Tiling, Stencils, Tensors, and more

J. “Ram” Ramanujam
Louisiana State University

Center for Comp. & Tech. (CCT)
School of Elec. Eng. & Comp. Sci.
ram@cct.lsu.edu

Acknowledgments

Collaborators

Albert Cohen (ENS Paris)
Franz Franchetti (CMU)
Louis-Noel Pouchet (OSU)
P. Sadayappan (OSU)
Robert Harrison (Stony Brook)
Fabrice Rastello (ENS Lyon)
Nasko Rountev (OSU)
Sven Verdoolaege (ENS)
Tobias Grosser (ETH)
Paul Kelly (Imperial)
Michelle Strout (Arizona)
S. Krishnamoorthy (PNNL)
Uday Bondhugula (IISc)
Muthu Baskaran (Reservoir)
…

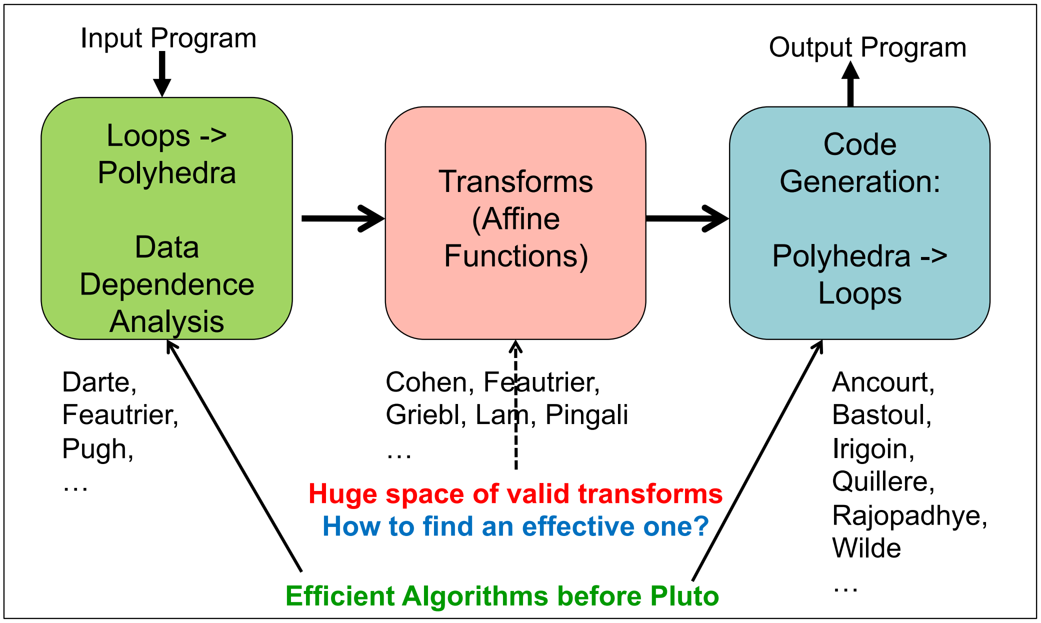
Carlo Bertolli
Fabio Luporini
Albert Hartono
Justin Holewinski
Venmugil Elango
Tom Henretty
Mahesh Ravishankar
Sanket Tavarageri
Richard Veras
Sameer Abu Asal
Rod Tohid
Ye Fang
Michal Brylinski
Zahra Khatemi
…

Funding

US National Science Foundation
US Army
US DOE
IBM
…

Quick Review of Tiling (ala Pluto)

Polyhedral Compiler Transformation



Input Program

Loops -> Polyhedra

Data Dependence Analysis

Transforms (Affine Functions)

Code Generation:

Polyhedra -> Loops

Output Program

Darte, Feautrier, Pugh, …

Cohen, Feautrier, Griebl, Lam, Pingali …

Ancourt, Bastoul, Irigoin, Quillere, Rajopadhye, Wilde …

Huge space of valid transforms

How to find an effective one?

Efficient Algorithms before Pluto

Pluto: generates efficient tiled, parallel output code for imperfect nests?

ϕ as an affine by-statement transform

- A one-dimensional affine transform for statement S_k is defined by:

$$\begin{aligned}\phi_{S_k}(\vec{i}) &= \left[c_1\ c_2\ \dots\ c_{m_{S_k}}\right]\left(\ \vec{i}\ \right) + c_0 \\ &\underset{\equiv}{=} \left[c_1\ c_2\ \dots\ c_{m_{S_k}}\ c_0\right]\begin{pmatrix}\vec{i}\\ 1\end{pmatrix}\end{aligned}$$

where $[c_0, c_1, c_2, \dots, c_{m_{S_k}}] \in \mathcal{Z}$.

- An affine transform

  = A new scanning hyperplane

  = A loop in the transformed space (with a particular property)

1-D Jacobi (imperfectly nested)

```
    for (t=1; t<M; t++) {
      for (i=2; i<N-1; i++) {
S:      b[i] = 0.333*(a[i-1]+a[i]+a[i+1]); }
      for (j=2; j<N-1; j++) {
T:      a[j] = b[j]; } }
```

$$\begin{bmatrix}\phi_S^1\\ \phi_S^2\end{bmatrix}\begin{pmatrix}t\\ i\\ 1\end{pmatrix} = \begin{bmatrix}1 & 0 & 0\\ 2 & 1 & 0\end{bmatrix}$$

$$\begin{bmatrix}\phi_T^1\\ \phi_T^2\end{bmatrix}\begin{pmatrix}t\\ j\\ 1\end{pmatrix} = \begin{bmatrix}1 & 0 & 0\\ 2 & 1 & 1\end{bmatrix}$$

Pluto: 1-D Jacobi (imperfectly nested)

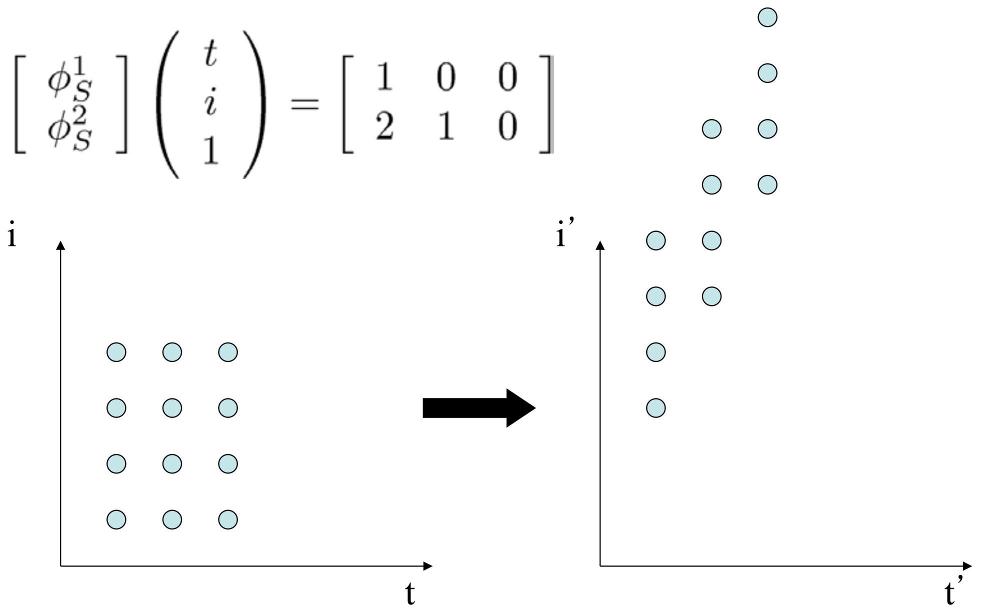
$$\begin{bmatrix} \phi_S^1 \\ \phi_S^2 \end{bmatrix} \begin{pmatrix} t \\ i \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \phi_T^1 \\ \phi_T^2 \end{bmatrix} \begin{pmatrix} t \\ j \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 1 \end{bmatrix}$$

- The resulting transformation is equivalent to a constant shift of one for T relative to S, fusion (j and i are named the same as a result), and skewing the fused i loop with respect to the t loop by a factor of two.
- The (1,0) hyperplane has the least communication: no dependence crosses more than one hyperplane instance along it.

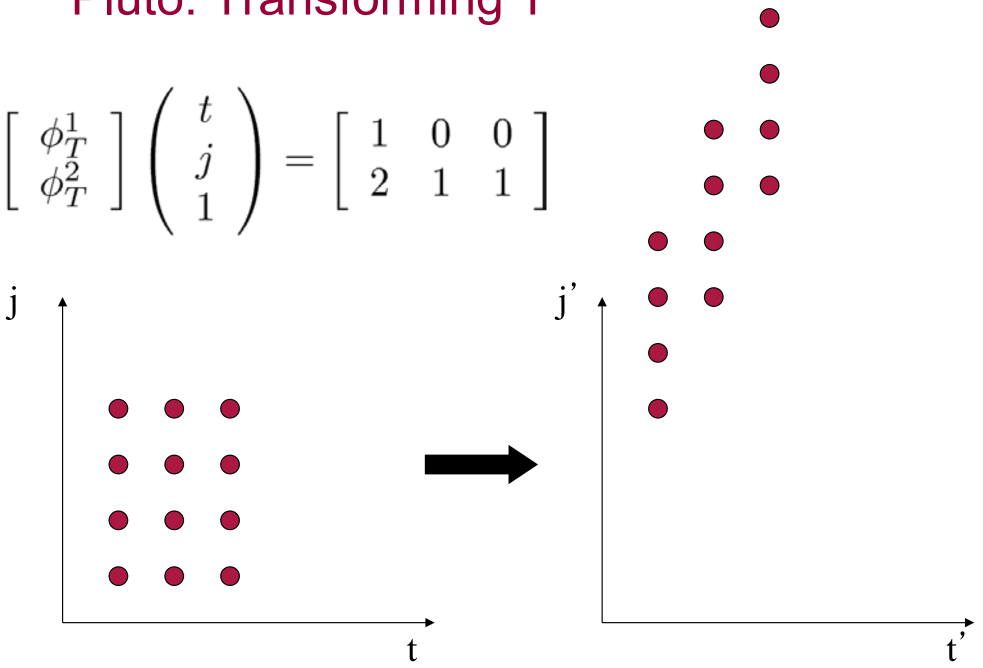
Pluto: Transforming S

$$\begin{bmatrix} \phi_S^1 \\ \phi_S^2 \end{bmatrix} \begin{pmatrix} t \\ i \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \end{bmatrix}$$

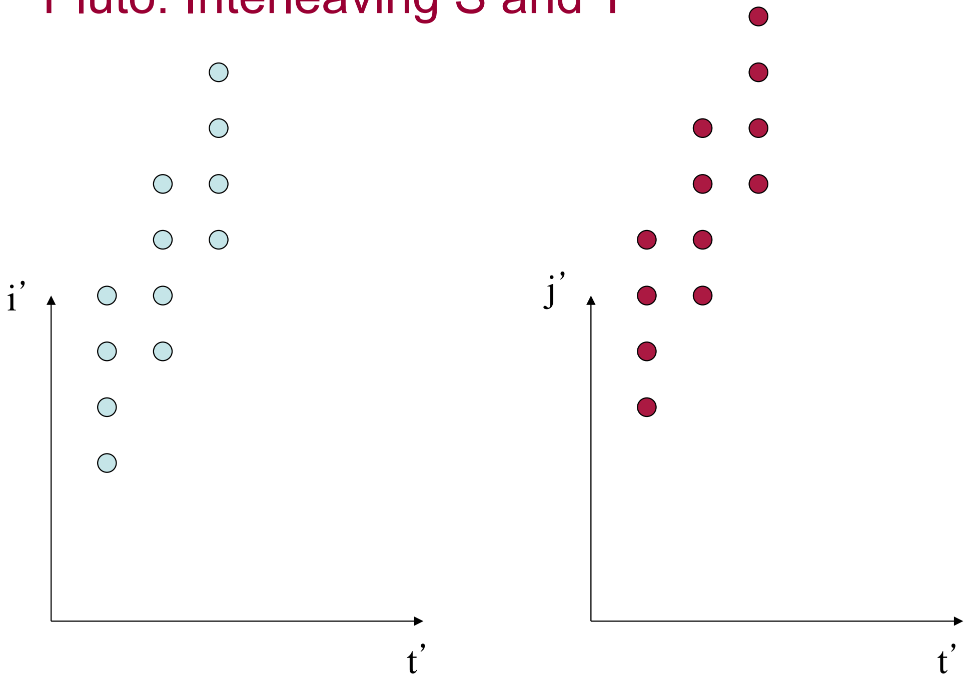


Pluto: Transforming T

$$\begin{bmatrix}\phi_T^1\\ \phi_T^2\end{bmatrix}\begin{pmatrix}t\\ j\\ 1\end{pmatrix} = \begin{bmatrix}1 & 0 & 0\\ 2 & 1 & 1\end{bmatrix}$$



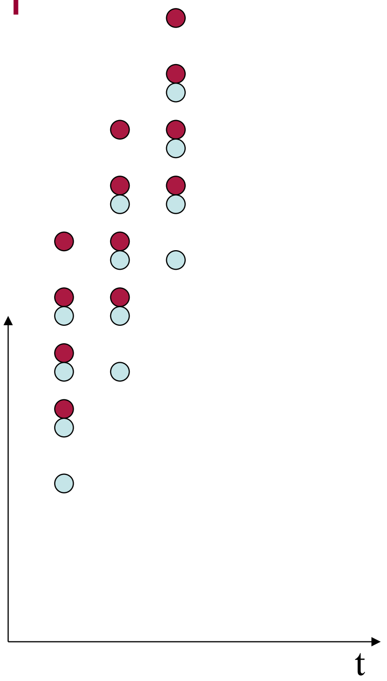
Pluto: Interleaving S and T



Pluto: Interleaving S and T

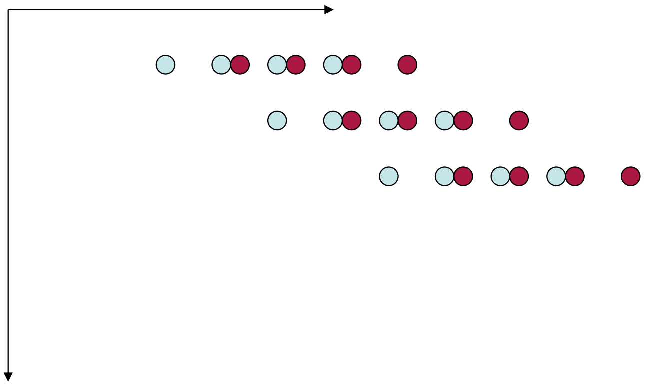
$$\begin{bmatrix}\phi_S^1\\ \phi_S^2\end{bmatrix}\begin{pmatrix}t\\ i\\ 1\end{pmatrix} = \begin{bmatrix}1&0&0\\ 2&1&0\end{bmatrix}$$

$$\begin{bmatrix}\phi_T^1\\ \phi_T^2\end{bmatrix}\begin{pmatrix}t\\ j\\ 1\end{pmatrix} = \begin{bmatrix}1&0&0\\ 2&1&1\end{bmatrix}$$



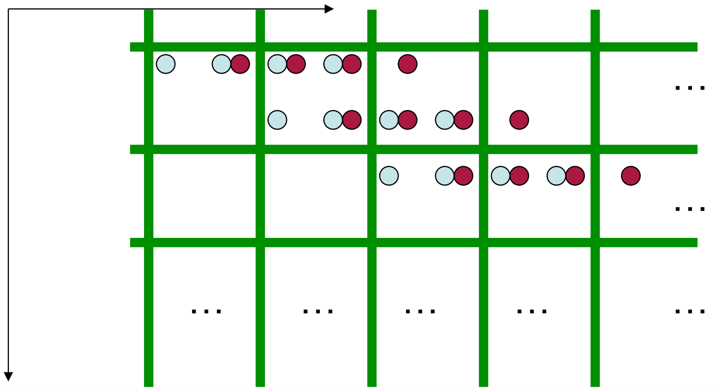
1-D Jacobi (imperfectly nested) – transformed code

```
    for (t0=0;t0<=M-1;t0++) {
S':   b[2]=0.333*(a[2-1]+a[2]+a[2+1]);
      for (t1=2*t0+3;t1<=2*t0+N-2;t1++) {
S:      b[-2*t0+t1]=0.333*(a[-2*t0+t1-1]+a[-2*t0+t1]
                          +a[-2*t0+t1+1]);
T:      a[-2*t0+t1-1]=b[-2*t0+t1-1]; }
T':   a[N-2]=b[N-2]; }
```

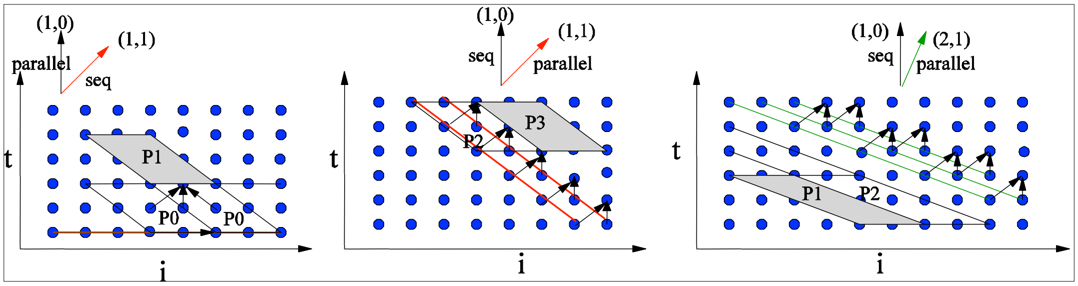


1-D Jacobi (imperfectly nested) – transformed tiled

```
    for (t0=0;t0<=M-1;t0++) {
S':   b[2]=0.333*(a[2-1]+a[2]+a[2+1]);
      for (t1=2*t0+3;t1<=2*t0+N-2;t1++) {
S:      b[-2*t0+t1]=0.333*(a[-2*t0+t1-1]+a[-2*t0+t1]
                              +a[-2*t0+t1+1]);
T:      a[-2*t0+t1-1]=b[-2*t0+t1-1]; }
T':   a[N-2]=b[N-2]; }
```



Pluto: Communication Volume & Reuse Distance



- $\phi(\mathbf{i}') - \phi(\mathbf{i})$ is an affine function that represents the component of a dependence along hyperplane ϕ
  - Communication volume (per unit area) at processor tile boundaries
  - Cache misses at local tile edges
  - Loads to a register tile

Stencil Computations

- Domain-Specific Language
- Tiling stencils
- Data Layouts
- Code Generation
- Higher Order Stencils: exploiting associativity, …

Why Domain-Specific Languages?

- Productivity
  - High level abstractions ease application development

Why Domain-Specific Languages?

- Productivity
  - High level abstractions ease application development
- Performance
  - Domain-specific semantics enables specialized optimizations
  - Constraints on specification enables more effective general-purpose transformations and tuning (tiling, fusion)

Why Domain-Specific Languages?

- Productivity
  - High level abstractions eases application development
- Performance
  - Domain-specific semantics enables specialized optimizations
  - Constraints on specification enables more effective general-purpose transformations and tuning (tiling, fusion)
- Portability
  - New architectures => changes only in domain-specific compiler, without any change in user application code

(Embedded) DSLs for Stencils

- Benefits of high-level specification of computations
  - Ease of use
    - For mathematicians/scientists creating the code
  - Ease of optimization
    - Facilitate loop and data transformations by compiler
    - Automatic transformation by compiler into parallel C/C++ code
- Embedded DSL provides flexibility
  - Generality of standard programming language (C, MATLAB) for non compute-intensive parts
  - Automated transformation of embedded DSL code for high performance on different target architectures
- Target architectures for Stencil DSL
  - Vector-SIMD (AVX, LRBNi, ..), GPU, FPGA, customized accelerators

Stencil DSL Example -- Standalone

```
int Nr; int Nc;
grid g [Nr][Nc];

double griddata a on g at 0,1;

pointfunction five_point_avg(p) {
  double ONE_FIFTH = 0.2;
  [1]p[0][0] = ONE_FIFTH*([0]p[-1][0] + [0]p[0][-1]
                 + [0]p[0][0] + [0]p[0][1] + [0]p[1][0]);
}

iterate 1000 {
  stencil jacobi_2d {
    [0     ][0:Nc-1] : [1]a[0][0] = [0]a[0][0];
    [Nr-1  ][0:Nc-1] : [1]a[0][0] = [0]a[0][0];
    [0:Nr-1][0     ] : [1]a[0][0] = [0]a[0][0];
    [0:Nr-1][Nc-1  ] : [1]a[0][0] = [0]a[0][0];
    [1:Nr-2][1:Nc-2] : five_point_avg(a);
  }

  reduction max_diff max {
    [0:Nr-1][0:Nc-1] : fabs([1]a[0][0] - [0]a[0][0]);
  }
} check (max_diff < .00001) every 4 iterations
```

Stencil DSL Example -- Standalone

```
int Nr; int Nc;
grid g [Nr][Nc];

double griddata a on g at 0,1;

pointfunction five_point_avg(p) {
  double ONE_FIFTH = 0.2;
  [1]p[0][0] = ONE_FIFTH*([0]p[-1][0] + [0]p[0][-1]
                 + [0]p[0][0] + [0]p[0][1] + [0]p[1][0]);
}

iterate 1000 {
  stencil jacobi_2d {
    [0     ][0:Nc-1] : [1]a[0][0] = [0]a[0][0];
    [Nr-1  ][0:Nc-1] : [1]a[0][0] = [0]a[0][0];
    [0:Nr-1][0     ] : [1]a[0][0] = [0]a[0][0];
    [0:Nr-1][Nc-1  ] : [1]a[0][0] = [0]a[0][0];
    [1:Nr-2][1:Nc-2] : five_point_avg(a);
  }

  reduction max_diff max {
    [0:Nr-1][0:Nc-1] : fabs([1]a[0][0] - [0]a[0][0]);
  }
} check (max_diff < .00001) every 4 iterations
```

Reference data over two time steps: current(0) and next (1)

Stencil DSL Example -- Standalone

```
int Nr; int Nc;
grid g [Nr][Nc];

double griddata a on g at 0,1;

pointfunction five_point_avg(p) {
  double ONE_FIFTH = 0.2;
  [1]p[0][0] = ONE_FIFTH*([0]p[-1][0] + [0]p[0][-1]
                 + [0]p[0][0] + [0]p[0][1] + [0]p[1][0]);
}

iterate 1000 {
  stencil jacobi_2d {
    [0     ][0:Nc-1] : [1]a[0][0] = [0]a[0][0];
    [Nr-1  ][0:Nc-1] : [1]a[0][0] = [0]a[0][0];
    [0:Nr-1][0     ] : [1]a[0][0] = [0]a[0][0];
    [0:Nr-1][Nc-1  ] : [1]a[0][0] = [0]a[0][0];
    [1:Nr-2][1:Nc-2] : five_point_avg(a);
  }

  reduction max_diff max {
    [0:Nr-1][0:Nc-1] : fabs([1]a[0][0] - [0]a[0][0]);
  }
} check (max_diff < .00001) every 4 iterations
```

Specify computations on borders

Stencil DSL – Embedded in C

```
int main() {
  int Nr = 256; int Nc = 256; int T = 100;
  double *a = malloc(Nc*Nr*sizeof(double));

#pragma sdsl start time_steps:T block:8,8,8 tile:1,3,1 time:4
  int Nr; int Nc;
  grid g [Nr][Nc];
  double griddata a on g at 0,1;
  pointfunction five_point_avg(p) {
    double ONE_FIFTH = 0.2;
    [1]p[0][0] = ONE_FIFTH*([0]p[-1][0] + [0]p[0][-1]
               + [0]p[0][0] + [0]p[0][1] + [0]p[1][0]); }
  iterate 1000 {
    stencil jacobi_2d {
      [0     ][0:Nc-1] : [1]a[0][0] = [0]a[0][0];
      [Nr-1  ][0:Nc-1] : [1]a[0][0] = [0]a[0][0];
      [0:Nr-1][0     ] : [1]a[0][0] = [0]a[0][0];
      [0:Nr-1][Nc-1  ] : [1]a[0][0] = [0]a[0][0];
      [1:Nr-2][1:Nc-2] : five_point_avg(a);}
    reduction max_diff max {
      [0:Nr-1][0:Nr-1] : fabs([1]a[0][0] - [0]a[0][0]);
    }
  } check (max_diff < .00001) every 4 iterations
#pragma sdsl end
}
```

Related Work

- 20+ publications over the last few years on optimizing stencil computations
- Some stencil DSLs and stencil compilers
  - Pochoir (MIT), PATUS (Basel), Mint (UCSD), Physis (Tokyo), Halide (MIT), Exastencils Project (Passau), …
- DSL Frameworks and libraries
  - SEJITS (LBL); Liszt, OptiML, OptiQL (Stanford), PyOP2/OP2 (Imperial College, Oxford)
- Our focus has been complementary: developing ***abstraction-specific*** compiler ***transformations*** matched to ***performance-critical characteristics*** of target architecture

Compilation of Stencil Codes

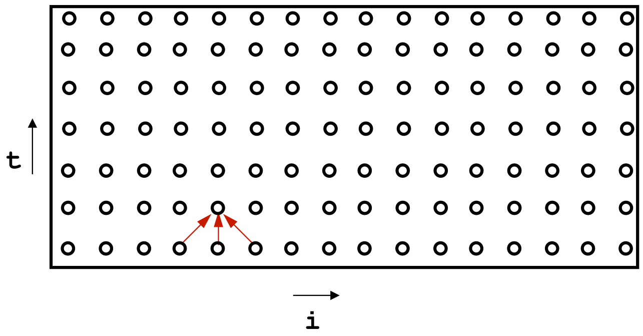
- Large class of applications
- Sweeps through a large data set
- Each data point: computed from "neighbors"
- Multiple time iterations
  – Repeated access to same data
- Pipelined parallel execution
- Example: One-dimensional Jacobi

```
for t = 1 to T
  for i = 1 to N
    B[i] = (A[i-1]+A[i]+A[i+1])/3
  for i = 1 to N
    A[i] = B[i]
```

```
for t = 1 to T
  for i = 1 to N
    A[t+1,i] =
       (A[t,i-1]+
        A[t,i]+A[t,i+1])/3
```

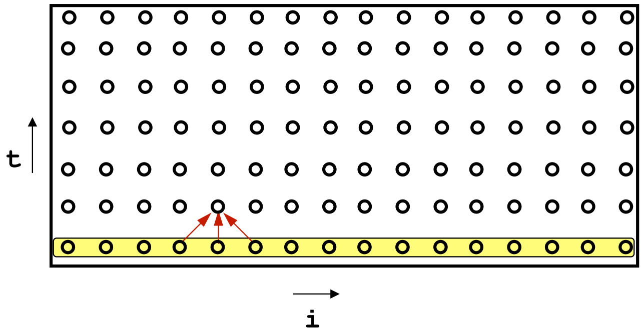
Motivation

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



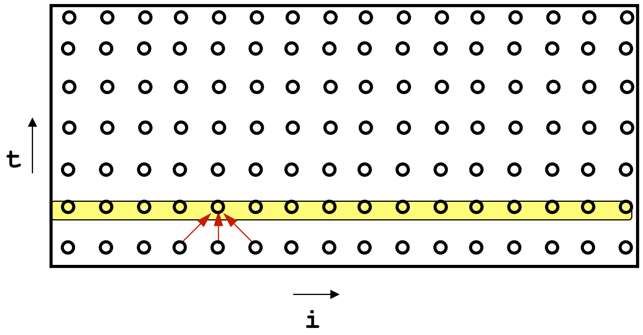
Motivation

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



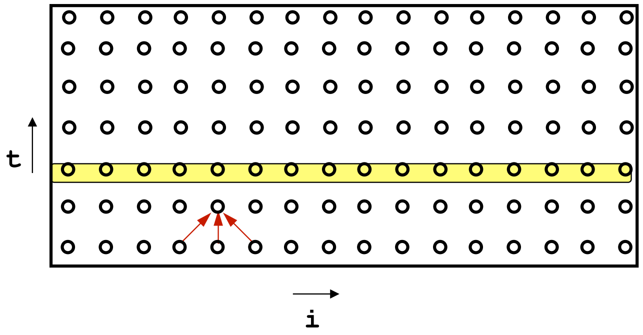
Motivation

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```

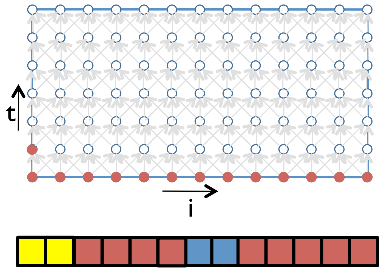


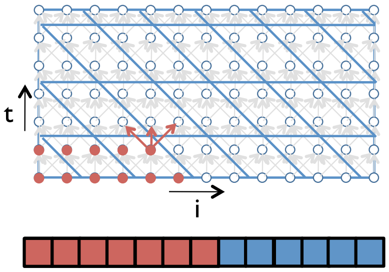
Motivation

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



Time Tiling (with 1-D array code)





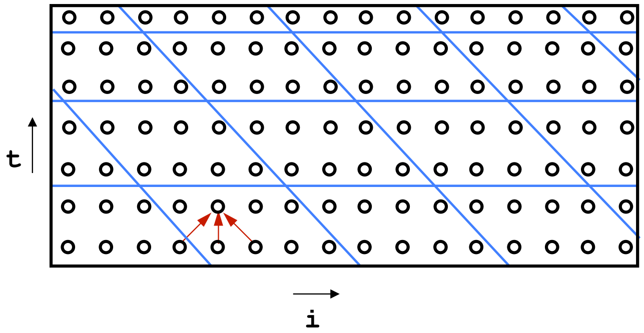
- Cache misses = Θ(TN)
- Concurrency in each t

- Cache misses = Θ(TN/B)
- No concurrent in a row

- Time tiling causes pipelined execution
- Solution: Adjust tiling – re-enable concurrent execution in a row of tiles

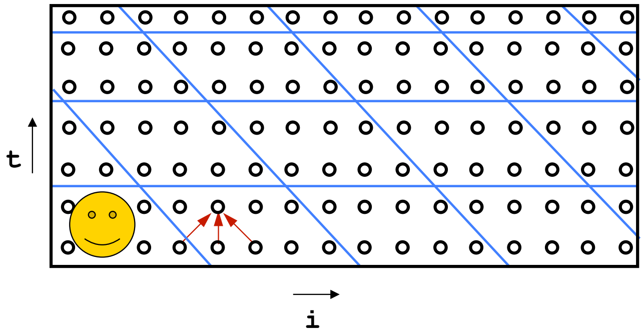
Motivation

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



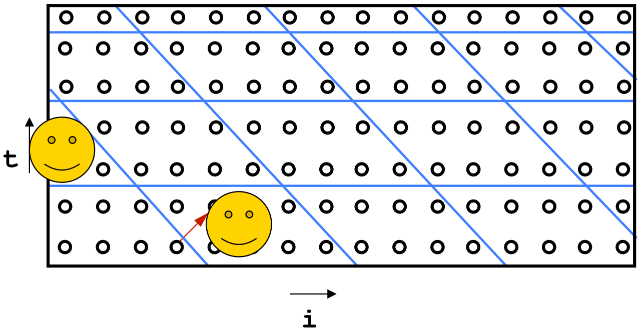
Motivation

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



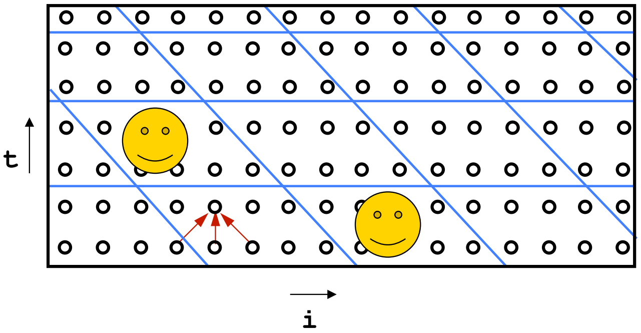
Motivation

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



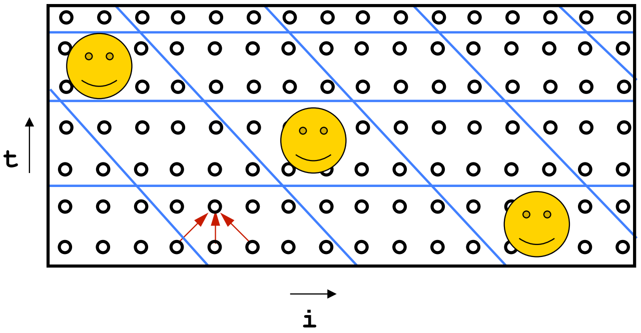
Motivation

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



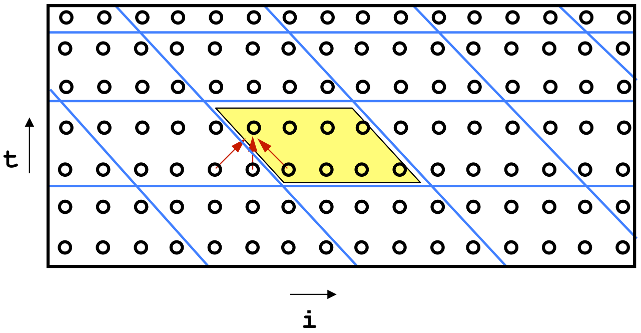
Motivation

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



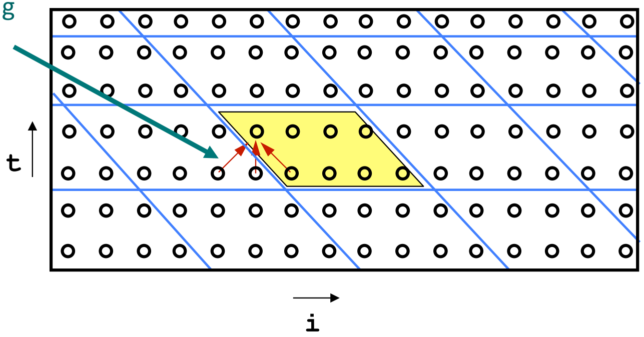
Motivation

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



Motivation

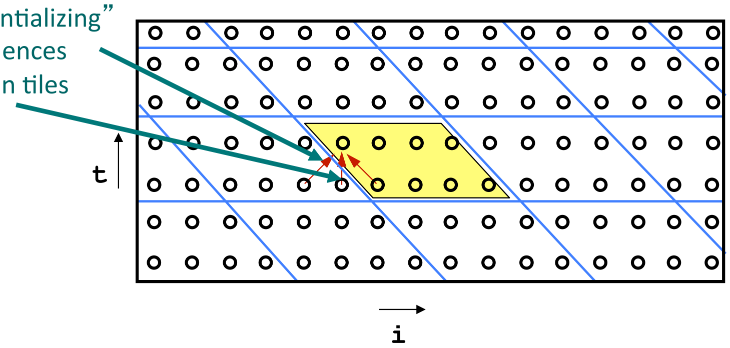
```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



“Sequentializing” dependence between tiles

Example

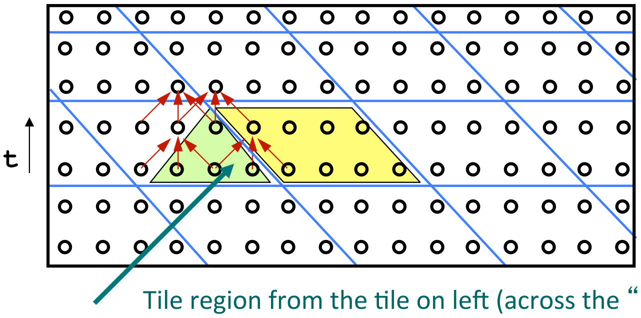
```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



“Sequentializing” dependences between tiles

Example

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```

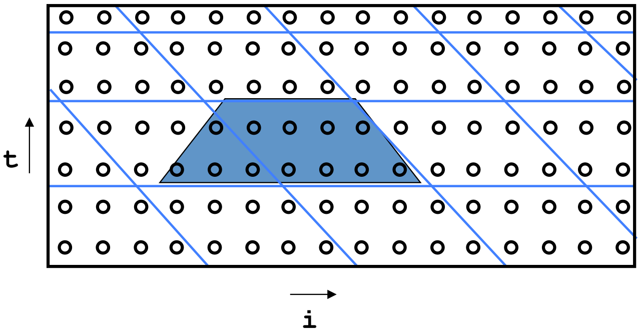


Tile region from the tile on left (across the "backface")
that needs to be finished before this tile can start

Overlapped Tiling

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```

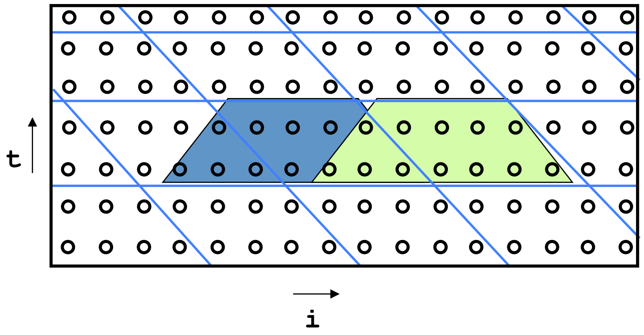
Overlapped Tiling



Overlapped Tiling

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```

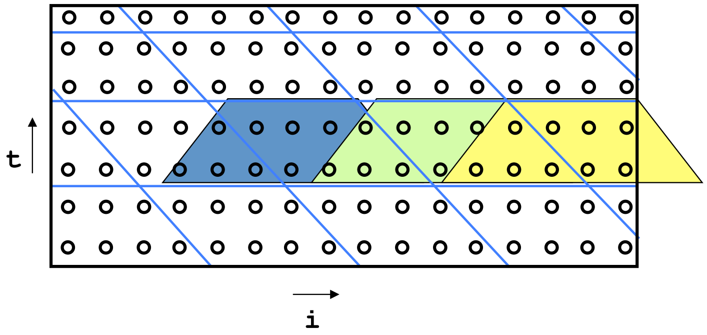
Overlapped Tiling



Overlapped Tiling

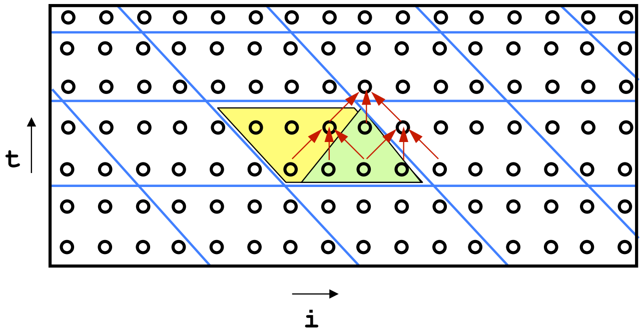
```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```

Overlapped Tiling



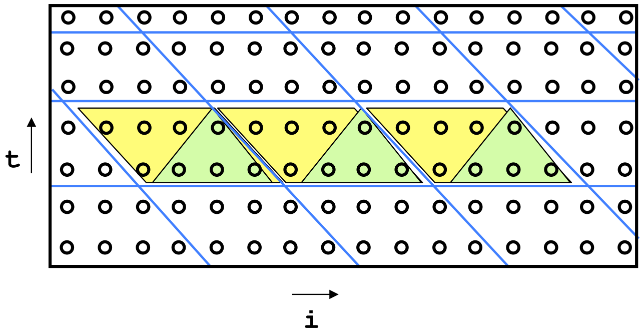
Split Tiling

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



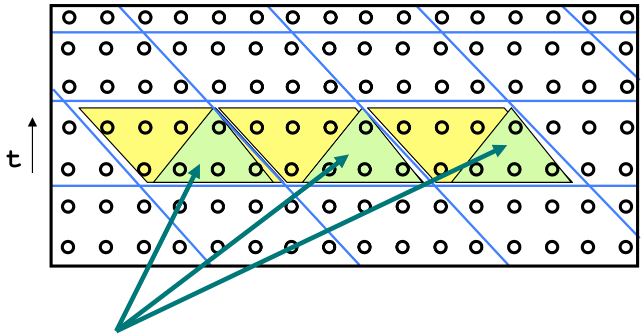
Split Tiling

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



Split Tiling

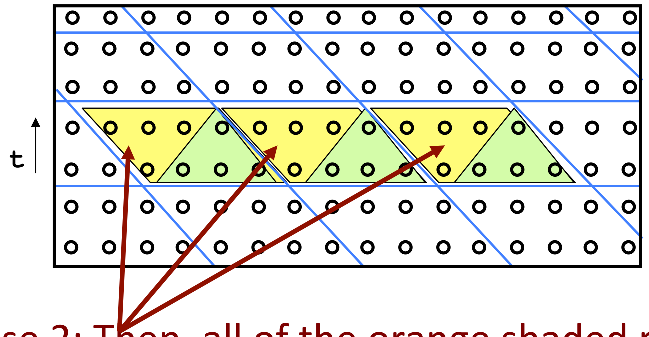
```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



Phase 1: All of the green shaded regions can be executed concurrently (first) once previous row of tiles are done

Example: Split Tiling

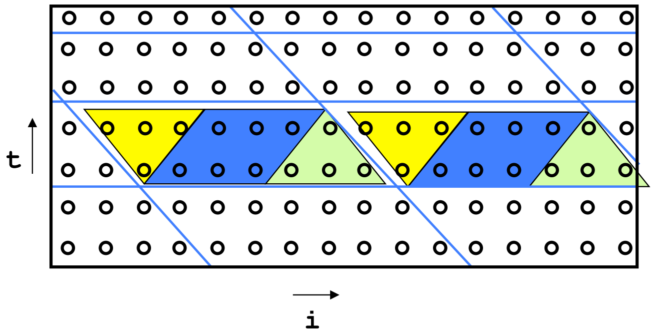
```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



Phase 2: Then, all of the orange shaded regions can be executed concurrently (next)

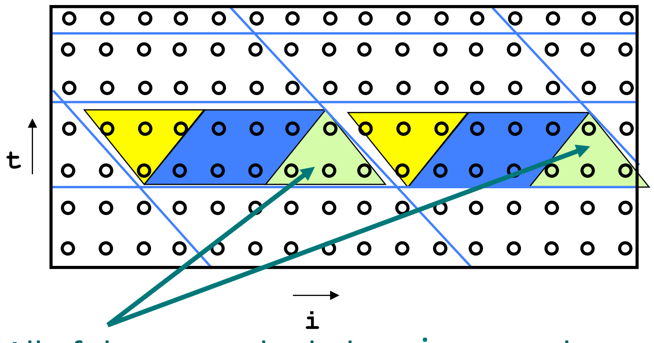
Split Tiling (no size assumptions)

```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



Split Tiling (no size assumptions)

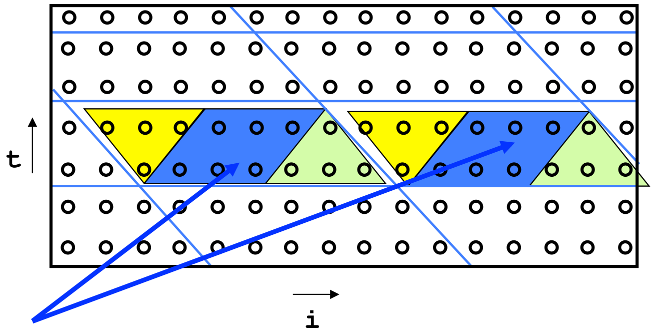
```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



Phase 1: All of the green shaded regions can be executed concurrently (first) once previous row of tiles are done

Split Tiling (no size assumptions)

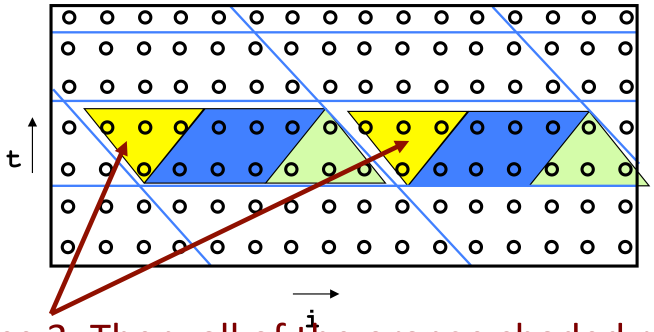
```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



Phase 2: All of the blue shaded regions can be executed concurrently (second)

Split Tiling (no size assumptions)

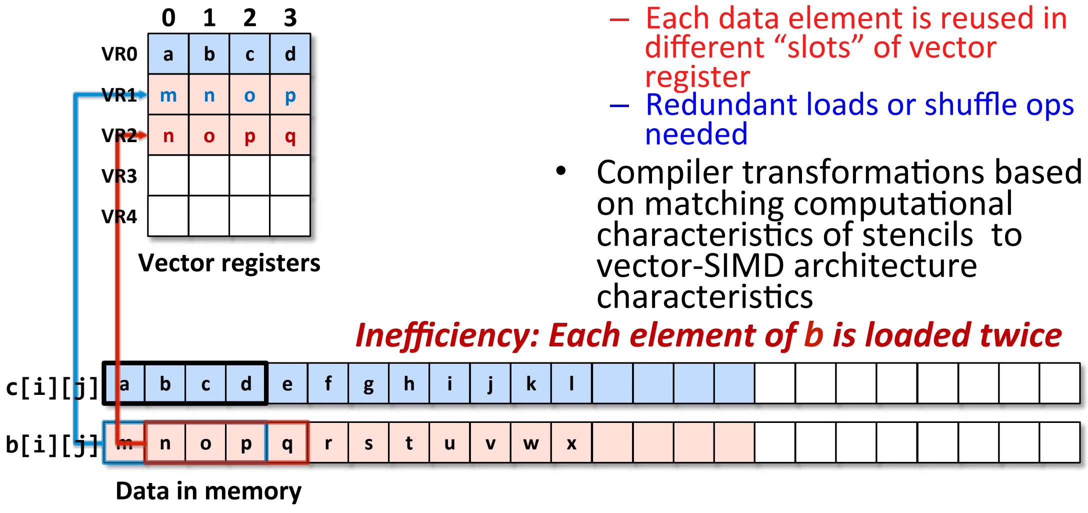
```
FOR t = 0 TO T-1
  FOR i = 1 TO N-1
    A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
```



Phase 3: Then, all of the orange shaded regions can be executed concurrently (next)

Stencils on Vector-SIMD Processors

```
for (i=0; i<H; ++i)
 for (j=0; j<W; ++j)
  c[i][j]+=b[i][j]+b[i][j+1];
```



	0	1	2	3
VR0	a	b	c	d
VR1	m	n	o	p
VR2	n	o	p	q
VR3				
VR4				

Vector registers

c[i][j]	a	b	c	d	e	f	g	h	i	j	k	l
b[i][j]	m	n	o	p	q	r	s	t	u	v	w	x

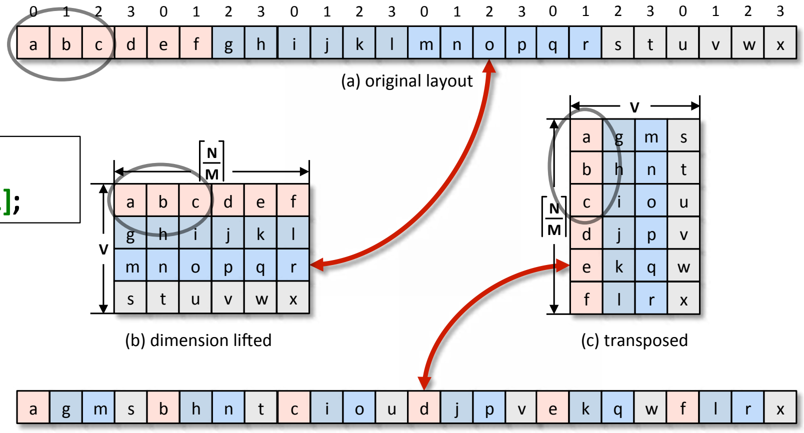
Data in memory

- Fundamental source of inefficiency with stencil codes on current short-vector SIMD ISAs (e.g. SSE, AVX …)
  - Concurrent operations on contiguous elements
  - Each data element is reused in different “slots” of vector register
  - Redundant loads or shuffle ops needed
- Compiler transformations based on matching computational characteristics of stencils to vector-SIMD architecture characteristics

Inefficiency: Each element of b is loaded twice

Data Layout Transformation

```
for (i = 0; i < N; ++i)
  a[i]=b[i-1]+b[i]+b[i+1];
```



0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x

(a) original layout

a	b	c	d	e	f
g	h	i	j	k	l
m	n	o	p	q	r
s	t	u	v	w	x

(b) dimension lifted

a	g	m	s
b	h	n	t
c	i	o	u
d	j	p	v
e	k	q	w
f	l	r	x

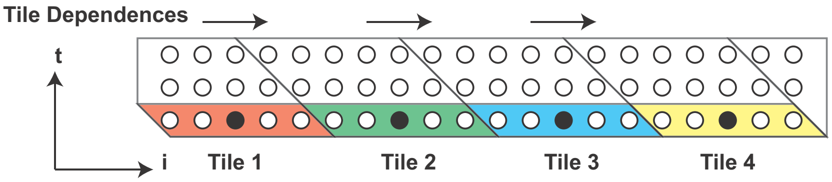
(c) transposed

a	g	m	s	b	h	n	t	c	i	o	u	d	j	p	v	e	k	q	w	f	l	r	x

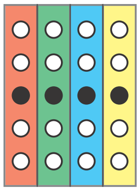
(d) transformed layout

- 1D vector in memory ⇔ (b) 2D logical view of same data
- (c) Transposed 2D array moves interacting elements into same slot of different vectors ⇔ (d) New 1D layout after transformation
- Boundaries need special handling

Standard Tiling with DLT



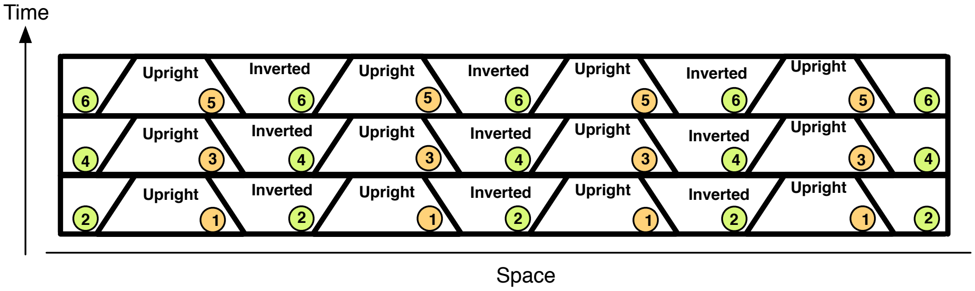
(a) Standard tiling -- Linear view



(b) Standard tiling -- DLT view (t=1)

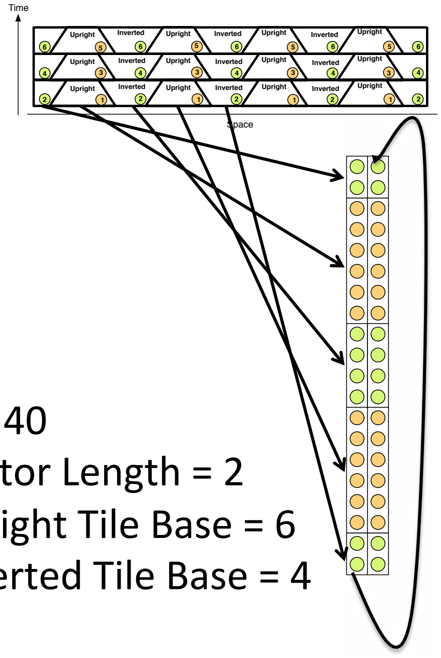
- Standard tiling cannot be used with the layout transform
- Inter-tile dependences prevent vectorization

Split Tiling



- Divide iteration space into upright and inverted tiles
- For each *tt* timesteps where *tt* = time tile size…
  - Execute upright tiles in parallel
  - Execute inverted tiles in parallel
- **Upright tile size increases with time tile size**

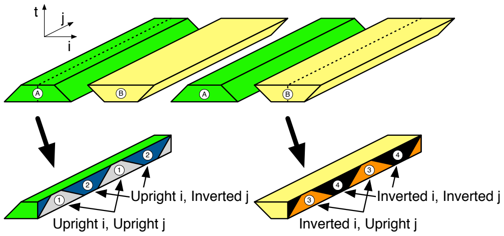
Split Tiling: DLT View



N = 40
Vector Length = 2
Upright Tile Base = 6
Inverted Tile Base = 4

- Tiles at t = 0
  - Orange upright tiles
  - Green inverted tiles
- Tiles in same vector slot
  - Compute multiple tiles in parallel
  - Some inverted tiles split DLT boundary

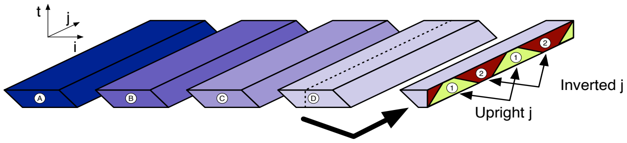
Nested Split Tiling



```
for tt
  parfor ii // (A) Upright i
    parfor jj // (1) Upright j
      for t { for i { for j {}}}};
    barrier();
    parfor jj // (2) Inverted j
      for t { for i { for j {}}}};
    barrier();
  parfor ii // (B) Upright j
    parfor jj // (3) Upright j
      for t { for i { for j {}}}};
    barrier();
    parfor jj // (4) Inverted j
      for t { for i { for j {}}}};
    barrier();
```

- Split-tile outermost space loop *d*
- Creates upright, inverted tiles which are each split-tiled on loop *d-1*
- Split-tiling proceeds recursively to innermost dimension
- But data footprint of tile grows in each spatial dimension, proportional to time-tile size

Hybrid Split Tiling

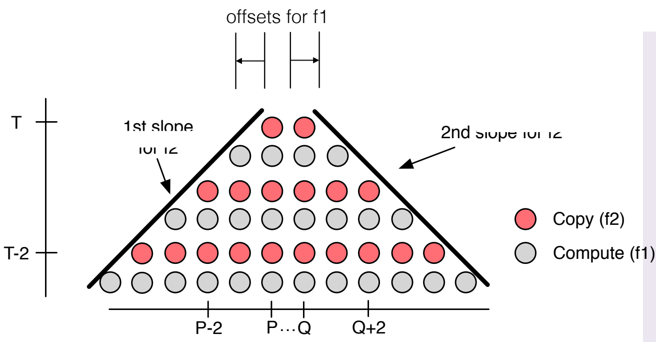


```
for tt
  for ii // (A) (B) (C) (D) Traditional i
    parfor jj // (1) Upright j
      for t { for i { for j {}}}};
    barrier();
    parfor jj // (2) Inverted j
      for t { for i { for j {}}}};
    barrier();
```

- **Parallelogram tile size along spatial dimensions are unconstrained by time tile size**
- Hybrid scheme: use parallelogram tiling for some spatial dimensions and split tiling for the rest
- **Allows smaller tile footprint for higher dimensional stencils**

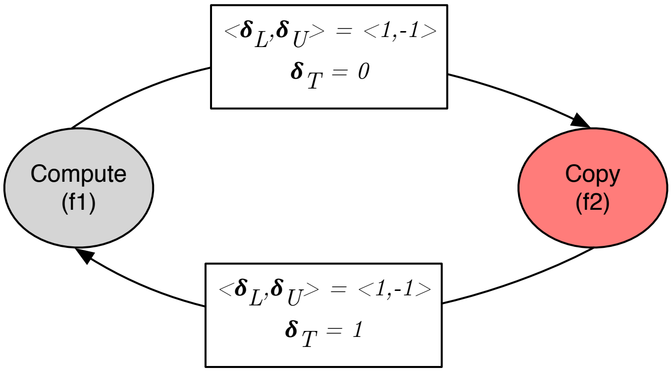
Back-Slicing Analysis

- Need to find geometric properties of split tiles
  - Slopes of tile in each dimension *d*
  - Offset of each statement w.r.t. tile start, tile end



```
for (t=0;t<100;++t) {
 for (i=1;i<999;++i)
  f1: a1[i]=0.33*(a0[i-1]+
                  a0[i  ]+
                  a0[i+1]);
 for (i=1;i<999;++i)
  f2: a0[i]=a1[i];
}
```

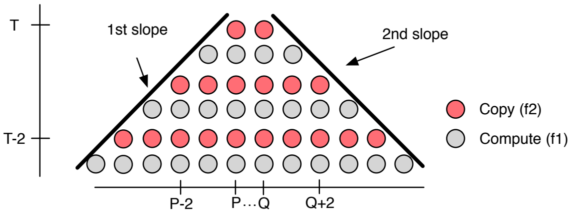
Dependence Summary Graph(DSG)

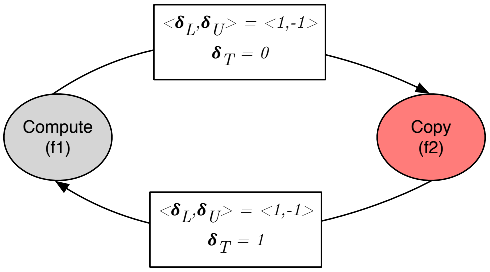


$<\delta_L,\delta_U> = <1,-1>$
$\delta_T = 0$
Compute
(f1)
Copy
(f2)
$<\delta_L,\delta_U> = <1,-1>$
$\delta_T = 1$

- Vertices represent statements
- Edges represent dependence summaries for each dimension
    - $<\delta_L, \delta_U>$ → max/min spatial components of flow and anti dependences
    - δ_T → Time distance between statements

Computing Slopes



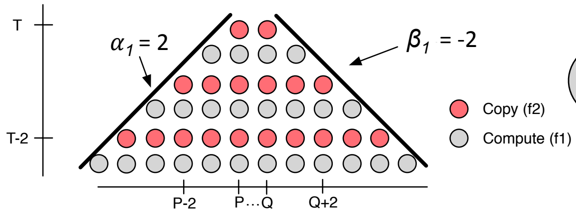


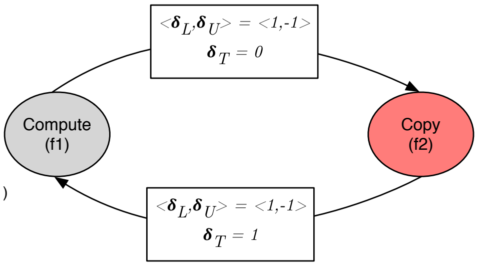
$<\delta_L,\delta_U> = <1,-1>$
$\delta_T = 0$
Compute
(f1)
Copy
(f2)
$<\delta_L,\delta_U> = <1,-1>$
$\delta_T = 1$

- Compute *cycle ratios* $\rho_L(C)$, $\rho_U(C)$ for each cycle C of the DSG

$$\rho_L(C) = \frac{\sum_C \delta_L}{\sum_C \delta_T} = \frac{2}{1} = 2 \qquad \rho_U(C) = \frac{\sum_C \delta_U}{\sum_C \delta_T} = \frac{-2}{1} = -2$$

Computing Slopes




$<\delta_L,\delta_U> = <1,-1>$
$\delta_T = 0$
Compute (f1)
Copy (f2)
$<\delta_L,\delta_U> = <1,-1>$
$\delta_T = 1$

- For each dimension *d* of the stencil…
  - Lower bound slope α_d is maximum cycle ratio
  - Upper bound slope β_d is minimum cycle ratio

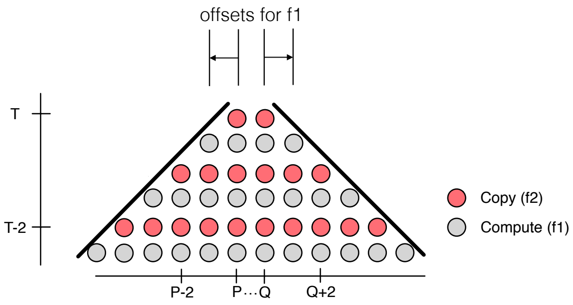
$$\alpha_d = \max\left(\rho_L(C)\right)\forall C \in DSG = 2$$
$$\beta_d = \min\left(\rho_U(C)\right)\forall C \in DSG = -2$$

Computing Offsets

- Build a system of validity constraints using loop bounds of upright tile code
- Results in system of linear inequalities

```
for (tt=...){
 for (ii=...){
  for (t=...){
   for (i=ii+oLF1+αL*(t-tt);
        i<ii+TU+oUF1+βU*(t-tt);
        ++i)
    f1: a1[i] = 0.33*(a0[i-1]+
                      a0[i  ]+
                      a0[i+1]);

   for (i=ii+oLF2+αL*(t-tt);
        i<ii+TU+oUF2+βU*(t-tt);
        ++i)
    f2: a0[i] = a1[i];
}}}
```



Computing Offsets

- For any pair of dependent statements, given a region over which the target statement is executed, the source statement should be executed over a region large enough to satisfy the dependence

<u>Lower Bound Constraints</u>

$$ii + o_L^{f1} + \alpha * t \le ii + o_L^{f2} + \alpha * t - 1$$

$$ii + o_L^{f2} + \alpha * (t-1) \le ii + o_L^{f1} + \alpha * t - 1$$

<u>Upper Bound Constraints</u>

$$ii + T_U + o_U^{f1} + \beta * t \ge ii + T_U + o_U^{f2} + \beta * t + 1$$

$$ii + T_U + o_U^{f2} + \beta * (t-1) \le ii + T_U + o_U^{f1} + \beta * t + 1$$

```
for (tt=...){
 for (ii=...){
  for (t=...){
   for (i=ii+o_LF1+α_L*(t-tt);
        i<ii+T_U+o_UF1+β_U*(t-tt);
        ++i)
    f1: a1[i] = 0.33*(a0[i-1]+
                      a0[i  ]+
                      a0[i+1]);

   for (i=ii+o_LF2+α_L*(t-tt);
        i<ii+T_U+o_UF2+β_U*(t-tt);
        ++i)
    f2: a0[i] = a1[i];
}}}
```

Computing Offsets

- Simplify to a system of difference constraints
- Solve with Bellman-Ford algorithm

<u>Lower Bound Constraints</u>

$$o_L^{f1} - o_L^{f2} \le -1$$

$$o_L^{f2} - o_L^{f1} \le \alpha - 1$$

<u>Upper Bound Constraints</u>

$$o_U^{f2} - o_U^{f1} \le -1$$

$$o_U^{f1} - o_U^{f2} \le -\beta - 1$$

<u>Lower Bound Offsets</u>

$$o_L^{f1} = -1$$

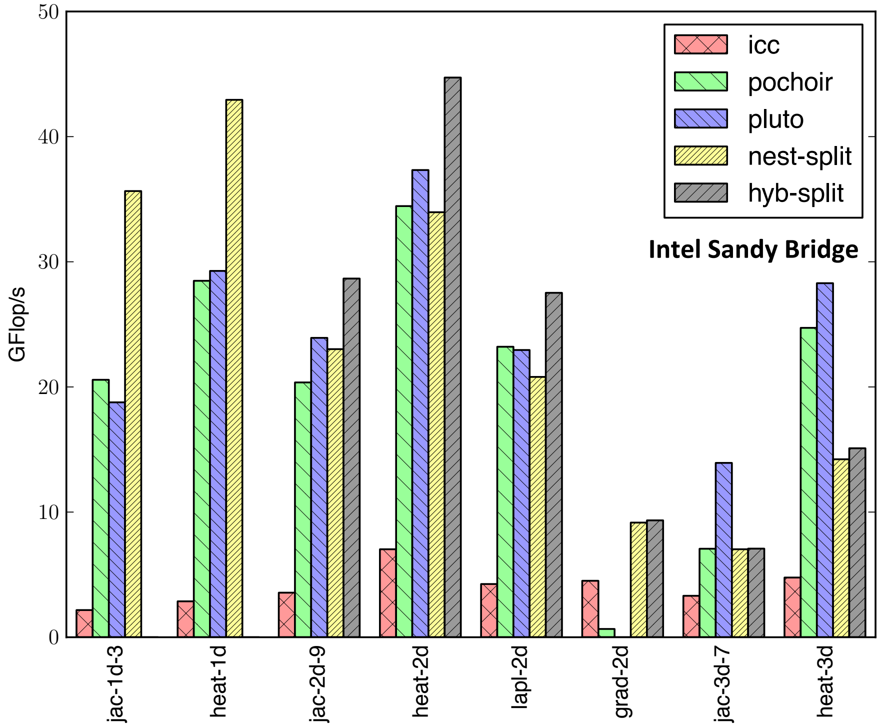
$$o_L^{f2} = 0$$

<u>Upper Bound Offsets</u>

$$o_U^{f1} = 1$$

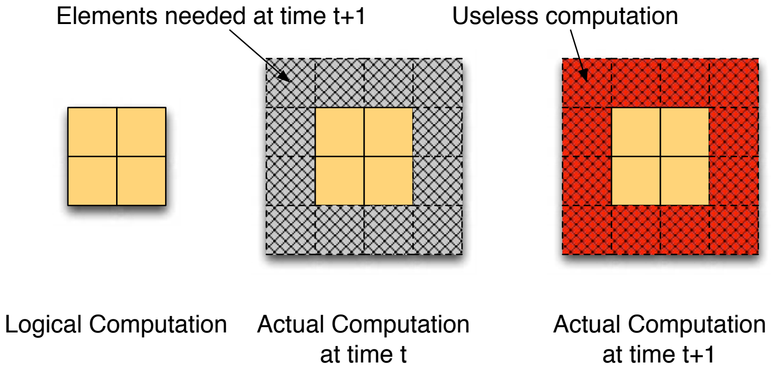
$$o_U^{f2} = 0$$

Stencils on Multicore CPU: Performance



Stencils on GPUs

- Vector-SIMD alignment problems non-existent
- Different optimization challenges: limited forms of synchronization, avoidance of thread divergence
- Overlapped tiling: *Redundantly* compute neighboring cells to avoid inter-thread-block sync, lower communication, and avoid thread divergence


Elements needed at time t+1

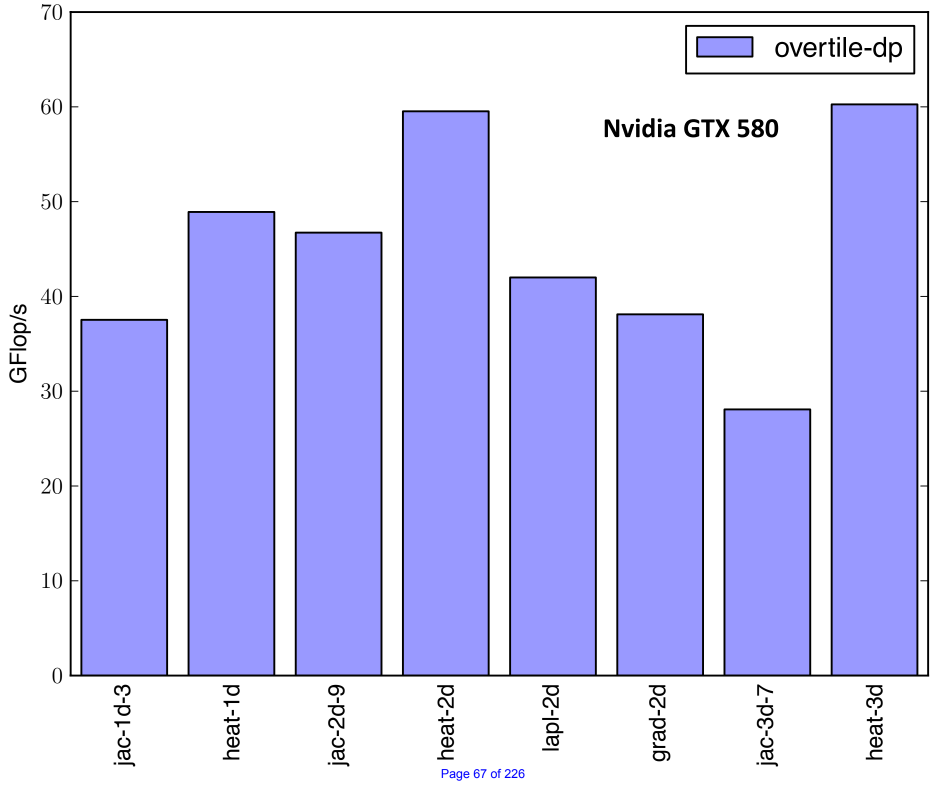
Useless computation

Logical Computation

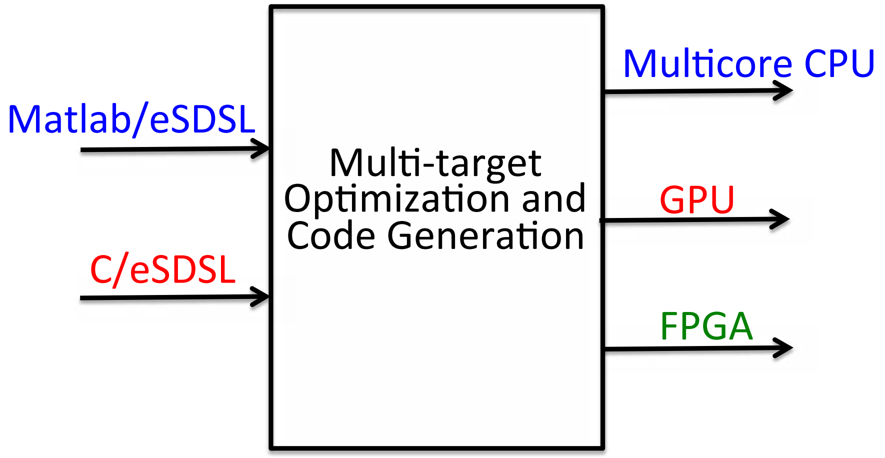
Actual Computation at time t

Actual Computation at time t+1

Stencils on GPU: Performance



Multi-Target Code Generation from SDSL



Matlab/eSDSL

C/eSDSL

Multi-target Optimization and Code Generation

Multicore CPU

GPU

FPGA

Summary so far …

- Overlapped and split tiling to recover concurrency (without startup overhead) in tiled execution of stencil computations.
- Stencil computations suffer from stream-alignment conflict for vector-SIMD ISAs
  - Data Layout Transformation to avoid the conflict
  - Split Tiling to enable concurrency along with DLT
- Overlapped tiling and split tiling on GPUs
- Performance improvement over state-of-the-art for 1D and 2D benchmarks
- Multi-target compiler for Stencil DSL in progress
- Recent work on related fusion and tiling for unstructured meshes (with Michelle Strout and Paul Kelly)

Higher Order Stencils Ain’t So Bad: A Framework for Enhancing Data Reuse via Associative Reordering

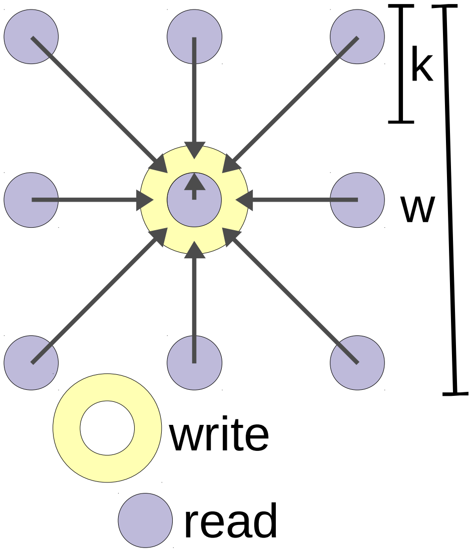
Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, **J. Ramanujam**, P. Sadayappan

The Ohio State University, Rice University,
ETH, INRIA, Louisiana State University

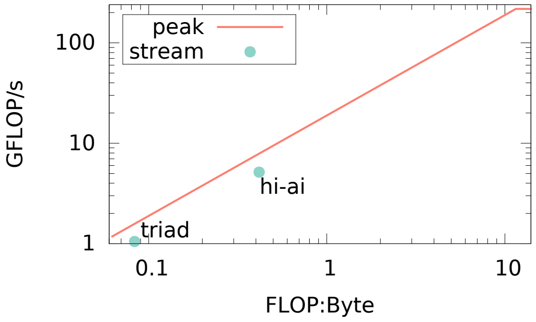
May 12, 2016

Stencils

```
for (i=k; i<N-k; i++)
 for (j=k; j<N-k; j++)
  for (ii=-k; ii<=k; ii++)
   for (jj=-k; jj<=k; jj++)
    OUT[i][j] +=
     IN[i+ii][j+jj]*C[ii][jj]
```



Roofline Model



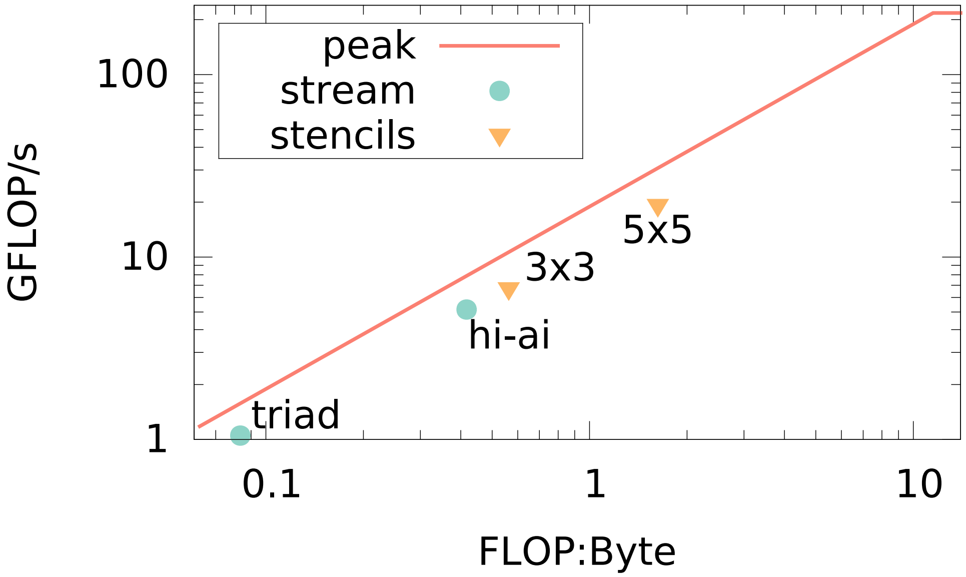
Triad

```
for (t=0; t<T; t++)
  for (i=0; i<N; i++)
    C[i] = A[i]*X + B[i]
```

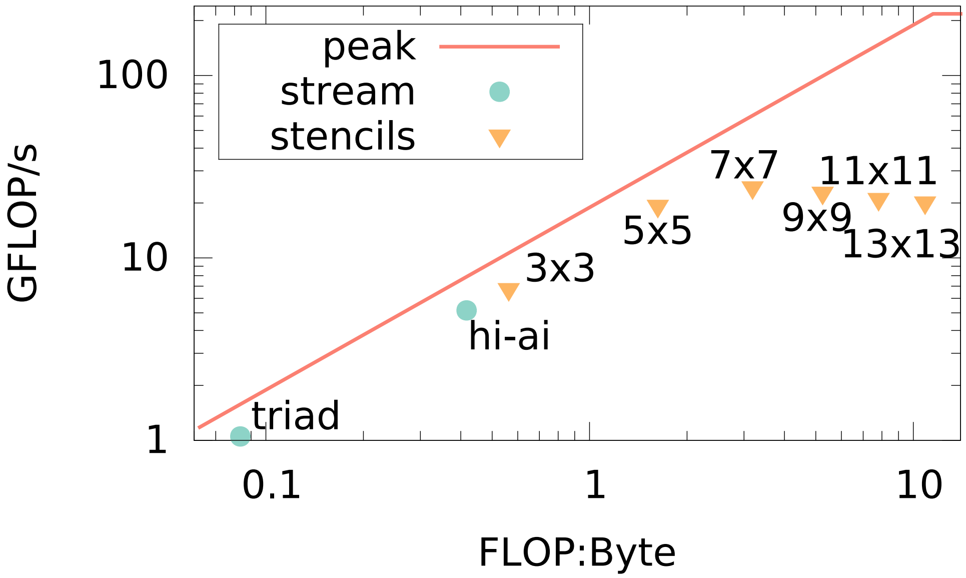
High arithmetic intensity triad

```
for (t=0; t<T; t++)
  for (i=0; i<N; i++)
    C[i] = A[i]*A[i] +
           A[i]*B[i] +
           B[i]*B[i] +
           A[i]*X +
           B[i]*Y + Z
```

Roofline Model Stencils

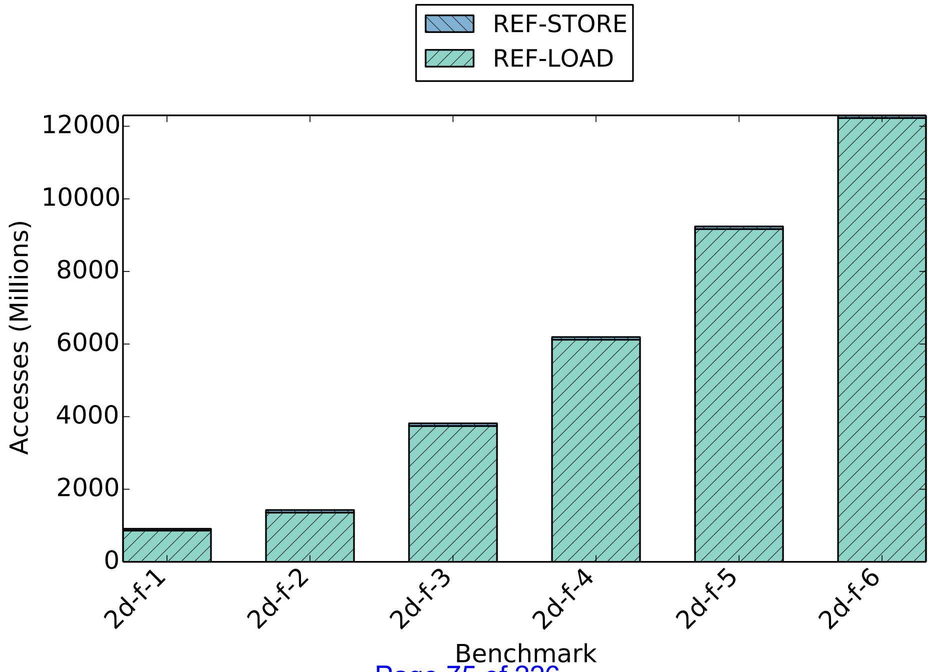


Roofline Model Stencils

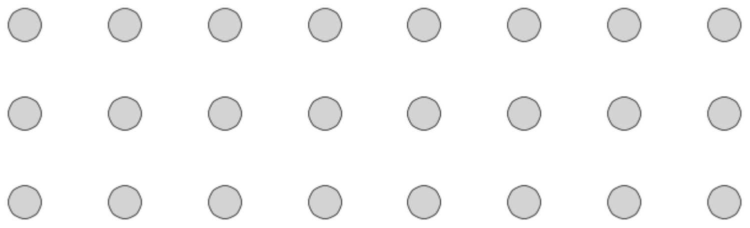


Problem: Performance *does not scale* with arithmetic intensity!

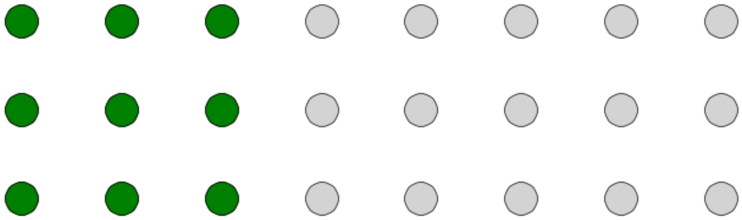
Bottleneck



Register reuse

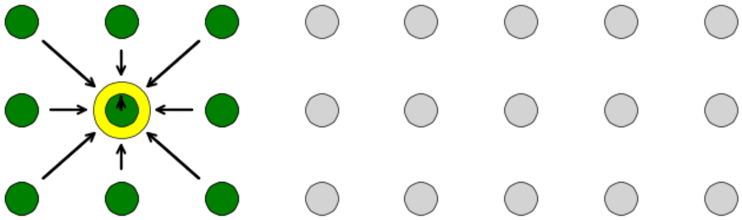


Register reuse



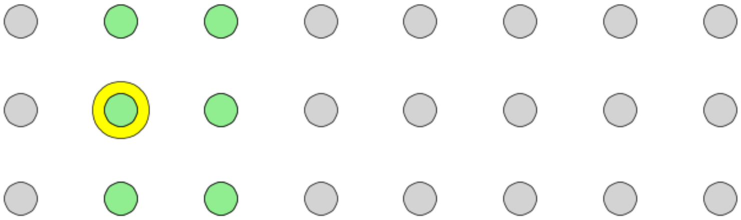
Page 77 of 226

Register reuse

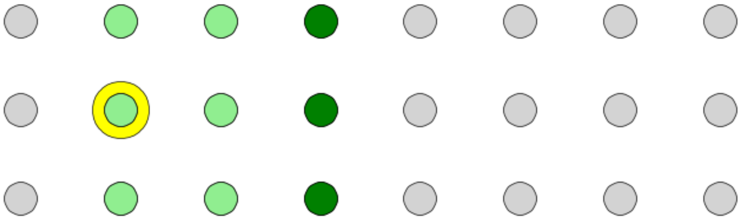


Page 78 of 226

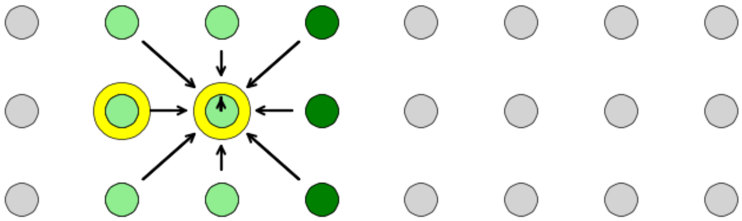
Register reuse



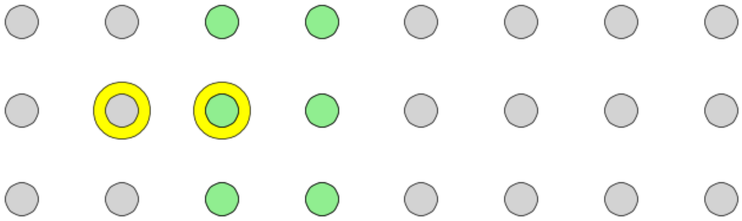
Register reuse



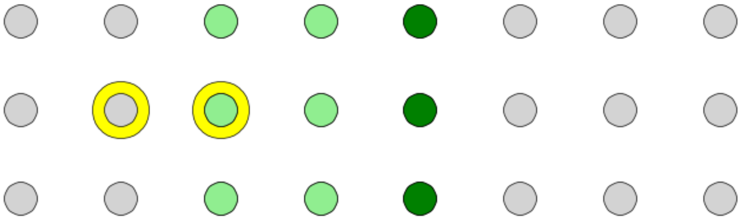
Register reuse



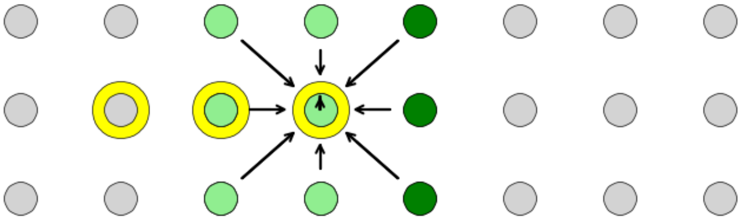
Register reuse



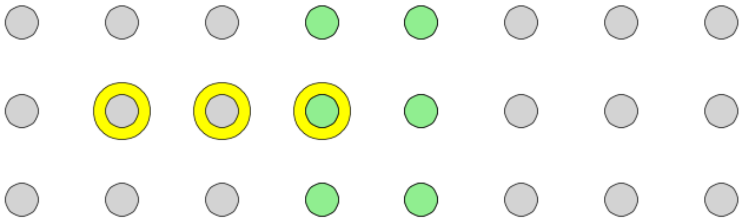
Register reuse



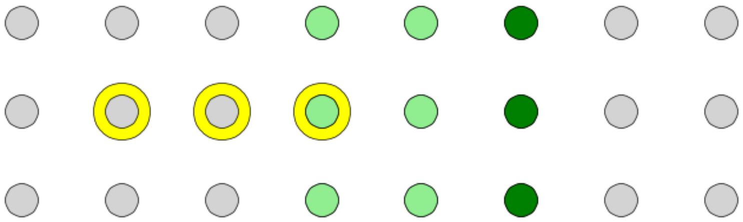
Register reuse



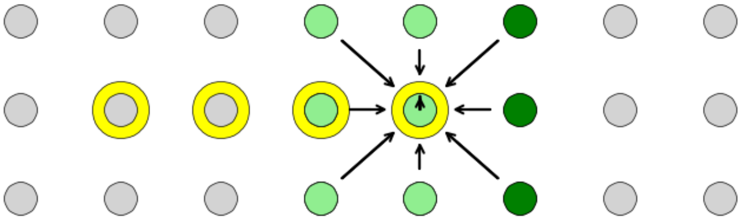
Register reuse



Register reuse

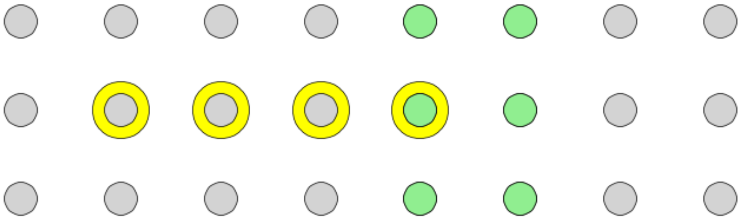


Register reuse



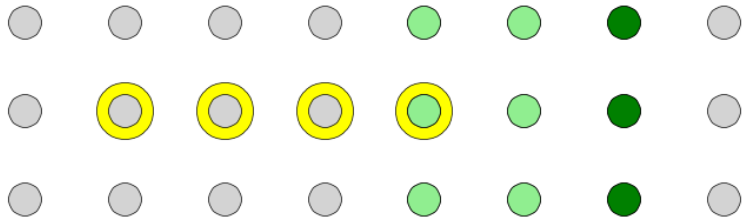
Page 87 of 226

Register reuse

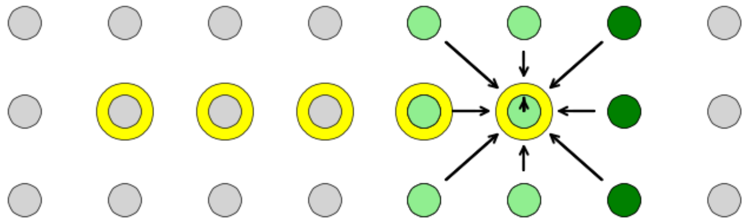


Page 88 of 226

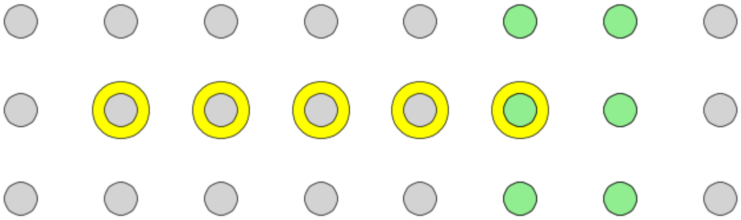
Register reuse



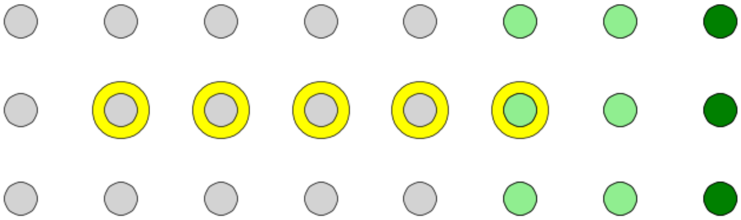
Register reuse



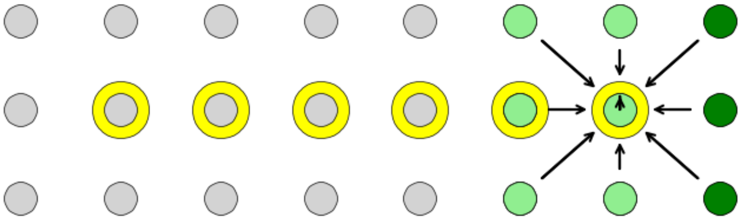
Register reuse



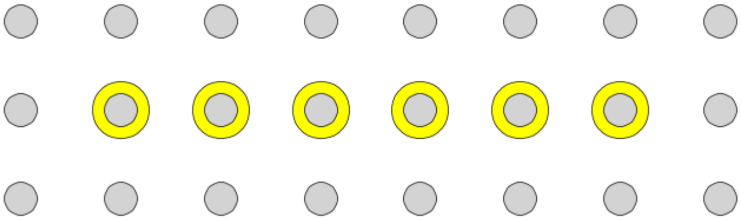
Register reuse



Register reuse



Register reuse



Contributions

1. **Identified Problem**:
   - *Register reuse* for stencil computations

Contributions

1. **Identified Problem**:
   - *Register reuse* for stencil computations
2. **Solution**:
   - Exploit *associativity & commutativity* to increase data-locality

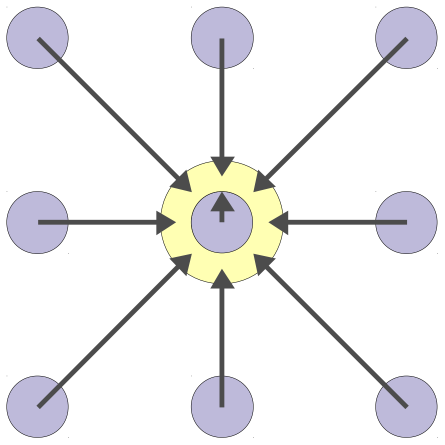
Contributions

1. **Identified Problem**:
   - *Register reuse* for stencil computations
2. **Solution**:
   - Exploit *associativity & commutativity* to increase data-locality
3. Cost model

Contributions

1. **Identified Problem**:
   - *Register reuse* for stencil computations
2. **Solution**:
   - Exploit *associativity & commutativity* to increase data-locality
3. Cost model
4. Experimental results

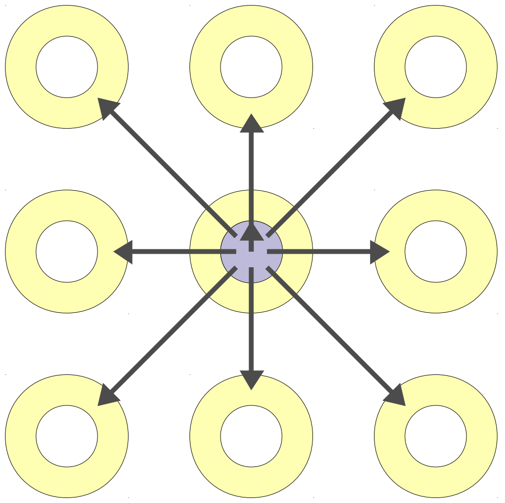
Gather-Gather



- w reads from *IN*
- 0 reads from *OUT*
- 1 write to *OUT*
- $w^2 - w + 1$ registers

```
for (i=k; i<N-k; i++)
 for (j=k; j<N-k; j++)
  for (ii=-k; ii<=k; ii++)
   for (jj=-k; jj<=k; jj++)
    OUT[i][j] +=
     IN[i+ii][j+jj]*C[ii][jj]
```

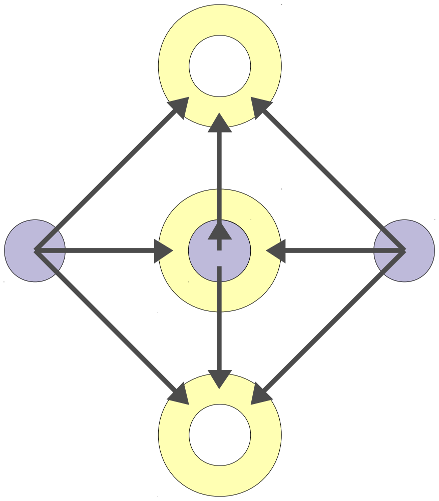
Scatter-Scatter



- 1 reads from *IN*
- $w - 1$ reads from *OUT*
- w write to *OUT*
- $w^2 - w + 1$ registers

```
for (i=k; i<N-k; i++)
 for (j=k; j<N-k; j++)
  for (ii=-k; ii<=k; ii++)
   for (jj=-k; jj<=k; jj++)
    OUT[i-ii][j-jj] +=
     IN[i][j]*C[ii][jj]
```

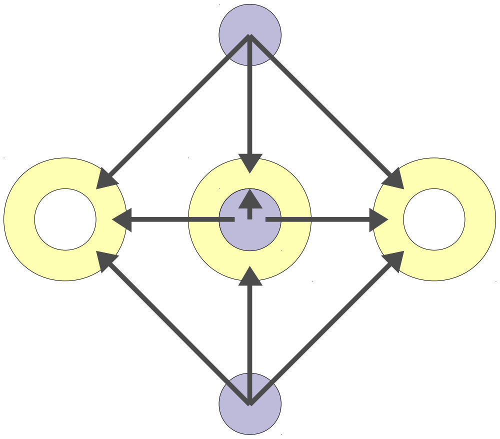
Gather-Scatter



- 1 reads from *IN*
- $w - 1$ reads from *OUT*
- w write to *OUT*
- $w + 1$ registers

```
for (i=1; i<N-1; i++)
 for (j=1; j<N-1; j++)
   t1 = t2 // IN[i][j-1]
   t2 = t3 // IN[i][j]
   t3 = IN[i][j+1]
   OUT[i-1][j] = t1 + t2 + t3
   OUT[i][j] = t1 + t2 + t3
   OUT[i+1][j] = t1 + t2 + t3
```

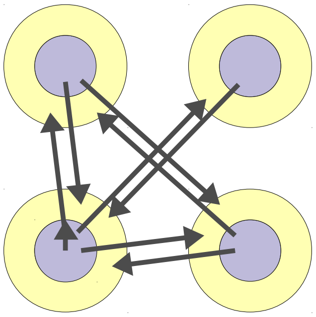
Scatter-Gather



- w reads from *IN*
- 0 reads from *OUT*
- 1 write to *OUT*
- $w + 1$ registers

```
for (i=1; i<N-1; i++)
 for (j=1; j<N-1; j++)
   x = IN[i-1][j] +
       IN[i][j] + IN[i+1][j]
   t1 = t2 + x
   t2 = t3 + x
   t3 = x
   OUT[i][j-1] = t1
```

Compact



- $\lceil w/2 \rceil$ reads from IN
- $w/2$ reads from OUT
- $w/2$ write to OUT
- $2 \cdot (w/2)^2$ registers

Multidimensional Retiming

```
for (i=W; i<X; i++)
    for (j=Y; j<Z; j++) {
R:      A[i][j] += C[i][j]
S:      B[i][j] += C[i][j+T]
    }
```

Original Code: `C[i][j]` and `C[i][j+T]` accessed in same iteration

Multidimensional Retiming

```
for (i=W; i<X; i++) {
    for (j=Y; j<Y+T; j++)
R1:     A[i][j] += C[i][j]
    for (j=Y+T; j<Z; j++) {
R2:     A[i][j] += C[i][j]
S1:     B[i][j-T] += C[i][j]
    }
    for (j=Z; j<Z+T; j++)
S2:     B[i][j-T] += C[i][j]
}
```

Retimed Code: `C[i][j]` and `C[i][j]` accessed in same iteration

Retiming Vectors

- Program contains multiple reduction statement
- Vector of loop offsets per statement
- Offsets can be applied polyhedrally to a statements schedule

```
for (i=1; i<N; i++)
  OUT[i] += IN[i-1]
  OUT[i] += IN[i]
  OUT[i] += IN[i+1]
```

Applying vectors $< -1 >$, $< 0 >$, $< 1 >$ becomes:

```
OUT[1] += IN[0]
for (i=1; i<N-1; i++)
  OUT[i+1] += IN[i]
  OUT[i]   += IN[i]
  OUT[i-1] += IN[i]
OUT[N-2] += IN[N-1]
```

Applicability

1. Loop bounds must be affine

Applicability

1. Loop bounds must be affine
2. Arrays and scalars only, no pointers

Applicability

1. Loop bounds must be affine
2. Arrays and scalars only, no pointers
3. Access functions do not need to be affine

Applicability

1. Loop bounds must be affine
2. Arrays and scalars only, no pointers
3. Access functions do not need to be affine
4. Functions must be side effect free

Applicability

1. Loop bounds must be affine
2. Arrays and scalars only, no pointers
3. Access functions do not need to be affine
4. Functions must be side effect free
5. Retiming changes order of operations

Applicability

1. Loop bounds must be affine
2. Arrays and scalars only, no pointers
3. Access functions do not need to be affine
4. Functions must be side effect free
5. Retiming changes order of operations
6. Semantics preserved when using an associative & commutative operator
    - for direct convolutions
    - for sum-of-product stencils

Framework Demo - Input

```
for (i=k; i<N-k; i++)
  for (j=k; j<N-k; j++)
    OUT[i][j] = 0
    OUT[i][j] += IN[i-1][j-1] * C[-1][-1]
    OUT[i][j] += IN[i-1][j]   * C[-1][0]
    OUT[i][j] += IN[i-1][j+1] * C[-1][1]
    OUT[i][j] += IN[i][j-1]   * C[0][-1]
    OUT[i][j] += IN[i][j]     * C[0][0]
    OUT[i][j] += IN[i][j+1]   * C[0][1]
    OUT[i][j] += IN[i+1][j-1] * C[1][-1]
    OUT[i][j] += IN[i+1][j]   * C[1][0]
    OUT[i][j] += IN[i+1][j+1] * C[1][1]
```

Framework Demo - Compact Representation

Compact Representation:

```
for (i=k; i<N-k; i++)
  for (j=k; j<N-k; j++)
    OUT[i][j] = 0
    for (ii=-k; ii<=k; ii++)
      for (jj=-k; jj<=k; jj++)
        OUT[i][j] += IN[i+ii][j+jj]*C[ii][jj]
```

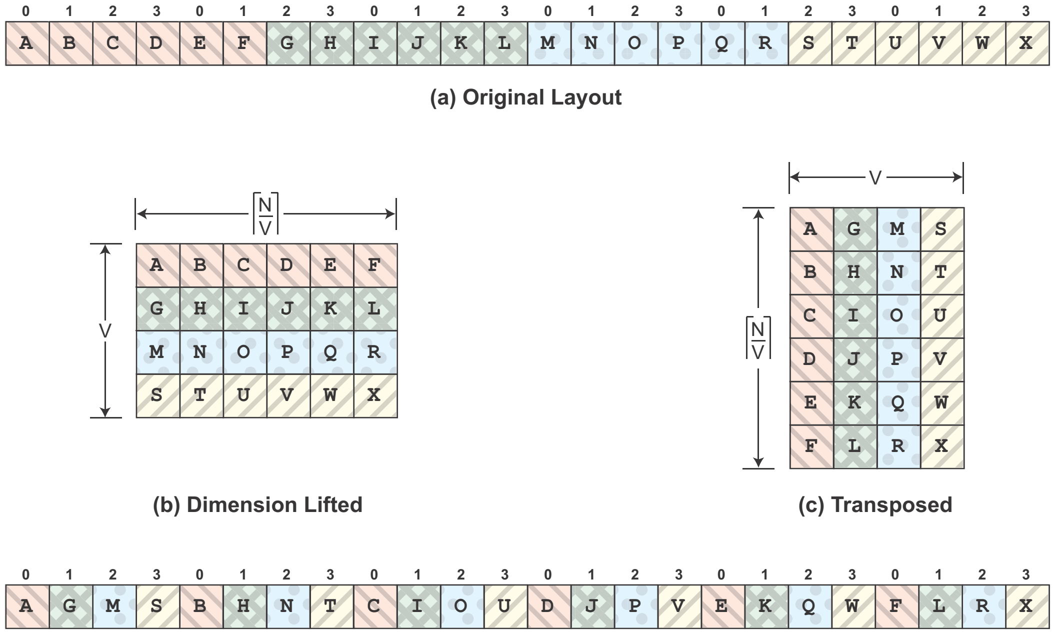
Retiming:

```
for (i=2*k; i<N-2*k; i++)
  for (j=k; j<N-k; j++)
    OUT[i+k][j] = 0
    for (ii=-k; ii<=k; ii++)
      for (jj=-k; jj<=k; jj++)
        OUT[i-ii][j] += IN[i][j+jj]*C[ii][jj]
```

Framework Demo - Prolog/Epilog

```
for (i=0; i<2*k; i++)
  for (j=k; j<N-k; j++)
    OUT[i+k][j] = 0
    for (ii=-k; ii<=-k+i; ii++)
      for (jj=-k; jj<=k; jj++)
        OUT[i-ii][j] += IN[i][j+jj]*C[ii][jj]
for (i=2*k; i<N-2*k; i++)
  for (j=k; j<N-k; j++)
    OUT[i+k][j] = 0
    for (ii=-k; ii<=k; ii++)
      for (jj=-k; jj<=k; jj++)
        OUT[i-ii][j] += IN[i][j+jj]*C[ii][jj]
for (i=N-2*k; i<N; i++)
  for (j=k; j<N-k; j++)
    for (ii=i-N+k+1; ii<=k; ii++)
      for (jj=-k; jj<=k; jj++)
        OUT[i-ii][j] += IN[i][j+jj]*C[ii][jj]
```

Dimension Lifted Transposition (CC'11)



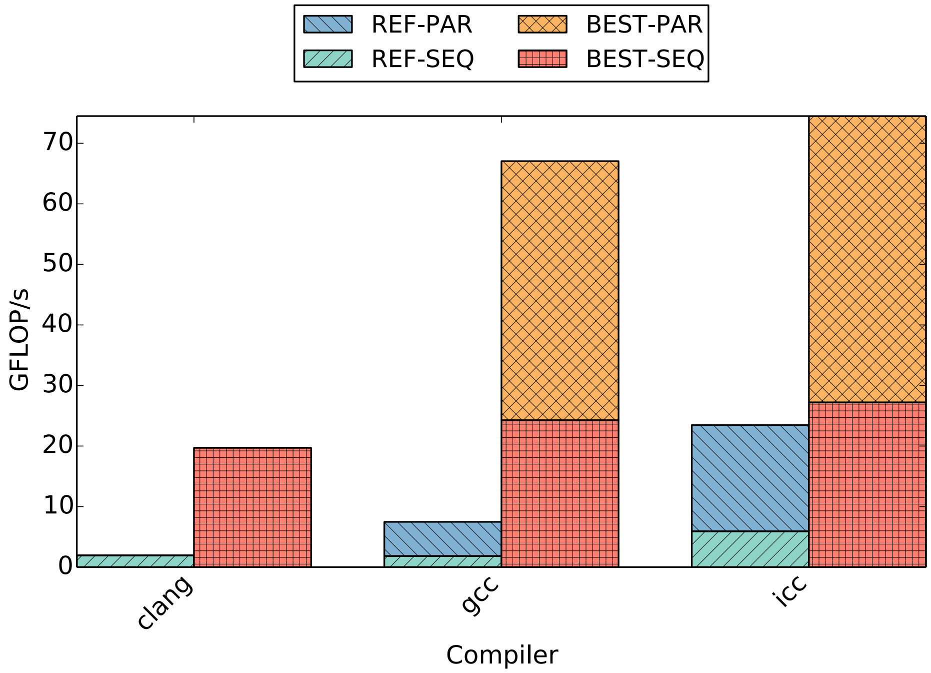
(a) Original Layout

(b) Dimension Lifted

(c) Transposed

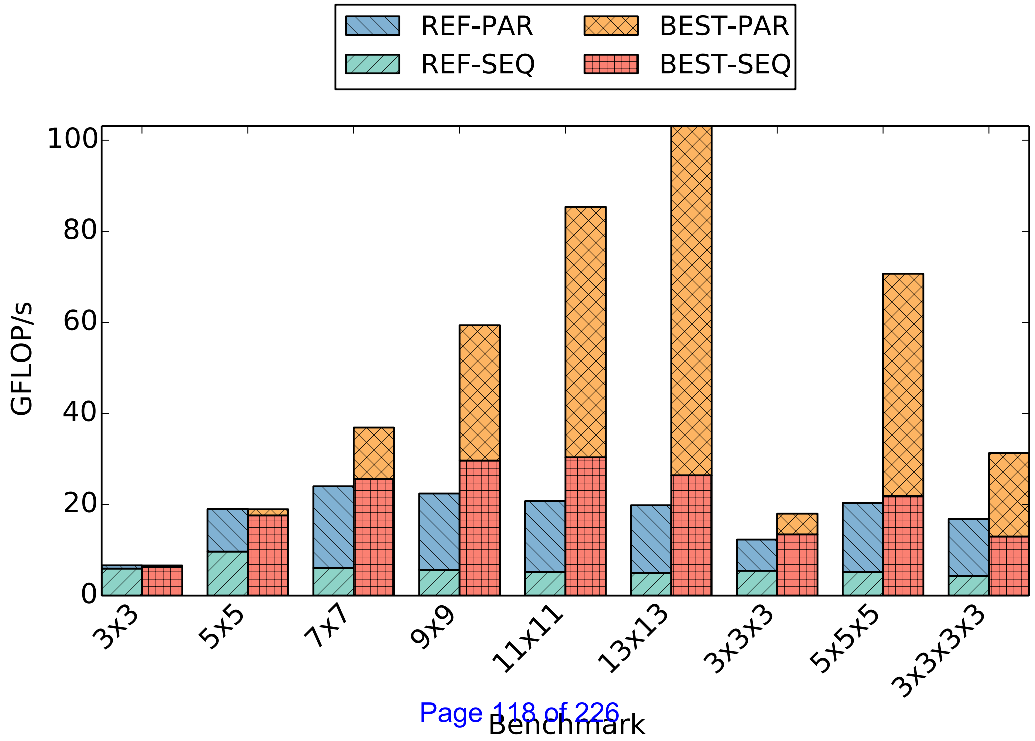
(d) Transformed Layout

Gradient Edge Detection (2d, 97-point)

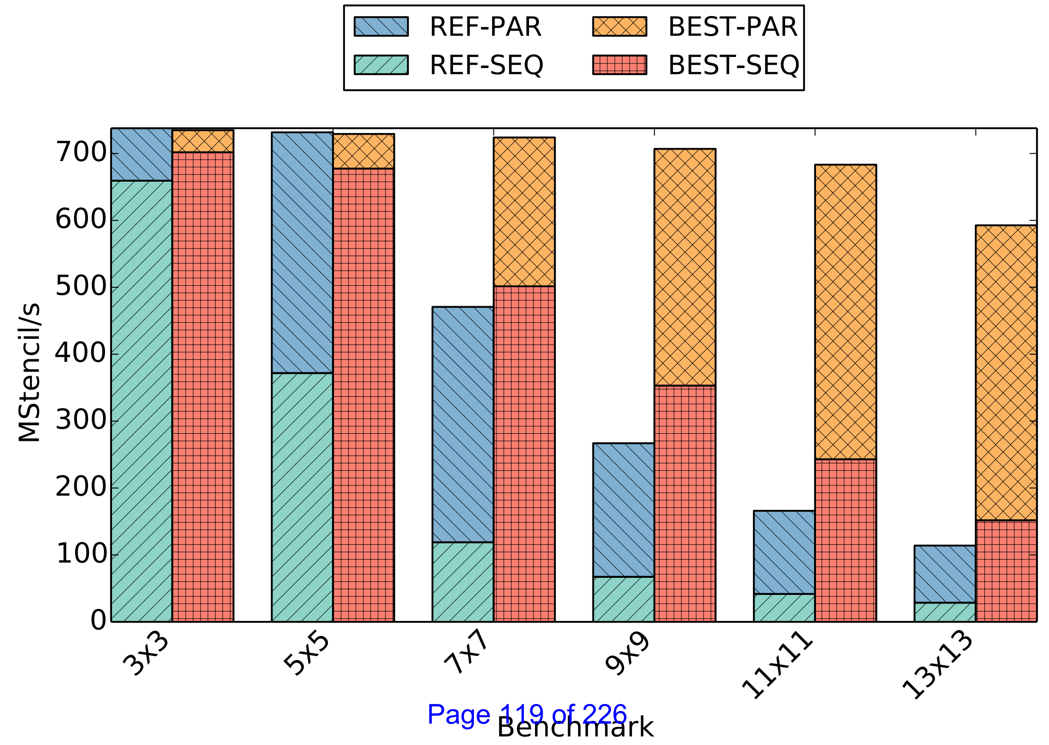


i7-4770K, ICC 13.1.3

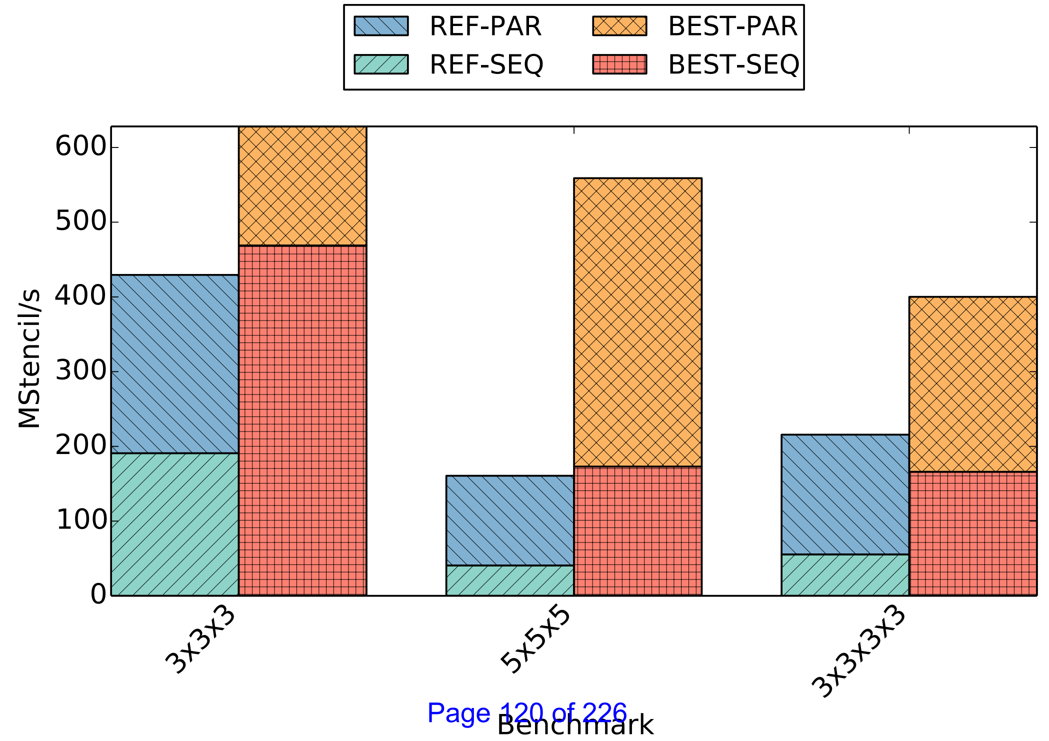
Synthetic Benchmarks Performance



Synthetic Benchmarks Rate (2d)



Synthetic Benchmarks Rate (3d & 4d)



Stencil Micro-Benchmarks

Ibiglaplace 2D, 97-point stencil for gradient edge detection

Inoise3 2D, 49-point stencil for noise cleaning

Drprj3 3D, 19-point stencil from NAS MG Benchmark

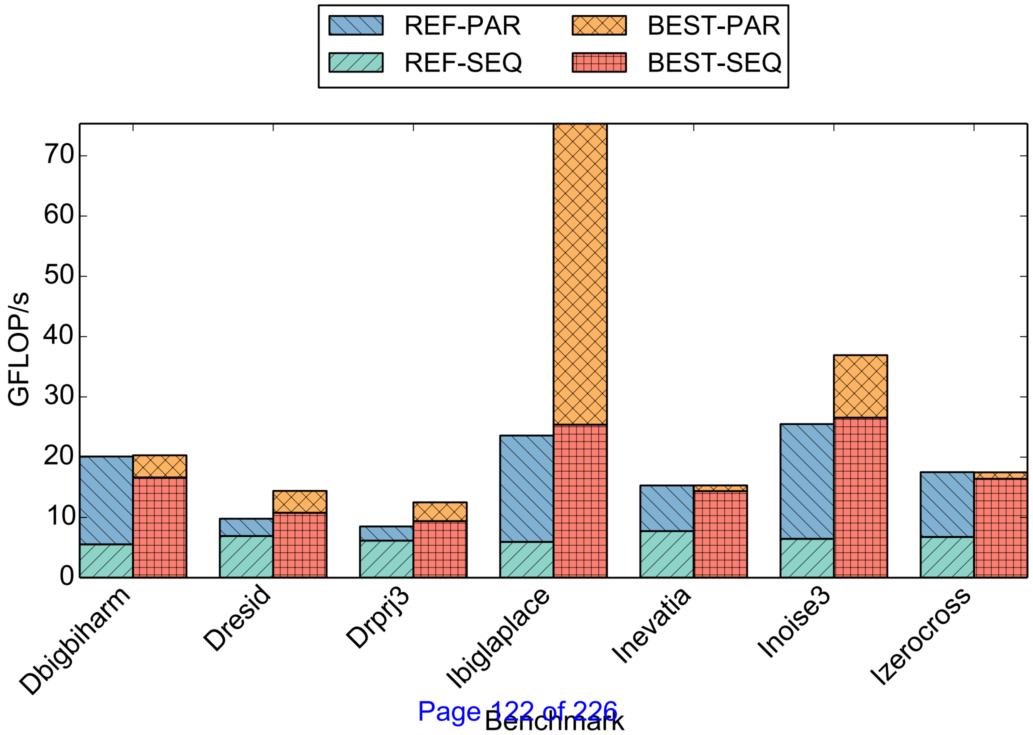
Dresid 3D, 21-point stencil from NAS MG Benchmark

Izerocross 2D, 25-point stencil for edge detection

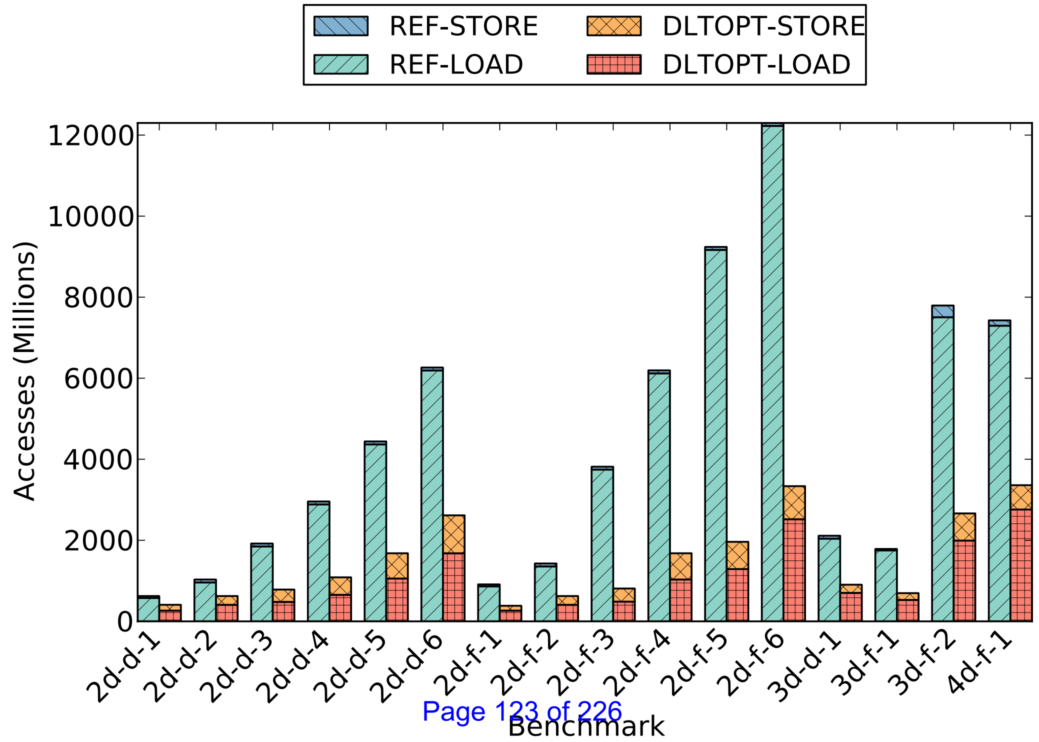
Dbigbiharm 2D, 25-point stencil for biharmonic operator

Inevatia 2D, 20-point stencil for gradient edge detection

Stencil Micro-Benchmarks

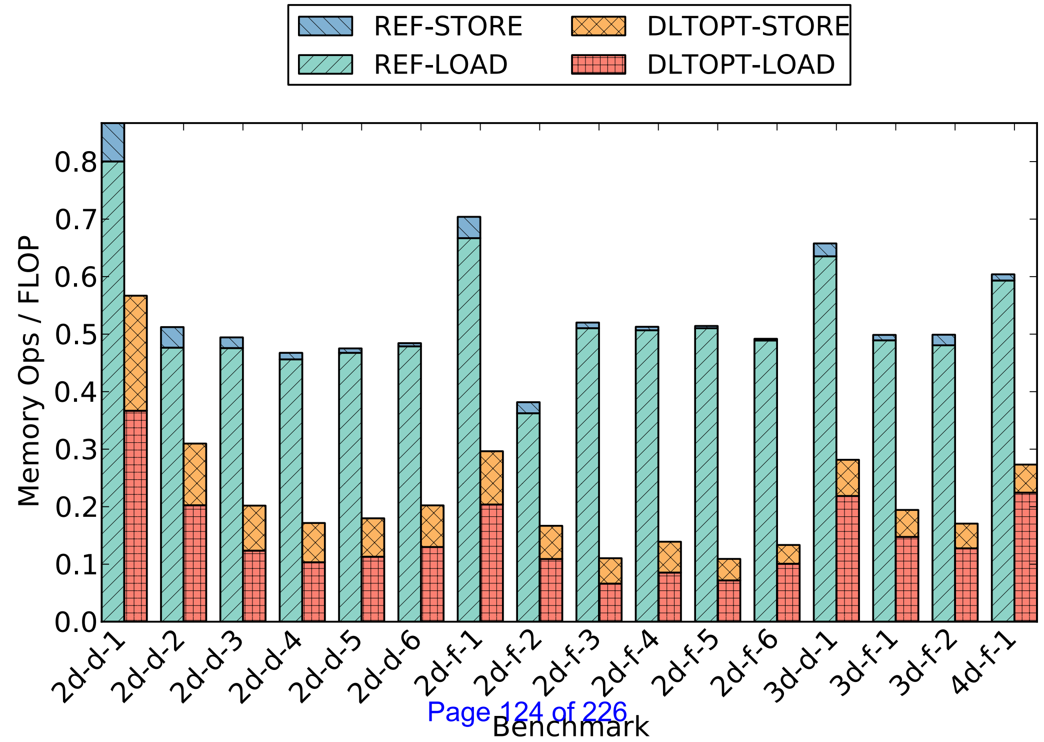


Memory Accesses



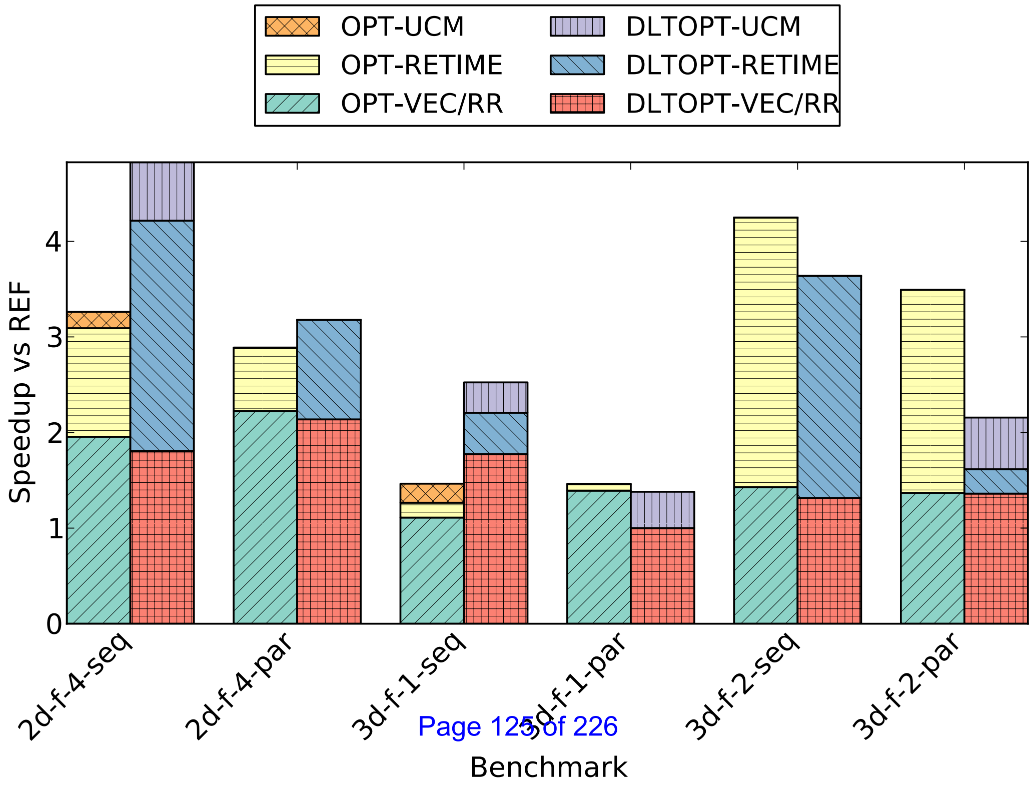
Page 123 of 226

Memory Ops per FLOP



Page 124 of 226

Impact of Transformations



Conclusion

1. High order stencils had low performance
    - Unable to *reuse registers*

Conclusion

1. High order stencils had low performance
   - Unable to *reuse registers*
2. Solved by reordering computation
   - Exploit *associativity and commutativity*
   - Formalization and cost model from retiming

Conclusion

1. High order stencils had low performance
   - Unable to *reuse registers*
2. Solved by reordering computation
   - Exploit *associativity and commutativity*
   - Formalization and cost model from retiming

3. Stencil/s maintained in higher order stencils
   - Allows scientists to use higher order stencils efficiently

Cross-loop Optimization of Arithmetic Intensity for Finite Element Local Assembly

Fabio Luporini, F. Rathgeber, G.-T. Bercea
D.A. Ham, P.H.J. Kelly
Imperial College London
J. "Ram" Ramanujam
Louisiana State University
Ana Lucia Varbanescu
University of Amsterdam

Lyon Spring School, May 2016

Goal: fast, automated resolution of PDEs

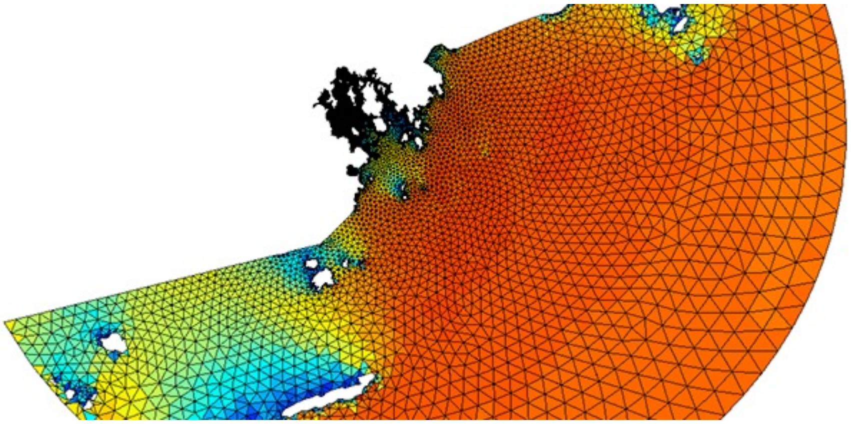


Image publicly available from http://www.bmtargoss.com/

Particularly interested in weather forecast
in a given time window (e.g., one hour)

Goal: fast, automated resolution of PDEs

Raise the level of abstraction (through domain-specific languages)

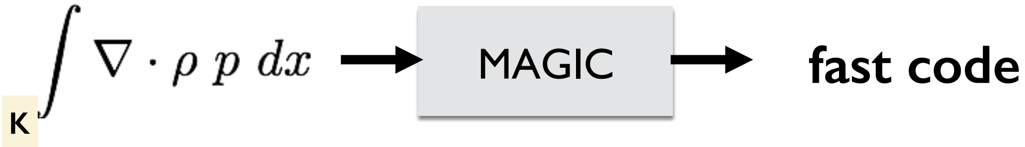
$$\int_K \nabla \cdot \rho\, p\, dx$$

+

Stack of optimizing compilers

↓

Faster code than you can reasonably write "by hand"



$\int_K \nabla \cdot \rho\, p\, dx$ → MAGIC → **fast code**

This part of the talk

- $\int \nabla \cdot \rho\, p\, dx$ → MAGIC from DSL for PDEs to loop chains
- MAGIC → fast code Tiling for unstructured meshes
- MAGIC → fast code COFFEE: expression compiler

THIS PART's MESSAGE (philosophy):

- Getting the abstraction right is key in designging and implementing the MAGIC
- The MAGIC enables automatic powerful cross-loop optimization, which means faster code than you can get when writing it by hand and "having faith" in your favorite compiler

From DSL to loop chains

Firedrake provides a DSL for finite element methods

```
phi, p = Function(mesh, …)
…
while not convergence:
{
  …
  phi -= dt / 2 * p
  if …:
    p += (assemble(dt*inner(nabla_grad(v),…))*dx)
  else:
    solve(…)
  …
  phi += dt / 2 * p
  …
}
…
```

Loop over the mesh!

Loop over the mesh!

Call to third party library!

Loop over the mesh!

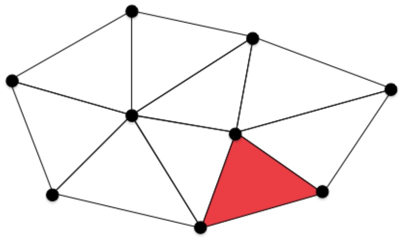
The resulting non-affine parallel-loops chain

```
while not convergence:
{
  forall cells
    …
    for i
      for j
        … expr(i, j)
    A[C[i]] = …

  forall edges
    A[E[i]] = …
    …

  function call !

  forall cells
    …
}
```



Dependencies through indirect memory accesses (C and E not known at compile time): break many compiler optimizations.

Computing expr can be so expensive, depending on the equation being solved, that the loop becomes compute-bound.

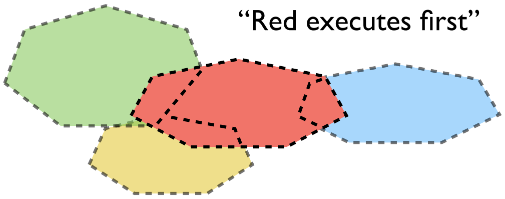
Towards tiling non-affine loops

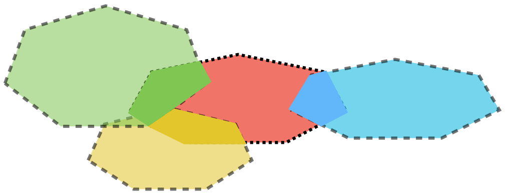
```
while not convergence:
{
  forall cells
    …
    for i
      for j
        … expr(i, j)
    A[C[i]] = …

  forall edges
    A[E[i]] = …
    …

  function call !

  forall cells
    …
}
```

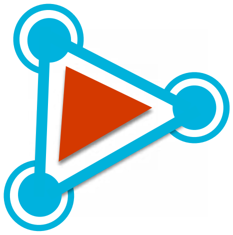




Generalized sparse tiling example

Par loop 1:

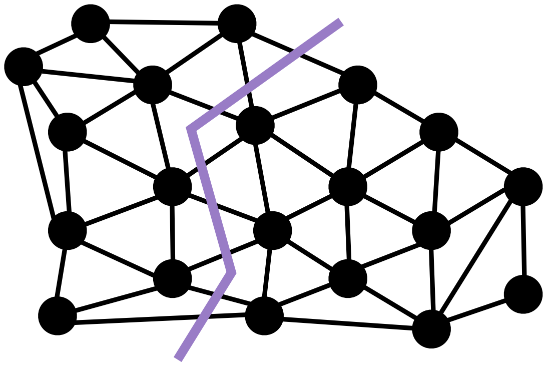
```
forall edges
  read local data
  increment adjacent vertices
```



Par loop 2:

```
forall cells
  read adjacent vertices
  write local data
```

Generalized sparse tiling example



```
forall edges
  read local data
  increment adjacent vertices
```

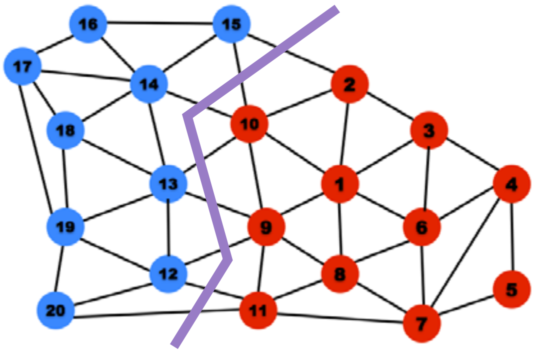
Seed (shared) set partitioning

```
forall cells
  read adjacent vertices
  write local data
```

1. Seed (shared) set partitioning

Partitions
(i.e. "base" tiles)
fit the cache!

Generalized sparse tiling example



```
forall edges
  read local data
  increment adjacent vertices
```

Seed (shared) set partitioning

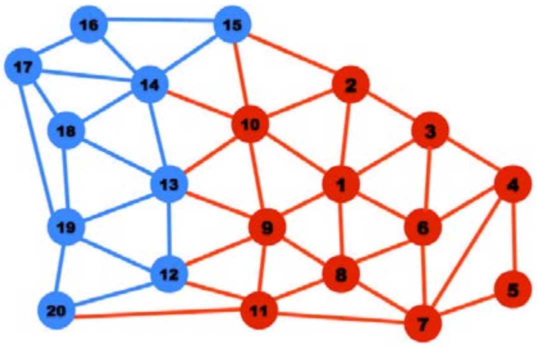
```
forall cells
  read adjacent vertices
  write local data
```

0. RED, 1 BLUE

1. Seed (shared) set partitioning **and coloring**
Lower color (number) => Higher scheduling priority

Property after executing the red edges:
all red vertices are updated, while blue ones are ***not***

Generalized sparse tiling example



```
forall edges
  read local data
  increment adjacent vertices
```

Seed (shared) set partitioning

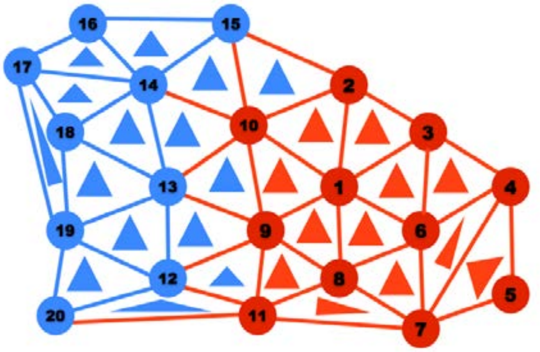
```
forall cells
  read adjacent vertices
  write local data
```

0. RED, 1 BLUE

1. Seed (shared) set partitioning and coloring
Lower number => Higher scheduling priority

2. First loop over edges, data-flow analysis:
assign MIN color over adjacent vertices => Property

Generalized sparse tiling example



```
forall edges
  read local data
  increment adjacent
vertices
```

Seed (shared) set partitioning

```
forall cells
  read adjacent vertices
  write local data
```

0. RED, 1 BLUE

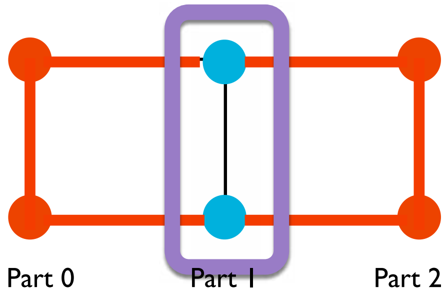
1. Seed (shared) set partitioning and coloring
Lower number => Higher scheduling priority

2. First loop over edges, data-flow analysis:
assign MIN color over adjacent vertices => Property

3. Second loop over cells, data-flow analysis:
Property => assign MAX color over adjacent vertices

Parallel execution: the coloring problem

The longer the loop chain, the larger the tile expansion



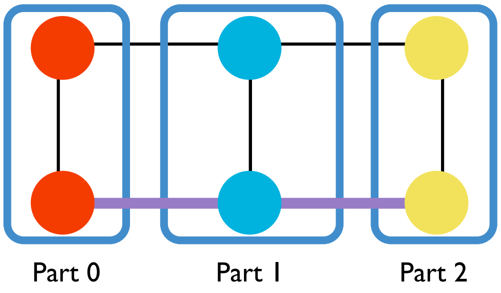
```
forall edges
```

0. RED, 1 BLUE

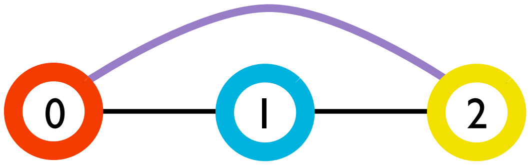
Race conditions are now possible!

Parallel execution: the coloring problem

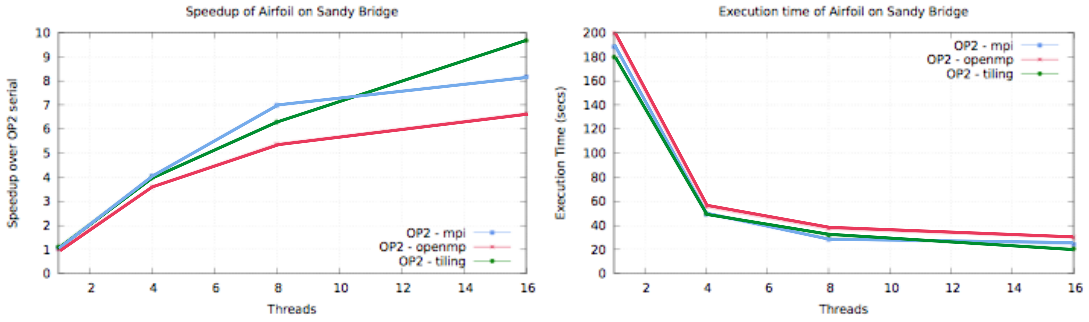
The longer the loop chain, the larger the tile expansion



Solution: Color the k-distant mesh instead (K = 2 here)

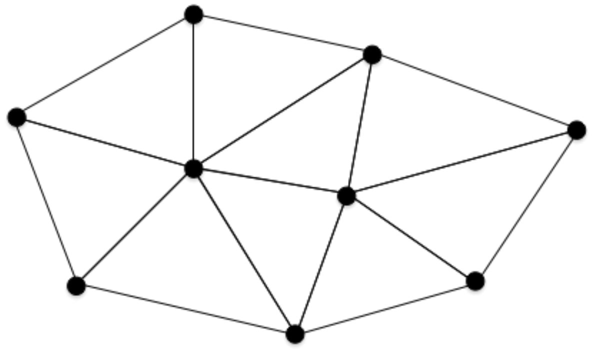


Performance evaluation - Airfoil



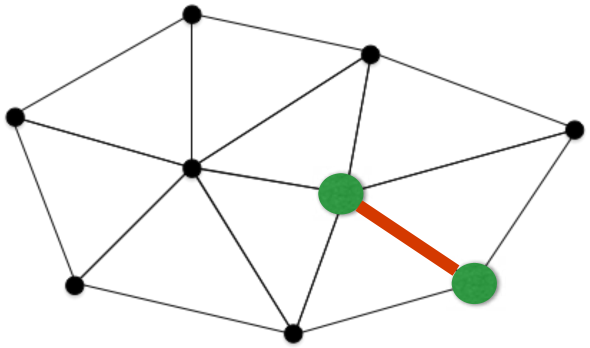
- Problem:
  - Semi-structured mesh, ~700000 quadrilateral cells
  - ~1.11x over MPI (no NUMA issue!), including inspector cost
  - Time stepping loop unrolled, 6 loops tiled
- Setup:
  - Intel Sandy Bridge (dual-socket 8-core Xeon E5-2680)
  - Intel compiler 13, -xAVX, -O3, -xHost

Unstructured meshes used for discretization



- To discretize a PDE's domain
- "Unstructured" implies the mesh connectivity can be practically expressed only through a graph abstraction (unlike structured stencils) or arrays of indices (e.g., `A[B[i]]`)
- Same program applied to different meshes, so the mesh (connectivity) is known only at run-time.

The right abstraction simplifies the analysis!



```
void incrVertices (
   double* e,
   double* v1,
   double* v2)
{
   *v1 += *e;
   *v2 += *e;
}
```

```
op_par_loop (incrVertices, edges
   op_arg_dat (edgesDat, -1, OP_ID,           OP_READ),
   op_arg_dat (vertexDat, 0, edges2vertices, OP_INC),
   op_arg_dat (vertexDat, 1, edges2vertices, OP_INC));
```

Optimizing arithmetic intensity in FEM assembly

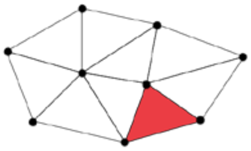
```
while not convergence:
{
  forall cells
    …
    for i
      for j
        … expr(i, j)
    A[C[i]] = …

  forall edges
    A[E[i]] = …
    …

  function call !

  forall cells
    …
}
```

- FEM execution time ~ assembly + solver (fun call)
- The numerical evaluation of integrals based on quadrature!



$$A^K_{ij} = \int_K w \nabla\phi^K_i \cdot \nabla\phi^K_j \,\mathrm{d}x$$

- Context: automated code generation for generic assembly operators; that is, "we abstract from the specific equation and discretization!"

Motivating Examples - 1

Depends on discretization employed; e.g., polynomial order

m, n, o rarely greater than 30
typically between 3 and 15

```
…
…
for (int ip  = 0; ip < m; ++ip) {
  …
  for (int j  = 0; j < n; ++j) {
    for (int k  = 0; k < o; ++k) {
      A[j][k] += (det * W[ip] * B[ip][k] * C[ip][j]);
    }
  }
}
…
```

Mass matrix operator

Motivating Examples - 2

m, n, o rarely greater than 30
typically between 3 and 15

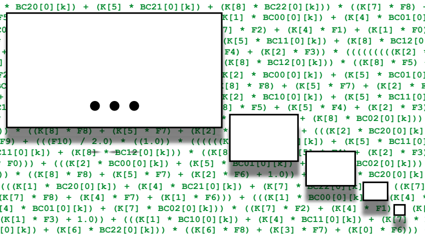
```
…
…
for (int ip  = 0; ip < m; ++ip) {
  …
  for (int j  = 0; j < n; ++j) {
    for (int k  = 0; k < o; ++k) {
      A[j][k] += (((B[ip][k] * B[ip][j]) + ((((K[2] * B0[ip]
[k]) + (K[5] * B1[ip][k]) + (K[8] * B2[ip][k])) * ((K[2] *
B0[ip][j]) + (K[5] * B1[ip][j]) + (K[8] * B2[ip][j]))) +
(((K[1] * B0[ip][k]) + (K[4] * B1[ip][k]) + (K[7] * B2[ip]
[k])) * ((K[1] * B0[ip][j]) + (K[4] * B1[ip][j]) + (K[7] *
B2[ip][j]))) + (((K[0] * B0[ip][k]) + (K[3] * B1[ip][k]) +
(K[6] * B2[ip][k])) * ((K[0] * B0[ip][j]) + (K[3] * B1[ip][j])
+ (K[6] * B2[ip][j])))) * F1 * F0)) * det * W[ip]);
    }
  }
}
…
```

Helmholtz operator

Motivating Examples - 3

m, n, o rarely greater than 30
typically between 3 and 15

```
…
for (int ip  = 0; ip < m; ++ip) {
  …
  for (int j  = 0; j < n; ++j) {
    for (int k  = 0; k < o; ++k) {
      A[j][k] += ((((K[2] * BC10[0][j]) + (K[5] * BC11[0][j]) + (K[8] * BC12[0][j])) * (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * ((((((K[8] * F2)
+ (K[5] * F1) + (K[2] * F0)) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[7] * F8) + (K[4] * F7) + (K[1] * F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[8] * F5) +
(K[5] * F4) + (K[2] * F3)) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0)) + ((((((K[8] * F2) + (K[5] * F1) + (K[2] * F0)) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) +
(((K[7] * F8) + (K[4] * F7) + (K[1] * F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[8] * F5) + (K[5] * F4) + (K[2] * F3)) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) +
1.0))) / 2.0))) * F9) + (((K[6] * F5) + (K[3] * F4) + (K[0] * F3)) * ((((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[6] * F8) + (K[3] * F7) + (K[0] * F6))) +
(((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) *
((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[0] * BC00[0][k]) + (K[3] * BC01[0][k]) + (K[6] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[0] * BC20[0][k]) +
(K[3] * BC21[0][k]) + (K[6] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[6] * F2) + (K[3] *
F1) + (K[0] * F0) + 1.0))) / 2.0)) + ((((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[6] * F8) + (K[3] * F7) + (K[0] * F6))) + (((K[0] * BC10[0][k]) + (K[3] *
BC11[0][k]) + (K[6] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[6] * F5) + (K[3] * F4) + (K[0] *
F3))) + (((K[0] * BC00[0][k]) + (K[3] * BC01[0][k]) + (K[6] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[0] * BC20[0][k]) + (K[3] * BC21[0][k]) + (K[6] * BC22[0]
[k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[6] * F2) + (K[3] * F1) + (K[0] * F0) + 1.0))) / 2.0)))
* F9) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((((((K[8] * F5) + (K[5] * F4) + (K[2] * F3)) * ((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[6] * F8) +
(K[3] * F7) + (K[0] * F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[8] * F2) + (K[5] * F1) + (K[2] * F0)) * ((K[6] * F2) + (K[3] * F1) + (K[0] * F0) + 1.0))) / 2.0)) +
(((((K[8] * F5) + (K[5] * F4) + (K[2] * F3)) * ((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[6] * F8) + (K[3] * F7) + (K[0] * F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)
+ (((K[8] * F2) + (K[5] * F1) + (K[2] * F0)) * ((K[6] * F2) + (K[3] * F1) + (K[0] * F0) + 1.0))) / 2.0))) * F9) + (((((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) *
((K[7] * F8) + (K[4] * F7) + (K[1] * F6))) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[1] * BC00[0][k]) +
(K[4] * BC01[0][k]) + (K[7] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[7] * F2) + (K[4] * F1) +
(K[1] * F0))) + (((K[1] * BC20[0][k]) + (K[4] * BC21[0][k]) + (K[7] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8]
* BC12[0][k])) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0) + (((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[7] * F8) + (K[4] * F7) + (K[1] *
F6))) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * ((K[8] * F5                      K[1] * BC00[0][k]) + (K[4] * BC01[0][k]) + (K[7] * BC02[0]
[k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC00[0][k]) + (K[5] * BC0                      7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[1] * BC20[0]
[k]) + (K[4] * BC21[0][k]) + (K[7] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] *                      (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[7] * F5) +
(K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0))) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) +                      F4) + (K[2] * F3)) * ((((((((K[2] * BC10[0][k]) + (K[5] *
BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2                      (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] *
F3))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] * F2                      K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0]
[k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC20[0][k]) + (K[5] * BC                       [8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] *
BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) +                       K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0]
[k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] * BC1                      8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC00[0]
[k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] *                           + (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[5]
* F1) + (K[2] * F0))) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] *        + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k])
+ (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0))) / 2.0))) * F9) + (((F10) / 2.0) * ((1.0)) * ((((((K        [k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0]
[k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8       + (K[2] * F3))) + (((K[2] * BC00[0]
[k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]      BC02[0][k])) * ((K[8] * F2) + (K[5]
* F1) + (K[2] * F0))) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0))      * BC20[0][k]) + (K[5] * BC21[0][k])
+ (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0))) / 2.0) + ((((K[1] * BC20[0][k]) + (K[4] * BC21[0][k]) + (K[7] *      ((K[7] * F8) + (K[4] * F7) +
(K[1] * F6))) + (((K[1] * BC20[0][k]) + (K[4] * BC21[0][k]) + (K[7] * BC22[0][k])) * ((K[7] * F8) + (K[4] * F7) + (K[1] * F6))) + (((K[1] * BC00[0][k      (K[4] * BC01[0][k]) + (K[7] *
BC02[0][k])) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[1] * BC00[0][k]) + (K[4] * BC01[0][k]) + (K[7] * BC02[0][k])) * ((K[7] * F2) + (K[4] * F1)   (K[1] * F0))) + (((K[1] *
BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) + 1.0)) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * ((K[7] *
F5) + (K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0) + ((((K[0] * BC20[0][k]) + (K[3] * BC21[0][k]) + (K[6] * BC22[0][k])) * ((K[6] * F8) + (K[3] * F7) + (K[0] * F6))) + (((K[0] * BC20[0][k])
+ (K[3] * BC21[0][k]) + (K[6] * BC22[0][k])) * ((K[6] * F8) + (K[3] * F7) + (K[0] * F6))) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((K[6] * F5) + (K[3] * F4)
+ (K[0] * F3))) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((K[6] * F5) ….
    }
  }
}
```



Hyperelasticity operator

What should we do with such expressions?

```
for (int ip  = 0; ip < m; ++ip) {
  …
  for (int j  = 0; j < n; ++j) {
    for (int k  = 0; k < o; ++k) {
      A[j][k] += (((B[ip][k] * B[ip][j]) + ((((K[2] * B0[ip][k]) + (K[5] * B1[ip]
[k]) + (K[8] * B2[ip][k])) * ((K[2] * B0[ip][j]) + (K[5] * B1[ip][j]) + (K[8] *
B2[ip][j]))) + (((K[1] * B0[ip][k]) + (K[4] * B1[ip][k]) + (K[7] * B2[ip][k])) *
((K[1] * B0[ip][j]) + (K[4] * B1[ip][j]) + (K[7] * B2[ip][j]))) + (((K[0] * B0[ip]
[k]) + (K[3] * B1[ip][k]) + (K[6] * B2[ip][k])) * ((K[0] * B0[ip][j]) + (K[3] *
B1[ip][j]) + (K[6] * B2[ip][j])))) * F1 * F0)) * det * W[ip]);
    }
  }
}
```

Key questions we address:

- Common sub-expressions
- Loop-invariants
- Re-association and factorization
- Vectorization

What can a compiler do for us?

Need to be tackled jointly, not individually

Optimizing for FLOPs

```
for i
 for j
  for k
   A[j][k] += B[i][j] * C[i][k] + (E[i][j]*β + F[i][j]*γ)
+
               (B[i][j] * D[i][k])*α
```

Innermost-loop invariant

```
for i
 for j
  for k
   A[j][k] += B[i][j] * C[i][k] + (E[i][j]*β + F[i][j]*γ)
+
               (B[i][j] * D[i][k])*α
```

Optimizing for FLOPs

OK, compilers do this easily…

```
for i
 for j
  tmp = (E[i][j]*β + F[i][j]*γ)
  for k
   A[j][k] += B[i][j] * C[i][k] + tmp +
               (B[i][j] * D[i][k])*α
```

… but need promotion for vectorization!
Important because of **small loops** and presence of **tens/hundreds of invariant sub-expressions**

```
for i
 for j
  TMP[j] = (E[i][j]*β + F[i][j]*γ)
 for j
  for k
   A[j][k] += B[i][j] * C[i][k] + TMP[j] +
               (B[i][j] * D[i][k])*α
```

Optimizing for FLOPs

```
for i
 for j
  TMP[j] = (E[i][j]*β + F[i][j]*γ)
 for j
  for k
   A[j][k] += B[i][j] * C[i][k] + TMP[j] +
              (B[i][j] * D[i][k])*α
```

```
for i
 for j
  TMP[j] = (E[i][j]*β + F[i][j]*γ)
 for j
  for k
   A[j][k] += B[i][j] * C[i][k] + TMP[j] +
              B[i][j] * (D[i][k]*α)
```

Optimizing for FLOPs

```
for i
 for j
  TMP[j] = (E[i][j]*β + F[i][j]*γ)
 for j
  for k
   A[j][k] += B[i][j] * (C[i][k] + D[i][k]*α) + TMP[j]
```

Outer-loop invariant: no way your compiler thinks “globally”

```
for i
 for j
  TMP[j] = (E[i][j]*β + F[i][j]*γ)
 for k
  TMP2[k] = (C[i][k] + D[i][k]*α)
 for j
  for k
   A[j][k] += B[i][j] * TMP2[k] + TMP[j]
```

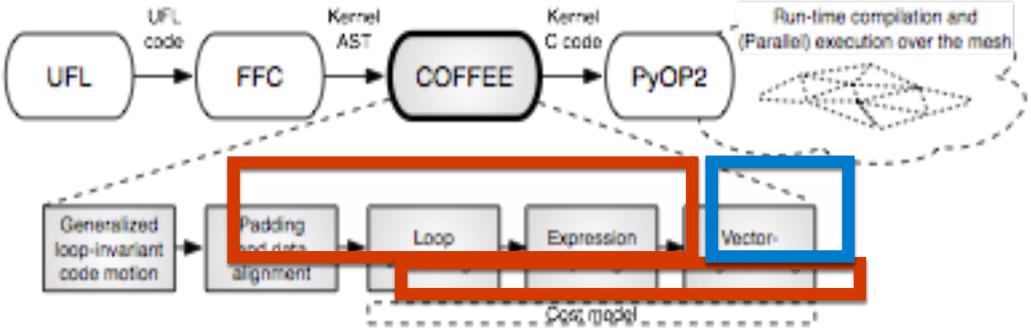
The COFFEE Project

- Embedded and actually used in Firedrake master!
- Could be integrated with FEniCS, because both framework use the same DSL compiler
- Therefore, potentially, a user space of ~1000 scientists!
- Of course, a lot still has to be done
- Source code is >5000 lines of Python code, and is becoming finite element independent

A COmpiler For Fast Expression Evaluation

Any partial differential equation expressible in Firedrake
A broad range of differential operators are supported

Many discretizations are supported (all affecting code generation), e.g., element type, polynomial order, etc.


UFL
UFL code
FFC
Kernel AST
COFFEE
Kernel C code
PyOP2
Run-time compilation and (Parallel) execution over the mesh
Generalized loop-invariant code motion
Padding and data alignment
Loop
Expression
Vector-
Cost model

Optimizing for ILP - register reuse

```
for i
 … hoisted stuff …
 for j
  for k
   A[j][k] += B[i][j] * TMP2[k] + TMP[j]
```

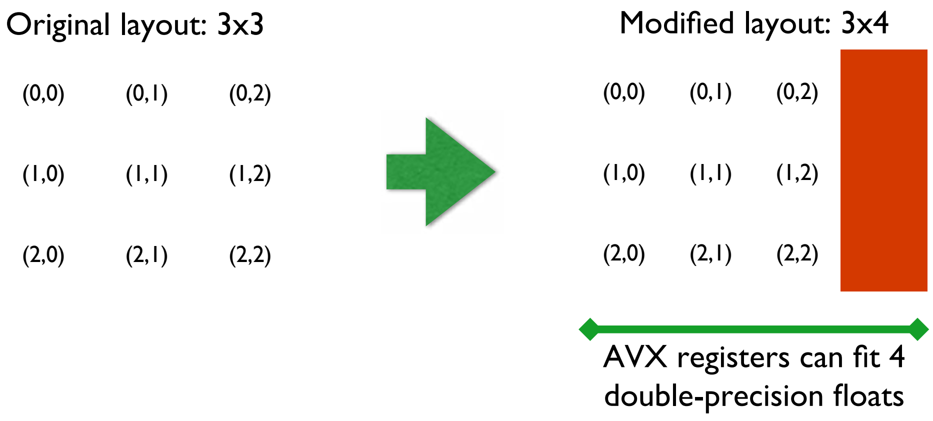
Associative operator

```
for i
 … hoisted stuff …
 for j
  for k
   A[j][k] += B[i][j] * TMP2[k]
 for j
  for k
   A[j][k] += TMP[j]
```

Expression splitting ~ loop fission for expressions, to increase register reuse when expressions are particularly complicated

Optimizing for ILP - SIMD - data alignment

Padding and data alignment for efficient SIMDization



Original layout: 3x3

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

Modified layout: 3x4

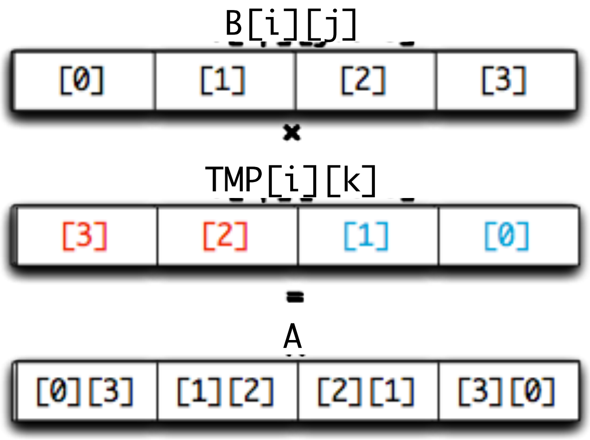
(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

AVX registers can fit 4 double-precision floats

- Ensure data alignment (efficient memory loads/stores **not** crossing cache boundaries)
- Small overhead due to restoring the storage layout

Optimizing for ILP - specialized SIMDization

```
for i = 0 < 4
  for j = 0 < 4
    for k = 0 < 4
      A[j][k] += B[i][j]*TMP[i][k]
```



B[i][j]

[0]	[1]	[2]	[3]

×

TMP[i][k]

[3]	[2]	[1]	[0]

=

A

[0][3]	[1][2]	[2][1]	[3][0]

A[4:4]

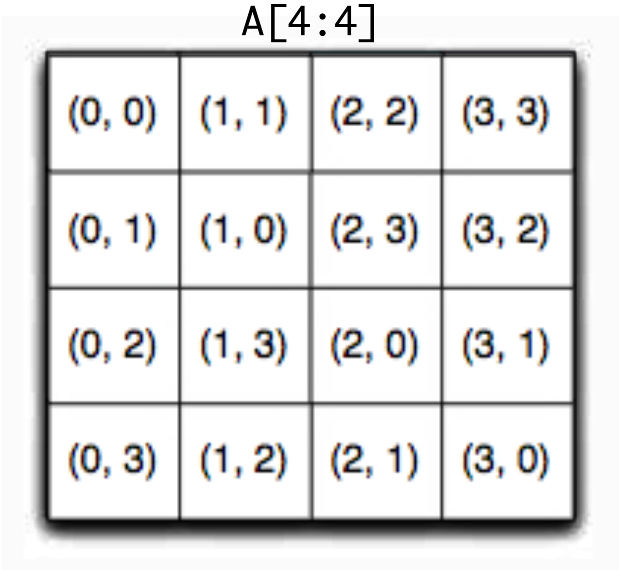
[0][0]	[1][1]	[2][2]	[3][3]
[0][1]	[1][0]	[2][3]	[3][2]
[0][2]	[1][3]	[2][0]	[3][1]
[0][3]	[1][2]	[2][1]	[3][0]

TOT = **2** mem loads

Optimizing for ILP - specialized SIMDization

Storage layout can be restored with a few vector shuffles

A[4:4]

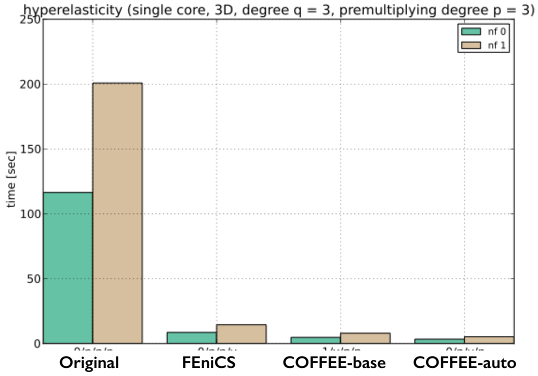


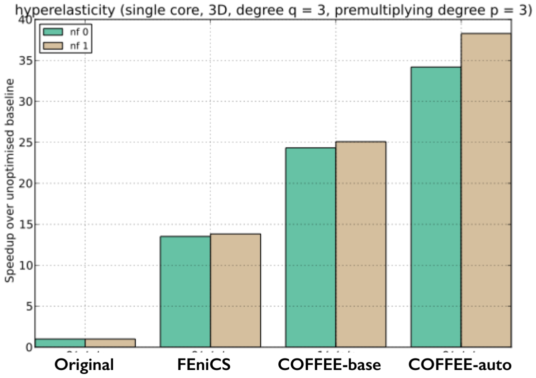
(0, 0)	(1, 1)	(2, 2)	(3, 3)
(0, 1)	(1, 0)	(2, 3)	(3, 2)
(0, 2)	(1, 3)	(2, 0)	(3, 1)
(0, 3)	(1, 2)	(2, 1)	(3, 0)

_mm256_unpackhi_pd
_mm256_unpackhi_pd
_mm256_unpacklo_pd
_mm256_unpacklo_pd

_mm256_permute2f128_pd
_mm256_permute2f128_pd
_mm256_permute2f128_pd
_mm256_permute2f128_pd

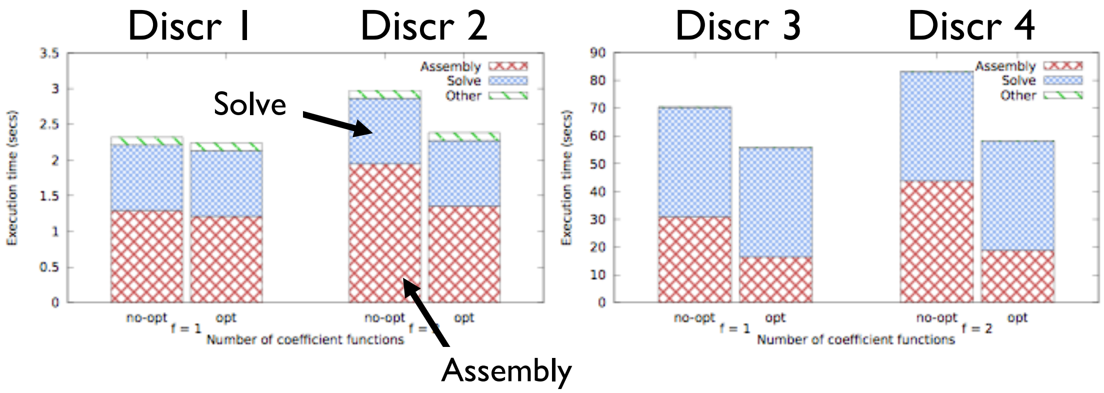
Assembly only performance evaluation





- Problem:
  - hyperelasticity, with 0 and 1 coefficient functions
  - polynomial order 3
  - mesh: small enough to fit the L2 cache of the architecture
  - Original, FEniCS-optimized, COFFEE-optimized, COFFEE-autotuned
- Setup:
  - Single core of an Intel Sandy Bridge (I7-2600 CPU @ 3.40GHz)
  - Intel compiler (version 14.1, -O3, -xAVX, -ip, -xHost)

Full application performance evaluation



- Problem:
  - linear elasticity with *f=1* and *f*=2 coefficient functions
  - polynomial order 1 (left fig) and 2 (right fig)
  - mesh: tetrahedral, 196608 elements (CG family)
  - max **application** speedup: 1.47x (but grows with complexity of equation!)
- Setup:
  - Single core of an Intel Sandy Bridge (I7-2600 CPU @ 3.40GHz)
  - Intel compiler (version 13.1, -O3, -xAVX, -ip, -xHost)

Summary

- What I've shown you is implemented.
  - COFFEE is used by Firedrake
    - automatically does the expression manipulation discussed
    - plus other "more domain-specific" stuff!
- Combining domain-specific and technology knowledge allows you to deliver optimizations more powerful than you can write by hand.
- Where are we going now?
  - Different discretizations => different loop nests
  - …

Automatic Synthesis of High-Performance Codes for Quantum Chemistry using the Tensor Contraction Engine (TCE)

Thanks to Collaborators

- **Louisiana State University**: G. Baumgartner, A. Allam, A. Panyala, H. Salamy, P. Bhattacharya
- **Ohio State University**: P. Sadayappan, D. Cociorva, C. Lam, R. Pitzer, A. Bibireata, X. Gao, S. Krishnan, A. Sibiryakov, L.-N. Pouchet, A. Rountev, and others
- **Pacific Northwest Labs**: S. Krishnamoorthy, J. Nieplocha
- **Oak Ridge National Labs**: R. Harrison, D. Bernholdt, V. Choppella
- **University of Waterloo**: M. Nooijen
- **University of Illinois**: S. Hirata
- **IISc**: U. Bondhugula
- **Reservoir Labs**: M. Baskaran
- **Intel**: Q. Lu, A. Hartono

Domain-Specific Optimizations

- Heterogeneity creates a software challenge
  - Multiple implementations for different system components, e.g. OpenMP (multicore), OpenCL (GPU), VHDL (FPGA)
- How can we **Write-Once-Execute-Anywhere**?

Domain-Specific Optimizations

- Heterogeneity creates a software challenge
  - Multiple implementations for different system components, e.g. OpenMP (multicore), OpenCL (GPU), VHDL (FPGA)
- How can we ~~**Write-Once-Execute-Anywhere**~~?

Domain-Specific Optimizations

- Heterogeneity creates a software challenge
  - Multiple implementations for different system components, e.g. OpenMP (multicore), OpenACC/OpenCL (GPU), VHDL (FPGA)
- How can we **Write-Once-Execute-Well-Anywhere**?
  - Too daunting a challenge for general-purpose languages
  - More promising for domain-specific approaches
- Examples of domain-specific computational abstractions
  - Tensor expressions
  - Affine computations (stencils, …)

Problem Domain: High-Accuracy Quantum Chemical Methods

- Coupled cluster methods are widely used for very high quality electronic structure calculations
- Typical Laplace factorized CCSD(T) term:

$$A3A = \tfrac{1}{2}\big(X_{ce,af}Y_{ae,cf} + X_{c\bar{e},a\bar{f}}Y_{a\bar{e},c\bar{f}} + X_{c\bar{e},\bar{a}f}Y_{\bar{a}\bar{e},cf}$$

$$+ X_{\bar{c}e,a\bar{f}}Y_{ae,\bar{c}\bar{f}} + X_{\bar{c}e,\bar{a}f}Y_{\bar{a}e,\bar{c}f} + X_{\bar{c}\bar{e},\bar{a}\bar{f}}Y_{\bar{a}\bar{e},\bar{c}\bar{f}}\big)$$

$$X_{ce,af} = t_{ij}^{ce}t_{ij}^{af} \qquad Y_{ae,cf} = \langle ab\|ek\rangle\langle cb\|fk\rangle$$

Typical methods will have tens to hundreds of such terms

- Indices i, j, k : *O* (O=100) values, a, b, c, e, f : *V* (V=3000)
- Term costs *O*(OV^5) ≈ 10^{19} FLOPs; Integrals ~ 1000 FLOPs each
- *O*(V^4) terms ~ 500 TB memory each

Time Crunch in Quantum Chemistry

Two major bottlenecks in computational chemistry

- Highly computationally intensive models
- Extremely time consuming to develop codes

Time Crunch in Quantum Chemistry

Two major bottlenecks in computational chemistry

- Highly computationally intensive models
- Extremely time consuming to develop codes

The vicious cycle of computational science

- More powerful computers make more accurate models computationally feasible :-)
- Efficient parallel implementation of complex models takes longer and longer
- Hence computational scientists spend more time with MPI programming, and less time doing science :-(

Time Crunch in Quantum Chemistry

Two major bottlenecks in computational chemistry

- Highly computationally intensive models
- Extremely time consuming to develop codes

The vicious cycle of computational science

- More powerful computers make more accurate models computationally feasible :-)
- But efficient parallel implementation of complex models takes longer and longer
- Hence computational scientists spend more time with MPI programming, and less time doing science :-(

- Coupled Cluster family of models in electronic structure theory
- Increasing number of terms => explosive increase in code complexity
- Theory well known for decades but efficient implementations took many years

Theory	#Terms	#F77Lines	Year
CCD	11	3209	1978
CCSD	48	13213	1982
CCSDT	102	33932	1988
CCSDTQ	183	79901	1992

Problems

Complexity of methods

- Implementation takes months
- Experimentation required to develop new methods

Our Solution

Tensor Contraction Engine

- Tensor contraction expressions as input
- (Fortran) source code as output

Generated code increases productivity

Problems

Complexity of methods

- Implementation takes months
- Experimentation required to develop new methods

Complexity of computers

- Different architectures have significantly different performance characteristics

Our Solution

Tensor Contraction Engine

- Tensor contraction expressions as input
- (Fortran) source code as output

Generated code increases productivity

Generate optimized code for target/Optimize generated code for target

Problems

Complexity of methods

- Implementation takes months
- Experimentation required to develop new methods

Complexity of computers

- Different architectures have significantly different performance characteristics

Our Solution

Tensor Contraction Engine

- Tensor contraction expressions as input
- (Fortran) source code as output

Generated code increases productivity

Generate optimized code for target/Optimize generated code for target

What's Novel?

Code generation merely for productivity, historically

- Imitate what a researcher would do – but quicker

We treat as a computer science problem

- Like a compiler
- Algorithmic choices explored rigorously and exhaustively

The Tensor Contraction Engine (TCE)

- User describes computational problem (tensor contractions, a la many-body methods) in a simple, high-level language
  - Similar to what might be written in papers
- Compiler-like tools translate high-level language into traditional Fortran (or C, or…) code
- Generated code is compiled and linked to libraries providing computational infrastructure
  - Code can be tailored to target architecture
- Two versions of TCE developed
  - Full exploitation of symmetry, but fewer optimizations (So Hirata)
  - Partial exploitation of symmetry, but more sophisticated optimizations
  - Used to implement over 20 models, included in NWChem
  - First parallel implementation for many of the methods

Addressing Programming Challenges

- **Productivity**
  - User writes simple, high-level code
  - Code generation tools do the tedious work
- **Complexity**
  - Significantly reduces complexity visible to programmer
- **Performance**
  - Perform (some important) optimizations prior to C/Fortran code generation
  - Automate many decisions humans make
  - Tailor generated code to target computer
  - Tailor generated code to specific problem

Problem: Tensor Contractions

- Formulas of the form

$$S_{abij} = \sum_{c,d,e,f,k,l} A_{acik} B_{befl} C_{dfjk} D_{cdel} + \cdots$$

- Multi-dimensional summation over products of large multi-dimensional arrays
- Tens of arrays and array indices, hundreds of terms
- Index ranges between 10 and 3000
- And this is still a simple model!

Application Domain

- Quantum chemistry, condensed matter physics
- Example: study chemical properties
- Typical program structure

```
quantum chemistry code;
while (not converged) {
        tensor contractions;
        quantum chemistry code;
}
```

- Bulk of computation in tensor contractions

High-Level Language for Tensor Contraction Expressions

```
range V = 3000;
range O = 100;

index a,b,c,d,e,f : V;
index i,j,k : O;

mlimit = 1000000000000;

function F1(V,V,V,O);
function F2(V,V,V,O);

procedure P(in T1[O,O,V,V], in T2[O,O,V,V], out X)=

begin
   X == sum[ sum[F1(a,b,e,k) * F2(c,b,f,k), {b,k}]
                 * sum[T1[i,j,c,e] * T2[i,j,a,f], {i,j}],
                 {a,e,c,f}];
end
```

$$A3A = \tfrac{1}{2}\big(X_{ce,af}Y_{ae,cf} + X_{c\bar{e},a\bar{f}}Y_{a\bar{e},c\bar{f}} + X_{\bar{c}\bar{e},af}Y_{\bar{a}\bar{e},cf} + X_{\bar{c}e,a\bar{f}}Y_{ae,\bar{c}\bar{f}} + X_{\bar{c}e,\bar{a}f}Y_{\bar{a}e,\bar{c}f} + X_{\bar{c}\bar{e},\bar{a}\bar{f}}Y_{\bar{a}\bar{e},\bar{c}\bar{f}}\big)$$

$$X_{ce,af} \boxed{= t_{ij}^{ce}t_{ij}^{af}} \qquad Y_{ae,cf} = \langle ab\|ek\rangle\langle cb\|fk\rangle$$

Tensor Contraction Expression

- Tensor:
  - multi-dimensional array

t_{ij}^{ab} → `t[a,b,i,j]`

- Tensor contraction expression:
  - multi-dimensional summation over products of large arrays

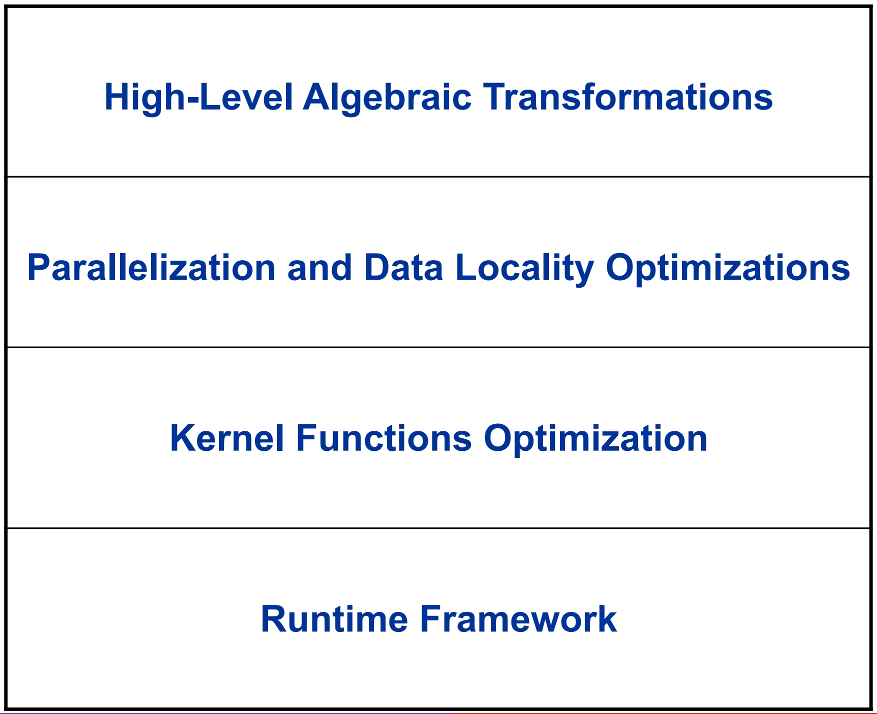
$$r_j^i = \sum_{a,b} t_b^a v_{aj}^{bi}$$ → `r[i,j]=sum[t[a,b]*v[b,i,a,j],{a,b}]`

```
for i=1 to Ni
  for j=1 to Nj
    for a=1 to Na
      for b=1 to Nb
        r[i,j] += t[a,b] * v[b,i,a,j]
```

CCSD Doubles Equation (Quantum Chemist's Eye Test Chart :-))

```
hbar[a,b,i,j] == sum[f[b,c]*t[i,j,a,c],{c}] -sum[f[k,c]*t[k,b]*t[i,j,a,c],{k,c}] +sum[f[a,c]*t[i,j,c,b],{c}] -sum[f[k,c]*t[k,a]*t[i,j,c,b],{k,c}] -
    sum[f[k,j]*t[i,k,a,b],{k}] -sum[f[k,c]*t[j,c]*t[i,k,a,b],{k,c}] -sum[f[k,i]*t[j,k,b,a],{k}] -sum[f[k,c]*t[i,c]*t[j,k,b,a],{k,c}]
    +sum[t[i,c]*t[j,d]*v[a,b,c,d],{c,d}] +sum[t[i,j,c,d]*v[a,b,c,d],{c,d}] +sum[t[j,c]*v[a,b,i,c],{c}] -sum[t[k,b]*v[a,k,i,j],{k}]
    +sum[t[i,c]*v[b,a,j,c],{c}] -sum[t[k,a]*v[b,k,j,i],{k}] -sum[t[k,d]*t[i,j,c,b]*v[k,a,c,d],{k,c,d}] -sum[t[i,c]*t[j,k,b,d]*v[k,a,c,d],{k,c,d}] -
    sum[t[j,c]*t[k,b]*v[k,a,c,i],{k,c}] +2*sum[t[j,k,b,c]*v[k,a,c,i],{k,c}] -sum[t[j,k,c,b]*v[k,a,c,i],{k,c}] -sum[t[i,c]*t[j,d]*t[k,b]*v[k,a,d,c],
    {k,c,d}] +2*sum[t[k,d]*t[i,j,c,b]*v[k,a,d,c],{k,c,d}] -sum[t[k,b]*t[i,j,c,d]*v[k,a,d,c],{k,c,d}] -sum[t[j,d]*t[i,k,c,b]*v[k,a,d,c],{k,c,d}]
    +2*sum[t[i,c]*t[j,k,b,d]*v[k,a,d,c],{k,c,d}] -sum[t[i,c]*t[j,k,d,b]*v[k,a,d,c],{k,c,d}] -sum[t[j,k,b,c]*v[k,a,i,c],{k,c}] -
    sum[t[i,c]*t[k,b]*v[k,a,j,c],{k,c}] -sum[t[i,k,c,b]*v[k,a,j,c],{k,c}] -sum[t[i,c]*t[j,d]*t[k,a]*v[k,b,c,d],{k,c,d}] -
    sum[t[k,d]*t[i,j,a,c]*v[k,b,c,d],{k,c,d}] -sum[t[k,a]*t[i,j,c,d]*v[k,b,c,d],{k,c,d}] +2*sum[t[j,d]*t[i,k,a,c]*v[k,b,c,d],{k,c,d}] -
    sum[t[j,d]*t[i,k,c,a]*v[k,b,c,d],{k,c,d}] -sum[t[i,c]*t[j,k,d,a]*v[k,b,c,d],{k,c,d}] -sum[t[i,c]*t[k,a]*v[k,b,c,j],{k,c}]
    +2*sum[t[i,k,a,c]*v[k,b,c,j],{k,c}] -sum[t[i,k,c,a]*v[k,b,c,j],{k,c}] +2*sum[t[k,d]*t[i,j,a,c]*v[k,b,d,c],{k,c,d}] -
    sum[t[j,d]*t[i,k,a,c]*v[k,b,d,c],{k,c,d}] -sum[t[j,c]*t[k,a]*v[k,b,i,c],{k,c}] -sum[t[j,k,c,a]*v[k,b,i,c],{k,c}] -sum[t[i,k,a,c]*v[k,b,j,c],{k,c}]
    +sum[t[i,c]*t[j,d]*t[k,a]*t[l,b]*v[k,l,c,d],{k,l,c,d}] -2*sum[t[k,b]*t[l,d]*t[i,j,a,c]*v[k,l,c,d],{k,l,c,d}] -2*sum[t[k,a]*t[l,d]*t[i,j,c,b]*v[k,l,c,d],
    {k,l,c,d}] +sum[t[k,a]*t[l,b]*t[i,j,c,d]*v[k,l,c,d],{k,l,c,d}] -2*sum[t[j,c]*t[l,d]*t[i,k,a,b]*v[k,l,c,d],{k,l,c,d}]
    -2*sum[t[j,d]*t[l,b]*t[i,k,a,c]*v[k,l,c,d],{k,l,c,d}] +sum[t[j,d]*t[l,b]*t[i,k,c,a]*v[k,l,c,d],{k,l,c,d}] -2*sum[t[i,c]*t[l,d]*t[j,k,b,a]*v[k,l,c,d],
    {k,l,c,d}] +sum[t[i,c]*t[l,a]*t[j,k,b,d]*v[k,l,c,d],{k,l,c,d}] +sum[t[i,c]*t[l,b]*t[j,k,d,a]*v[k,l,c,d],{k,l,c,d}]
    +sum[t[i,k,c,d]*t[j,l,b,a]*v[k,l,c,d],{k,l,c,d}] +4*sum[t[i,k,a,c]*t[j,l,b,d]*v[k,l,c,d],{k,l,c,d}] -2*sum[t[i,k,c,a]*t[j,l,b,d]*v[k,l,c,d],{k,l,c,d}]
    -2*sum[t[i,k,a,b]*t[j,l,c,d]*v[k,l,c,d],{k,l,c,d}] -2*sum[t[i,k,a,c]*t[j,l,d,b]*v[k,l,c,d],{k,l,c,d}] +sum[t[i,k,c,a]*t[j,l,d,b]*v[k,l,c,d],{k,l,c,d}]
    +sum[t[i,c]*t[j,d]*t[k,l,a,b]*v[k,l,c,d],{k,l,c,d}] +sum[t[i,j,c,d]*t[k,l,a,b]*v[k,l,c,d],{k,l,c,d}] -2*sum[t[i,j,c,b]*t[k,l,a,d]*v[k,l,c,d],{k,l,c,d}]
    -2*sum[t[i,j,a,c]*t[k,l,b,d]*v[k,l,c,d],{k,l,c,d}] +sum[t[j,c]*t[k,b]*t[l,a]*v[k,l,c,i],{k,l,c}] +sum[t[l,c]*t[j,k,b,a]*v[k,l,c,i],{k,l,c}]
    -2*sum[t[l,a]*t[j,k,b,c]*v[k,l,c,i],{k,l,c}] +sum[t[l,a]*t[j,k,c,b]*v[k,l,c,i],{k,l,c}] -2*sum[t[k,c]*t[j,l,b,a]*v[k,l,c,i],{k,l,c}]
    +sum[t[k,a]*t[j,l,b,c]*v[k,l,c,i],{k,l,c}] +sum[t[k,b]*t[j,l,c,a]*v[k,l,c,i],{k,l,c}] +sum[t[j,c]*t[l,k,a,b]*v[k,l,c,i],{k,l,c}]
    +sum[t[i,c]*t[k,a]*t[l,b]*v[k,l,c,j],{k,l,c}] +sum[t[l,c]*t[i,k,a,b]*v[k,l,c,j],{k,l,c}] -2*sum[t[l,b]*t[i,k,a,c]*v[k,l,c,j],{k,l,c}]
    +sum[t[l,b]*t[i,k,c,a]*v[k,l,c,j],{k,l,c}] +sum[t[i,c]*t[k,l,a,b]*v[k,l,c,j],{k,l,c}] +sum[t[j,c]*t[l,d]*t[i,k,a,b]*v[k,l,d,c],{k,l,c,d}]
    +sum[t[j,d]*t[l,b]*t[i,k,a,c]*v[k,l,d,c],{k,l,c,d}] +sum[t[j,d]*t[l,a]*t[i,k,c,b]*v[k,l,d,c],{k,l,c,d}] -2*sum[t[i,k,c,d]*t[j,l,b,a]*v[k,l,d,c],
    {k,l,c,d}] -2*sum[t[i,k,a,c]*t[j,l,b,d]*v[k,l,d,c],{k,l,c,d}] +sum[t[i,k,c,a]*t[j,l,b,d]*v[k,l,d,c],{k,l,c,d}] +sum[t[i,k,a,b]*t[j,l,c,d]*v[k,l,d,c],
    {k,l,c,d}] +sum[t[i,k,c,b]*t[j,l,d,a]*v[k,l,d,c],{k,l,c,d}] +sum[t[i,k,a,c]*t[j,l,d,b]*v[k,l,d,c],{k,l,c,d}] +sum[t[k,a]*t[l,b]*v[k,l,i,j],{k,l}]
    +sum[t[k,l,a,b]*v[k,l,i,j],{k,l}] +sum[t[k,b]*t[l,d]*t[i,j,a,c]*v[l,k,c,d],{k,l,c,d}] +sum[t[k,a]*t[l,d]*t[i,j,c,b]*v[l,k,c,d],{k,l,c,d}]
    +sum[t[i,c]*t[l,d]*t[j,k,b,a]*v[l,k,c,d],{k,l,c,d}] -2*sum[t[i,c]*t[l,a]*t[j,k,b,d]*v[l,k,c,d],{k,l,c,d}] +sum[t[i,c]*t[l,a]*t[j,k,d,b]*v[l,k,c,d],
    {k,l,c,d}] +sum[t[i,j,c,b]*t[k,l,a,d]*v[l,k,c,d],{k,l,c,d}] +sum[t[i,j,a,c]*t[k,l,b,d]*v[l,k,c,d],{k,l,c,d}] -2*sum[t[l,c]*t[i,k,a,b]*v[l,k,c,j],
    {k,l,c}] +sum[t[l,b]*t[i,k,a,c]*v[l,k,c,j],{k,l,c}] +sum[t[l,a]*t[i,k,c,b]*v[l,k,c,j],{k,l,c}] +v[a,b,i,j]
```

Multi-Level Optimization Framework



High-Level Algebraic Transformations

Parallelization and Data Locality Optimizations

Kernel Functions Optimization

Runtime Framework

Algebraic Transformations: Operation Minimization

$$S_{abij} = \sum_{c,d,e,f,k,l} A_{acik} B_{befl} C_{dfjk} D_{cdel}$$

- Requires $4 * N^{10}$ operations if indices a-l have range N
- Using associative, commutative, distributive laws acceptable

Algebraic Transformations: Operation Minimization

$$S_{abij} = \sum_{c,d,e,f,k,l} A_{acik} B_{befl} C_{dfjk} D_{cdel}$$

$$S_{abij} = \sum_{c,d,e,f,k,l} A_{acik} C_{dfjk} B_{b\underline{e}f\underline{l}} D_{cd\underline{e}\underline{l}}$$

Algebraic Transformations: Operation Minimization

$$S_{abij} = \sum_{c,d,e,f,k,l} A_{acik} B_{befl} C_{dfjk} D_{cdel}$$

$$S_{abij} = \sum_{c,d,e,f,k,l} A_{acik} C_{dfjk} B_{b\underline{e}f\underline{l}} D_{cd\underline{e}\underline{l}}$$

$$S_{abij} = \sum_{c,d,f,k} A_{acik} C_{dfjk} \left(\sum_{e,l} B_{befl} D_{cdel} \right)$$

$$S_{abij} = \sum_{c,k} A_{acik} \left(\sum_{d,f} C_{dfjk} \left(\sum_{e,l} B_{befl} D_{cdel} \right) \right)$$

Algebraic Transformations: Operation Minimization

$$S_{abij} = \sum_{c,d,e,f,k,l} A_{acik} B_{befl} C_{dfjk} D_{cdel}$$

- Requires $4 * N^{10}$ operations if indices a-l have range N
- Using associative, commutative, distributive laws acceptable
- **Optimal formula sequence requires only $6 * N^6$ operations (but more memory)**

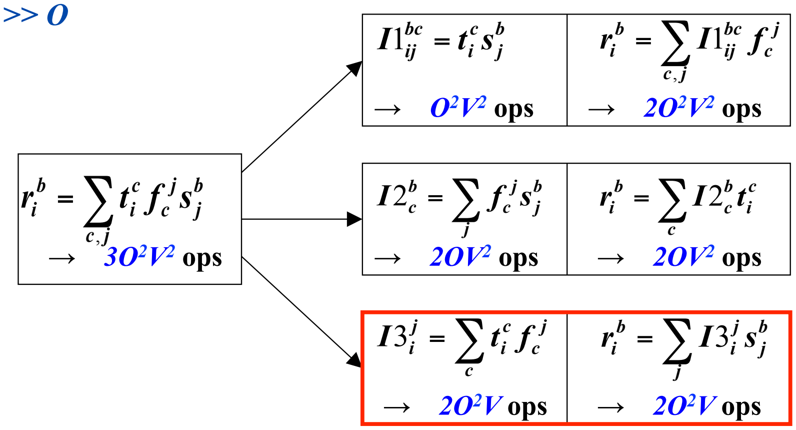
$$T1_{bcdf} = \sum_{e,l} B_{befl} D_{cdel}$$

$$T2_{bcjk} = \sum_{d,f} T1_{bcdf} C_{dfjk}$$

$$S_{abij} = \sum_{c,k} T2_{bcjk} A_{acik}$$

Single-Term Optimization (Binarization)

- ***b, c*** : range ***V*** (# virtual orbitals)
  i, j : range ***O*** (# occupied orbitals)
  V >> ***O***



$$r_i^b = \sum_{c,j} t_i^c f_c^j s_j^b \rightarrow 3O^2V^2 \text{ ops}$$

$$I1_{ij}^{bc} = t_i^c s_j^b \rightarrow O^2V^2 \text{ ops} \qquad r_i^b = \sum_{c,j} I1_{ij}^{bc} f_c^j \rightarrow 2O^2V^2 \text{ ops}$$

$$I2_c^b = \sum_j f_c^j s_j^b \rightarrow 2OV^2 \text{ ops} \qquad r_i^b = \sum_c I2_c^b t_i^c \rightarrow 2OV^2 \text{ ops}$$

$$I3_i^j = \sum_c t_i^c f_c^j \rightarrow 2O^2V \text{ ops} \qquad r_i^b = \sum_j I3_i^j s_j^b \rightarrow 2O^2V \text{ ops}$$

- Reduce the operation count from $3O^2V^2$ to $4O^2V$.
- Algorithms: *dynamic programming* (for small cases) and *heuristic search* (for large cases)

Multi-Term Optimization (Factorization)

- Unoptimized:

$$r_{ij}^{ab} = \sum_{c,d} t_i^c s_j^d v_{cd}^{ab} + \sum_{c,d} u_{ij}^{cd} v_{cd}^{ab} \quad \rightarrow \quad 2O^2V^4 + 3O^2V^4 \text{ ops}$$

- Single-term optimization:

$$r_{ij}^{ab} = \sum_{d} \left(\sum_{c} t_i^c v_{cd}^{ab} \right) s_j^d + \sum_{c,d} u_{ij}^{cd} v_{cd}^{ab} \quad \rightarrow \quad 2O^2V^4 + 2OV^4 + 2O^2V^3 \text{ ops}$$

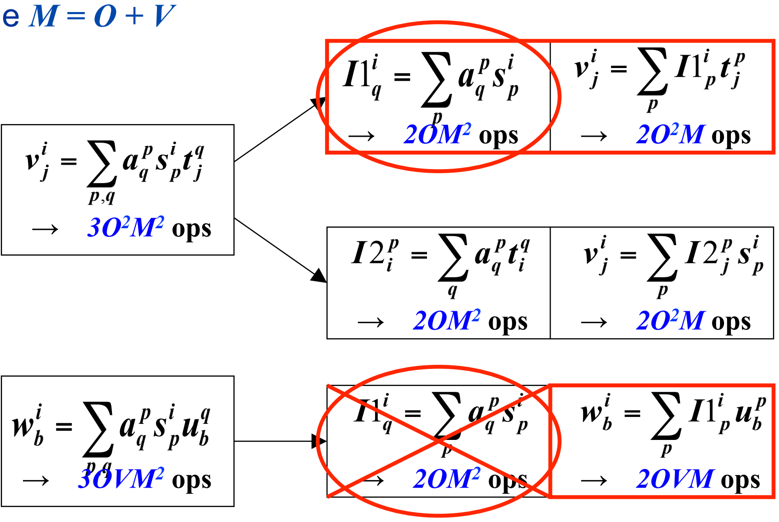
- Factorization:

$$r_{ij}^{ab} = \sum_{c,d} \left(t_i^c s_j^d + u_{ij}^{cd} \right) v_{cd}^{ab} \quad \rightarrow \quad 2O^2V^4 + O^2V^2 \text{ ops}$$

- Improved operation count over single-term optimization.

Common Subexpression Elimination

- p, q : range $M = O + V$



$v_j^i = \sum_{p,q} a_q^p s_p^i t_j^q \rightarrow 3O^2M^2$ ops

$I1_q^i = \sum_p a_q^p s_p^i \rightarrow 2OM^2$ ops | $v_j^i = \sum_p I1_p^i t_j^p \rightarrow 2O^2M$ ops

$I2_i^p = \sum_q a_q^p t_i^q \rightarrow 2OM^2$ ops | $v_j^i = \sum_p I2_j^p s_p^i \rightarrow 2O^2M$ ops

$w_b^i = \sum_{p,q} a_q^p s_p^i u_b^q \rightarrow 3OVM^2$ ops

$\cancel{I1_q^i = \sum_p a_q^p s_p^i \rightarrow 2OM^2 \text{ ops}}$ | $w_b^i = \sum_p I1_p^i u_b^p \rightarrow 2OVM$ ops

- Improves operation count by $2OM^2$.

Algebraic Transformation: Summary

$$S(a,b,i,j) = \sum_{c,d,e,f,k,l} A(a,c,i,k)B(b,e,f,l)C(d,f,j,k)D(c,d,e,l)$$

- Requires 4 * N^{10} operations if indices a-l have range N
- Optimized form requires only 6 * N^6 operations

$$T1(b,c,d,f) = \sum_{e,l} B(b,e,f,l)D(c,d,e,l)$$

$$T2(b,c,j,k) = \sum_{d,f} T1(b,c,d,f)C(d,f,j,k)$$

$$S(a,b,i,j) = \sum_{c,k} T2(b,c,j,k)A(a,c,i,k)$$

- Optimization Problem: Given an input tensor-contraction expression, find equivalent form that minimizes # operations
  - Problem is NP-hard; efficient pruning search strategy developed, that has been very effective in practice
- However, storage requirements increase after operation minimization

Memory Minimization: Compute by Parts (Loop Fusion)

$$T1_{bcdf} = \sum_{e,l} B_{befl} D_{cdel}$$

$$T2_{bcjk} = \sum_{d,f} T1_{bcdf} C_{dfjk}$$

$$S_{abij} = \sum_{c,k} T2_{bcjk} A_{acik}$$

Formula sequence

Memory Minimization: Compute by Parts (Loop Fusion)

$$T1_{bcdf} = \sum_{e,l} B_{befl} D_{cdel}$$
$$T2_{bcjk} = \sum_{d,f} T1_{bcdf} C_{dfjk}$$
$$S_{abij} = \sum_{c,k} T2_{bcjk} A_{acik}$$

Formula sequence

```
T1 = 0; T2 = 0; S = 0
for b, c, d, e, f, l
    $T1_{bcdf}$ += $B_{befl}\ D_{cdel}$
for b, c, d, f, j, k
    $T2_{bcjk}$ += $T1_{bcdf}\ C_{dfjk}$
for a, b, c, i, j, k
    $S_{abij}$ += $T2_{bcjk}\ A_{acik}$
```

Unfused code

Memory Minimization: Compute by Parts (Loop Fusion)

$$T1_{bcdf} = \sum_{e,l} B_{befl} D_{cdel}$$
$$T2_{bcjk} = \sum_{d,f} T1_{bcdf} C_{dfjk}$$
$$S_{abij} = \sum_{c,k} T2_{bcjk} A_{acik}$$

Formula sequence

```
T1 = 0; T2 = 0; S = 0
for b, c, d, e, f, l
    $T1_{bcdf}$ += $B_{befl}\ D_{cdel}$
for b, c, d, f, j, k
    $T2_{bcjk}$ += $T1_{bcdf}\ C_{dfjk}$
for a, b, c, i, j, k
    $S_{abij}$ += $T2_{bcjk}\ A_{acik}$
```

Unfused code

```
S = 0
for b, c
    T1f = 0; T2f = 0
    for d, e, f, l
        $T1f_{df}$ += $B_{befl}\ D_{cdel}$
    for d, f, j, k
        $T2f_{jk}$ += $T1f_{df}\ C_{dfjk}$
    for a, i, j, k
        $S_{abij}$ += $T2f_{jk}\ A_{acik}$
```

(Partially) Fused code

Memory Minimization: Loop Fusion

```
T1 = 0; T2 = 0; S = 0
for b, c, d, e, f, l
    $T1_{bcdf} += B_{befl} D_{cdel}$
for b, c, d, f, j, k
    $T2_{bcjk} += T1_{bcdf} C_{dfjk}$
for a, b, c, i, j, k
    $S_{abij} += T2_{bcjk} A_{acik}$
```

Unfused code

```
S = 0
for b, c
    T1f = 0; T2f = 0
    for d, e, f, l
        $T1f_{df} += B_{befl} D_{cdel}$
    for d, f, j, k
        $T2f_{jk} += T1f_{df} C_{dfjk}$
    for a, i, j, k
        $S_{abij} += T2f_{jk} A_{acik}$
```

(Partially) Fused code

```
S = 0
for b, c
    T1f = 0; T2f = 0
    for d, f
        for e, l
            $T1f += B_{befl} D_{cdel}$
        for j, k
            $T2f_{jk} += T1f\, C_{dfjk}$
    for a, i, j, k
        $S_{abij} += T2f_{jk} A_{acik}$
```

Fully Fused code

- Optimization Problem: Given an operation-minimized sequence of tensor-contractions, find “best” set of loops to fuse, to minimize memory access overhead
  – Problem is NP-hard; heuristics and pruning search used

Operation Minimal Form

```
for a, e, c, f
    for i, j
        $X_{aecf} += T_{ijae} T_{ijcf}$
for c, e, b, k
    $T1_{cebk} = f1(c, e, b, k)$
for a, f, b, k
    $T2_{afbk} = f2(a, f, b, k)$
for c, e, a, f
    for b, k
        $Y_{ceaf} += T1_{cebk} T2_{afbk}$
for c, e, a, f
    $E += X_{aecf} Y_{ceaf}$
```

Inputs

External function calls

Output

array	space	time
X	V^4	V^4O^2
T1	V^3O	$C_{f1}V^3O$
T2	V^3O	$C_{f2}V^3O$
Y	V^4	V^5O
E	1	V^4

a .. f: range V = 1000 .. 3000
i .. k: range O = 30 .. 100

Memory-Minimal Form

```
for a, f, b, k
    $T2_{afbk} = f2(a, f, b, k)$
for c, e
    for b, k
        $T1_{bk} = f1(c, e, b, k)$
    for a, f
        for i, j
            $X \mathrel{+}= T_{ijae} T_{ijcf}$
        for b, k
            $Y \mathrel{+}= T1_{bk} T2_{afbk}$
        $E \mathrel{+}= X Y$
```

Fusion of loops allows reduction of rank of arrays

array	space	time
X	1	V^4O^2
T1	VO	$C_{f1}V^3O$
T2	V^3O	$C_{f2}V^3O$
Y	1	V^5O
E	1	V^4

a .. f: range V = 3000

i .. k: range O = 100

Redundant Computation Allows Full Fusion

```
for a, e, c, f
    for i, j
        $X \mathrel{+}= T_{ijae} T_{ijcf}$
    for b, k
        $T1 = f1(c, e, b, k)$
        $T2 = f2(a, f, b, k)$
        $Y \mathrel{+}= T1\ T2$
    $E \mathrel{+}= X Y$
```

array	space	time
X	1	V^4O^2
T1	1	$C_{f1}V^5O$
T2	1	$C_{f2}V^5O$
Y	1	V^5O
E	1	V^4

Tiling to Reduce Recomputation

Loop over tiles

```
for $a^t, e^t, c^t, f^t$
    for a, e, c, f
        for i, j
            $X_{aecf} += T_{ijae} T_{ijcf}$
    for b, k
        for c, e
            $T1_{ce} = f1(c, e, b, k)$
        for a, f
            $T2_{af} = f2(a, f, b, k)$
        for c, e, a, f
            $Y_{ceaf} += T1_{ce} T2_{af}$
    for c, e, a, f
        $E += X_{aecf} Y_{ceaf}$
```

array	space	time
X	B^4	V^4O^2
T1	B^2	$C_{f1}(V/B)^2V^3O$
T2	B^2	$C_{f2}(V/B)^2V^3O$
Y	B^4	V^5O
E	1	V^4

Tiling further improves locality

High-Performance Tensor Computations

- Tensor computations expressible as nested loops operating on multi-dimensional arrays. We see several possible approaches
  - Use a compiler optimization framework to automatically optimize loops with complex nesting structure (***motivation for our work on PLUTO, a polyhedral optimizer***)
  - Exploit BLAS (***we discuss this next***)
- BLAS + Index Permutations
  - Highly-tuned GEMM routines in the BLAS library can be used since a tensor contraction is essentially a generalized matrix multiplication.
  - GEMM requires a two-dimensional view of the input matrices:
    - Summation and non-summation indices should be grouped into two contiguous sets.
    - Index permutation is needed to reshape the arrays.
  - Goal: Minimize the execution time of the generated code

One Approach: BLAS + Index Permutations

- Key aspects of this approach
  - Optimize a sequence of calls using information about the performance of these routines.
  - Provide portable performance across architectures.
- Two types of constituent operations:
  - Generalized Matrix Multiplication (GEMM)
  - Index Permutation
- Challenge: Useful, combinable empirical performance-model of constituent operations.
  - Optimize index permutation + choice of GEMM
  - Sequence of tensor contractions
  - Exploiting parallelism

Example: BLAS + index permutations

A contraction example:

$$E(i,j,c)=\sum_{a,b}[A(a,b,c)\times B(a,i)\times C(b,j)]$$

All indices range over N, an operation-minimal evaluation sequence is:

$$T1(i,b,c)=\sum_{a}[A(a,b,c)\times B(a,i)]$$

$$E(i,j,c)=\sum_{b}[T1(i,b,c)\times C(b,j)]$$

Example: BLAS + index permutations

Many ways of generating code, two of them are:

1:

Reshape A: (a,b,c) → (c,b,a)

GEMM: B(a,i) x A(cb,a) → T1(i,cb); with (t,t)

GEMM: C(b,j) x T1(ic,b) → E(j,ic); with (t,t)

Reshape E: (j,i,c) → (i,j,c)

2:

GEMM: A(a,bc)xB(a,i)→T1(bc,i); with (t,n)

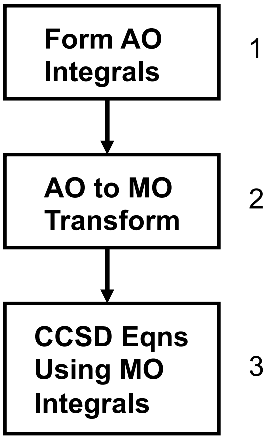
GEMM: T1(b,ci)xC(b,j)→E(ci,j); with (t,n)

Reshape E: (c,i,j) → (i,j,c)

Neither one is better than the other for all the array sizes!

Operation Minimization Experiments

- Combined optimization across three steps
  - Normally separately (manually) optimized
  - Each step uses tensor expressions
- Exp. 1: Combine 2 and 3
  - Feed Optimizer expressions for AO-to-MO transform, along with CCSD Equations
- Exp. 2: Combine 1, 2, & 3
  - Cholesky decomposition for forming AO integrals; combine all three steps

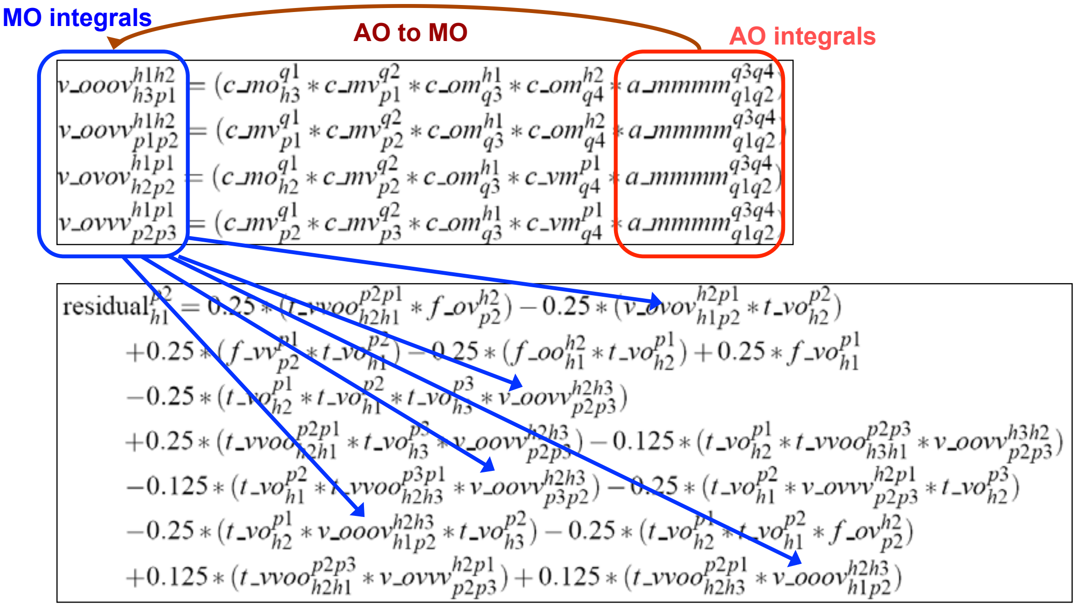


Form AO Integrals 1

AO to MO Transform 2

CCSD Eqns Using MO Integrals 3

Standard Two-Step CCSD T1



MO integrals

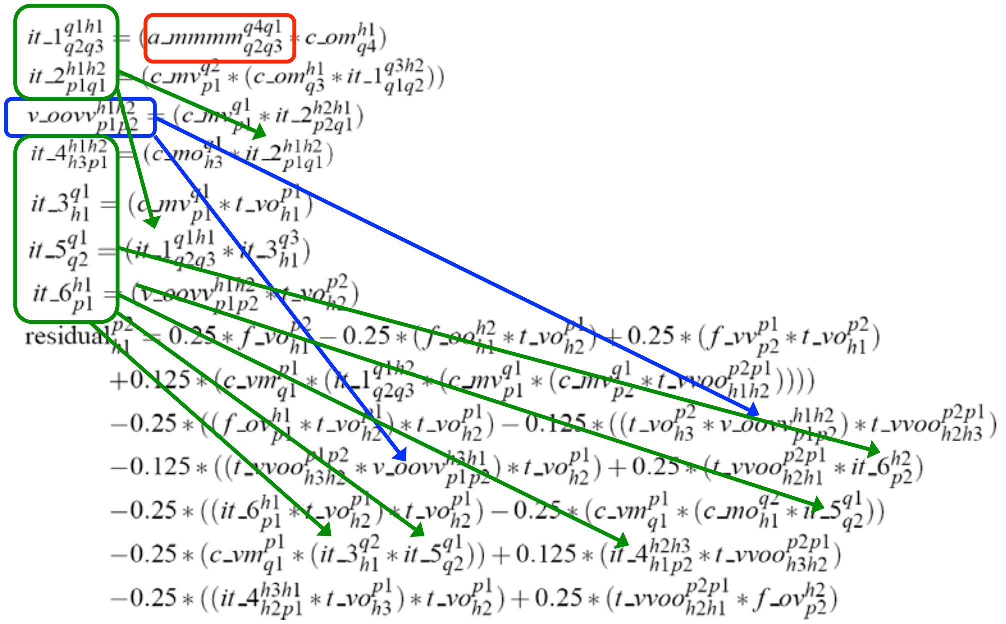
AO to MO

AO integrals

$$
\begin{aligned}
v\_ooov^{h1h2}_{h3p1} &= (c\_mo^{q1}_{h3} * c\_mv^{q2}_{p1} * c\_om^{h1}_{q3} * c\_om^{h2}_{q4} * a\_mmmm^{q3q4}_{q1q2})\\
v\_oovv^{h1h2}_{p1p2} &= (c\_mv^{q1}_{p1} * c\_mv^{q2}_{p2} * c\_om^{h1}_{q3} * c\_om^{h2}_{q4} * a\_mmmm^{q3q4}_{q1q2})\\
v\_ovov^{h1p1}_{h2p2} &= (c\_mo^{q1}_{h2} * c\_mv^{q2}_{p2} * c\_om^{h1}_{q3} * c\_vm^{p1}_{q4} * a\_mmmm^{q3q4}_{q1q2})\\
v\_ovvv^{h1p1}_{p2p3} &= (c\_mv^{q1}_{p2} * c\_mv^{q2}_{p3} * c\_om^{h1}_{q3} * c\_vm^{p1}_{q4} * a\_mmmm^{q3q4}_{q1q2})
\end{aligned}
$$

$$
\begin{aligned}
\text{residual}^{p2}_{h1} = {} & 0.25 * (t\_vvoo^{p2p1}_{h2h1} * f\_ov^{h2}_{p2}) - 0.25 * (v\_ovov^{h2p1}_{h1p2} * t\_vo^{p2}_{h2})\\
& +0.25 * (f\_vv^{p1}_{p2} * t\_vo^{p2}_{h1}) - 0.25 * (f\_oo^{h2}_{h1} * t\_vo^{p1}_{h2}) + 0.25 * f\_vo^{p1}_{h1}\\
& -0.25 * (t\_vo^{p1}_{h2} * t\_vo^{p2}_{h1} * t\_vo^{p3}_{h3} * v\_oovv^{h2h3}_{p2p3})\\
& +0.25 * (t\_vvoo^{p2p1}_{h2h1} * t\_vo^{p3}_{h3} * v\_oovv^{h2h3}_{p2p3}) - 0.125 * (t\_vo^{p1}_{h2} * t\_vvoo^{p2p3}_{h3h1} * v\_oovv^{h3h2}_{p2p3})\\
& -0.125 * (t\_vo^{p2}_{h1} * t\_vvoo^{p3p1}_{h2h3} * v\_oovv^{h2h3}_{p3p2}) - 0.25 * (t\_vo^{p2}_{h1} * v\_ovvv^{h2p1}_{p2p3} * t\_vo^{p3}_{h2})\\
& -0.25 * (t\_vo^{p1}_{h2} * v\_ooov^{h2h3}_{h1p2} * t\_vo^{p2}_{h3}) - 0.25 * (t\_vo^{p1}_{h2} * t\_vo^{p2}_{h1} * f\_ov^{h2}_{p2})\\
& +0.125 * (t\_vvoo^{p2p3}_{h2h1} * v\_ovvv^{h2p1}_{p2p3}) + 0.125 * (t\_vvoo^{p2p1}_{h2h3} * v\_ooov^{h2h3}_{h1p2})
\end{aligned}
$$

Combined AO-to-MO & CCSD T1



$$
\begin{aligned}
it\_1^{q1h1}_{q2q3} &= (a\_mmmm^{q4q1}_{q2q3} * c\_om^{h1}_{q4})\\
it\_2^{h1h2}_{p1q1} &= (c\_mv^{q2}_{p1} * (c\_om^{h1}_{q3} * it\_1^{q3h2}_{q1q2}))\\
v\_oovv^{h1h2}_{p1p2} &= (c\_mv^{q1}_{p1} * it\_2^{h2h1}_{p2q1})\\
it\_4^{h1h2}_{h3p1} &= (c\_mo^{q1}_{h3} * it\_2^{h1h2}_{p1q1})\\
it\_3^{q1}_{h1} &= (c\_mv^{q1}_{p1} * t\_vo^{p1}_{h1})\\
it\_5^{q1}_{q2} &= (it\_1^{q1h1}_{q2q3} * it\_3^{q3}_{h1})\\
it\_6^{h1}_{p1} &= (v\_oovv^{h1h2}_{p1p2} * t\_vo^{p2}_{h2})
\end{aligned}
$$

$$
\begin{aligned}
\text{residual}^{p2}_{h1} = {} & 0.25 * f\_vo^{p2}_{h1} - 0.25 * (f\_oo^{h2}_{h1} * t\_vo^{p1}_{h2}) + 0.25 * (f\_vv^{p1}_{p2} * t\_vo^{p2}_{h1})\\
& +0.125 * (c\_vm^{p1}_{q1} * (it\_1^{q1h2}_{q2q3} * (c\_mv^{q1}_{p1} * (c\_mv^{q1}_{p2} * t\_vvoo^{p2p1}_{h1h2}))))\\
& -0.25 * ((f\_ov^{h1}_{p1} * t\_vo^{p1}_{h2}) * t\_vo^{p1}_{h2}) - 0.125 * ((t\_vo^{p2}_{h3} * v\_oovv^{h1h2}_{p1p2}) * t\_vvoo^{p2p1}_{h2h3})\\
& -0.125 * ((t\_vvoo^{p1p2}_{h3h2} * v\_oovv^{h3h1}_{p1p2}) * t\_vo^{p1}_{h2}) + 0.25 * (t\_vvoo^{p2p1}_{h2h1} * it\_6^{h2}_{p2})\\
& -0.25 * ((it\_6^{h1}_{p1} * t\_vo^{p1}_{h2}) * t\_vo^{p1}_{h2}) - 0.25 * (c\_vm^{p1}_{q1} * (c\_mo^{q2}_{h1} * it\_5^{q1}_{q2}))\\
& -0.25 * (c\_vm^{p1}_{q1} * (it\_3^{q2}_{h1} * it\_5^{q1}_{q2})) + 0.125 * (it\_4^{h2h3}_{h1p2} * t\_vvoo^{p2p1}_{h3h2})\\
& -0.25 * ((it\_4^{h3h1}_{h2p1} * t\_vo^{p1}_{h3}) * t\_vo^{p1}_{h2}) + 0.25 * (t\_vvoo^{p2p1}_{h2h1} * f\_ov^{h2}_{p2})
\end{aligned}
$$

Considering CCSD Iterations

$$v\_ooov^{h1h2}_{h3p1} = (c\_mo^{q1}_{h3} * c\_mv^{q2}_{p1} * c\_om^{h1}_{q3} * c\_om^{h2}_{q4} * a\_mmmm^{q3q4}_{q1q2})$$
$$v\_oovv^{h1h2}_{p1p2} = (c\_mv^{q1}_{p1} * c\_mv^{q2}_{p2} * c\_om^{h1}_{q3} * c\_om^{h2}_{q4} * a\_mmmm^{q3q4}_{q1q2})$$
$$v\_ovov^{h1p1}_{h2p2} = (c\_mo^{q1}_{h2} * c\_mv^{q2}_{p2} * c\_om^{h1}_{q3} * c\_vm^{p1}_{q4} * a\_mmmm^{q3q4}_{q1q2})$$
$$v\_ovvv^{h1p1}_{p2p3} = (c\_mv^{q1}_{p2} * c\_mv^{q2}_{p3} * c\_om^{h1}_{q3} * c\_vm^{p1}_{q4} * a\_mmmm^{q3q4}_{q1q2})$$

$$\begin{aligned}
\text{residual}^{p2}_{h1} = {} & 0.25 * (t\_vvoo^{p2p1}_{h2h1} * f\_ov^{h2}_{p2}) - 0.25 * (v\_ovov^{h2p1}_{h1p2} * t\_vo^{p2}_{h2}) \\
& +0.25 * (f\_vv^{p1}_{p2} * t\_vo^{p2}_{h1}) - 0.25 * (f\_oo^{h2}_{h1} * t\_vo^{p1}_{h2}) + 0.25 * f\_vo^{p1}_{h1} \\
& -0.25 * (t\_vo^{p1}_{h2} * t\_vo^{p2}_{h1} * t\_vo^{p3}_{h3} * v\_oovv^{h2h3}_{p2p3}) \\
& +0.25 * (t\_vvoo^{p2p1}_{h2h1} * t\_vo^{p3}_{h3} * v\_oovv^{h2h3}_{p2p3}) - 0.125 * (t\_vo^{p1}_{h2} * t\_vvoo^{p2p3}_{h3h1} * v\_oovv^{h3h2}_{p2p3}) \\
& -0.125 * (t\_vo^{p2}_{h1} * t\_vvoo^{p3p1}_{h2h3} * v\_oovv^{h2h3}_{p3p2}) - 0.25 * (t\_vo^{p2}_{h1} * v\_ovvv^{h2p1}_{p2p3} * t\_vo^{p3}_{h2}) \\
& -0.25 * (t\_vo^{p1}_{h2} * v\_ooov^{h2h3}_{h1p2} * t\_vo^{p2}_{h3}) - 0.25 * (t\_vo^{p1}_{h2} * t\_vo^{p2}_{h1} * f\_ov^{h2}_{p2}) \\
& +0.125 * (t\_vvoo^{p2p3}_{h2h1} * v\_ovvv^{h2p1}_{p2p3}) + 0.125 * (t\_vvoo^{p2p1}_{h2h3} * v\_ooov^{h2h3}_{h1p2})
\end{aligned}$$

… Other computations that modify tensors t_vo etc.

Optimized CCSD T1

$$it\_1^{q1h1}_{q2q3} = (a\_mmmm^{q4q1}_{q2q3} * c\_om^{h1}_{q4})$$
$$it\_2^{h1h2}_{p1q1} = (c\_mv^{q2}_{p1} * (c\_om^{h1}_{q3} * it\_1^{q3h2}_{q1q2}))$$
$$v\_oovv^{h1h2}_{p1p2} = (c\_mv^{q1}_{p1} * it\_2^{h2h1}_{p2q1})$$
$$it\_4^{h1h2}_{h3p1} = (c\_mo^{q1}_{h3} * it\_2^{h1h2}_{p1q1})$$

Unchanged every iteration; compute only once

$$it\_3^{q1}_{h1} = (c\_mv^{q1}_{p1} * t\_vo^{p1}_{h1})$$
$$it\_5^{q1}_{q2} = (it\_1^{q1h1}_{q2q3} * it\_3^{q3}_{h1})$$
$$it\_6^{h1}_{p1} = (v\_oovv^{h1h2}_{p1p2} * t\_vo^{p2}_{h2})$$

Re-compute every iteration

$$\begin{aligned}
\text{residual}^{p2}_{h1} = {} & 0.25 * f\_vo^{p2}_{h1} - 0.25 * (f\_oo^{h2}_{h1} * t\_vo^{p1}_{h2}) + 0.25 * (f\_vv^{p1}_{p2} * t\_vo^{p2}_{h1}) \\
& +0.125 * (c\_vm^{p1}_{q1} * (it\_1^{q1h2}_{q2q3} * (c\_mv^{q1}_{p1} * (c\_mv^{q1}_{p2} * t\_vvoo^{p2p1}_{h1h2})))) \\
& -0.25 * ((f\_ov^{h1}_{p1} * t\_vo^{p1}_{h2}) * t\_vo^{p1}_{h2}) - 0.125 * ((t\_vo^{p2}_{h3} * v\_oovv^{h1h2}_{p1p2}) * t\_vvoo^{p2p1}_{h2h3}) \\
& -0.125 * ((t\_vvoo^{p1p2}_{h3h2} * v\_oovv^{h3h1}_{p1p2}) * t\_vo^{p1}_{h2}) + 0.25 * (t\_vvoo^{p2p1}_{h2h1} * it\_6^{h2}_{p2}) \\
& -0.25 * ((it\_6^{h1}_{p1} * t\_vo^{p1}_{h2}) * t\_vo^{p1}_{h2}) - 0.25 * (c\_vm^{p1}_{q1} * (c\_mo^{q2}_{h1} * it\_5^{q1}_{q2})) \\
& -0.25 * (c\_vm^{p1}_{q1} * (it\_3^{q2}_{h1} * it\_5^{q1}_{q2})) + 0.125 * (it\_4^{h2h3}_{h1p2} * t\_vvoo^{p2p1}_{h3h2}) \\
& -0.25 * ((it\_4^{h3h1}_{h2p1} * t\_vo^{p1}_{h3}) * t\_vo^{p1}_{h2}) + 0.25 * (t\_vvoo^{p2p1}_{h2h1} * f\_ov^{h2}_{p2})
\end{aligned}$$

… Other computations that modify tensors t_vo etc.

Impact of Optimizations

CCSD T1 *(O=10, V=500)*

Iteration Count		Operation Count	Reduction Factor
1 (Brueckner)	Separated steps	5.36 x 10^{12}	1
	Combined Opt	1.51 x 10^{12}	3.55
10	Separated steps	5.63 x 10^{12}	1
	Combined Opt	2.26 x 10^{12}	2.49

CCSD T2

Iteration Count	Expanded MO Tensors	Operation Count	Reduction Factor
1	Seperated Steps	2.85 x 10^{14}	1
	Combined Opt.	1.93 x 10^{13}	14.75
10	Separated Steps	4.22 x 10^{14}	1
	Combined Opt.	1.67 x 10^{14}	2.53

Experiment 2

- ***Cholesky decomposition*** to compute AO basis integral tensors.

$$a_{rs}^{pq} = \sum_{z} u_{z}^{pq} u_{rs}^{z}$$

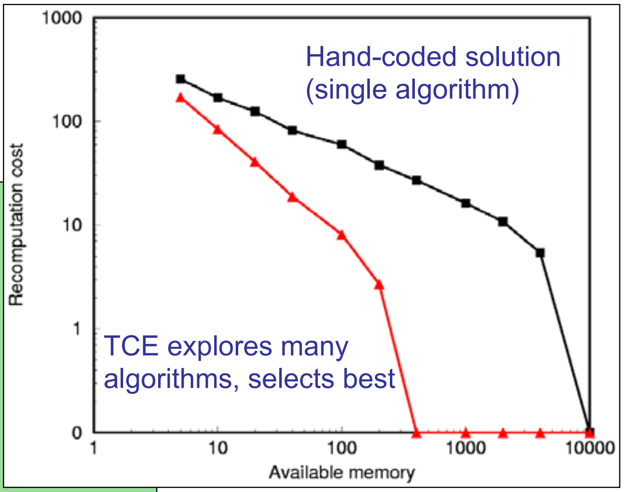
Equation	Number of terms	Expanded MO Integrals	AO Integrals
CCSD E	5	v_vvoo	a_mmmm
CCSD T1	26	v_vvov, v_ovvo, v_ovov, v_vvoo, v_ovoo	a_mmmm
CCSD T2	57	v_oooo, v_ooov, v_ovoo, v_oovv, v_ovov, v_ovvo, v_vvoo, v_ovvv, v_vvov, v_vvvv	a_mmmm

- Index ranges $\boldsymbol{O} = \mathbf{100},\ \boldsymbol{V} = \mathbf{5000},\ \boldsymbol{M} = \boldsymbol{O} + \boldsymbol{V},\ \boldsymbol{Z} = \mathbf{10}\ (\boldsymbol{O} + \boldsymbol{V})$

Impact of Optimizations

CCSD T2			
Iteration Count	Optimization	Operation Count	Reduction Factor
1	Separated Optimization	1.15e+20	1
	Combine AO-to-MO and CCSD	8.77e+19	1.31
	Cholesky-AO and AO-to-MO	8.39e+19	1.37
	Combining all three steps	4.87e+18	23.70
10	Separated Optimization	2.77e+20	1
	Combine AO-to-MO and CCSD	2.52e+20	1.10
	Cholesky-AO and AO-to-MO	2.41e+20	1.15
	Combining all three steps	4.75e+19	5.83

Space-time Trade-offs



Hand-coded solution (single algorithm)

TCE explores many algorithms, selects best

```
range V = 3000;
range O = 100;

index a,b,c,d,e,f : V;
index i,j,k : O;

mlimit = 1000000000000;

function F1(V,V,V,O);
function F2(V,V,V,O);

procedure P(in T1[O,O,V,V], in T2[O,O,V,V], out X)=

begin
   X == sum[ sum[F1(a,b,f,k) * F2(c,e,b,k), {b,k}]
                    * sum[T1[i,j,a,e] * T2[i,j,c,f], {i,j}],
                    {a,e,c,f}];
end
```

$$A3A = \tfrac{1}{2}\big(X_{ce,af}Y_{ae,cf} + X_{c\bar{e},a\bar{f}}Y_{a\bar{e},c\bar{f}} + X_{c\bar{e},\bar{a}f}Y_{\bar{a}\bar{e},cf} + X_{\bar{c}e,a\bar{f}}Y_{ae,\bar{c}\bar{f}} + X_{\bar{c}e,\bar{a}f}Y_{\bar{a}e,\bar{c}f} + X_{\bar{c}\bar{e},\bar{a}\bar{f}}Y_{\bar{a}\bar{e},\bar{c}\bar{f}}\big)$$

$$X_{ce,af} = t_{ij}^{ce}t_{ij}^{af} \qquad Y_{ae,cf} = \langle ab\|ek\rangle\langle cb\|fk\rangle$$

Experiments: Index Permute + BLAS

$$
\begin{aligned}
T1(a,q,r,s) &= \sum_p C4(p,a) * A(p,q,r,s) \\
T2(a,b,r,s) &= \sum_q C3(q,b) * T1(a,q,r,s) \\
T3(a,b,c,s) &= \sum_r C2(r,c) * T2(a,b,r,s) \\
B(a,b,c,d) &= \sum_s C1(s,d) * T3(a,b,c,s)
\end{aligned}
$$

- Atomic-Orbital to Molecular-Orbital Integral transform: very important transformation in quantum chemistry codes
- Tensors (double precision elements):
  - Sequential experiments: $N_p = N_q = N_r = N_s = N_a = N_b = N_c = N_d = 64$
  - Parallel experiments: $N_p = N_q = N_r = N_s = N_a = N_b = N_c = N_d = 96$

Experiments: Index Permute + BLAS

- Sequential results: the improvement is 20%

Unoptimized (sec.)				Optimized (sec.)			
GEMM	Index Permutation	Exec. Time	GFLOPS	GEMM	Index Permutation	Exec. Time	GFLOPS
10.06	2.58	12.64	2.07	10.58	0.0	10.58	2.48

- Parallel results on 4 processors: the improvement is 78%

Unoptimized (sec.)				Optimized (sec.)			
GEMM	Index Permutation	Exec. Time	GFLOPS	GEMM	Index Permutation	Exec. Time	GFLOPS
12.23	7.74	19.97	3.27	7.57	3.64	11.21	5.83

TCE: Summary of Work Done So Far

- Two versions of TCE developed
- Full exploitation of symmetry, but fewer optimizations (So Hirata)
- Partial exploitation of symmetry, but more sophisticated optimizations
- First parallel implementation for many of the chemistry methods
- Used to implement over 20 models, included in NWChem, a computational chemistry software distributed by Pacific Northwest Lab in US
- NWChem contains about 1M lines of human-generated code and over 2M lines of machine-generated code from TCE
- "The resulting scientific capabilities would have taken many man-decades of effort; instead, new theories / models can be tested in a day on a full-scale system" – Robert Harrison

TCE: More Challenges

- Tensors are not always dense!
- Here are some challenges
  - Exploiting symmetry
  - Exploiting sparsity
  - Exploiting block-sparsity (RINO: Regular Inner Nonregular Outer computations)
- Appears to require combination of domain-specific information, architecture-aware optimizations, and machine-specific optimizations

TCE: Ongoing and Future Work

- Problem: block-sparse and anti-symmetric tensors
- More sophisticated performance models
- Parallel code generation
  - Data distribution interacts w/ memory minimization
  - Multi-level parallelism needed for block-sparse tensors
- Use of PLUTO to drive optimizations in TCE after algebraic-optimizations (and perhaps memory minimization)
- Chemistry-specific optimizations
- Apply to tensor computations from other fields: materials science, nuclear physics

Summary

- The "power wall" has led to a major shift in architecture and is making heterogeneous computing essential
- Architectural diversity and heterogeneous computing create huge software challenges
- Domain-specific computing is a promising approach to effectively handle architectural diversity and heterogeneous computing
  - Productivity, portability, performance
  - Write-once-execute-well-anywhere
- **Close interaction between domain experts, systems software experts, and architects is essential**

Harrison’s Thoughts on DSLs

- **“Clearly, domain specific languages will be an integral part of future computational science and we note that several of the HPCS languages had at their core the idea of being extensible and readily specialized to new fields. However, translating the narrow success of the TCE into broad relevance remains a challenge**.
  - For instance, how can application scientists make effective use of the optimization and compilation tools of computer science without having a computer scientist at their side?
  - What elements are in common between languages tailored to chemistry or material science or linguistics or forestry?
  - How do we ensure that such programs can inter-operate when composing multi-physics applications?”

Further Reading

- Review of Tiling:
  - U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A Practical and Automatic Polyhedral Program Optimization System,” in *Proc. ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI’08),* pp. 101–113, Tucson, June 2008.
  - U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, “Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model,” in *Proc. CC 2008 - International Conference on Compiler Construction,* (L. Hendren Ed.), Lecture Notes in Computer Science, Vol. 4959, pp. 132–146, Springer-Verlag, 2008.

Further Reading

- Stencils:
  - S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev and P. Sadayappan, "Effective Automatic Parallelization of Stencil Computations," in *Proc. ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 07),* San Diego, June 2007.
  - T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, “Data Layout Transformation for Stencil Computations on Short SIMD Architectures,” in Proc. CC 2011 - International Conference on Compiler Construction, (J. Knoop Ed.), Lecture Notes in Computer Science, Vol. 6601, pp. 223–242, Springer-Verlag, 2011.
  - T. Henretty, R. Veras, F. Franchetti, L.N. Pouchet, J. Ramanujam and P. Sadayappan, “A Domain-Specific Language and Compiler for Stencil Computations on Short-Vector SIMD and GPU Architectures,” in *17th Workshop on Compilers for Parallel Computing (CPC 2013),* Lyon, France, July 2013.

Further Reading

- Stencils (continued):
  - T. Henretty, J. Holewinski, R. Veras, F. Franchetti, L.N. Pouchet, J. Ramanujam, A. Rountev and P. Sadayappan, “A Stencil Compiler for Short-Vector SIMD Architectures,” in *Proc. 27th ACM International Conference on Supercomputing*, Eugene, OR, June 2013.
  - A. Cohen, T. Grosser, P. Kelly, J. Ramanujam, P. Sadayappan, and S.Verdoolaege, “Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles to Reconcile Parallelism and Locality, Avoiding Divergence and Load Imbalance,” in *Proc. 6th Workshop on General Purpose Processing Using GPUs (GPGPU-6)*, held with ASPLOS '13, March 2013.
  - K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, “A Framework for Enhancing Data Reuse via Associative Reordering,” *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2014)*, pp. 65–76, Edinburgh, UK, June 2014.

Further Reading

- Irregular codes (finite-elements code generation, runtime compilation, …) :
  - M. Strout, F. Luporini, C. Krieger, C. Bertolli, G.-T. Bercea, C. Olschanowsky, J. Ramanujam, and P. Kelly, “Generalizing Run-time Tiling with the Loop Chain Abstraction,” in *Proc. 28th IEEE International Parallel & Distributed Processing Symposium*, Phoenix, AZ, April 2014.
  - F. Luporini, A.L. Varbanescu, F. Rathgeber, G.-T. Bercea, J. Ramanujam, D.A. Ham, and P.H.J. Kelly, “Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly,” *ACM Transactions on Architecture and Code Optimization,* vol. 11, no. 4, 57:1–57:25, January 2015.
  - M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, “Automatic Parallelization of a Class of Irregular Loops for Distributed Memory Systems,” in *ACM Transactions on Parallel Computing*, vol. 1, no. 1, pp. 7:1–7:37, September 2014.

Further Reading

- Tensor Contraction Engine (TCE):
  - A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, D. Bernholdt, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan, “Performance Optimization of Tensor Contraction Expressions for Many Body Methods in Quantum Chemistry,” *The Journal of Physical Chemistry A,* vol. 113, no. 45, pp. 12715–12723, 2009.
  - Q. Lu, X. Gao, S. Krishnamoorthy, G. Baumgartner, J. Ramanujam, and P. Sadayappan, “Empirical Performance Model-Driven Data Layout Optimization and Library Call Selection for Tensor Contraction Expressions,” *Journal of Parallel and Distributed Computing*, vol. 72, no. 3, pp. 338–352, March 2012.
  - A. Auer, G. Baumgartner, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Automatic Code Generation for Many-Body Electronic Structure Methods: The Tensor Contraction Engine," *Molecular Physics,* vol. 104, no. 2, pp. 211--228, January 2006.

Further Reading

- Tensor Contraction Engine (TCE) -- continued:

  - G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Synthesis of High-Performance Parallel Programs for a Class of ab initio Quantum Chemistry Models," *Proceedings of the IEEE,* vol. 93, no. 2, pp. 276-292, February 2005.
  - S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C. Lam, J. Ramanujam, P. Sadayappan, and V. Choppella, "Efficient Synthesis of Out-of-Core Algorithms Using a Nonlinear Optimization Solver," *Journal of Parallel and Distributed Computing,* vol. 66, no. 5, pp. 659-673, May 2006.
  - D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, R. Harrison and R. Pitzer, "A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry," in *Proceedings of Supercomputing 2002 (SC2002),* November 2002.

Further Reading

- Tensor Contraction Engine (TCE) -- continued:

  - D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison, "Space-time trade-off optimization for a class of electronic structure calculations," in *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI),* pp. 177-186, Berlin, Germany, June 2002.
  - D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison, "Towards Automatic Synthesis of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization," in *Proc. of the Intl. Conf. on High Performance Computing,* Lecture Notes in Comp. Sci,, Vol. 2228, pp. 237-248, Springer-Verlag, 2001.
  - A. Bibireata, S. Krishnan, G. Baumgartner, D. Cociorva, C. Lam, P. Sadayappan, J. Ramanujam, D. Bernholdt, and V. Choppella, "Memory-Constrained Data Locality Optimization for Tensor Contractions," in *Languages and Compilers for Parallel Computing,* (L. Rauchwerger et al. Eds.), LNCS, Vol. 2958, pp. 93-108, Springer-Verlag, 2004.