# *Performance Portability Challenges for Exascale Computing*

P. (Saday) Sadayappan
Ohio State University

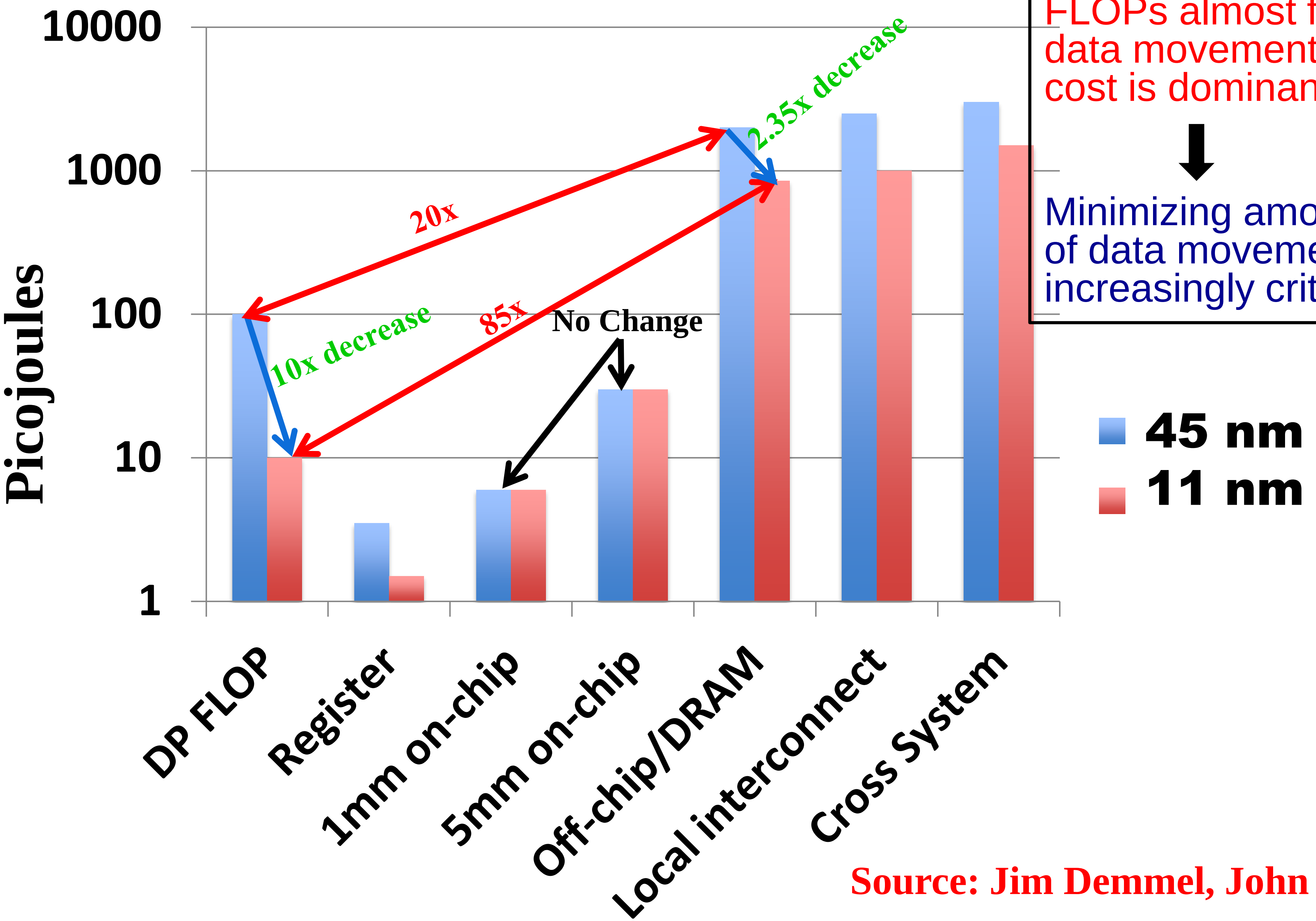
SCEC 2018
December 13, 2018

# High-Performance Software Development Challenge

- Low Performance
  - Very challenging to achieve high performance for GPUs and FPGAs
  - Requires understanding of low-level arch. details
- Low Productivity
  - Need to program using different programming models: OpenMP for multicores, CUDA/OpenCL for GPUs, Verilog/VHDL for FPGAs
  - Steep learning curve: CUDA known by few; Verilog/VHDL known by fewer
  - Parallel programming is much more difficult than sequential C/C++
- No Portability
  - Multiple versions of code must be maintained for different platforms
- Challenges will get worse in the future: compilers must do more!
  - Research Direction 1: Understanding Data Movement Complexity
  - Research Direction 2: Domain/Pattern-Specific Transformation/Code-Gen

# Research Direction 1: Data Movement Complexity

# Data Movement Cost: Energy Trends



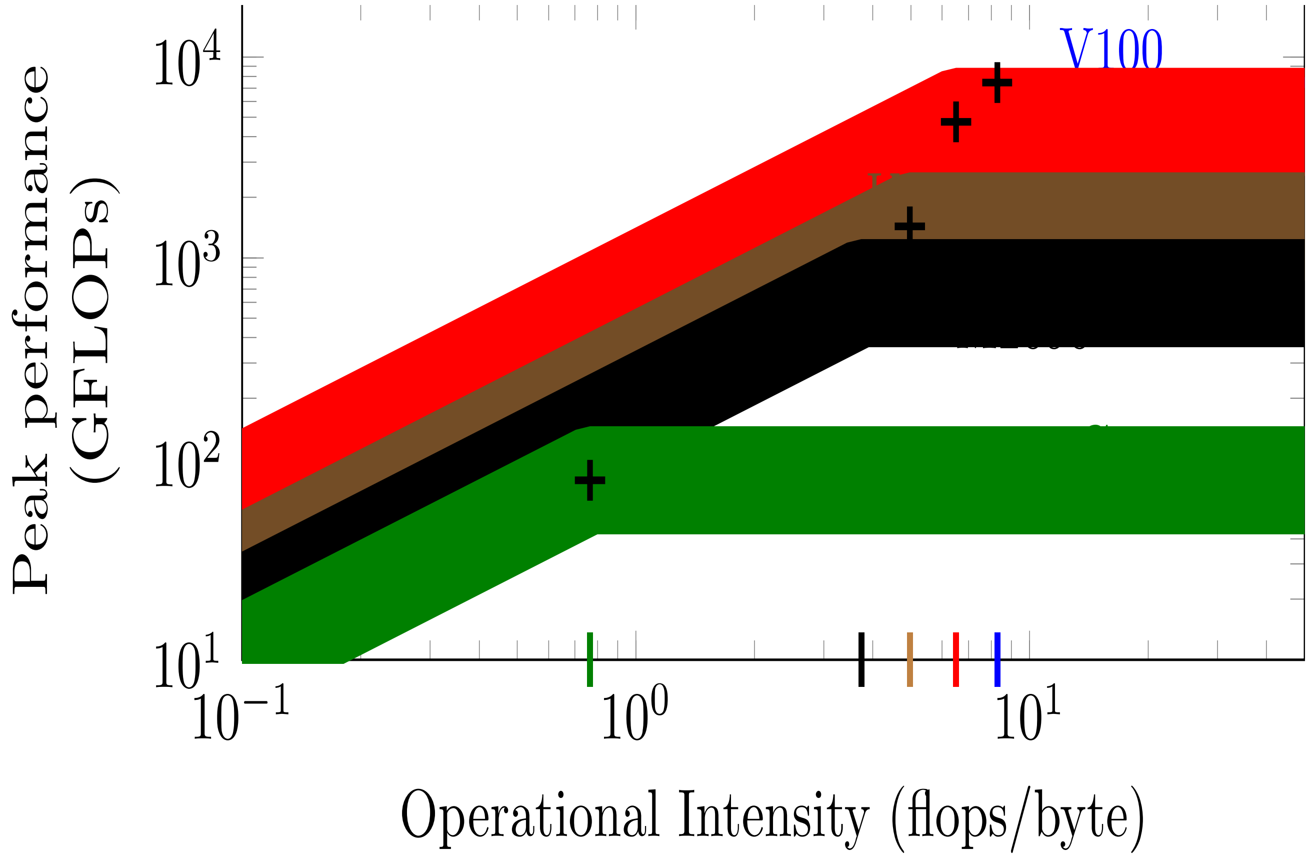

FLOPs almost free; data movement cost is dominant

⬇

Minimizing amount of data movement increasingly critical

**Source: Jim Demmel, John Shalf**

# Data Movement Cost: Performance Trends




- Nvidia GPUs over 5 generations: Fermi, Maxwell, Kepler, Pascal, Volta
- Peak GFLOPs and Peak Mem BW have both increased
- But machine balance (Peak_GFLOPs/Peak_BW) has steadily risen => more and more constrained by data movement

# Computational vs. Data Movement Complexity

```
for (i=1; i<N-1; i++)
  for (j=1;j<N-1; j++)
    A[i][j] = A[i][j-1] + A[i-1][j];
```

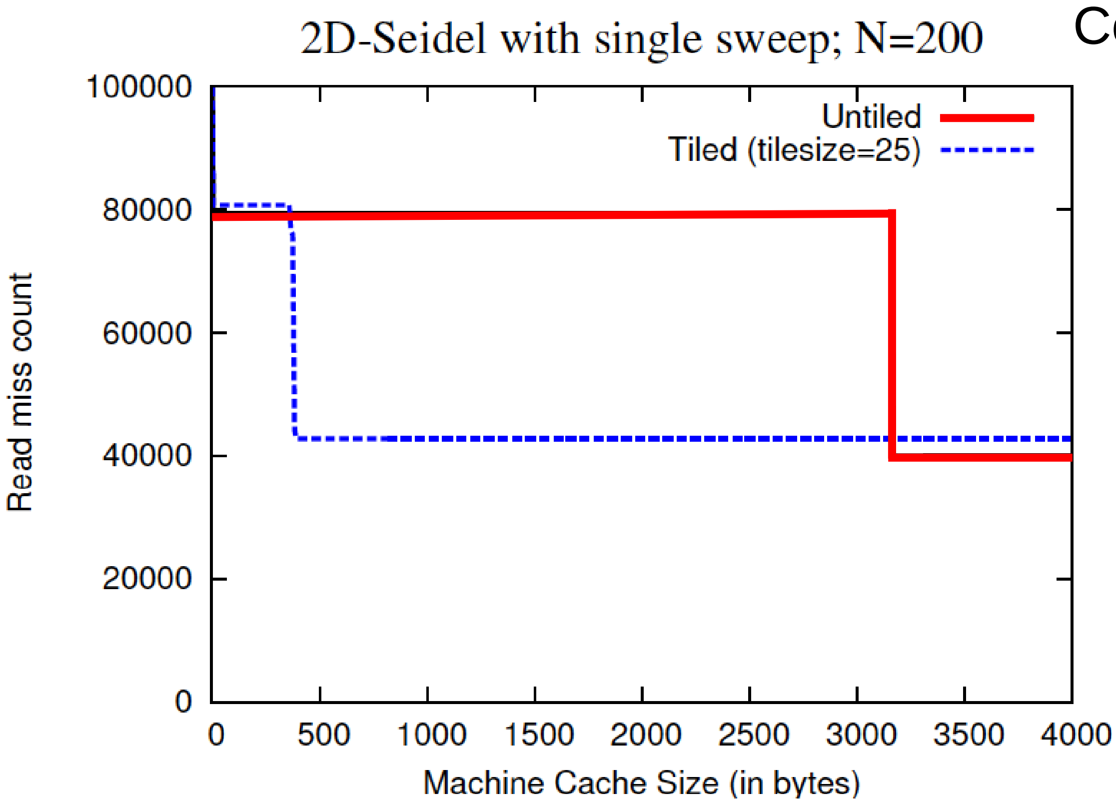
Untiled version

Comp. complexity: $(N-1)^2$ Ops

```
for(it = 1; it<N−1; it +=B)
 for(jt = 1; jt<N−1; jt +=B)
  for(i = it; i < min(it+B, N−1); i++)
   for(j = jt; j < min(jt+B, N−1); j++)
    A[i][j] = A[i−1][j] + A[i][j−1];
```

Tiled Version

Comp. complexity: $(N-1)^2$ Ops




1. Data movement cost is different for two comp. equiv. versions
2. Also depends on cache size

Question: Can we achieve lower cache misses than this tiled version? How can we know when much further improvement is not possible?

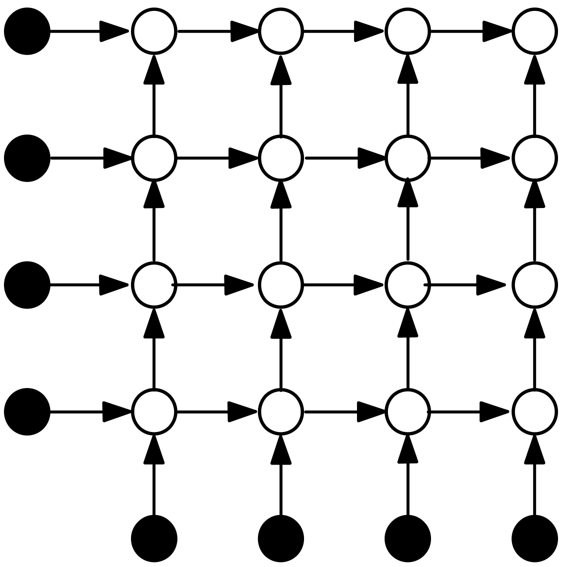
Question: What is the lowest achievable data movement cost among all possible equivalent versions of a computation?

**Current tools do not address this question**

# Modeling Data Movement Complexity: CDAG

```
for (i=1; i<N-1; i++)
  for (j=1;j<N-1; j++)
    A[i][j] = A[i][j-1] + A[i-1][j];
```

```
for(it = 1; it<N−1; it +=B)
  for(jt = 1; jt<N−1; jt +=B)
    for(i = it; i < min(it+B, N−1); i++)
      for(j = jt; j < min(jt+B, N−1); j++)
        A[i][j] = A[i−1][j] + A[i][j−1];
```



CDAG for N=6

**Develop upper bounds on min-cost**

**Minimum possible data movement cost?**

**No known effective solution to problem**

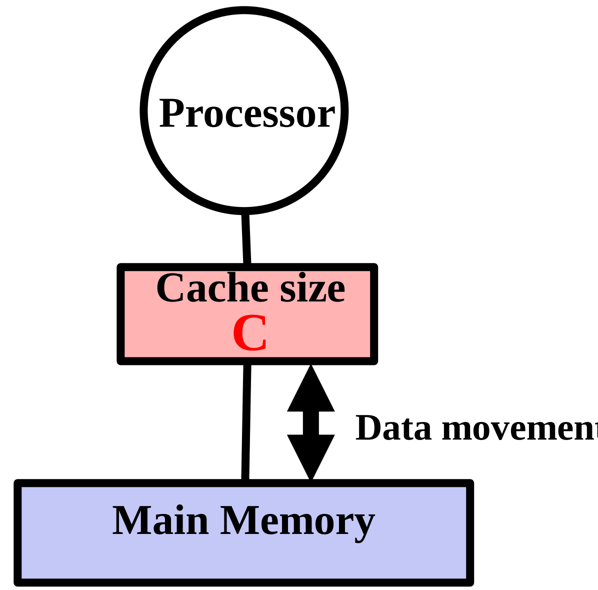
**Develop lower bounds on min-cost**

Hong and Kung: The Red-Blue Pebble Game, STOC 1981

# Lower Bounds: Matrix Multiplication

```
for (i=0; i<N; i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      C[i][j] += A[i][k]*B[k][j];
```

- Hong/Kung [STOC 1981]: Any valid implementation of the standard mat-mult algorithm on a system with cache capacity C will require $\Omega(N^3/\sqrt{C})$ volume of data movement between main-memory and cache
- Irony et al. [JPDC 2004]: Lower bound with scaling constant: $\frac{1}{2\sqrt{2}} \frac{N^3}{\sqrt{C}} - C$
- Dongarra et al. [JFOCS 2008]: Improved constant from $1/(2\sqrt{2})$ to $1.83$
- Smith, Van de Geijn; Langou: 1.83->2
- Efficient tiled execution has data volume of $2N^3/\sqrt{C}$ => Minimal possible data movement!
- Open problem: Bounds for other algorithms
  - Tight data-movement lower bounds (with scaling constants) are unknown for most algorithms



# Example: Erroneous Roofline Limit

- Recent paper contrasts achieved performance by new GPU implementations of sparse-dense matrix-multiplication (SpMM) with GPU “roofline” limits
- But OI used is not a proper upper-bound for SpMM, because it is based on pessimistic reasoning about possible data reuse
- Leads to incorrect conclusion that developed implementation is quite close to “best” possible
  - >150 GFLOPs measured on same GPU with Nvidia’s cuSPARSE SpMM

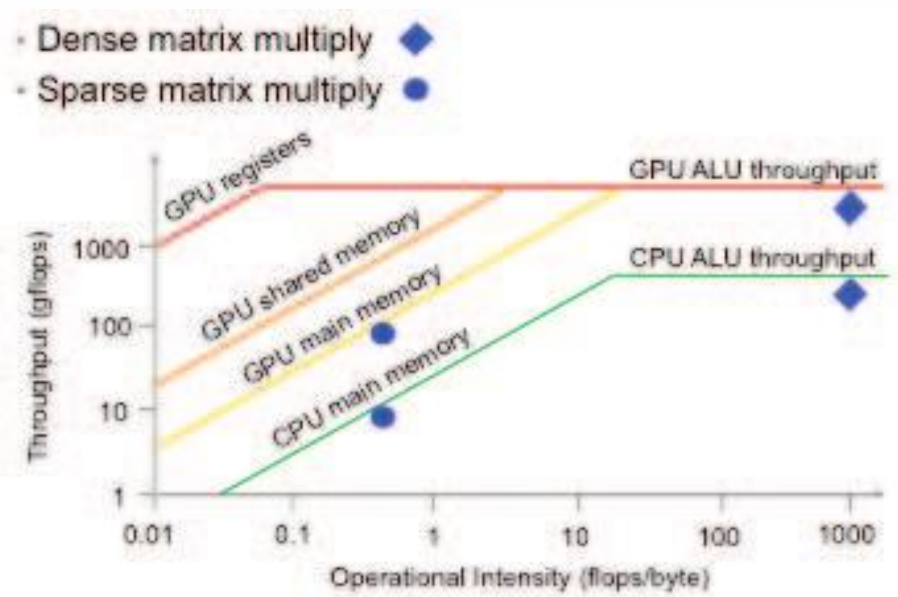
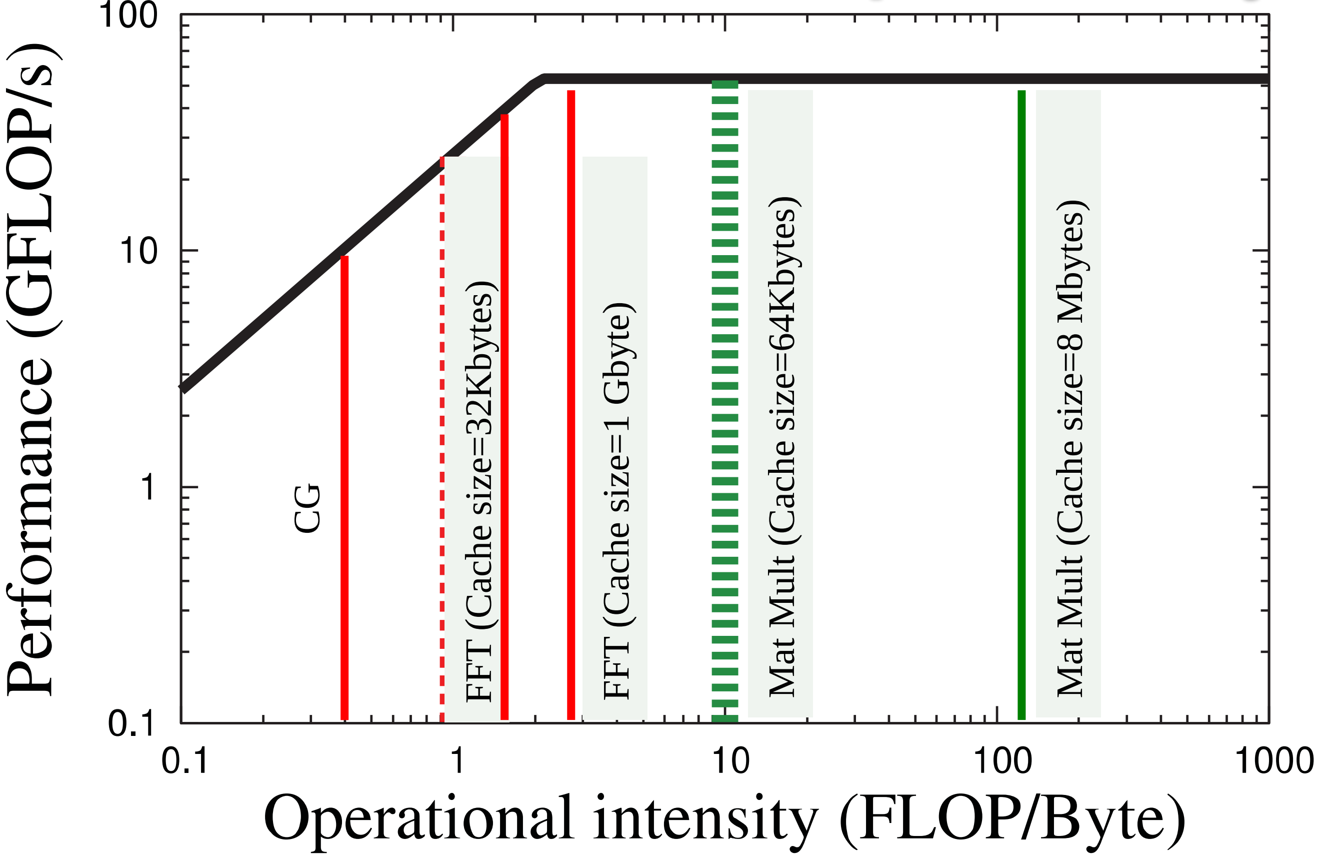



TABLE I. ROOFLINES AND OBSERVED PERFORMANCE FOR SPARSE MATRIX PRIMITIVES

| | Operation | Roofline Limit | Observed Performance |
|---|---|---|---|
| SpMM: Sparse-Dense MM | A * S | 72 gflops | 45 gflops |
| | A * $S^T$ | 72 gflops | 31 gflops |
| SDDMM: Sampled Dense-Dense MM | S ° (A * B) | 72 gflops | 63 gflops |

# I/O Lower Bounds => Op. Intensity Upper Bounds




**Upper-bound on operational intensity is a function of cache capacity**

| Algorithm | #Float-Ops | I/O Lower Bound | OI Upper Bound | |
|---|---|---|---|---|
| Matrix Multiplication | $2N^3$ | $16*N^3/C^{1/2}$ bytes | $(1/8)*C^{1/2}$ | ☺ |
| FFT | $2*N*\log_2 N$ | $16*N*\log_2 N/\log_2 C$ | $(1/8)*\log_2 C$ | ☹ |
| Conj. Gradient (2D Heat Eqn.) | $20*N^2*T$ | $48*N^2*T$ | 5/12 | ☹ |

# Why Optimizing Data Movement is Fundamentally Hard

- Suppose FLOPs were expensive relative to data movement
  - Efficient functions can be simply composed, because computational complexity is additive
  - Let $fopt_1$ and $fopt_2$ be efficient implementations of functions $f_1$ and $f_2$
  - An efficient implementation for ($f_1$ o $f_2$) can be simply constructed by composing the individual implementations: ($fopt_1$ o $fopt_2$) [concatenate CDAGs]
- Parallelization across multiple cores
  - If FLOP costs dominate data movement costs, main parallelization issue is load-balancing of work across cores
  - If $fopt_1$ and $fopt_2$ are each individually load-balanced, so will ($fopt_1$ o $fopt_2$)
- But what about the current reality: FLOPs are cheap, but data movement is expensive?
  - Problem: Data movement complexity is NOT additive under composition

# Composing Operations: Computation vs. Data Movement

- Computational complexity is additive when composing operations

  S1: r1 = f1(a1,..an);

  S2: r2 = f2(r1,b1,…bm);

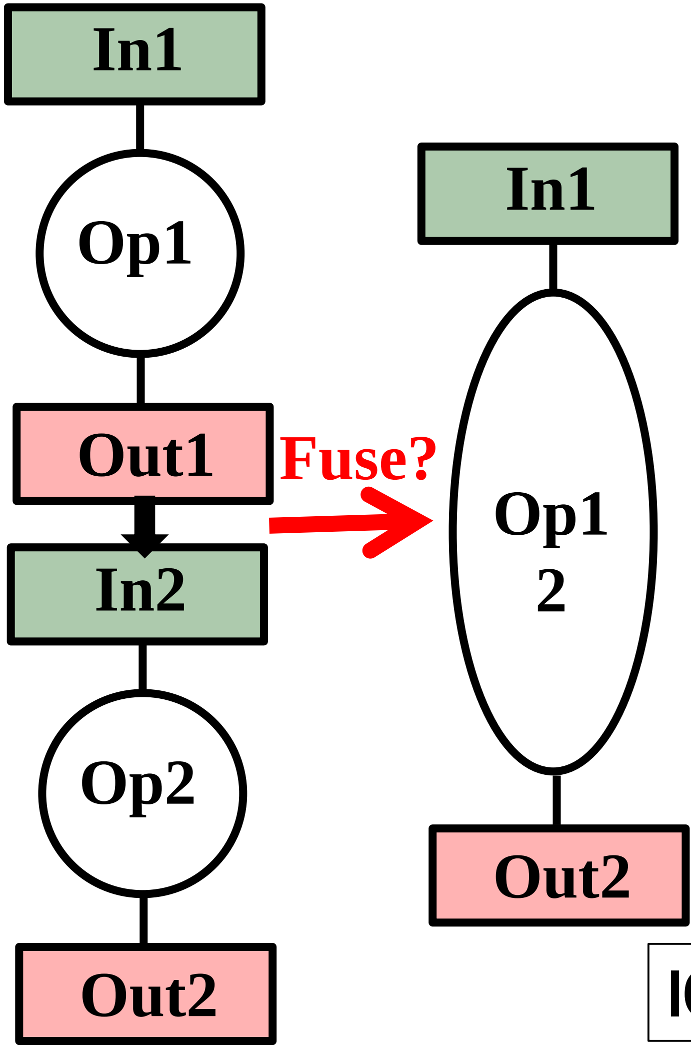
  - comp-cost(S1;S2) = comp-cost(S1)+comp-cost(S2)

- But data-movement complexity is not additive with composition

  - min-data-mvmt(S1;S2) can be less than min-data-mvmt(S1)+min-data-mvmt(S2)

| Operation | Composition | Comp-Cost | Minimum Data-Mvmt |
| --- | --- | --- | --- |
| Dot-Product | N scalar mult-adds | O(N) | O(N) |

# Lower Bounds Analysis: When is Fusion Useful?




**Lower Bounds can be composed**

$IO_{LB}(Op12) = IO_{LB}(Op1) + IO_{LB}(Op2) – 2 |Out1|$

**Max I/O reduction from fusion is twice size of Out1**

Fusion cannot be useful when:

$IO_{LB}(Op1) + IO_{LB}(Op2)$ is much larger than $|Out1|$

**Lower bounds analysis can help prune many configs.**

# Optimizing the Four-Index Integral Transform

- 4-D integral tensor (A) transformed from one basis to another (C) using transformer B
- Implemented as sequence of four tensor contractions
- Combinatorially explosive number of fusion/tiling/distribution choices
- NWChem comp. chem. suite implements 15 different variants of 4-index transform; none optimal
- New “communication-optimal” distributed 4-index transform developed by OSU/PNNL collaboration
  - Space of configs. pruned by data mvmt. lower bounds analysis
  - Significant improvement over previous NWChem versions
  - Incorporated into NWChem

$$\boxed{C[\alpha,\beta,\gamma,\delta] = \sum_{i,j,k,l} A[i,j,k,l].B[\alpha,i].B[\beta,j].B[\gamma,k].B[\delta,l]}$$

$$\Downarrow$$

$$\boxed{O1[\alpha,j,k,l]} = \sum_i A[i,j,k,l].B[\alpha,i]$$

$$\boxed{O2[\alpha,\beta,k,l]} = \sum_j \boxed{O1[\alpha,j,k,l]}.B[\beta,j]$$

$$\boxed{O3[\alpha,\beta,\gamma,l]} = \sum_k \boxed{O2[\alpha,\beta,k,l]}.B[\gamma,k]$$

$$C[\alpha,\beta,\gamma,\delta] = \sum_l \boxed{O3[\alpha,\beta,\gamma,l]}.B[\delta,l]$$

# Open Questions

- Can tools be developed to automatically characterize data movement complexity of algorithms?

- Can a general methodology be developed for use of data-movement lower-bounds in guiding design-space exploration?

- Can data-movement lower-bounds be used for algorithm-architecture co-design?
  - Example: Are 16 registers too few for efficient implementation of a CNN (Convolutional Neural Network) kernel?

- Can data movement constraints of irregular/sparse applications be characterized and used for optimization?

# Research Direction: Pattern-Specific Optimization

# Portability: OpenACC and OpenMP-Offload

- Directive-based prog. models for GPU/Accelerator offload
  - Spec. of computation in source code very similar to sequential code
  - Directives specify parts of code to be offloaded to GPU
  - User can optionally control when data is moved between CPU/GPU

**OpenMP**

```c
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

**OpenACC**

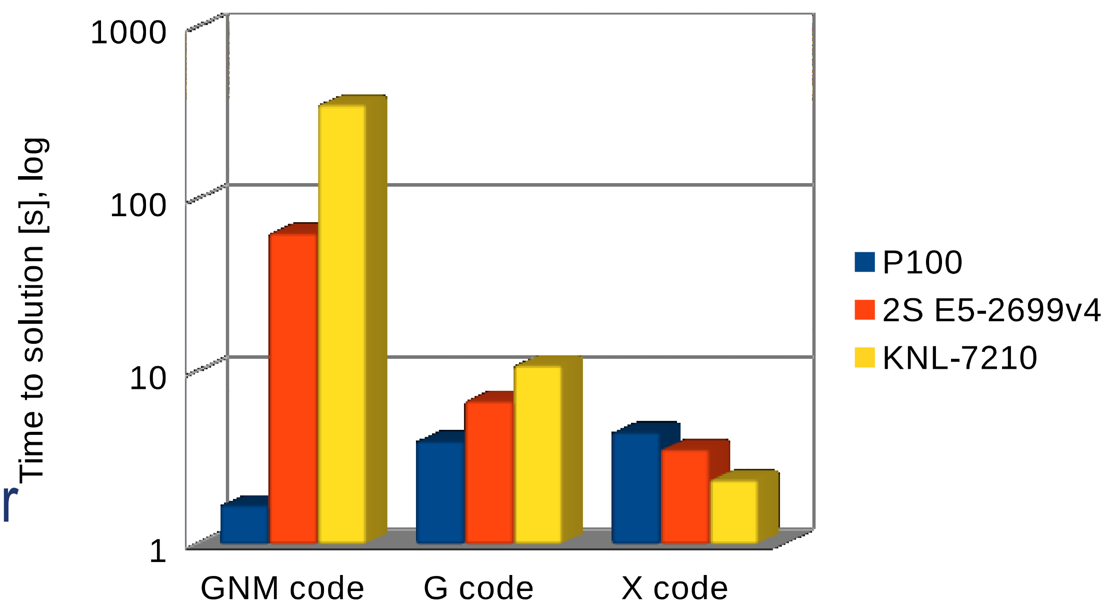
```c
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc parallel loop
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

Source: Jeff Larkin

# Case Study: OpenACC and OpenMP

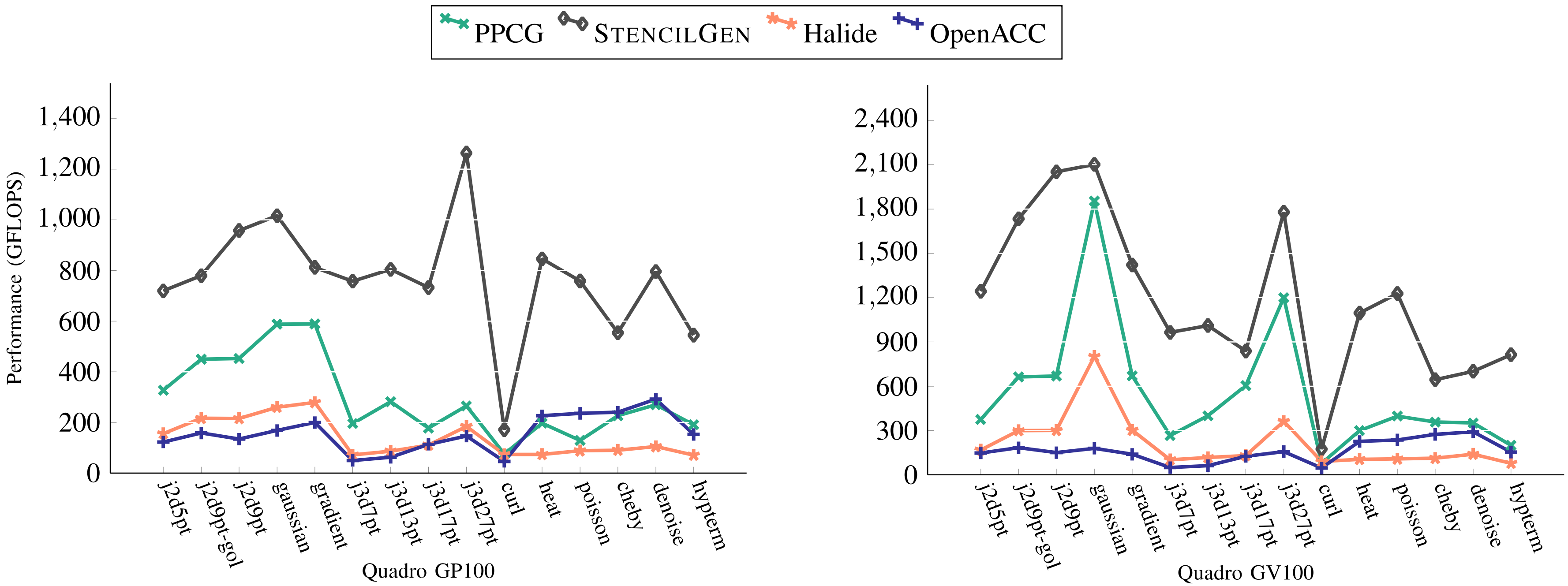
- Directive based optimization of radiation scheme ACRANEB2 in Danish weather prediction model: KNL, Pascal GPU, Xeon
  - Poulsen and Berg, http://www.dmi.dk/fileadmin/user_upload/Rapporter/TR2017/SR17-22.pdf
- Conclusion: Even with directive-based models, achieving high performance requires different source-code versions
  - Loops had to be rearranged and data-structure layouts changed
  - Performance difference for variants can be huge

- X: Code version tuned for KNL
- G: Code tuned GPU with same data structures
- GNM: GPU-tuned, with transposed data structures
  - Good perf. boost on GPU, but about 100x slowdown for KNL!



# Performance: Stencil DSL vs. General-Purpose

- DSL-generated GPU code achieves much higher performance




| Benchmark | N | T | k | FPP | Benchmark | N | T | k | FPP | Benchmark | N | T | k | FPP |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| j2d5pt | $8192^2$ | 4 | 1 | 10 | j3d7pt | $512^3$ | 4 | 1 | 13 | heat | $512^3$ | 4 | 1 | 15 |
| j2d9pt-gol | $8192^2$ | 4 | 1 | 18 | j3d13pt | $512^3$ | 4 | 2 | 25 | poisson | $512^3$ | 4 | 1 | 21 |
| j2d9pt | $8192^2$ | 4 | 2 | 18 | j3d17pt | $512^3$ | 4 | 1 | 28 | cheby | $512^3$ | 4 | 1 | 39 |
| gaussian | $8192^2$ | 4 | 2 | 50 | j3d27pt | $512^3$ | 4 | 1 | 54 | denoise | $512^3$ | 4 | 2 | 62 |
| gradient | $8192^2$ | 4 | 1 | 18 | curl | $450^3$ | 1 | 1 | 36 | hypterm | $300^3$ | 1 | 4 | 358 |

N: Domain Size, T: Time Tile Size, k: Stencil Order, FPP: FLOPs per Point

# Domain-Specific Optimization: Tensor Contractions

$$C_{ijkl} = \sum_{mn} A_{imkn} \cdot B_{jnlm}$$

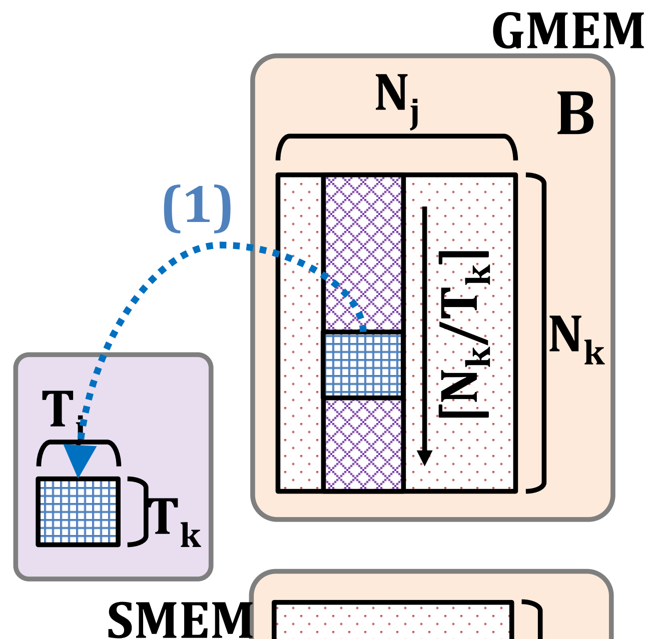
```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  for (k=0; k<N; k++)
   for (l=0; l<N; l++)
    for (m=0; m<N; m++)
     for (n=0; n<N; n++)
      C[i][j][k][l] += A[i][m][k][n]*B[j][n][l][m];
```

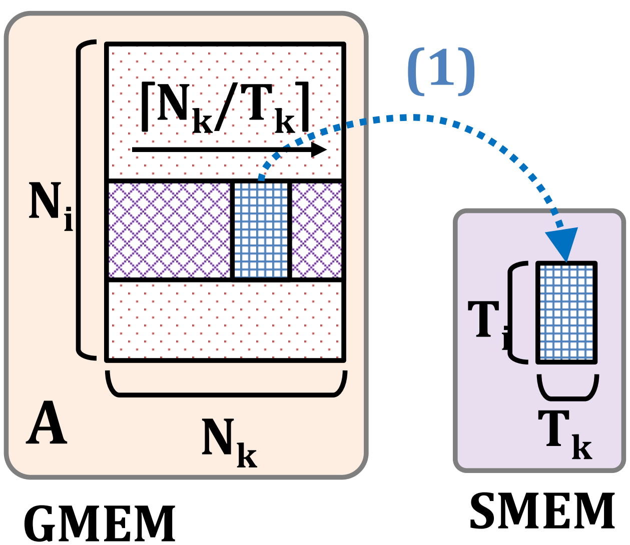
- Tensor contraction is high-dimension analog of matrix-matrix product
- Each loop index appears in exactly two tensors
  - "Contraction index" appears only in input (rhs) tensors: {m, n}
  - "External index": appears in output tensor and one input tensor: {i, k} {j, l}
- TensorGen project (OSU/PNNL) is developing domain-specific compiler for multi-target (GPU, multi/manycore CPU) optimization of arbitrary tensor contractions
  - Specialized schema for optimized data movement/buffering

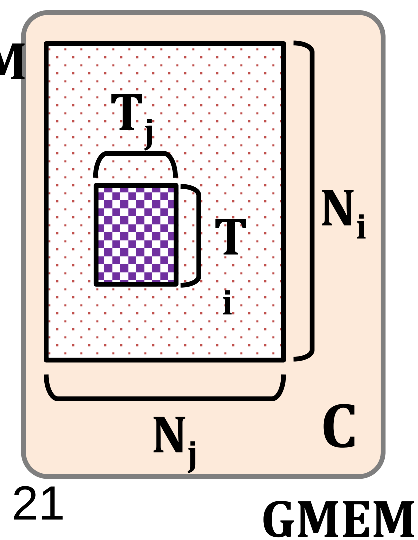
# Matrix Multiplication Schema

## C[i][j] += A[i][k]*B[k][j]

- A 2D thread-block computes a 2D slice (TixTj) of C using a Ti x Nk slice of A and a Nk x Tj slice of B
- Registers are used to hold the TixTj slice of C
- A TixTk slice of A and a TkxTj slice of B are loaded into Shared Memory (1)



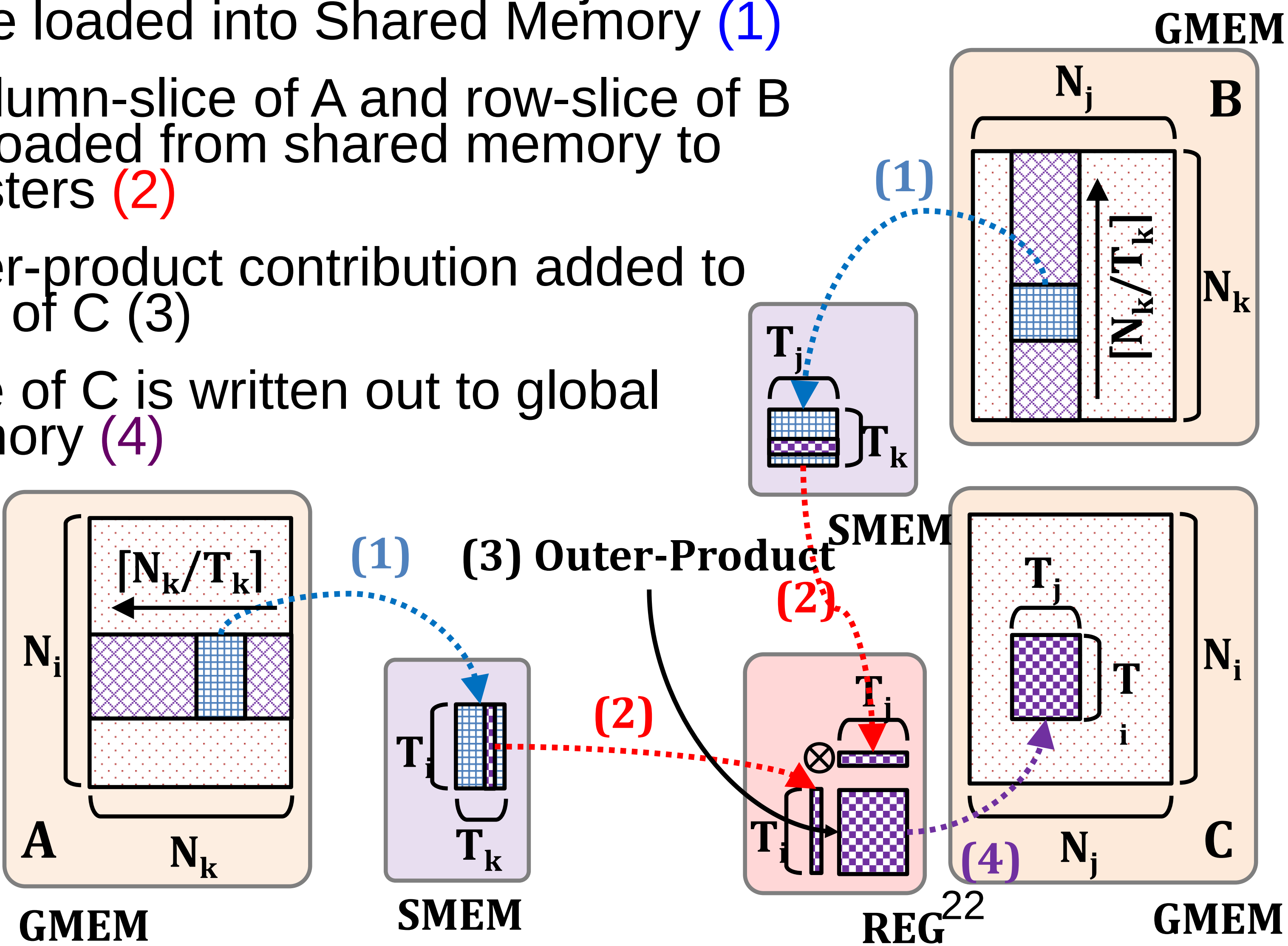



# Matrix Multiplication Schema

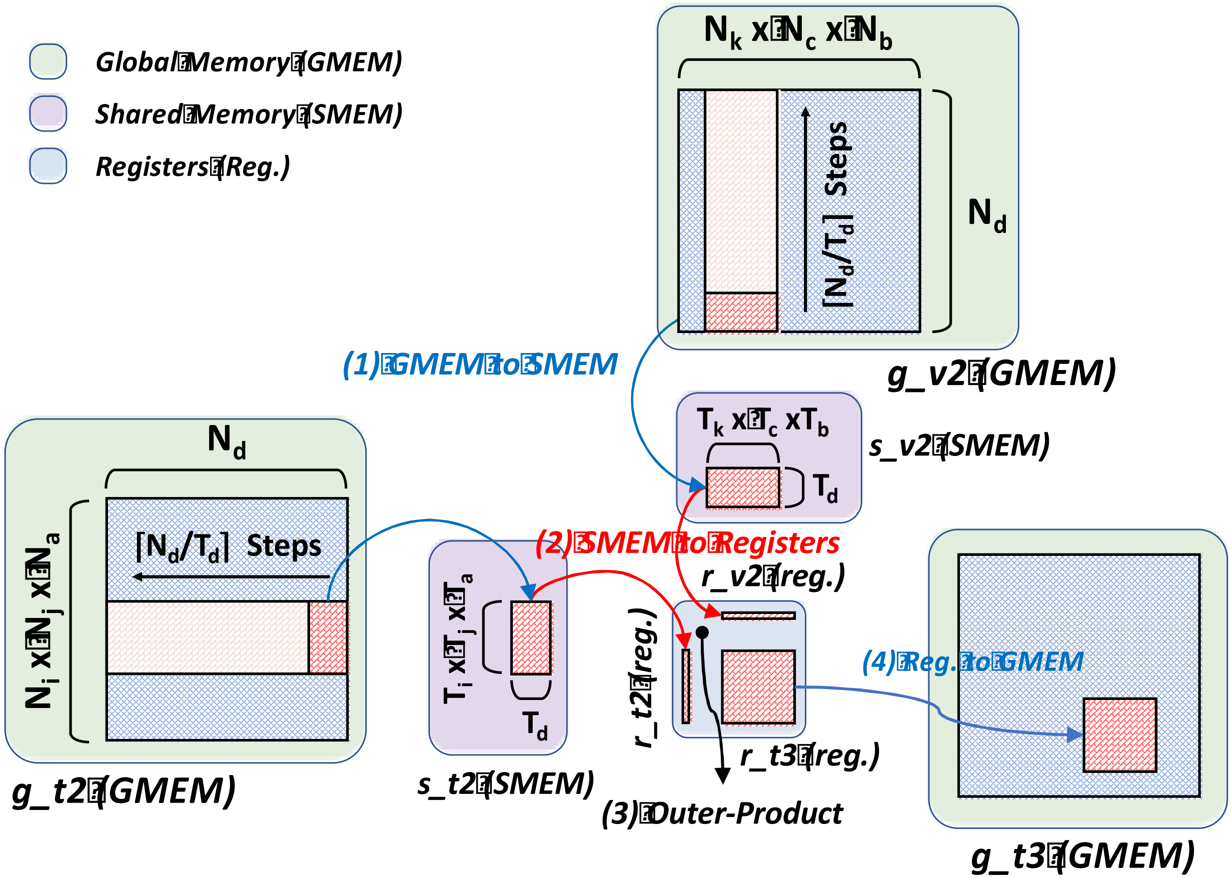
## C[i][j] += A[i][k]*B[k][j]

- A TixTk slice of A and a TkxTj slice of B are loaded into Shared Memory (1)
- A column-slice of A and row-slice of B are loaded from shared memory to registers (2)
- Outer-product contribution added to slice of C (3)
- Slice of C is written out to global memory (4)



# Generalizing for Arbitrary Tensor Contractions

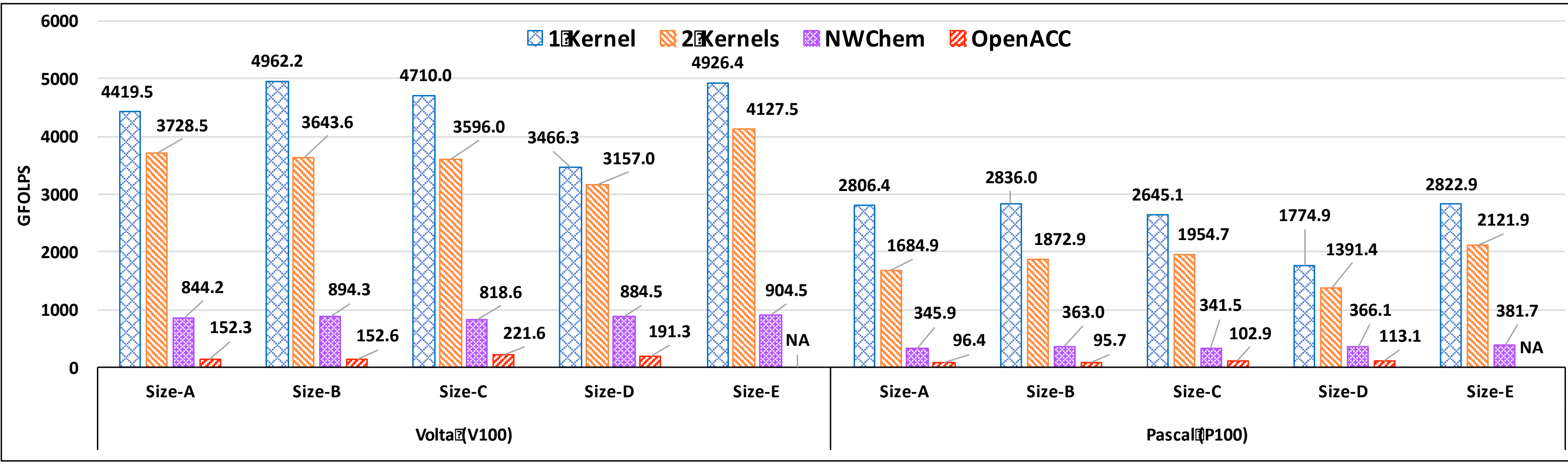
t3[k, j, i, c, b, a] -= t2[d, a, i, j] * v2[d, k, c, b]




- Custom optimizer exploits "orthogonal reuse directions" property
- t2 reuse: {b,c,k}; v2 reuse: {a,i,j}; t3 reuse: {d} (reduction)
- 2D multi-level tiling (shared-memory + registers); streamed tiling along {d}
- Slice of t3 held in register tiles; maximize reuse of data slices of t2 and v2

# CCSD(T) Tensor Contractions in NWChem

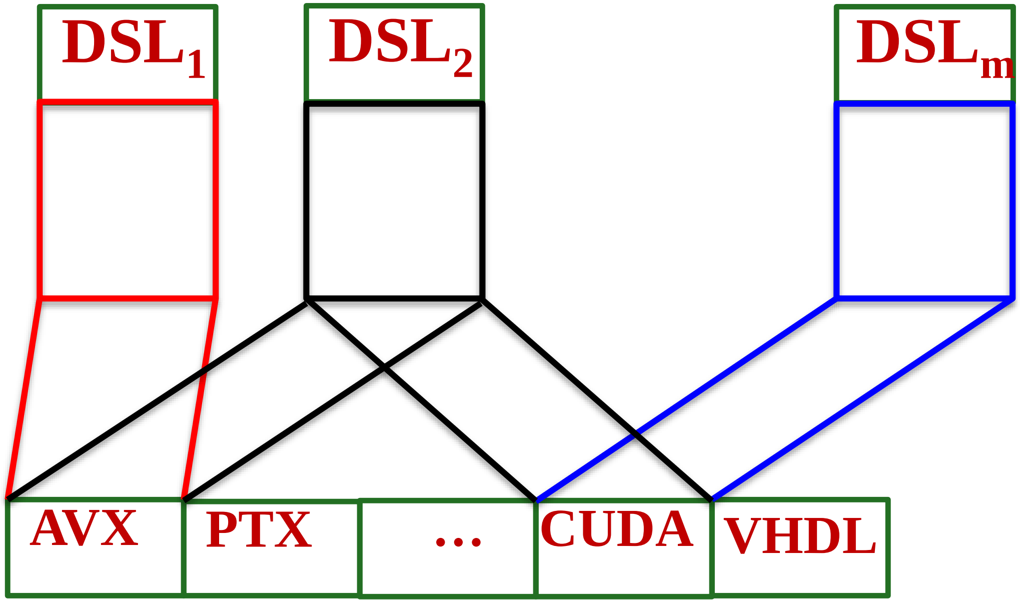
| | | | |
|---|---|---|---|
| **sd1_1** | $t3[k,j,i,c,b,a] -= t2[l,a,b,i] * v2[k,j,c,l]$ | **sd2_1** | $t3[k,j,i,c,b,a] -= t2[d,a,i,j] * v2[d,k,c,b]$ |
| **sd1_2** | $t3[k,j,i,c,b,a] += t2[l,a,b,j] * v2[k,i,c,l]$ | **sd2_2** | $t3[k,j,i,c,b,a] -= t2[d,a,j,k] * v2[d,i,c,b]$ |
| **sd1_3** | $t3[k,j,i,c,b,a] -= t2[l,a,b,k] * v2[j,i,c,l]$ | **sd2_3** | $t3[k,j,i,c,b,a] += t2[d,a,i,k] * v2[d,j,c,b]$ |
| **sd1_4** | $t3[k,j,i,c,b,a] -= t2[l,b,c,i] * v2[k,j,a,l]$ | **sd2_4** | $t3[k,j,i,c,b,a] += t2[d,b,i,j] * v2[d,k,c,a]$ |
| **sd1_5** | $t3[k,j,i,c,b,a] += t2[l,b,c,j] * v2[k,i,a,l]$ | **sd2_5** | $t3[k,j,i,c,b,a] += t2[d,b,j,k] * v2[d,i,c,a]$ |
| **sd1_6** | $t3[k,j,i,c,b,a] -= t2[l,b,c,k] * v2[j,i,a,l]$ | **sd2_6** | $t3[k,j,i,c,b,a] -= t2[d,b,i,k] * v2[d,j,c,a]$ |
| **sd1_7** | $t3[k,j,i,c,b,a] += t2[l,a,c,i] * v2[k,j,b,l]$ | **sd2_7** | $t3[k,j,i,c,b,a] -= t2[d,c,i,j] * v2[d,k,b,a]$ |
| **sd1_8** | $t3[k,j,i,c,b,a] -= t2[l,a,c,j] * v2[k,i,b,l]$ | **sd2_8** | $t3[k,j,i,c,b,a] -= t2[d,c,j,k] * v2[d,i,b,a]$ |
| **sd1_9** | $t3[k,j,i,c,b,a] += t2[l,a,c,k] * v2[j,i,b,l]$ | **sd2_9** | $t3[k,j,i,c,b,a] += t2[d,c,i,k] * v2[d,j,b,a]$ |




- CCSD(T) is an accurate but extremely compute-intensive method in NWChem
- New fused GPU kernels significantly outperform current GPU code in NWChem
- Code is being incorporated into NWChenEX

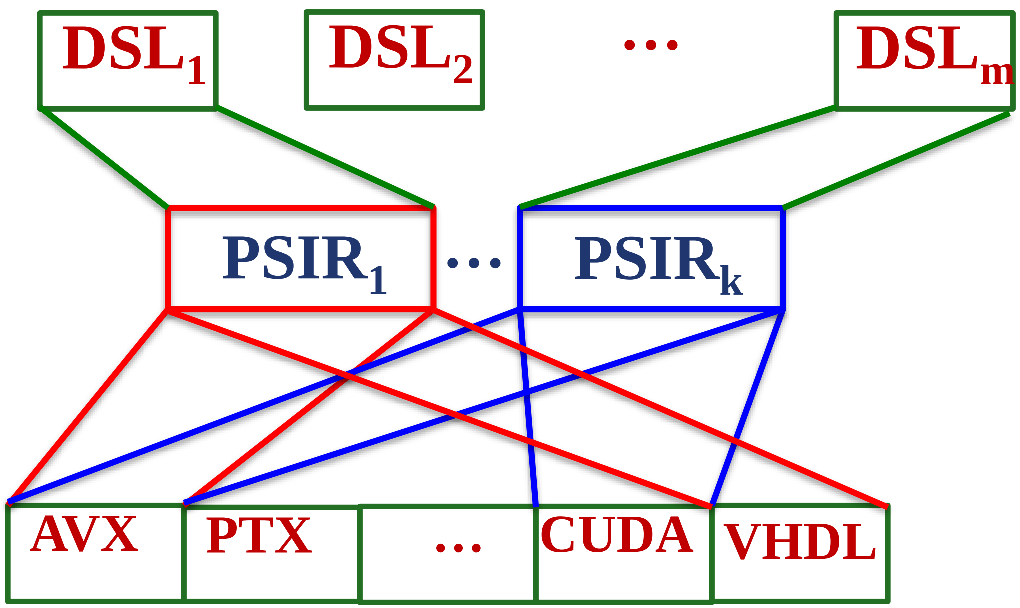
# Isolated Domain-Specific Compilers and Libraries

- Multi-target DSLs achieve performance & portability by:
  - Use of appropriate internal representation of computation that facilitates effective choice of mapping/scheduling of computation/data
  - Separation of high-level target-independent decisions from low-level platform-specific choices
  - Use of platform-specific code-schema driven by key performance factors
- But each Domain-Specific compiler/library today is a stand-alone system
  - No common infrastructure support for building new DSLs



# Intermediate Pattern-Specific Layers?

- Can a small number of Pattern-Specific IRs be identified?
  - DSLs perform domain-specific transformations and generate suitable PSIR
  - Pattern-Specific Compiler performs platform-specific optimizations/transformations for different target platforms



# Matrices/Tensors: Orthogonal Aligned Reuse Pattern

- Many matrix/tensor computations in computational science and data-science/ML have the following characteristics
  - Access functions for matrix/tensor indices are simply surrounding loop indices
    - C[i][j] += A[i][k]*B[k][j]
    - A[i][r] += T[i][j][k]*B[j][r]*C[k][r]
  - Reuse of data elements occurs only along iteration space axes (1D) or product-space of axes (2D and higher reuse)
  - All surrounding loops represent reuse directions for one or more arrays
  - Optimal tile size for innermost tiling loop is always one (or vector-length if that loop is innermost intra-tile loop) => streaming
  - Permutation of outer tiling loops have negligible effect on data mvmt. vol.
- Significant promise for efficient data-volume based model-driven loop transformation and multi-target code generation for this class of computations

# Summary

- End of Moore's Law implies greater customization and need to make more efficient use of limited resources
- Achieving performance, productivity, and portability will be even more challenging => Compilers must play a bigger role
- Fundamental bottleneck: data movement (FLOPs are relatively cheap)
  - Need advances in understading inherent data movement complexity of algorithms
- Domain/pattern-specific compiler optimization is a promising direction
  - Need to identify a small number of computational patterns with wide coverage, and pattern-specific compiler transformation strategies for the patterns
- Many challenges and opportunities: exciting time to work on compiler research!

# Acknowledgment


- Many thanks to many collaborators, especially Sriram Krishnamoorthy, Louis-Noel Pouchet, Nasko Rountev, Uday Bondhugula, Albert Cohen, Jason Cong, Franz Franchetti, John Owens, Srini Parthasarathy, Dan Quinlan, J. Ramanujam, Fabrice Rastello, Vivek Sarkar, …; and numerous current and former graduate students and post-docs including Venmugil Elango, Tom Henretty, Justin Holewinski, Changwan Hong, Jinsung Kim, Martin Kong, Rakshith Kunchum, Emre Kurt, Israt Nisa, Samyam Rajbhandari, Mahesh Ravishankar, Prashant Rawat, Aravind Sukumaran-Rajam, Kevin Stock, ...

- Grateful thanks for funding from the U.S. National Science Foundation, U.S. Department of Energy, and DARPA