Achieving Performance Portability for Extreme Heterogeneity

Mary Hall

WACCPD

November 13, 2020

Collaborators and Acknowledgements

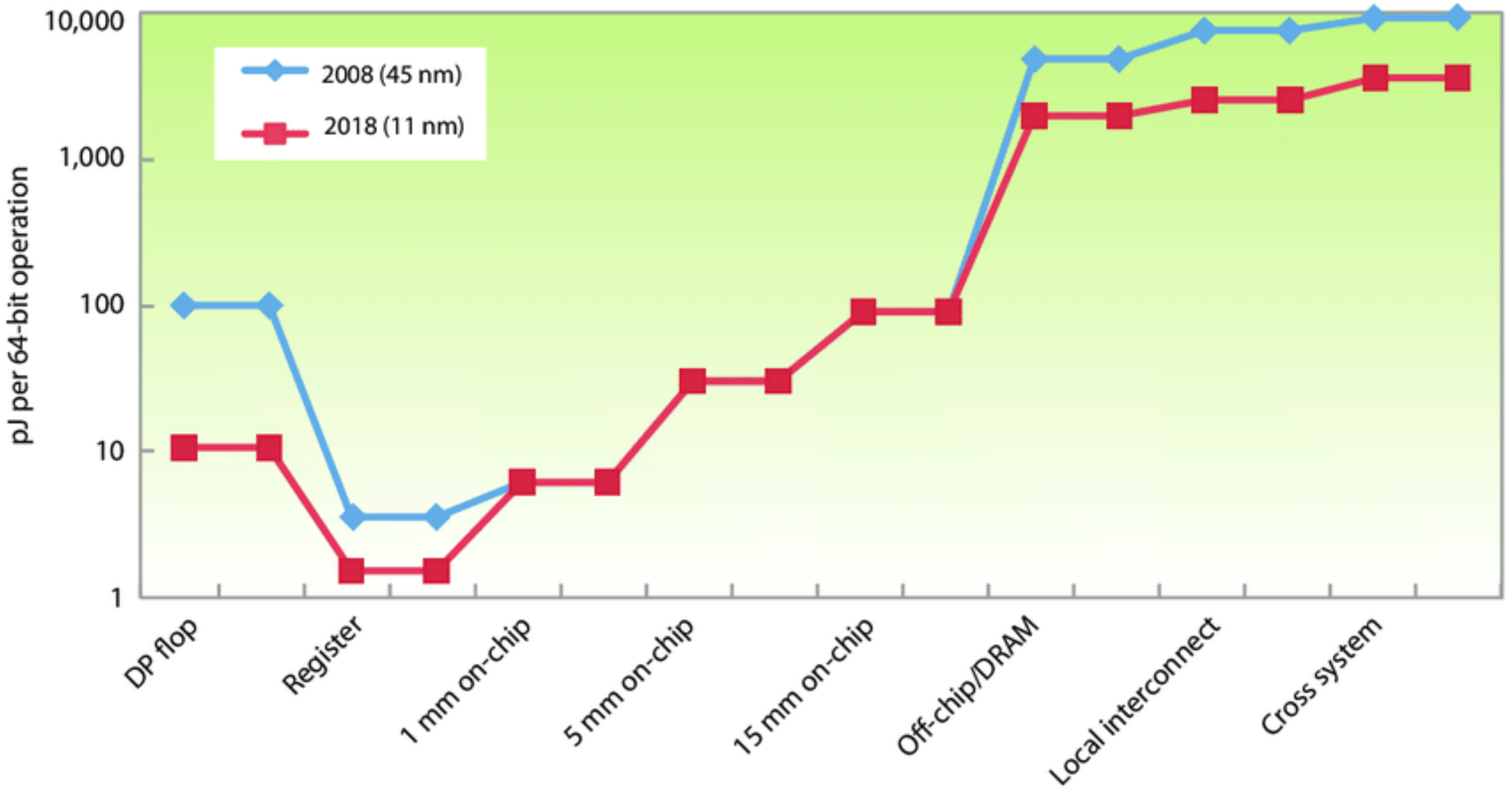
University of Utah (students)
David Fridlander, Rajath Javali, Vinu Sreenivasan, Hong Yeung
Lawrence Livermore National Laboratory
Tom Scogland, Bronis de Supinski
Lawrence Berkeley National Laboratory
Sam Williams, Hans Johansen
Argonne National Laboratory
Prasanna Balaprakash, Xingfu Wu, Brice Videau

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

This research used resources in Lawrence Berkeley National Laboratory and the National Energy Research Scientific Computing Center, which are supported by the U.S. Department of Energy Office of Science's Advanced Scientific Computing Research program under contract number DE-AC02-05CH11231.

Current Context

Cost of Data Movement



Source: P. Kogge and J. Shalf, "Exascale Computing Trends: Adjusting to the "New Normal"' for Computer Architecture," in *Computing in Science & Engineering, Nov.-Dec. 2013.*

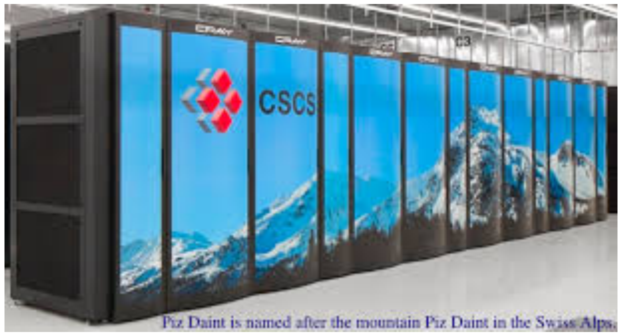
Performance Portability Challenge

Can the same program perform well on diverse supercomputing platforms? (e.g., Top 500 list, top500.org)

#1: Summit, IBM Power9+V100 GPUs

#3: TaihuLight, Sunway

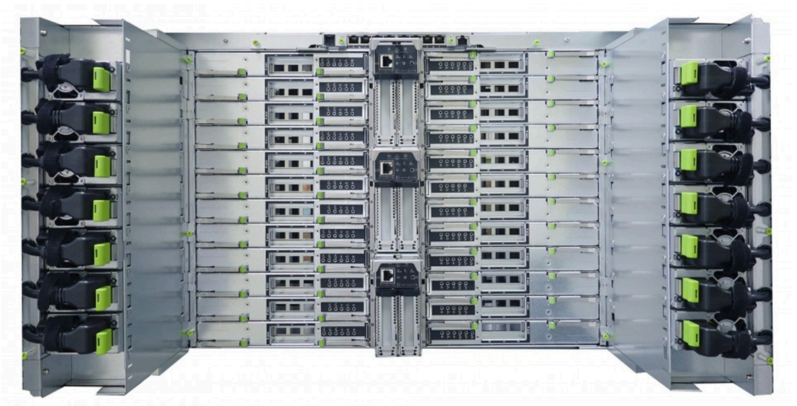
#4: Tianhe-2, Intel Xeon Phis


CSCS
Piz Daint is named after the mountain Piz Daint in the Swiss Alps.

#6: Piz Daint, Intel Xeon+P100 GPUs

What's Coming Next?

Aurora, Intel Xeon + Intel X Compute

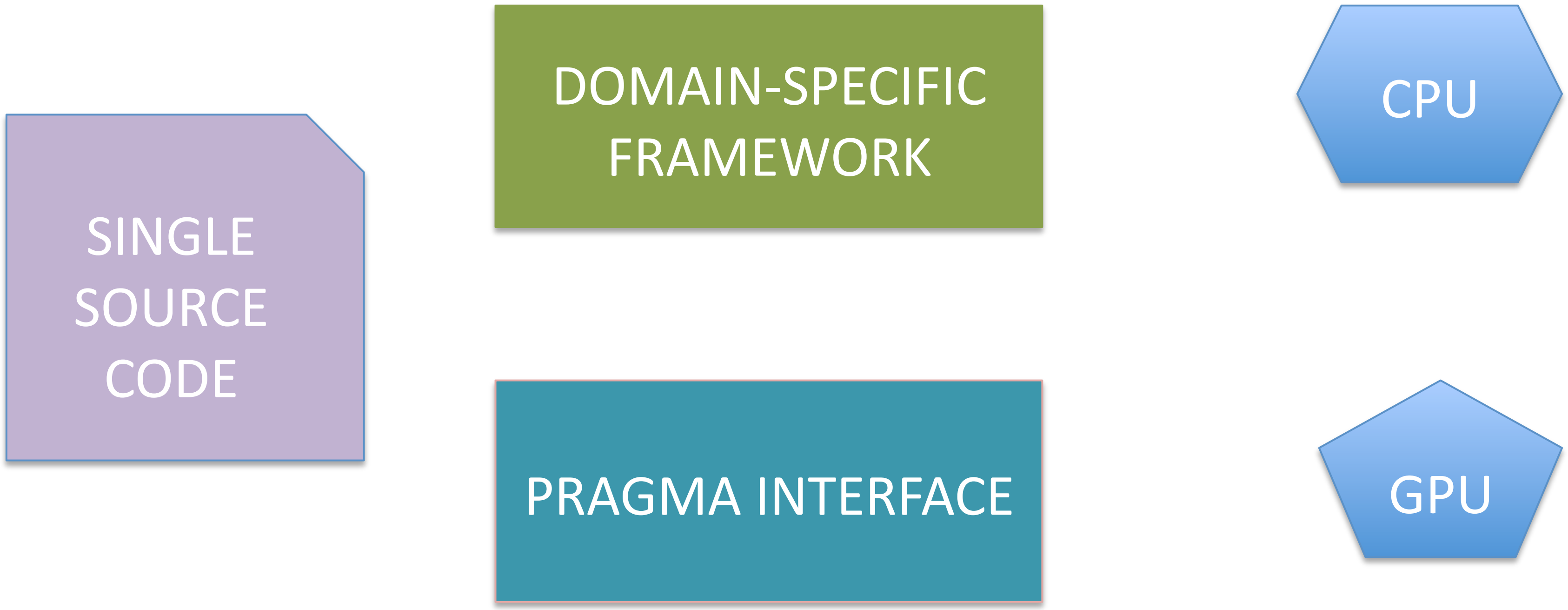


Fugaku (Riken), ARM + custom optimizations

Frontier, AMD EPYC CPU + AMD GPU

Single-Source Performance Portability

Single Source Performance Portability



Pragma Interface

Key Idea

- Add pragmas to existing (sequential) code
- Programmer productivity
- Achieve high performance across platforms

Examples

- OpenMP
- OpenACC
- Compiler pragmas, e.g., Clang/Polly

Domain-Specific Programming System

Key Idea

- Customize optimization for a specific application domain, easier than general purpose
- Programmer Productivity
- Achieve high performance and performance portability

Examples

- Stencils: ExaStencils, YASK, Open Climate Compiler, Pochoir
- Sparse linear algebra: MT1, Bernoulli, Taco
- Tensor contraction: TCE
- Big Data: Map-reduce, Spark
- Deep Learning: TensorFlow, PyTorch

Alternate Solutions

- Portability Frameworks, e.g., Kokkos, RAJA
- Parallelizing Compiler Technology, e.g., MLIR

Overlap, and ***all*** should be part of ecosytem

Solution 1:

Pragma Autotuning

Pragma-Based Node Solutions

Concerns

- Same OpenMP directives across different platforms?
- Programmer writes different directives to port code?
- Reduces value of using a productivity programming model?

Solution

- Use autotuning to identify best-performing OpenMP pragmas
- ***From descriptive to prescriptive OpenMP***

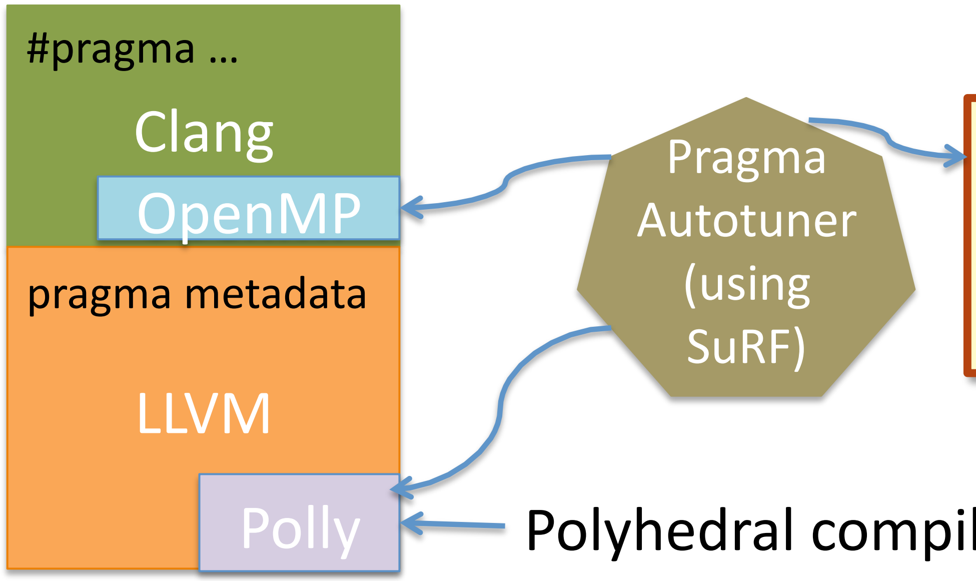
This talk focuses on OpenMP, but other pragma extensions can use the same approach.

Pragma Autotuner

Search Using Random Forest (SuRF) for autotuning search
ytopt toolkit

```
/* Clang/Polly example */
#pragma clang loop unroll(4)
for (int i = 0; i < n; i+=1)  Statement(i);
```

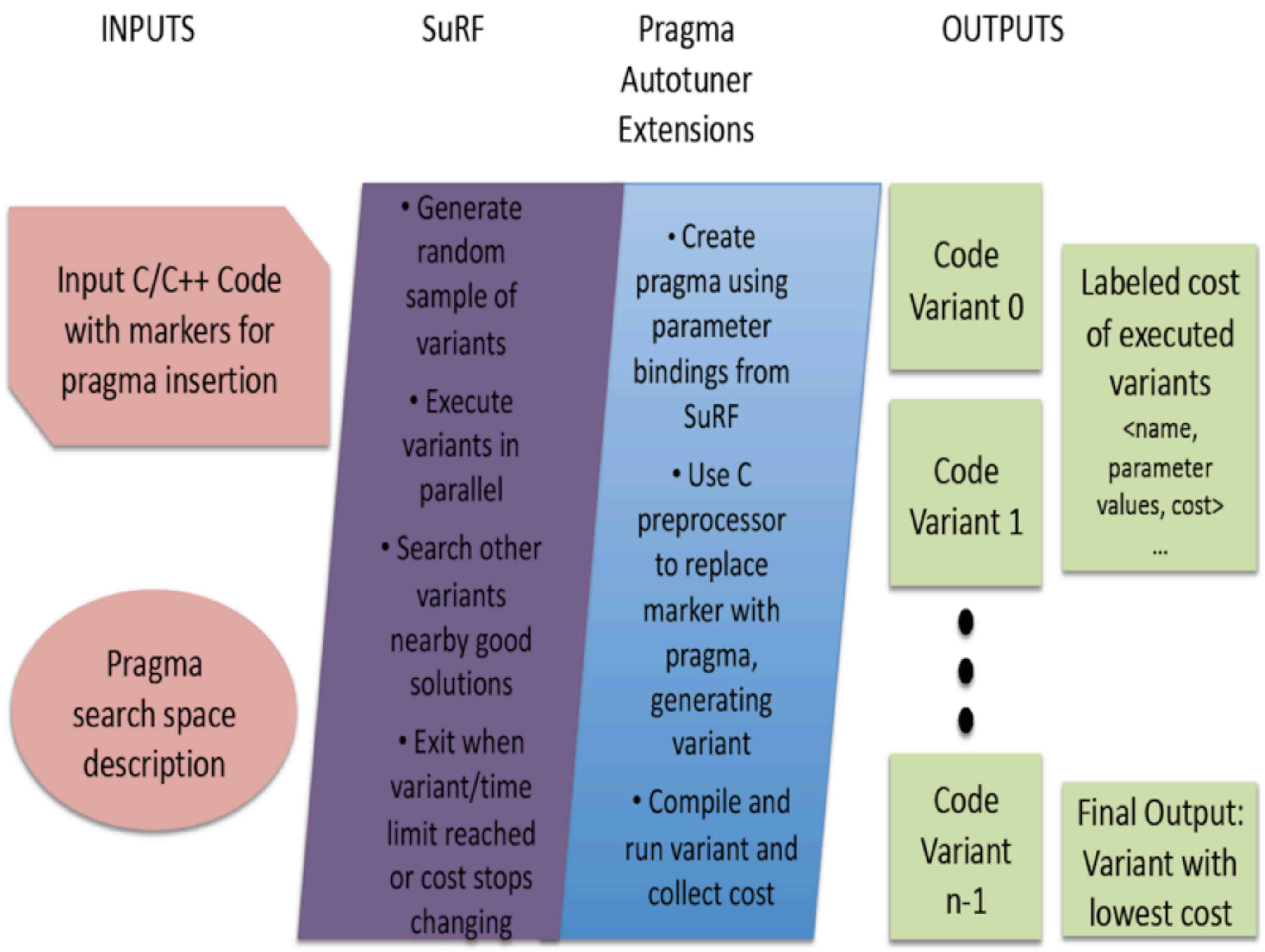
```
/* OpenMP example */
#pragma omp parallel loop
for (int i = 0; i < n; i+=1)  Statement(i);
```



```
/* OpenMP example */
#pragma omp target team distribute
parallel for
for (int i = 0; i < n; i+=1)  Statement(i);
```

Polyhedral compiler in LLVM

System Overview


INPUTS
SuRF
Pragma Autotuner Extensions
OUTPUTS
Input C/C++ Code with markers for pragma insertion
Pragma search space description
• Generate random sample of variants
• Execute variants in parallel
• Search other variants nearby good solutions
• Exit when variant/time limit reached or cost stops changing
• Create pragma using parameter bindings from SuRF
• Use C preprocessor to replace marker with pragma, generating variant
• Compile and run variant and collect cost
Code Variant 0
Code Variant 1
Code Variant n-1
Labeled cost of executed variants <name, parameter values, cost> …
Final Output: Variant with lowest cost

From Descriptive to Prescriptive OpenMP

OpenMP
loop

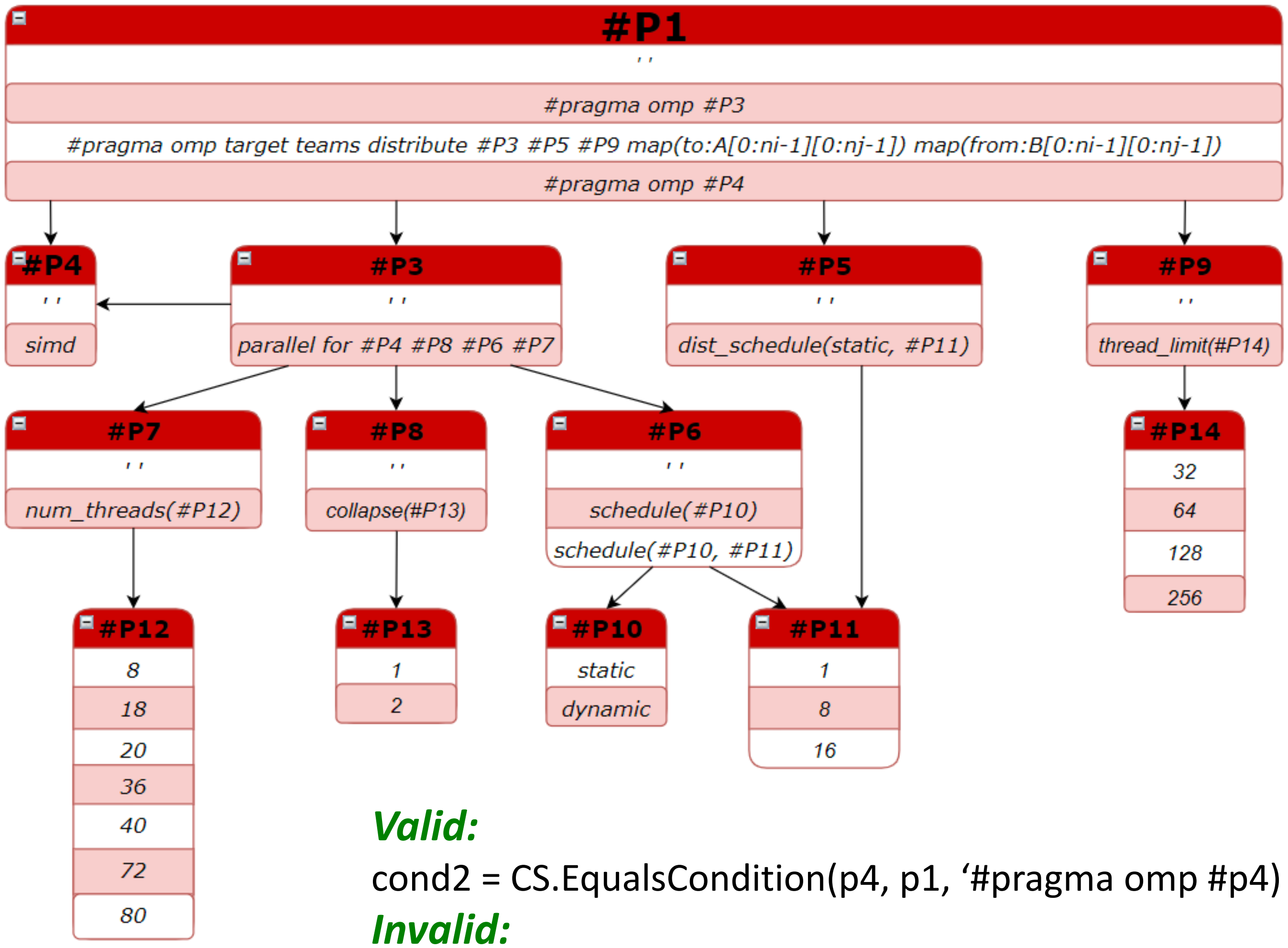
A `loop` construct specifies that the iterations of the associated loops may execute concurrently and permits the encountering thread(s) to execute the loop accordingly.

Use autotuning to identify a replacement OpenMP construct that is more prescriptive

convolution_2d.c (Polybench):

```
int i, j;
// P0
#pragma omp parallel loop
 for (i = 1; i < _PB_NI - 1; ++i) {
  // P1
  #pragma omp loop
  for (j = 1; j < _PB_NJ - 1; ++j) {
          B[i][j] =  0.2 * A[i-1][j-1] + 0.5 * A[i-1][j] + -0.8 * A[i-1][j+1]
           + -0.3 * A[ i ][j-1] + 0.6 * A[ i ][j] + -0.9 * A[ i ][j+1]
           +  0.4 * A[i+1][j-1] + 0.7 * A[i+1][j] +  0.1 * A[i+1][j+1];
  }
 }
```

OpenMP Pragma Search Space



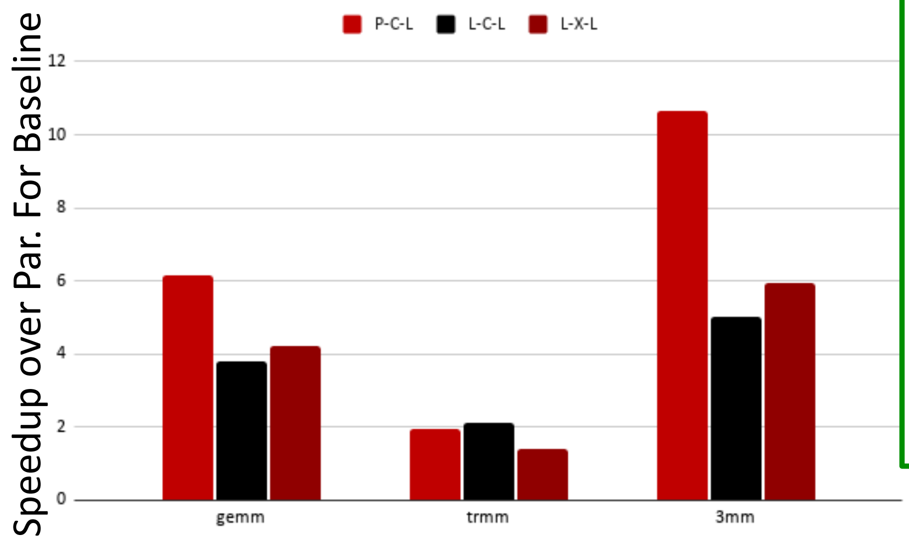
Valid:

cond2 = CS.EqualsCondition(p4, p1, ‘#pragma omp #p4)

Invalid:

Forbidden_clause = CS.ForbiddenAndConjunction(CS.ForbiddenEqualsClause(p1, ‘#pragma omp #P4’), CS.ForbiddenEqualsClause(p4, ‘ ‘)).

Performance Results



Differences:

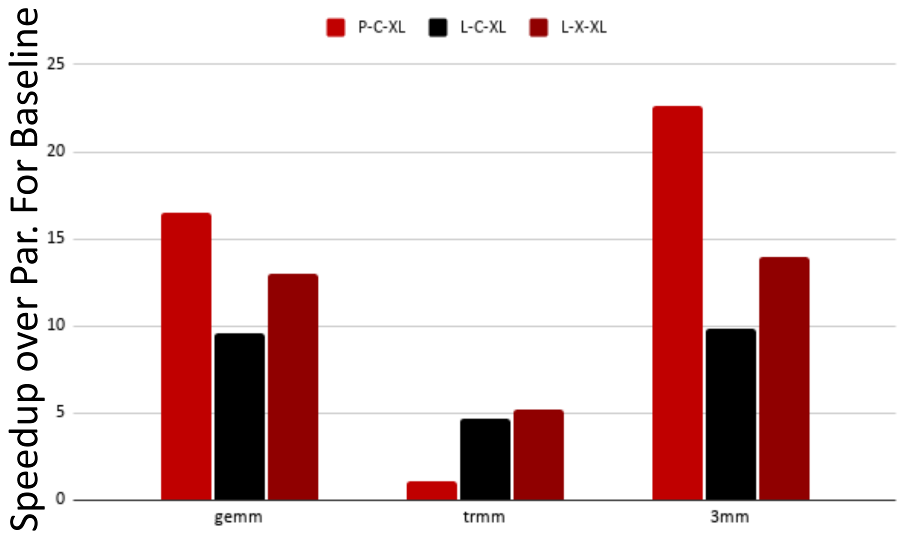
#pragma omp parallel for simd num_threads(72)

#pragma omp target teams distribute parallel for collapse(2) num_threads(40) dist_schedule(static, 8) thread_limit(32) is_device_ptr(A, B)

#pragma omp target teams distribute parallel for **simd** collapse(2) schedule(static)   dist_schedule(static, 16) thread_limit(32) is_device_ptr(A, B)

Differences Result From:

- Parallelism granularity and architecture
- Compiler capability (e.g., simd)



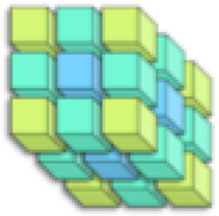
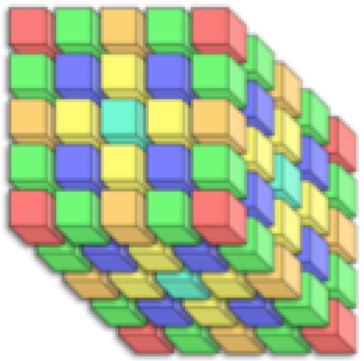
Results on Pascal and Lassen, LLNL, using Clang and XLC compilers, L and XL inputs

Solution 2:

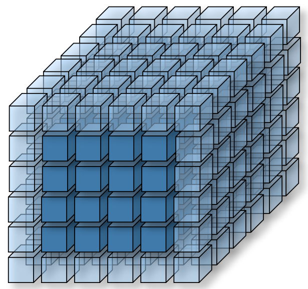
Domain-Specific Optimization of Data Movement

Stencil Computations

- Solve partial differential equations
  - Outputs computed from neighbors in multi-dimensional space
  - Multiplied by coefficient
- Access pattern arises in convolutions too
- Number of inputs related to order of stencil
  - Low order – memory bound
  - High order – compute intensive

Packed Data Layouts: a Small Unit of Data and Work



Idea

- Domain-specific programming system designed around a unit of ***data*** and ***parallel work***
- ***Data layout*** for each node is a collection of these units
- Flexible organization and adaptivity addresses ***performance portability***

Result

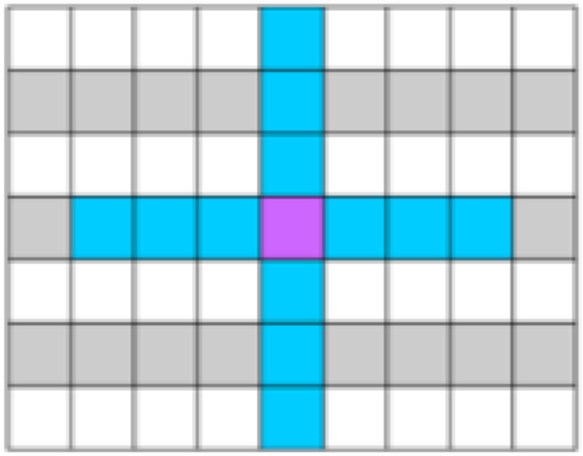
- Speeds up data movement
- Reduces need for data movement
- Reduces on-node data movement for communication, improving strong scaling

Sources of Data Movement

- H→ : Horizontal data movement, across nodes via interconnect
- V↑ : Vertical data movement, through a node's memory system

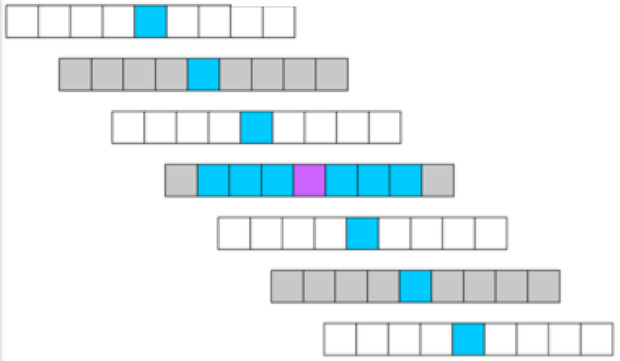
Description	Example
H→ : NODE ⇔ NODE	Send data from one node's memory to another's
V↑ : Memory ⇔ Cache	Load data into cache
V↑ : Cache ⇔ Register	Load data resident in cache into a (vector) register
V↑ : Global Memory ⇔ TLB	Lookup page table in memory to cache virtual to physical address mapping
V↑ : CPU ⇔ GPU	Load data from GPU memory into host CPU memory
H→ : GPU ⇔ GPU	Communicating GPU data to other nodes' GPUs

Vertical Data Movement for Stencils



Example: 13-point stencil

Out[i][j] = coeff*(In[i][j-3]+
In[i][j-2]+ … In[i][j+3]+
In[i-3][j]+In[i-2][j]+…
In[i+2][j]+ In[i+3][j]);



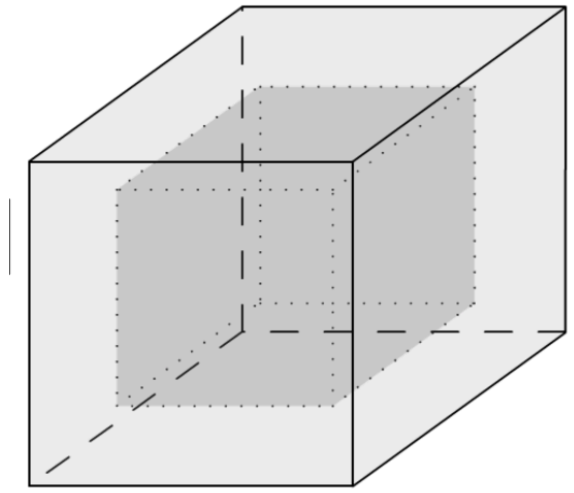
13-point stencil
7 distinct address streams

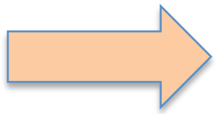
Vertical data movement impact

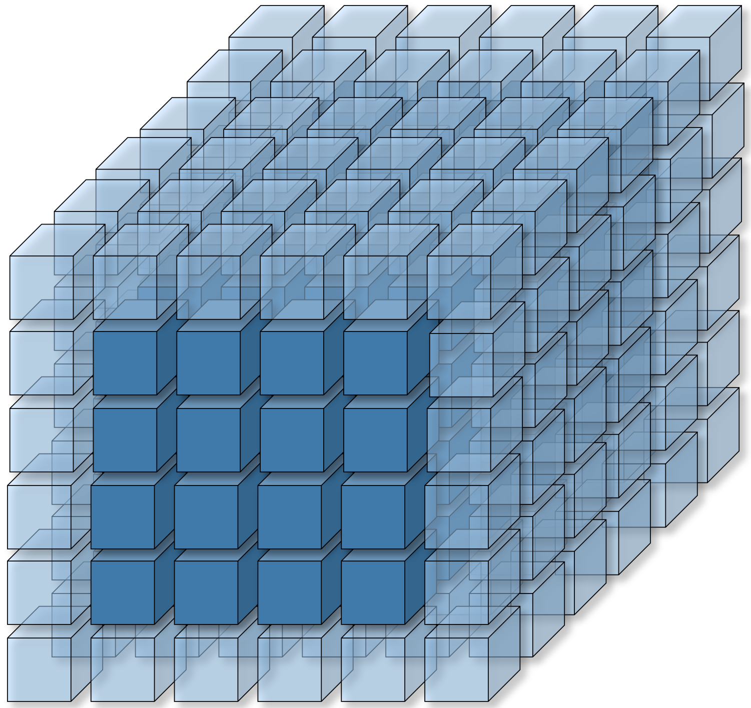
- Capacity misses in caches and TLB
- Limits hardware prefetching effectiveness
- Reordering in registers

Many-core parallelism & tiling make this worse

Solution: Brick Data Layout







Brick Data Layout + Code Generator

- A brick is a mini (e.g., 8x8x8) subdomain without a ghost zone
- Application of a stencil reaches into other bricks (affinity important)
- Implemented with contiguous storage and adjacency lists

[Zhao et al., PP3HPC 2018] [Zhao et al., SC 2019]

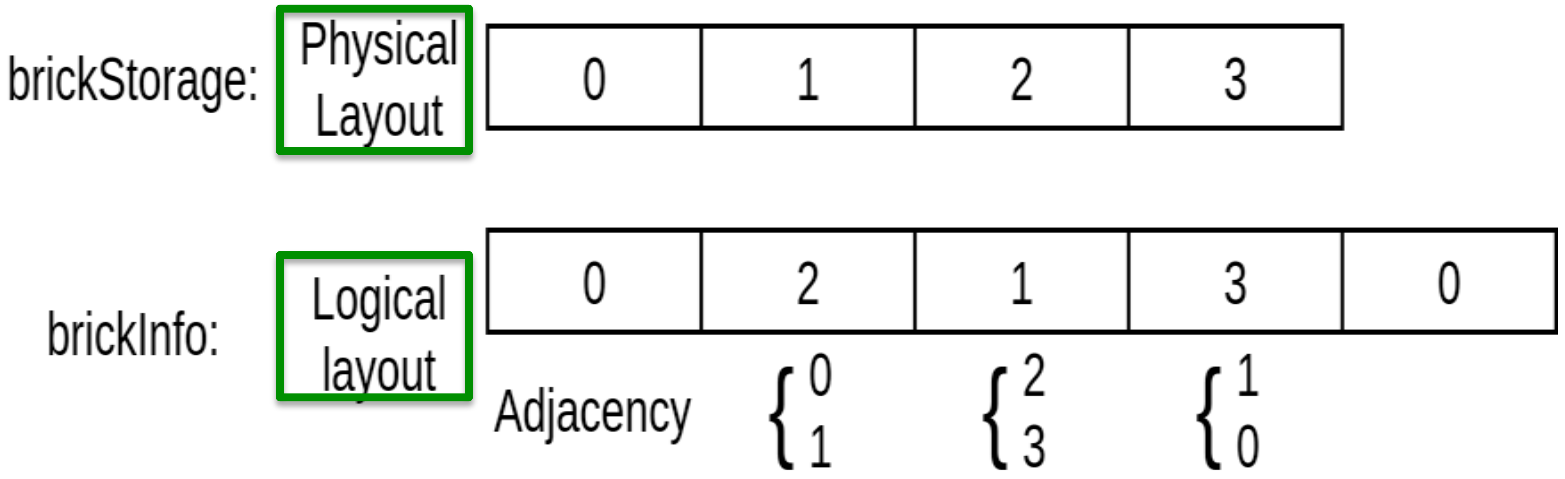
Brick Library Example

- Operates on brick input and output arrays In/Out
- Accesses outside of brick **b** are automatically resolved
- Stencil for code generation expressed in Python
- DAG representation for performance portable “vector” code generation

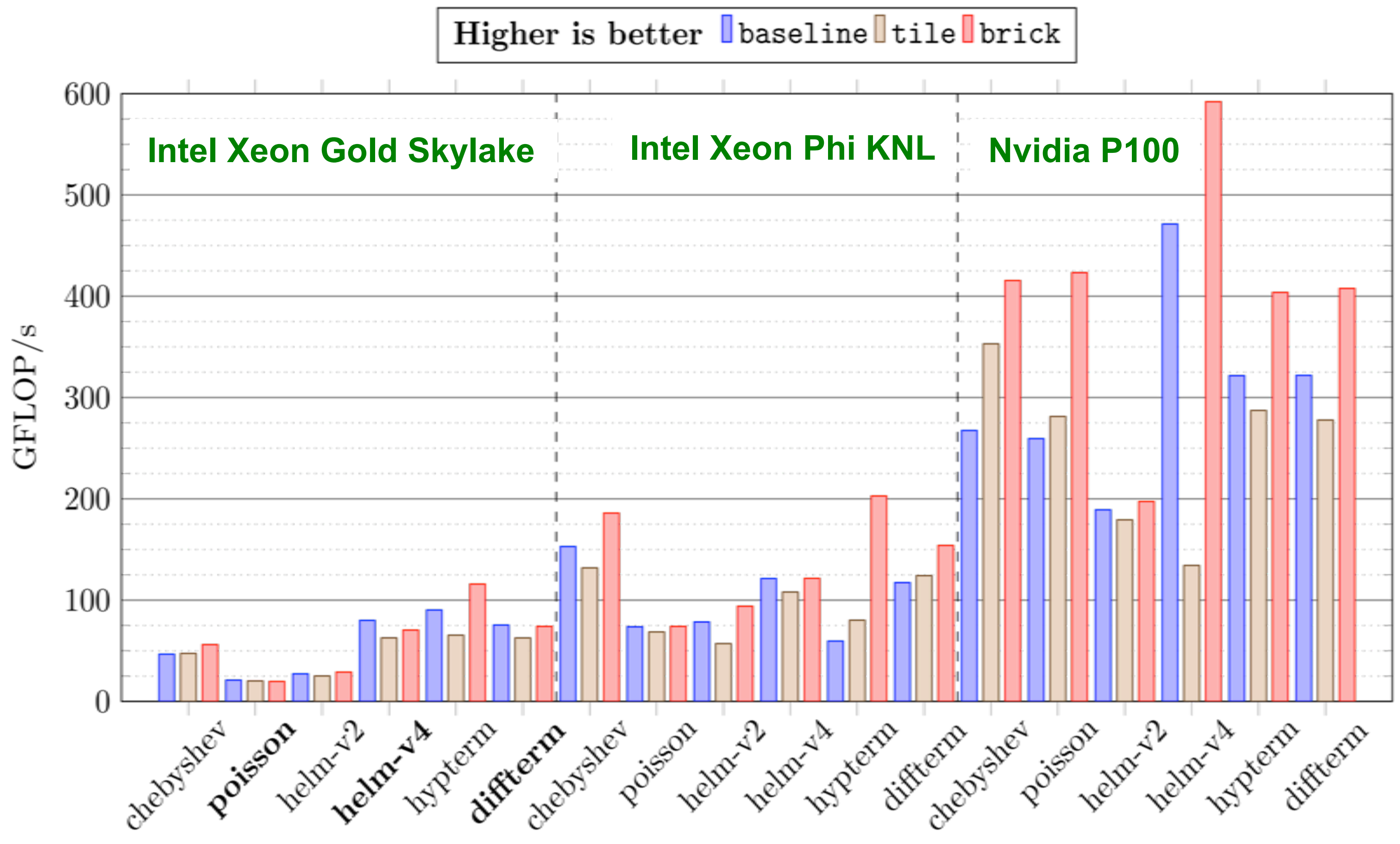
```
Brick<Dim<8,8>, Dim<2,2>>
In(&brickInfo, brickStorage, 0);
...
for (long b: allbricks)
  for (long j = 0; j < 8; ++j)
    for (long i = 0; i < 8; ++i)
      Out[b][j][i] = In[b][j][i] * coeff[0] +
          In[b][j][i+1] * coeff[1] +
          In[b][j][i-1] * coeff[2] +
          In[b][j+1][i] * coeff[3] +
          In[b][j-1][i] * coeff[4];
```

Aggregating Brick Collection

- Collection of neighboring bricks co-located for thread/node
- Indirection permits different ***physical*** layout from ***logical*** organization

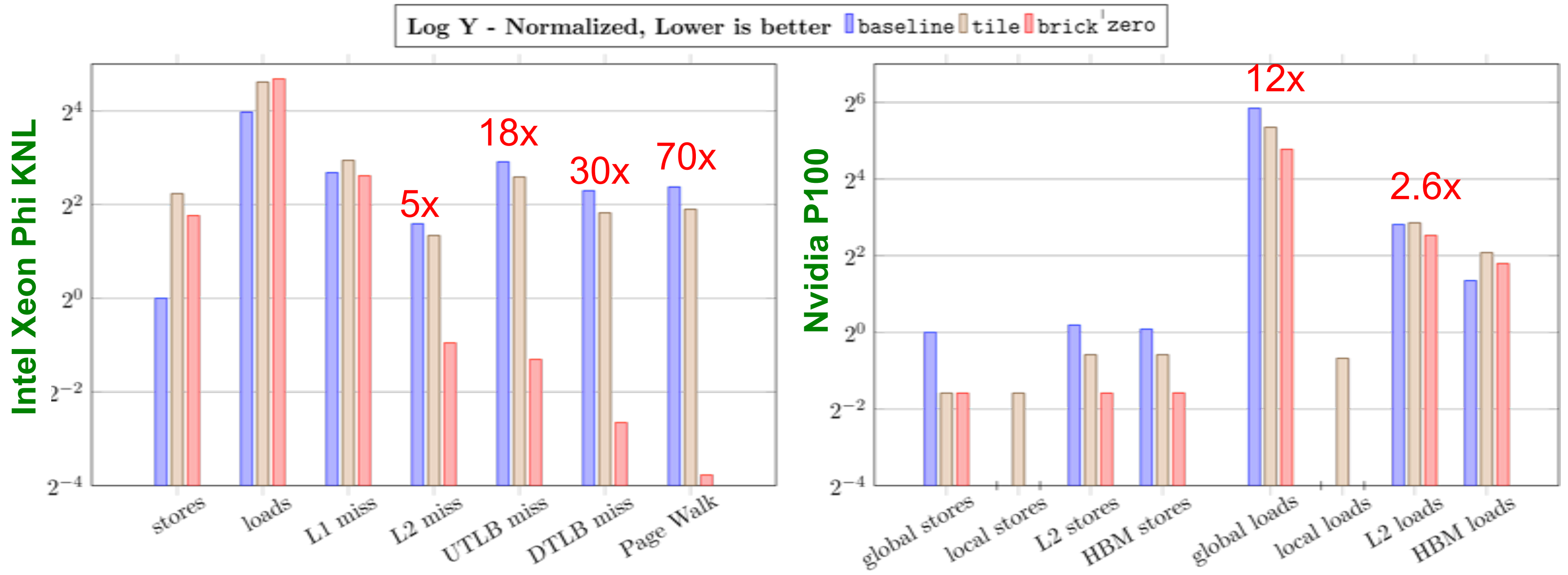


Single Node Performance



Maximum	1.2x	3.4x	1.6x
Average	~1.06x	~1.52x	~1.33x

Single Node Hypterm: Bricks Reduce Vertical Data Movement



Much better cache locality
Much less TLB pressure

Much better register reuse

T. Zhao, P. Basu, S. Williams, M. Hall, and H. Johansen. 2019. Exploiting reuse and vectorization in blocked stencil computations on CPUs and GPUs. SC'19.

Overlap of Approaches

Complementary

DOMAIN-SPECIFIC FRAMEWORK

- Provide data layout abstraction and primitives
- Provide domain-specific optimization and code generation

PRAGMA INTERFACE

- Thread and simd code generator for DSL
- Use for portions of code outside supported domains

Closing Remarks

- Different implementations are needed for different platforms …
- … but single source performance portability is possible
- Lots of exciting work ahead in the programming system ecosystem (compiler, DSL, portability, pragmas)
- Data movement drives performance portability, how to support data layout?