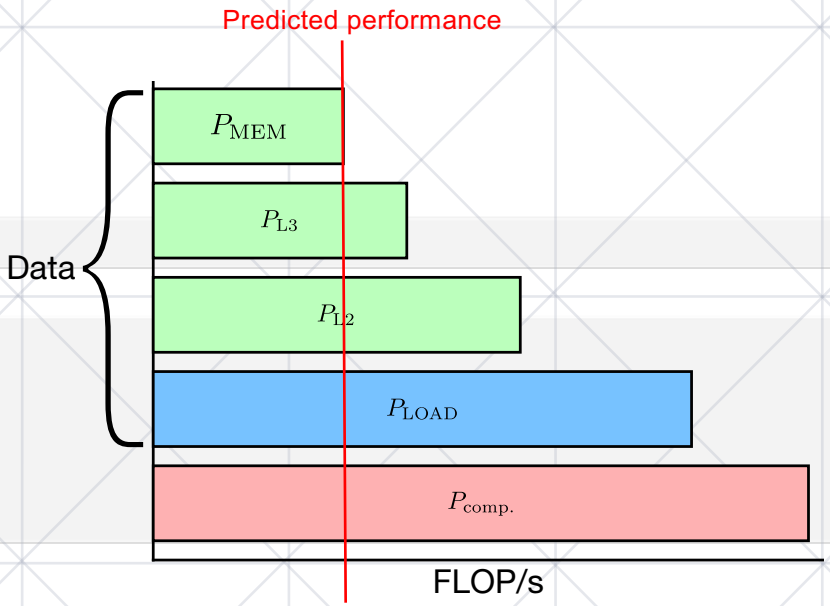


ROOFLINE AND ECM

Automatic Loop Kernel Analysis and
Performance Modeling with Kerncraft

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Roofline



Roofline

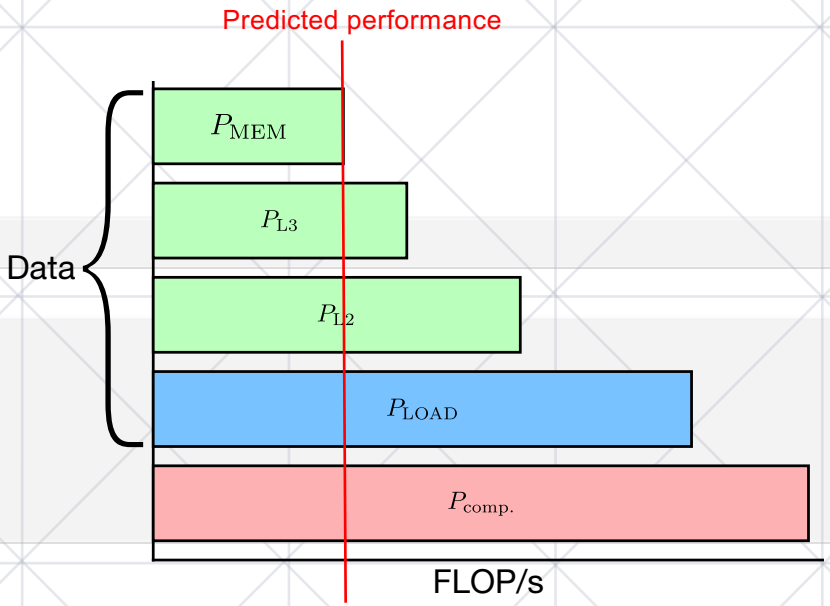
- All memory levels are separate bottlenecks

$$P = \min(P_{comp.}, I \bullet b_s)$$

$P_{comp.}$	Peak performance [FLOP/s]
I	Operational Intensity [FLOP/B]
b_s	Peak bandwidth [B/s]

- Bandwidths are measured by suitable benchmarks

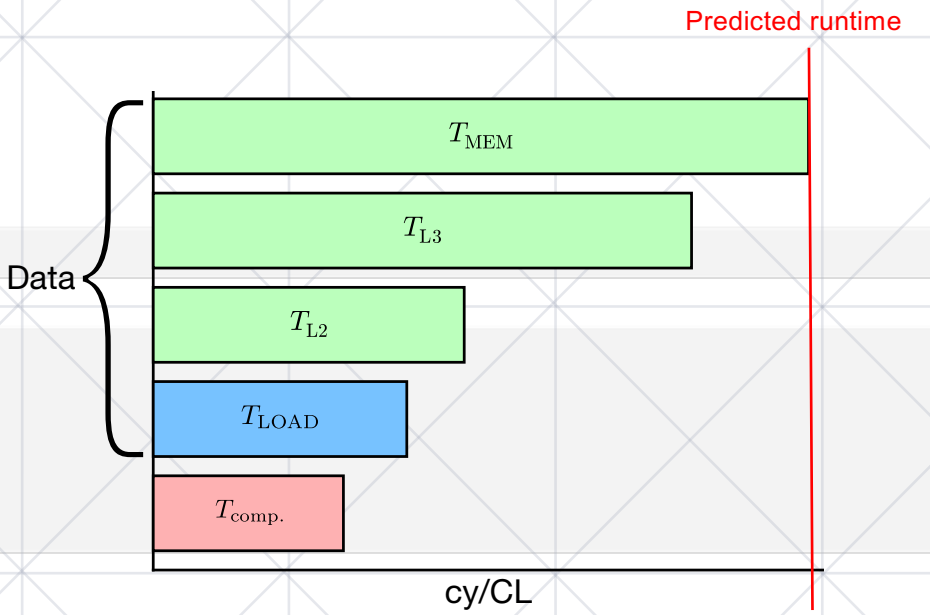
Roofline: Performance vs Time



Roofline

- CPU frequency?
  - → Cycles!
- Basic memory units?
  - → Cache Lines! (64 Byte)

Roofline: Performance vs Time



Roofline

- CPU frequency?
  - → Cycles!
- Basic memory units?
  - → Cache Lines! (1 CL=64 B)
  - → 1 unit of work = 1 CL

→ Cycle / Cache Line (cy/CL)

- Lower is better

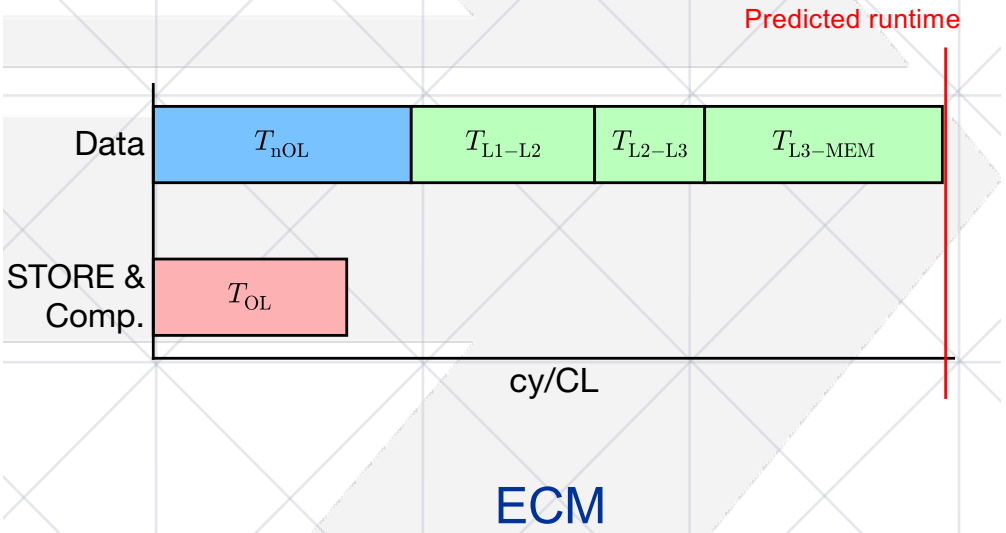
Execution-Cache-Memory (ECM) Model

- Memory and cache levels **contribute** to runtime

T_{OL} computation & stores
T_{nOL} loads from L1
$T_{L1\text{-}L2}$ loads from L2 into L1
$T_{L2\text{-}L3}$ loads from L3 into L2
$T_{L3\text{-}MEM}$ loads from main memory into L3

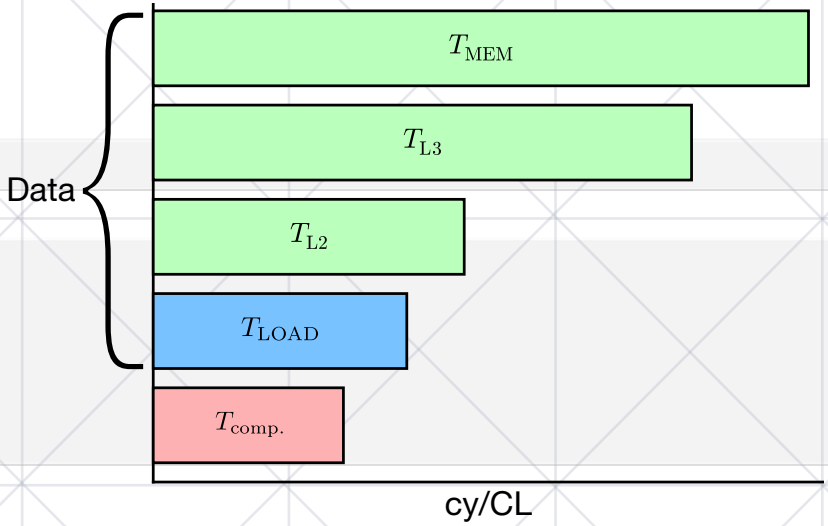
$\{ T_{OL} \,||\, T_{nOL} \mid T_{L1\text{-}L2} \mid T_{L2\text{-}L3} \mid T_{L3\text{-}MEM} \}$

- One measured input: full-socket mem. bandwidth

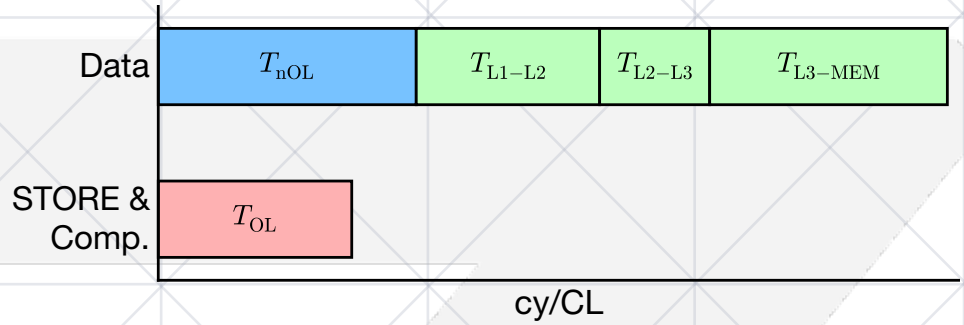


ECM

Roofline and ECM



Roofline



ECM

Performance Modeling

```
double a[5000], b[5000];
double s;

for(i=0; i<5000; ++i)
    a[i] = s * b[i];
```

STREAM Scale

- For each cache line (8 it.):
  - 1 CL is stored
  - 8 FLOP
  - 1 CL are loaded

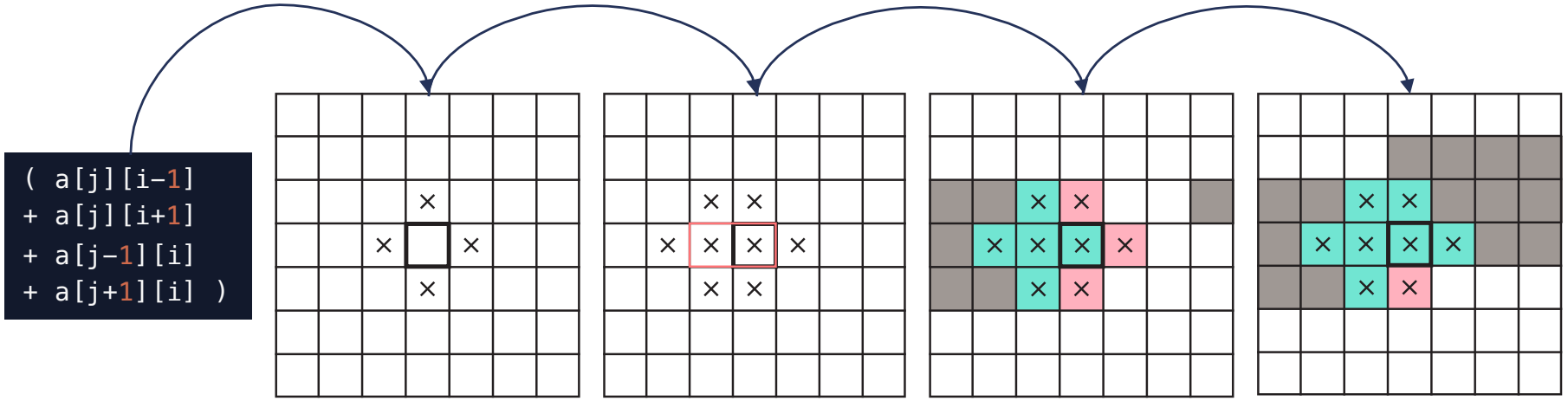
```
double a[5000][5000];
double b[5000][5000];
double s;

for(j=1; j<5000-1; ++j)
    for(i=1; i<5000-1; ++i)
        b[j][i] = ( a[j][i-1] + a[j][i+1]
                  + a[j-1][i] + a[j+1][i] )
                  * s;
```

2D 5-point Stencil

- For each cache line (8 it.):
  - 1 CL is stored
  - 32 FLOP
  - Up to 3 CL are loaded

Up to 3?

Layer Conditions



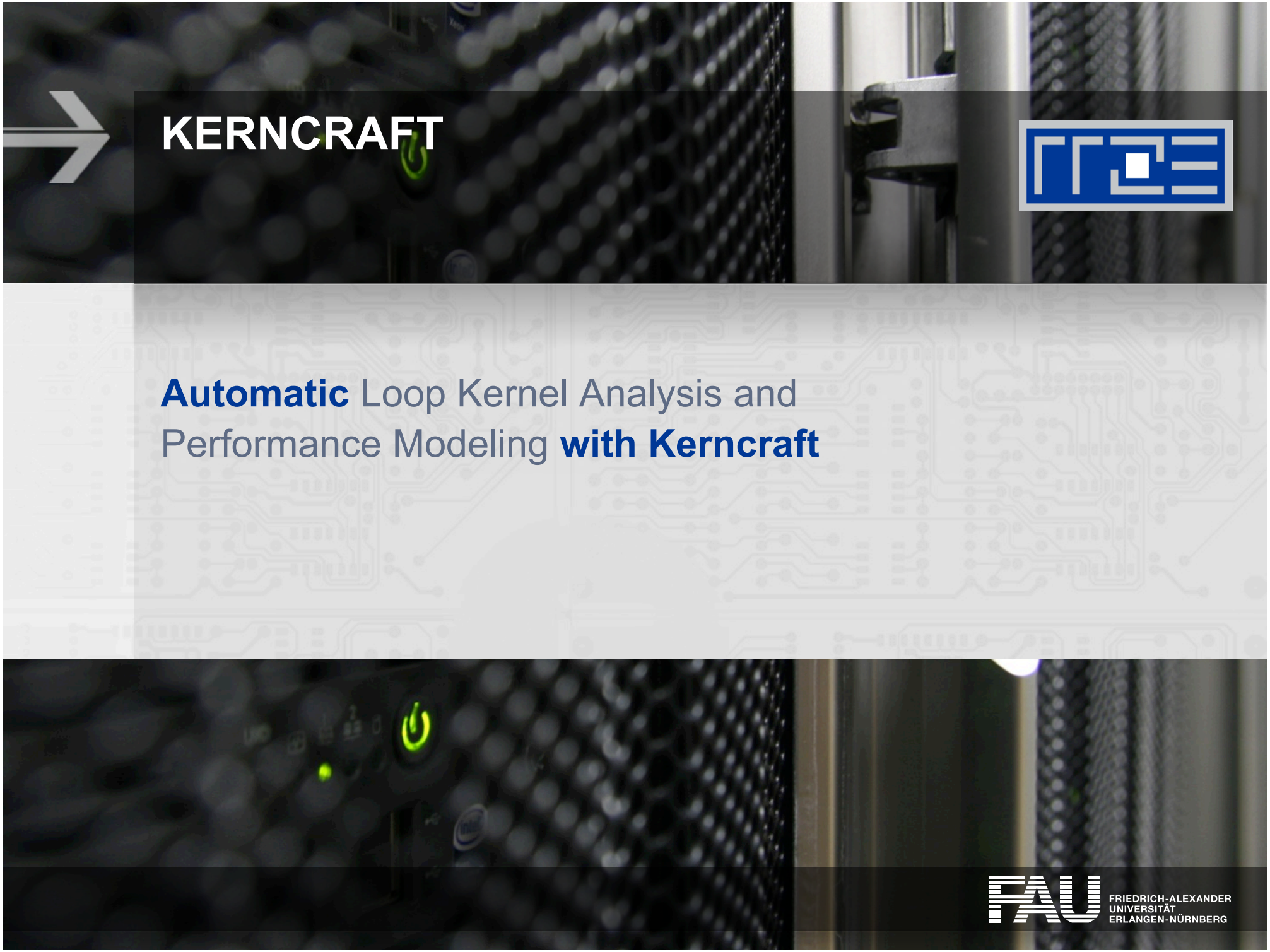
1D layer condition: stencil-width * stencil-height < cache-size

2D layer condition: stencil-height * matrix-width < cache-size

nD layer condition: $C_{\text{req.}} = \left(\sum L_{\text{rel.offsets}} + \max(L_{\text{rel.offsets}}) * n_{\text{slices}} \right) * s$

Performance Modeling

```
double U[M][N][N];
double V[M][N][N];
double ROC[M][N][N];
double c0, c1, c2, c3, c4, lap;

for(int k=4; k < M-4; k++) {
    for(int j=4; j < N-4; j++) {
        for(int i=4; i < N-4; i++) {
            lap = c0 * V[k][j][i]
                + c1 * ( V[ k ][ j ][i+1] + V[ k ][ j ][i-1])
                + c1 * ( V[ k ][j+1][ i ] + V[ k ][j-1][ i ])
                + c1 * ( V[k+1][ j ][ i ] + V[k-1][ j ][ i ])
                + c2 * ( V[ k ][ j ][i+2] + V[ k ][ j ][i-2])
                + c2 * ( V[ k ][j+2][ i ] + V[ k ][j-2][ i ])
                + c2 * ( V[k+2][ j ][ i ] + V[k-2][ j ][ i ])
                + c3 * ( V[ k ][ j ][i+3] + V[ k ][ j ][i-3])
                + c3 * ( V[ k ][j+3][ i ] + V[ k ][j-3][ i ])
                + c3 * ( V[k+3][ j ][ i ] + V[k-3][ j ][ i ])
                + c4 * ( V[ k ][ j ][i+4] + V[ k ][ j ][i-4])
                + c4 * ( V[ k ][j+4][ i ] + V[ k ][j-4][ i ])
                + c4 * ( V[k+4][ j ][ i ] + V[k-4][ j ][ i ]);
            U[k][j][i] = 2.f * V[k][j][i] - U[k][j][i]
                       + ROC[k][j][i] * lap;
}}}
```

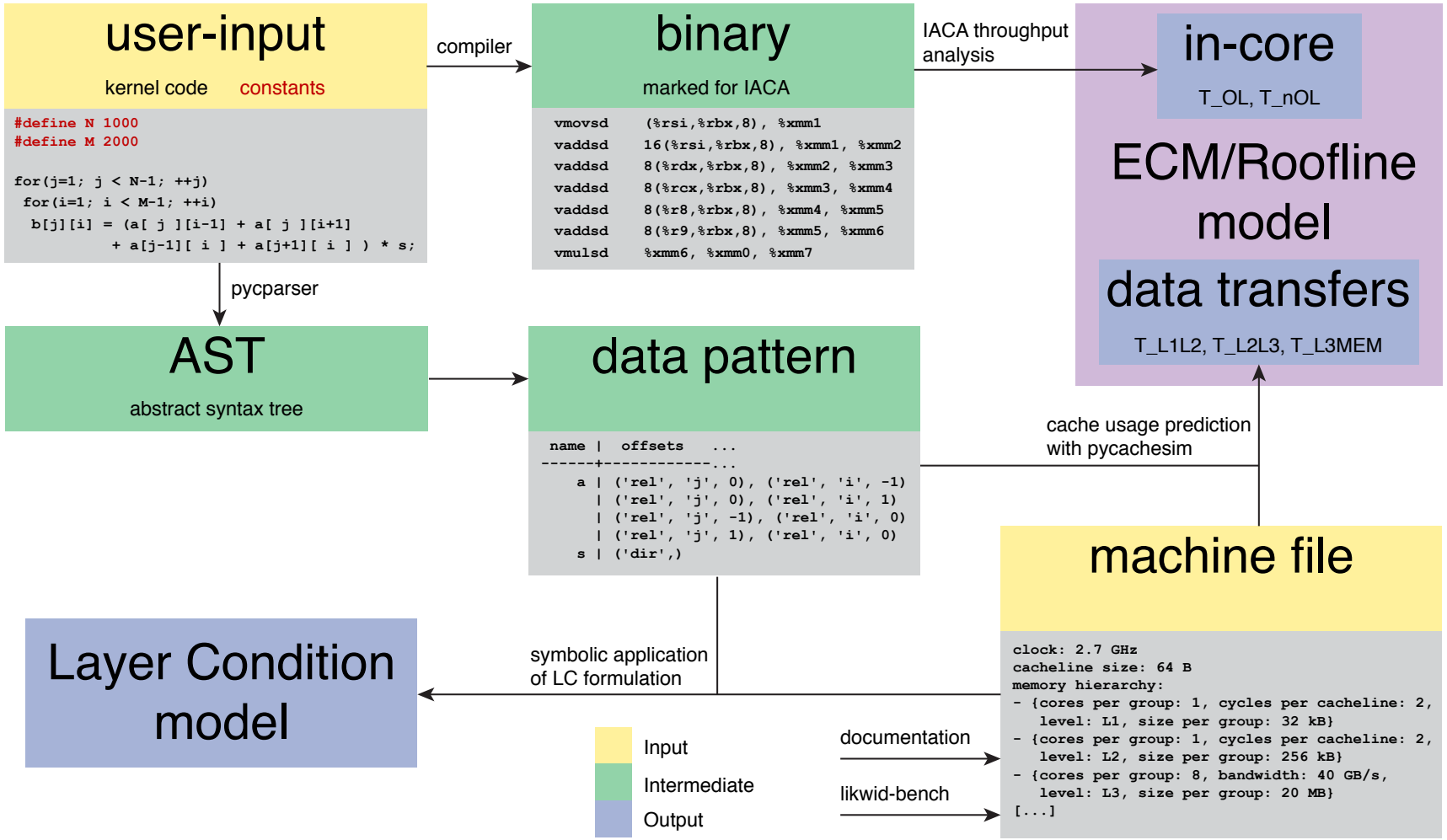


KERNCRAFT

Automatic Loop Kernel Analysis and Performance Modeling **with Kerncraft**

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Kerncraft


user-input
kernel code
constants
#define N 1000
#define M 2000
for(j=1; j < N-1; ++j)
for(i=1; i < M-1; ++i)
b[j][i] = (a[j][i-1] + a[j][i+1]
+ a[j-1][i] + a[j+1][i]) * s;
compiler
binary
marked for IACA
vmovsd (%rsi,%rbx,8), %xmm1
vaddsd 16(%rsi,%rbx,8), %xmm1, %xmm2
vaddsd 8(%rdx,%rbx,8), %xmm2, %xmm3
vaddsd 8(%rcx,%rbx,8), %xmm3, %xmm4
vaddsd 8(%r8,%rbx,8), %xmm4, %xmm5
vaddsd 8(%r9,%rbx,8), %xmm5, %xmm6
vmulsd %xmm6, %xmm0, %xmm7
IACA throughput analysis
in-core
T_OL, T_nOL
ECM/Roofline model
data transfers
T_L1L2, T_L2L3, T_L3MEM
pycparser
AST
abstract syntax tree
data pattern
name | offsets ...
a | ('rel', 'j', 0), ('rel', 'i', -1)
| ('rel', 'j', 0), ('rel', 'i', 1)
| ('rel', 'j', -1), ('rel', 'i', 0)
| ('rel', 'j', 1), ('rel', 'i', 0)
s | ('dir',)
cache usage prediction with pycachesim
machine file
clock: 2.7 GHz
cacheline size: 64 B
memory hierarchy:
- {cores per group: 1, cycles per cacheline: 2, level: L1, size per group: 32 kB}
- {cores per group: 1, cycles per cacheline: 2, level: L2, size per group: 256 kB}
- {cores per group: 8, bandwidth: 40 GB/s, level: L3, size per group: 20 MB}
[...]
Layer Condition model
symbolic application of LC formulation
Input
Intermediate
Output
documentation
likwid-bench

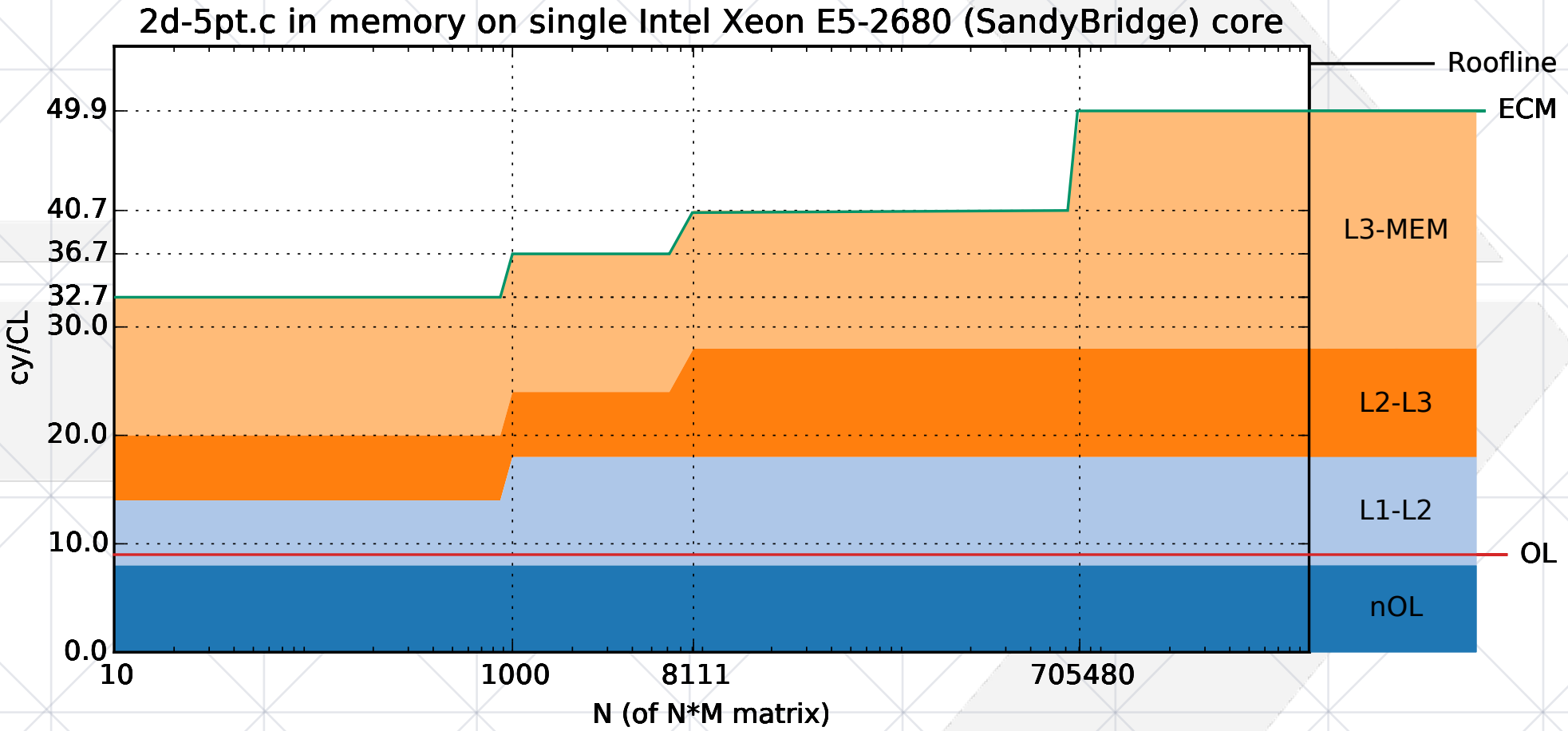
Kerncraft – Output

ECM model: { T_{OL} || T_{nOL} | T_{L1-L2} | T_{L2-L3} | T_{L3-MEM} }

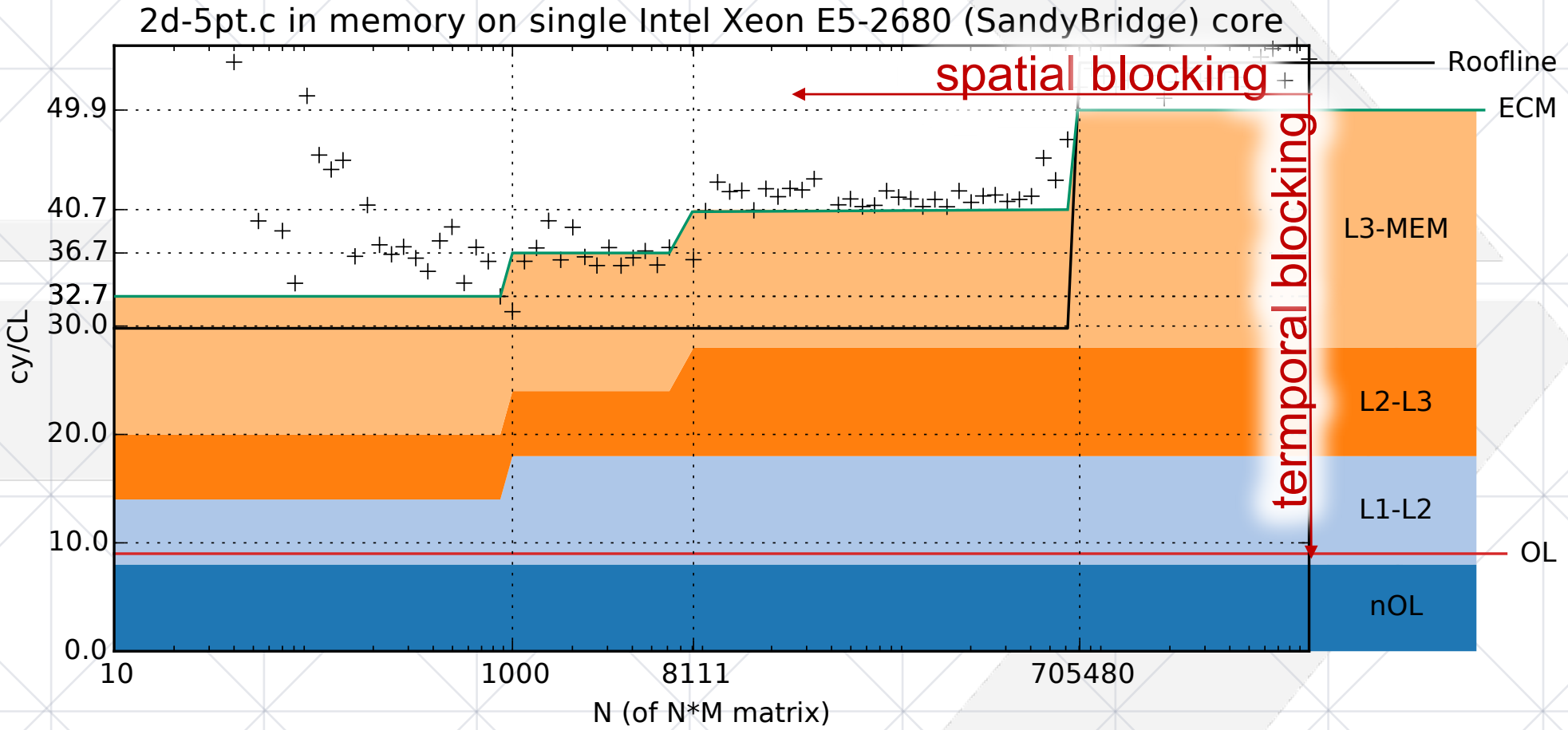
```
$ kerncraft --machine snb.yaml 2d-5pt.c --pmodel ECM -D N 5000 -D M 500
================================================================================
kernels/2d-5pt.c
================================================================================
{ 9.0 || 8.0 | 10
{ 9.0 \ 18.00 \ 2
saturating at 3 c
$
$ kerncraft --mac                                          cy/CL -D N 5000 -D M 500
================================================================================
kernels/2d-5pt.c
================================================================================
Cache or mem boun
29.79 cy/CL due to L3-MEM transfer bottleneck (bw from copy benchmark)
Arithmetic Intensity: 0.17 FLOP/b
$
```

```
double a[M][N];
double b[M][N];
double s;

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

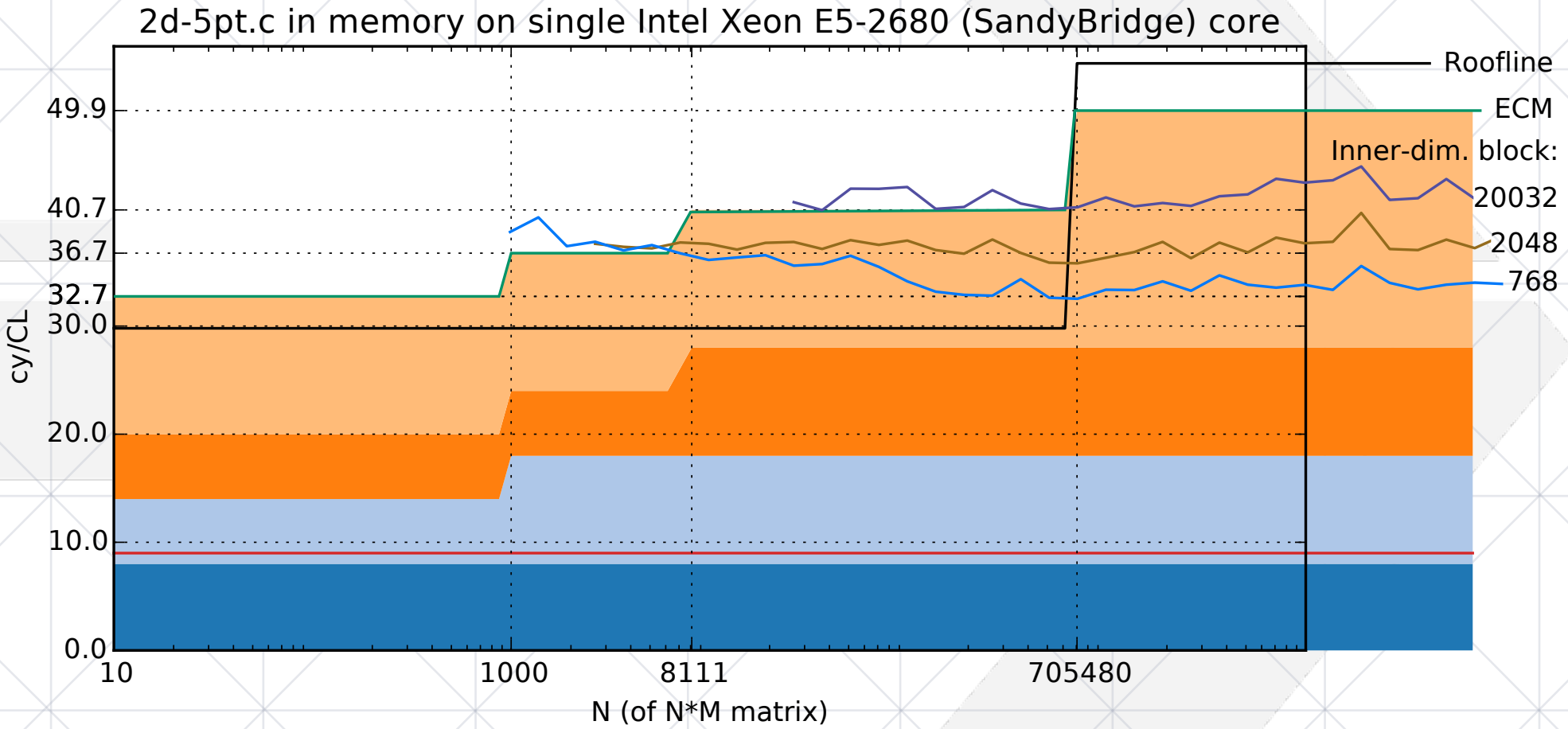
Kerncraft – Results



Kerncraft – Results



Kerncraft – Spatial Blocking



Kerncraft – Verbose Output

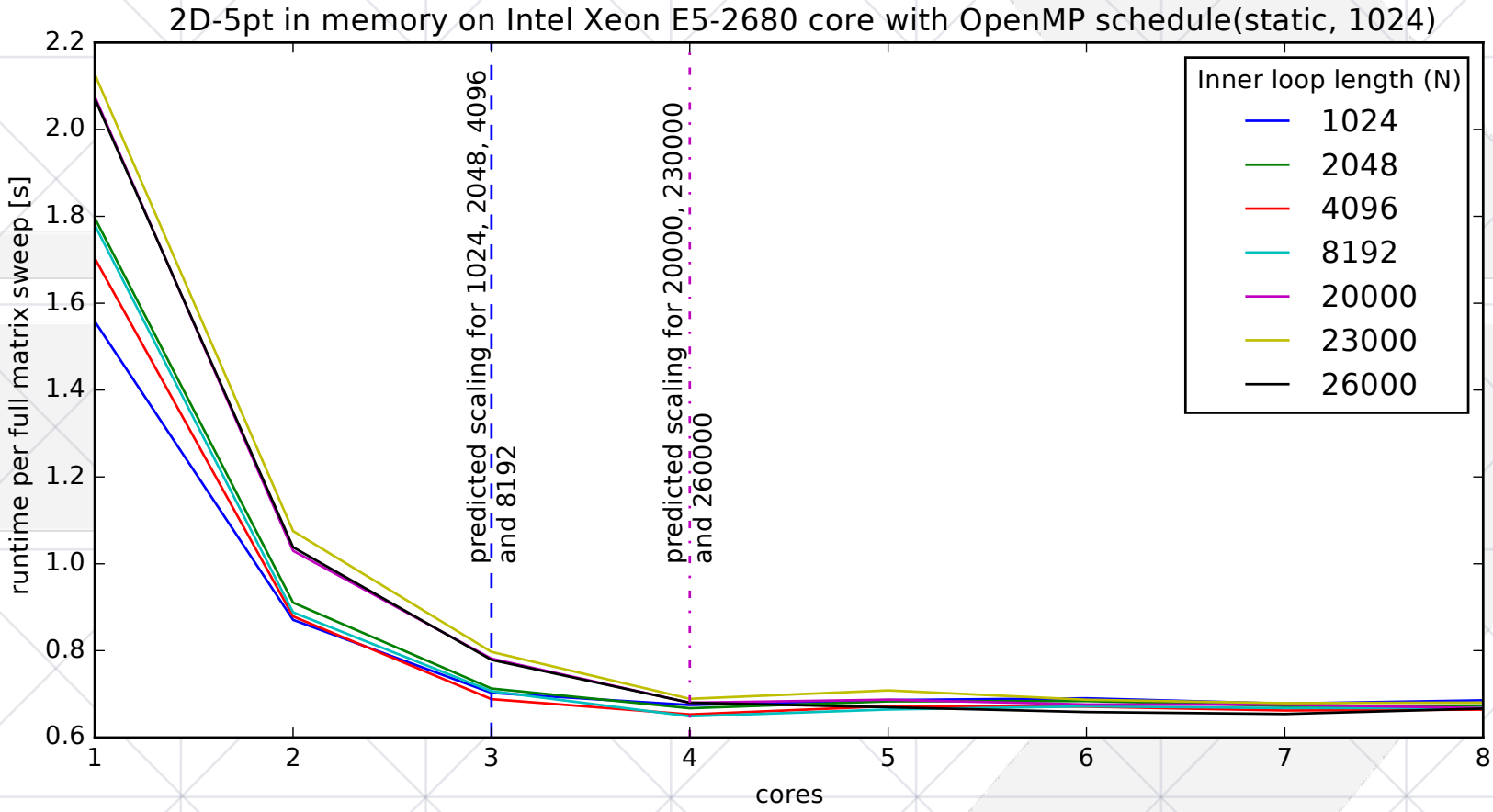
Layer condition analysis:

```
$ kerncraft --machine snb.yaml 2d-5pt.c --pmodel LC
[…]
1D Layer-Condition:
L1: unconditionally fulfilled
L2: unconditionally fulfilled
L3: unconditionally fulfilled
2D Layer-Condition:
L1: N <= 1024
L2: N <= 8192
L3: N <= 655360
$
```

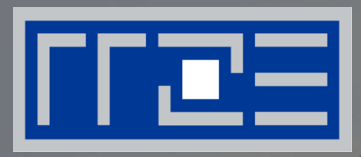
Also available as web-based calculator:

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

Kerncraft – In-Socket Scaling

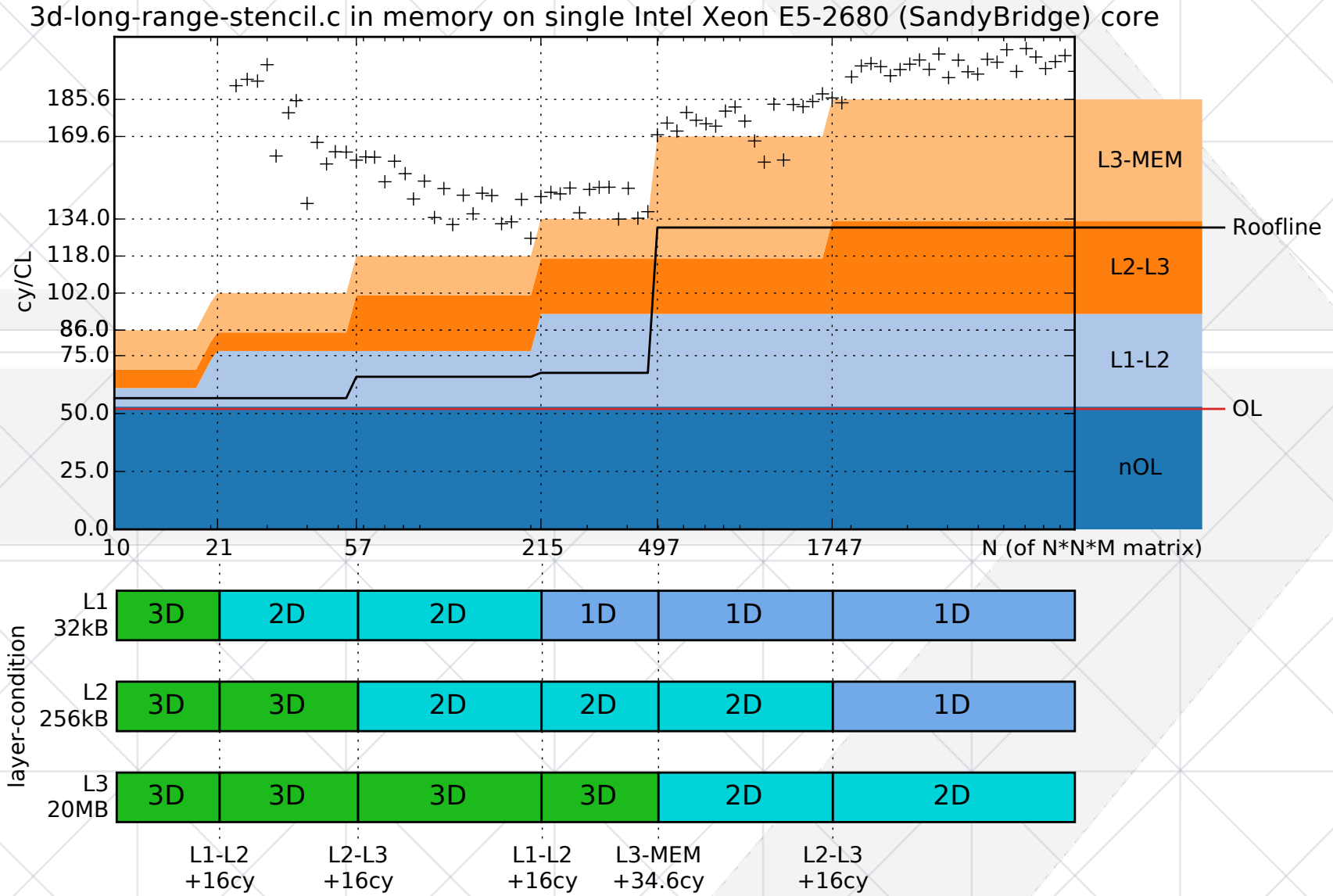


3D-LONG-RANGE EXAMPLE



```
double U[M][N][N];
double V[M][N][N];
double ROC[M][N][N];
double c0, c1, c2, c3, c4, lap;

for(int k=4; k < M-4; k++) {
    for(int j=4; j < N-4; j++) {
        for(int i=4; i < N-4; i++) {
            lap = c0 * V[k][j][i]
                + c1 * ( V[ k ][ j ][i+1] + V[ k ][ j ][i-1])
                + c1 * ( V[ k ][j+1][ i ] + V[ k ][j-1][ i ])
                + c1 * ( V[k+1][ j ][ i ] + V[k-1][ j ][ i ])
                + c2 * ( V[ k ][ j ][i+2] + V[ k ][ j ][i-2])
                + c2 * ( V[ k ][j+2][ i ] + V[ k ][j-2][ i ])
                + c2 * ( V[k+2][ j ][ i ] + V[k-2][ j ][ i ])
                + c3 * ( V[ k ][ j ][i+3] + V[ k ][ j ][i-3])
                + c3 * ( V[ k ][j+3][ i ] + V[ k ][j-3][ i ])
                + c3 * ( V[k+3][ j ][ i ] + V[k-3][ j ][ i ])
                + c4 * ( V[ k ][ j ][i+4] + V[ k ][ j ][i-4])
                + c4 * ( V[ k ][j+4][ i ] + V[ k ][j-4][ i ])
                + c4 * ( V[k+4][ j ][ i ] + V[k-4][ j ][ i ]);
            U[k][j][i] = 2.f * V[k][j][i] - U[k][j][i]
                       + ROC[k][j][i] * lap;
}}}
```

3D-long-range Example



Benefits

- Guiding optimizations
- Hardware-software co-design
- Energy optimized computing
- Deeper understanding of code and hardware interactions

Kerncraft...

- is a white-box utility
- takes some of the pain out of performance modeling
- is free (as in free beer and freedom)
- is NOT for inexperienced programmers
- is NOT a fully-automated jack-of-all-trades yielding better performance

Open Source


Fork me on GitHub
https://github.com/RRZE-HPC/kerncraft
Licensed under AGPLv3

Outlook

- Replacement for IACA (under investigation)
- Support for non-Intel Architectures (AMD and POWER8)
  - Depends on:
    - Support for non-inclusive cache-architectures and (work in progress)
    - ECM model support
    - Replacement for IACA
- Phenomenological performance modeling with LIKWID
- LLVM integration with polyhedral model
  - Import of kernels embedded in large code bases
  - Automatic tiling during compilation
- Irregular Performance Modeling (e.g., graph algorithms)

Outlook – Breadth-First-Search

- Assumptions:
  - More work, will lead to longer execution time
  - Difference in time, can be modeled by additional work
- Basic Model for BFS-TD

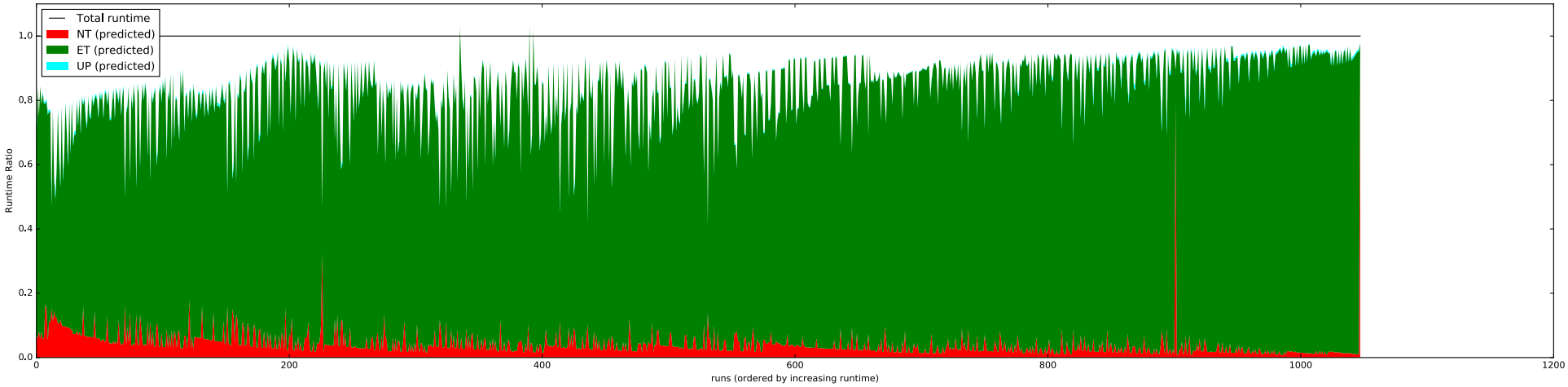
  t_{NT}(#nodes) +
  t_{ET}(#edgestraversed) +
  t_{UP}(#nodesupdated) = t

```
int64_t Step(const Graph &g,
             int64_t lvl,
             pvector<int64_t> &levels,
             pvector<NodeID> &parent) {

  int64_t changed = 0;
  for(NodeID u = 0; u < g.num_nodes();
      u++) {

    if (levels[u] != lvl)
      continue;
    // Node Traversal (NT) until here
    for(NodeID v : g.in_neigh(u)) {
      if(levels[v] < 0) {
        // Edge Traversal (ET) until here
        levels[v] = lvl + 1;
        changed += 1;
        // Update (UP) until here
      }
    }
  }
  return changed;
}
```

Outlook – Breadth-First-Search

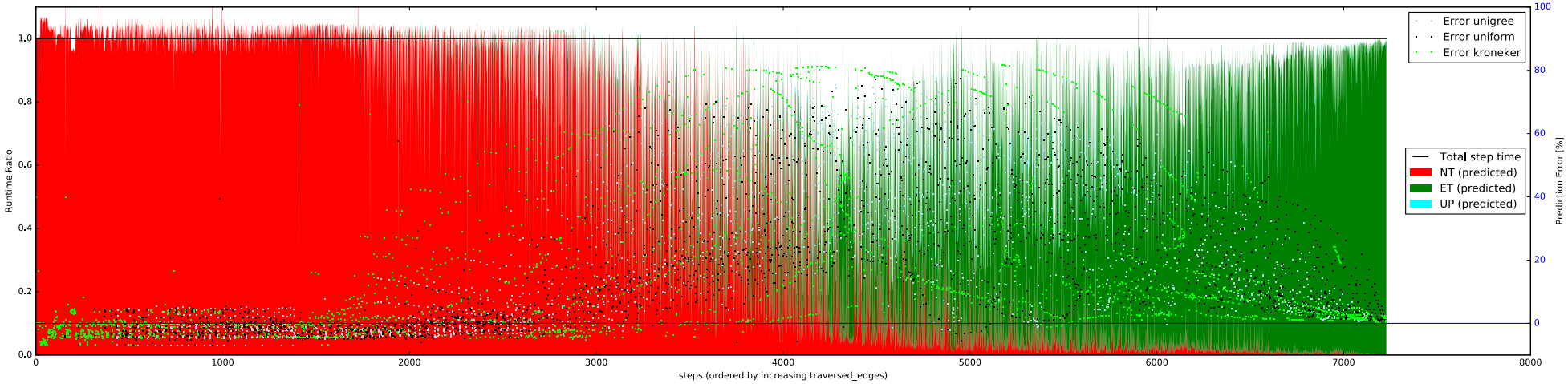
Outlook – Breadth-First-Search

- Naïve prediction model:



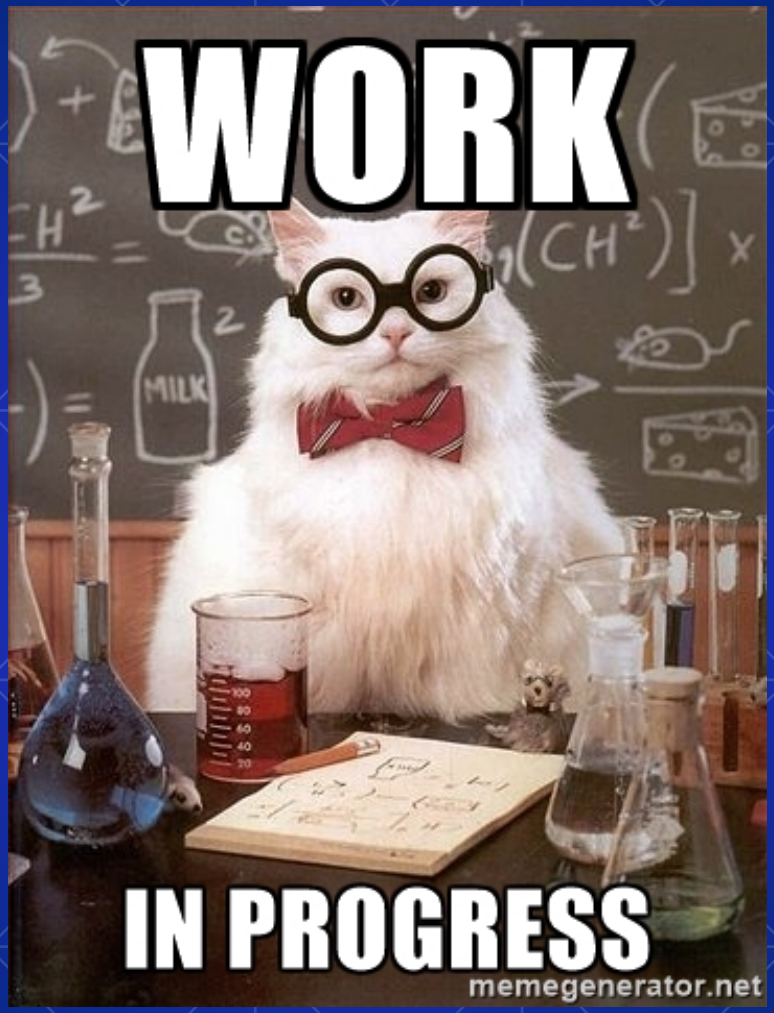
Outlook – Breadth-First-Search

- Naïve prediction model, stepwise:



Outlook

- Base model on queue network theory
- Major challenge:
  Which graph properties to take into account?
- Many common–but unsupported–building blocks.
  - Indirect accesses (graph in CSR format)
  - Branches
  - Pointer chasing (Shiloach-Vishkin for Connected-Components)
  - …


WORK
IN PROGRESS
memegenerator.net

Thank You for Your Attention!

Julian Hammer <julian.hammer@fau.de>
RRZE High Performance Computing Group
http://www.rrze.fau.de/hpc

Kerncraft – In-core Prediction

1. Transform
2. Compile to assembly
3. Mark inner loop
4. Extract unrolling factor
5. Compile to binary
6. Analyze with IACA

```
double a[N][N];
double b[N][N];
double s;

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

Kerncraft – In-core Prediction

1. **Transform**
   - Linearized arrays and malloc
2. Compile to assembly
3. Mark inner loop
4. Extract unrolling factor
5. Compile to binary
6. Analyze with IACA

```
#include <stdlib.h>

void dummy(double *);
extern int var_false;

int main(int argc, char **argv) {
  const int N = atoi(argv[2]);
  const int M = atoi(argv[1]);
  double *a = _mm_malloc(
    (sizeof(double)) * (M * N), 32);
  for (int i = 0; i < (M * N); ++i) a[i] = 0.23;
  if (var_false) dummy(a);

  double *b = _mm_malloc(
    (sizeof(double)) * (M * N), 32);
  for (int i = 0; i < (M * N); ++i) b[i] = 0.23;
  if (var_false) dummy(b);

  double s = 0.23;
  if (var_false) dummy(&s);

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

Kerncraft – In-core Prediction

1. Transform
2. Compile to assembly
3. **Mark inner loop**
   - Detection heuristic
4. Extract unrolling factor
5. Compile to binary
6. Analyze with IACA

```
[...]
..B1.25:
       vmovddup  %xmm1, %xmm0
        movslq    %r9d, %rdi
        vinsertf128 $1, %xmm0, %ymm0, %ymm0
        movl      $111, %ebx # INSERTED BY KERNCRAFT
        .byte     100        # INSERTED BY KERNCRAFT
        .byte     103        # INSERTED BY KERNCRAFT
        .byte     144        # INSERTED BY KERNCRAFT

..B1.26:
        vmovupd   (%rbx,%rsi,8), %xmm2
        vmovupd   16(%rbx,%rsi,8), %xmm3
        vmovupd   32(%rbx,%rsi,8), %xmm14
[...]
        vmulpd    %ymm7, %ymm0, %ymm8
        vmovupd   %ymm8, 104(%r8,%rsi,8)
        addq      $16, %rsi
        cmpq      %rdi, %rsi
        jb        ..B1.26
        movl      $222, %ebx # INSERTED BY KERNCRAFT
        .byte     100        # INSERTED BY KERNCRAFT
        .byte     103        # INSERTED BY KERNCRAFT
        .byte     144        # INSERTED BY KERNCRAFT
..B1.28:
[...]
```

Kerncraft – In-core Prediction

1. Transform
2. Compile to assembly
3. Mark inner loop
4. **Extract unrolling factor**
   - From mem. ref. increments
5. Compile to binary
6. Analyze with IACA

```
[...]
..B1.25:
       vmovddup  %xmm1, %xmm0
        movslq    %r9d, %rdi
        vinsertf128 $1, %xmm0, %ymm0, %ymm0
        movl      $111, %ebx # INSERTED BY KERNCRAFT
        .byte     100        # INSERTED BY KERNCRAFT
        .byte     103        # INSERTED BY KERNCRAFT
        .byte     144        # INSERTED BY KERNCRAFT

..B1.26:
        vmovupd   (%rbx,%rsi,8), %xmm2
        vmovupd   16(%rbx,%rsi,8), %xmm3
        vmovupd   32(%rbx,%rsi,8), %xmm14
[...]
        vmulpd    %ymm7, %ymm0, %ymm8
        vmovupd   %ymm8, 104(%r8,%rsi,8)
        addq      $16, %rsi
        cmpq      %rdi, %rsi
        jb        ..B1.26
        movl      $222, %ebx # INSERTED BY KERNCRAFT
        .byte     100        # INSERTED BY KERNCRAFT
        .byte     103        # INSERTED BY KERNCRAFT
        .byte     144        # INSERTED BY KERNCRAFT
..B1.28:
[...]
```

Kerncraft – In-core Prediction

1. Transform
2. Compile to assembly
3. Mark inner loop
4. Extract unrolling factor
5. Compile to binary
6. **Analyze with IACA**

   → 2D and 3D are LOADs

```
Throughput Analysis Report
--------------------------
Block Throughput: 18.90 Cycles
Throughput Bottleneck: FrontEnd, PORT2_AGU, PORT3_AGU

Port Binding In Cycles Per Iteration:
---------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D |
---------------------------------------------------------
| Cycles | 10.1   0.0  | 12.0 | 18.0   16.0 | 18.0  16.0|
---------------------------------------------------------

------------------------
|  Port  |  4   |  5   |
------------------------
| Cycles | 8.0  | 11.9 |
------------------------
```

Kerncraft – Cache Prediction

1. Parse kernel code
2. Enforce restrictions
3. Extract data accesses
4. Calculate cache accesses
    1. Compile offsets to fill all cache levels
    2. Reset cache simulator stats
    3. Execute next cache line accesses
    4. Check for hits/misses

```
double a[N][N];
double b[N][N];
double s;

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

Kerncraft – Cache Prediction

1. Parse kernel code
2. Enforce restrictions
3. **Extract data accesses**
4. Calculate cache accesses
   1. Compile offsets to fill all cache levels
   2. Reset cache simulator stats
   3. Execute next cache line accesses
   4. Check for hits/misses

```
variables:     name |   type size
              ------+-------------------------
                 a | double [M, N]
                 s | double None
                 b | double [M, N]


loop stack:             idx |        min        max       step
                        ----+-----------------------------------
                          j |          1      M - 1          1
                          i |          1      N - 1          1


data sources:          name |  offsets   ...
                       -----+-----------...
                          a | [j, i - 1]
                            | [j, i + 1]
                            | [j - 1, i]
                            | [j + 1, i]
                          s | None


data destinations:     name |  offsets   ...
                       -----+-----------...
                          b | [j, i]


constants:     name | value
              ------+-----------
                   N | 511
                   M | 511
```