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)  Carlo Bertolli
Franz Franchetti (CMU)    Fabio Luporini
Louis-Noel Pouchet (OSU) Albert Hartono
P. Sadayappan (OSU)       Justin Holewinski
Robert Harrison (Stony Brook) Venmugil Elango
Fabrice Rastello (ENS Lyon) Tom Henretty
Nasko Rountev (OSU)       Mahesh Ravishankar
Sven Verdoolaege (ENS)    Sanket Tavarageri
Tobias Grosser (ETH)      Richard Veras
Paul Kelly (Imperial)     Sameer Abu Asal
Michelle Strout (Arizona) Rod Tohid
S. Krishnamoorthy (PNNL) Ye Fang
Uday Bondhugula (IISc)    Michal Brylinski
Muthu Baskaran (Reservoir) 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

Darte, Feautrier, Pugh, ...

Cohen, Feautrier, Griebl, Lam, Pingali ...

Huge space of valid transforms
How to find an effective one?

Efficient Algorithms before Pluto

Transforms (Affine Functions)

Output Program

Code Generation:
Polyhedra -> Loops

Ancourt, Bastoul, Irigoin, Quillere, Rajopadhye, Wilde ...

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:

$$
\phi_{S_k}(i) = \begin{bmatrix}
c_1 & c_2 & \ldots & c_{m_{S_k}}
\end{bmatrix}
\begin{pmatrix}
i
\end{pmatrix} + c_0
\equiv \begin{bmatrix}
c_1 & c_2 & \ldots & c_{m_{S_k}} & c_0
\end{bmatrix}
\begin{pmatrix}
i
1
\end{pmatrix}
$$

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

- An affine transform
  - = A new scanning hyperplane
  - = A loop in the transformed space (with a particular property)

---

1-D Jacobi (imperfectly nested)

```c
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];
    }
}
```

$$
\begin{bmatrix}
\phi_1^S & \phi_2^S \\
\phi_1^T & \phi_2^T
\end{bmatrix}
\begin{pmatrix}
t \\
i \\
t \\
j
\end{pmatrix}
= \begin{bmatrix}
1 & 0 & 0 \\
2 & 1 & 0 \\
1 & 0 & 0 \\
2 & 1 & 1
\end{bmatrix}
$$
Pluto: 1-D Jacobi (imperfectly nested)

\[
\begin{bmatrix}
\phi_1^S \\
\phi_2^S \\
\phi_T^S \\
\phi_T^S
\end{bmatrix}
\begin{pmatrix}
t \\
i \\
1
\end{pmatrix}
= \begin{bmatrix}
1 & 0 & 0 \\
2 & 1 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
\phi_1^T \\
\phi_2^T \\
\phi_T^T
\end{bmatrix}
\begin{pmatrix}
t \\
j \\
1
\end{pmatrix}
= \begin{bmatrix}
1 & 0 & 0 \\
2 & 1 & 1
\end{bmatrix}
\]

- 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 T

\[
\begin{bmatrix}
\phi^1_T \\
\phi^2_T
\end{bmatrix}
\begin{pmatrix}
t \\
j \\
1
\end{pmatrix}
= 
\begin{bmatrix}
1 & 0 & 0 \\
2 & 1 & 1
\end{bmatrix}
\]

Pluto: Interleaving S and T
Pluto: Interleaving S and T

\[
\begin{bmatrix}
\phi_1^{S} \\
\phi_2^{S}
\end{bmatrix}
\begin{pmatrix}
t \\i \\
1
\end{pmatrix}
=
\begin{bmatrix}
1 & 0 & 0 \\
2 & 1 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
\phi_1^{T} \\
\phi_2^{T}
\end{bmatrix}
\begin{pmatrix}
t \\j \\
1
\end{pmatrix}
=
\begin{bmatrix}
1 & 0 & 0 \\
2 & 1 & 1
\end{bmatrix}
\]

1-D Jacobi (imperfectly nested) – transformed code

```c
for (t0=0; t0<=M-1; t0++) {
    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

\[
\begin{align*}
\text{for } (t0=0; t0<=M-1; t0++) & \{ \\
& \text{for } (t1=2*t0+3; t1<=2*t0+N-2; t1++) \{ \\
& \text{\textbf{S}} & : b[-2*t0+t1]=0.333*(a[-2*t0+t1-1]+a[-2*t0+t1]+a[-2*t0+t1+1]); \\
& \text{\textbf{T}} & : a[-2*t0+t1-1]=b[-2*t0+t1-1]; \\
& \text{\textbf{T}}' & : a[N-2]=b[N-2]; \\
\} \\
\end{align*}
\]

Pluto: Communication Volume & Reuse Distance

- \( \phi(i') - \phi(i) \) is an affine function that represents the component of a dependence along hyperplane \( \phi \)
  - 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)

• 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

```c
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

```c
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

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

Page 21 of 226

Stencil DSL Example -- Standalone

```c
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

Page 22 of 226
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

```plaintext
for t = 1 to T
  for i = 1 to N
  for i = 1 to N
    A[i] = B[i]
```

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

\[
\text{FOR } t = 0 \text{ TO } T-1 \\
\text{FOR } i = 1 \text{ TO } N-1 \\
A[t+1,i] = (A[t,i-1] + A[t,i] + A[t,i+1]) / 3
\]

### Time Tiling (with 1-D array code)

- Cache misses = \(\Theta(TN)\)
- Concurrency in each \(t\)
- Time tiling causes pipelined execution
- **Solution**: Adjust tiling – re-enable concurrent execution in a row of tiles

\[
\text{Cache misses } = \Theta(TN/B) \\
\text{No concurrent in a row}
\]
Motivation

\[
\text{FOR } t = 0 \text{ TO } T-1 \\
\text{FOR } i = 1 \text{ TO } N-1 \\
\quad A[t+1,i] = (A[t,i-1] + A[t,i] + A[t,i+1]) / 3
\]
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 \( N-1 \)
\[ A[t+1,i] = \frac{(A[t,i-1]+A[t,i]+A[t,i+1])}{3} \]
Motivation

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

Example

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

For $t = 0$ to $T-1$

For $i = 1$ to $N-1$

$$A[t+1,i] = \frac{A[t,i-1] + A[t,i] + A[t,i+1]}{3}$$

Overlapped Tiling

For $t = 0$ to $T-1$

For $i = 1$ to $N-1$

$$A[t+1,i] = \frac{A[t,i-1] + A[t,i] + A[t,i+1]}{3}$$
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

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

\[
\text{FOR } t = 0 \text{ TO } T-1 \\
\text{FOR } i = 1 \text{ TO } N-1 \\
A[t+1,i]=(A[t,i-1]+A[t,i]+A[t,i+1])/3
\]
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

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

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 $N-1$
    $A[t+1,i] = (A[t,i-1] + A[t,i] + A[t,i+1])/3$

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)

\[
\text{FOR } t = 0 \text{ TO } T-1 \\
\text{FOR } i = 1 \text{ TO } N-1 \\
A[t+1,i] = (A[t,i-1] + A[t,i] + A[t,i+1])/3
\]

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

Split Tiling (no size assumptions)

\[
\text{FOR } t = 0 \text{ TO } T-1 \\
\text{FOR } i = 1 \text{ TO } N-1 \\
A[t+1,i] = (A[t,i-1] + A[t,i] + A[t,i+1])/3
\]

Phase 3: Then, all of the orange shaded regions can be executed concurrently (next)
Stencils on Vector-SIMD Processors

- 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

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

---

Data Layout Transformation

- 1D vector in memory $\Leftrightarrow$ (b) 2D logical view of same data
- (c) Transposed 2D array moves interacting elements into same slot of different vectors $\Leftrightarrow$ (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 $tt = \text{time tile size}...$
  - Execute upright tiles in parallel
  - Execute inverted tiles in parallel
- Upright tile size increases with time tile size
Split Tiling: DLT View

- 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

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

Nested Split Tiling

- 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

- 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
Dependence Summary Graph (DSG)

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

Computing Slopes

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

$$\rho_L(C) = \sum_c \frac{\delta_L}{\delta_T} = \frac{2}{1} = 2$$
$$\rho_U(C) = \sum_c \frac{\delta_U}{\delta_T} = \frac{-2}{1} = -2$$
Computing Slopes

- For each dimension $d$ of the stencil...
  - Lower bound slope $\alpha_d$ is maximum cycle ratio
  - Upper bound slope $\beta_d$ is minimum cycle ratio

$$\alpha_d = \max(\rho_L(C)) \forall C \in DSG = 2$$

$$\beta_d = \min(\rho_U(C)) \forall C \in DSG = -2$$

Computing Offsets

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

```plaintext
for (tt=...){
    for (ii=...){
        for (t=...){
            for (i=ii+oLF1+\alpha L*(t-tt); i<ii+TU+oUF1+\beta U*(t-tt); ++i)
                f1: a1[i] = 0.33*(a0[i-1]+a0[i ]+a0[i+1]);
        }
    }
}}

for (i=ii+oLF2+\alpha L*(t-tt); i<ii+TU+oUF2+\beta U*(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

Lower Bound Constraints
\[ ii + o_{L}^{f1} + \alpha \cdot t \leq ii + o_{L}^{f2} + \alpha \cdot t - 1 \]
\[ ii + o_{L}^{f2} + \alpha \cdot (t-1) \leq ii + o_{L}^{f1} + \alpha \cdot t - 1 \]

Upper Bound Constraints
\[ ii + T_{U} + o_{U}^{f1} + \beta \cdot t \geq ii + T_{U} + o_{U}^{f2} + \beta \cdot t + 1 \]
\[ ii + T_{U} + o_{U}^{f2} + \beta \cdot (t-1) \leq ii + T_{U} + o_{U}^{f1} + \beta \cdot t + 1 \]

Simplify to a system of difference constraints

Solve with Bellman-Ford algorithm

Lower Bound Constraints
\[ o_{L}^{f1} - o_{L}^{f2} \leq -1 \]
\[ o_{L}^{f2} - o_{L}^{f1} \leq -\alpha - 1 \]

Upper Bound Constraints
\[ o_{U}^{f2} - o_{U}^{f1} \leq -1 \]
\[ o_{U}^{f1} - o_{U}^{f2} \leq -\beta - 1 \]

Bellman-Ford

Lower Bound Offsets
\[ o_{L}^{f1} = -1 \]
\[ o_{L}^{f2} = 0 \]

Upper Bound Offsets
\[ o_{U}^{f1} = 1 \]
\[ o_{U}^{f2} = 0 \]
Stencils on Multicore CPU: Performance

![Graph showing performance comparison between different tools]

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
Stencils on GPU: Performance

Nvidia GTX 580

GFlop/s

jac-1d-3  heat-1d  jac-2d-9  heat-2d  lapl-2d  grad-2d  jac-3d-7  heat-3d

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)
Stencils

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

**High arithmetic intensity triad**
```c
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
```
Problem: Performance does not scale with arithmetic intensity!
Register reuse

Page 76 of 226
Register reuse
Contributions

Identified Problem:
- Register reuse for stencil computations

Solution:
- Exploit associativity & commutativity to increase data-locality
Contributions

- Identified Problem:
  - Register reuse for stencil computations

- Solution:
  - Exploit associativity & commutativity to increase data-locality

- Cost model

Contributions

- Identified Problem:
  - Register reuse for stencil computations

- Solution:
  - Exploit associativity & commutativity to increase data-locality

- Cost model

- Experimental results
Gather-Gather

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

1. $for$ $(i=k; i<N-k; i++)$
2. $for$ $(j=k; j<N-k; j++)$
3. $for$ $(ii=-k; ii<=k; ii++)$
4. $for$ $(jj=-k; jj<=k; jj++)$
5. $OUT[i][j] +=$
6. $IN[i+ii][j+jj]*C[ii][jj]$

Scatter-Scatter

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

1. $for$ $(i=k; i<N-k; i++)$
2. $for$ $(j=k; j<N-k; j++)$
3. $for$ $(ii=-k; ii<=k; ii++)$
4. $for$ $(jj=-k; jj<=k; jj++)$
5. $OUT[i-ii][j-jj] +=$
6. $IN[i][j]*C[ii][jj]$
Gather-Scatter

- 1 reads from \( IN \)
- \( w - 1 \) reads from \( OUT \)
- \( w \) write to \( OUT \)
- \( w + 1 \) registers

\[
\begin{align*}
  &\text{for } (i=1; i<N-1; i++) \\
  &\text{for } (j=1; j<N-1; j++) \\
  &t1 = t2 \; \text{// } \text{IN}[i][j-1] \\
  &t2 = t3 \; \text{// } \text{IN}[i][j] \\
  &t3 = \text{IN}[i][j+1] \\
  &\text{OUT}[i-1][j] = t1 + t2 + t3 \\
  &\text{OUT}[i][j] = t1 + t2 + t3 \\
  &\text{OUT}[i+1][j] = t1 + t2 + t3
\end{align*}
\]

Scatter-Gather

- \( w \) reads from \( IN \)
- 0 reads from \( OUT \)
- 1 write to \( OUT \)
- \( w + 1 \) registers

\[
\begin{align*}
  &\text{for } (i=1; i<N-1; i++) \\
  &\text{for } (j=1; j<N-1; j++) \\
  &x = \text{IN}[i-1][j] + \\
  &\quad \text{IN}[i][j] + \text{IN}[i+1][j] \\
  &t1 = t2 + x \\
  &t2 = t3 + x \\
  &t3 = x \\
  &\text{OUT}[i][j-1] = t1
\end{align*}
\]
Multidimensional Retiming

```plaintext
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

```c
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]
  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

```c
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:

```c
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

Loop bounds must be affine

Arrays and scalars only, no pointers
Applicability

- Loop bounds must be affine
- Arrays and scalars only, no pointers
- Access functions do not need to be affine
- 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
6. Semantics preserved when using an associative & commutative operator
   - for direct convolutions
   - for sum-of-product stencils
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]

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=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]

(*Page 115 of 226*)

Dimension Lifted Transposition (CC’11)
Gradient Edge Detection (2d, 97-point)

![Graph showing performance comparison between different compilers and benchmarks.]

Compiler

i7-4770K, ICC 13.1.3

Page 117 of 226

Synthetic Benchmarks Performance

![Graph showing performance comparison between different benchmarks and architectures.]

Page 118 of 226
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
Inevati3 $2D$, 20-point stencil for gradient edge detection
Conclusion

- High order stencils had low performance
  - Unable to reuse registers
Conclusion

- High order stencils had low performance
  - Unable to *reuse registers*
- Solved by reordering computation
  - Exploit *associativity and commutativity*
  - Formalization and cost model from retiming
- Stencil/s maintained in higher order stencils
  - Allows scientists to use higher order stencils efficiently
Goal: fast, automated resolution of PDEs

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)

Stack of optimizing compilers

Faster code than you can reasonably write “by hand”

\[ \int_k \nabla \cdot \rho p \, dx \rightarrow \text{MAGIC} \rightarrow \text{fast code} \]

This part of the talk

\[ \int_k \nabla \cdot \rho p \, dx \rightarrow \text{MAGIC} \]

- from DSL for PDEs to loop chains
- fast code
- Tiling for unstructured meshes
- fast code
- COFFEE: expression compiler

THIS PART’s MESSAGE (philosophy):

- Getting the abstraction right is key in designing 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

\[ \phi, p = \text{Function}(\text{mesh}, \ldots) \]  

\[ \text{while not convergence:} \]  

\[ \{ \]  

\[ \phi \leftarrow \frac{dt}{2} \cdot p \]  

\[ \text{if } \ldots : \]  

\[ p \leftarrow (\text{assemble}(dt \cdot \text{inner}(\text{nabla_grad}(v), \ldots)) \cdot dx) \]  

\[ \text{else:} \]  

\[ \text{solve}(\ldots) \]  

\[ \phi \leftarrow \frac{dt}{2} \cdot p \]  

\[ \} \]  

The resulting non-affine parallel-loops chain

\[ \text{while not convergence:} \]  

\[ \{ \]  

\[ \text{forall cells} \]  

\[ \ldots \]  

\[ \text{for } i \]  

\[ \text{for } j \]  

\[ \ldots \text{expr}(i, j) \]  

A[C[i]] = \ldots

\[ \text{forall edges} \]  

A[E[i]] = \ldots

\[ \ldots \]  

function call !

\[ \text{forall cells} \]  

\[ \ldots \]  

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 !
}

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

1. Seed (shared) set partitioning

forall edges
   read local data
   increment adjacent vertices

forall cells
   read adjacent vertices
   write local data

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

Generalized sparse tiling example

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

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

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!

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!

- Context: automated code
  generation for generic assembly operators; that is,
  “we abstract from the specific equation and discretization!”
Motivating Examples - 1

for (int ip = 0; ip < m; ++ip) {
    for (int j = 0; j < n; ++j) {
        for (int k = 0; k < o; ++k) {
        }
    }
}

Motivating Examples - 2

for (int ip = 0; ip < m; ++ip) {
    for (int j = 0; j < n; ++j) {
        for (int k = 0; k < o; ++k) {
        }
    }
}

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

Mass matrix operator

Helmholtz operator
Motivating Examples - 3

```c
for (int ip = 0; ip < m; ++ip) {
    ...
    for (int j = 0; j < n; ++j) {
        for (int k = 0; k < o; ++k) {
        }
    }
}
```

What should we do with such expressions?

```c
for (int ip = 0; ip < m; ++ip) {
    ...
    for (int j = 0; j < n; ++j) {
        for (int k = 0; k < o; ++k) {
        }
    }
}
```

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

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

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

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

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 $j$
    $\text{TMP}[j] = (E[i][j]*\beta + F[i][j]*\gamma)$
for $j$
    for $k$
        $A[j][k] += B[i][j] * C[i][k] + \text{TMP}[j] + (B[i][j] * D[i][k]) * \alpha$

for $i$
for $j$
    $\text{TMP}[j] = (E[i][j]*\beta + F[i][j]*\gamma)$
for $j$
    for $k$
        $A[j][k] += B[i][j] * C[i][k] + \text{TMP}[j] + B[i][j] * (D[i][k] * \alpha)$

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

for $i$
for $j$
    $\text{TMP}[j] = (E[i][j]*\beta + F[i][j]*\gamma)$
for $j$
    for $k$
        $\text{TMP2}[k] = (C[i][k] + D[i][k] * \alpha)$
for $j$
    for $k$
        $A[j][k] += B[i][j] * \text{TMP2}[k] + \text{TMP}[j]$
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.
Optimizing for ILP - register reuse

for i
  ... hoisted stuff ...
for j
  for k
    \( A[j][k] += B[i][j] \times TMP2[k] + TMP[j] \)

for i
  ... hoisted stuff ...
for j
  for k
    \( A[j][k] += B[i][j] \times TMP2[k] \)
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

\[
\begin{array}{ccc}
(0,0) & (0,1) & (0,2) \\
(1,0) & (1,1) & (1,2) \\
(2,0) & (2,1) & (2,2) \\
\end{array}
\]

Modified layout: 3x4

\[
\begin{array}{ccc}
(0,0) & (0,1) & (0,2) \\
(1,0) & (1,1) & (1,2) \\
(2,0) & (2,1) & (2,2) \\
\end{array}
\]

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]

TOT = 2 mem loads

Optimizing for ILP - specialized SIMDization
Storage layout can be restored with a few vector shuffles
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
  • …
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-Well-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:

\[
A_{A3A} = \frac{1}{2} \left( X_{ce,af} Y_{ae,cf} + X_{ce,aJ} Y_{ae,cJ} + X_{ce,aE} Y_{ae,cE} 
+ X_{ce,aF} Y_{ae,cF} + X_{ae,CF} Y_{ae,cF} + X_{ae,eC} Y_{ae,cE} \right)
\]

\[
X_{ce,af} = t_{ij}^c t_{ij}^a Y_{ae,cf} = \langle ab|ek\rangle \langle cb|fk\rangle
\]

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

---

**Time Crunch in Quantum Chemistry**

**Two major bottlenecks in computational chemistry**

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

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

<table>
<thead>
<tr>
<th>Theory</th>
<th>#Terms</th>
<th>#F77Lines</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCD</td>
<td>11</td>
<td>3209</td>
<td>1978</td>
</tr>
<tr>
<td>CCSD</td>
<td>48</td>
<td>13213</td>
<td>1982</td>
</tr>
<tr>
<td>CCSDT</td>
<td>102</td>
<td>33932</td>
<td>1988</td>
</tr>
<tr>
<td>CCSDTQ</td>
<td>183</td>
<td>79901</td>
<td>1992</td>
</tr>
</tbody>
</table>
**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

<table>
<thead>
<tr>
<th>Complexity of methods</th>
<th>Our Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation takes months</td>
<td>Tensor Contraction Engine</td>
</tr>
<tr>
<td>Experimentation required to develop new methods</td>
<td>• Tensor contraction expressions as input</td>
</tr>
<tr>
<td></td>
<td>• (Fortran) source code as output</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Complexity of computers</th>
<th>Generated code increases productivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different architectures have significantly different performance characteristics</td>
<td>Generate optimized code for target/Optimize generated code for target</td>
</tr>
</tbody>
</table>

### 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

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

- 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

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

- Bulk of computation in tensor contractions

---

High-Level Language for Tensor Contraction Expressions

```plaintext
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
```
Tensor Contraction Expression

- Tensor:
  - multi-dimensional array

\[ t^{ab}_{ij} \rightarrow t[a,b,i,j] \]

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

\[ t^i_j = \sum_{a,b} t^{a}_{b} v^{b}_{aj} \rightarrow r[i,j] = \sum[t[a,b] \cdot v[b,i,a,j],{a,b}] \]

for \( i=1 \) to \( N_i \)

for \( j=1 \) to \( N_j \)

for \( a=1 \) to \( N_a \)

for \( b=1 \) to \( N_b \)

\[ r[i,j] \leftrightarrow t[a,b] \cdot v[b,i,a,j] \]

---

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

- Tensor:
  - multi-dimensional array

\[ \sum[\text{products of large arrays}] \]

\[ r[i,j] += t[a,b] \cdot v[b,i,a,j] \]

for \( b=1 \) to \( N_b \)

for \( i=1 \) to \( N_i \)

\[ r[i,j] = \sum[t[a,b] \cdot v[b,i,a,j],{a,b}] \]

Page 181 of 226

---

Page 182 of 226
### Multi-Level Optimization Framework

<table>
<thead>
<tr>
<th>High-Level Algebraic Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallelization and Data Locality Optimizations</td>
</tr>
<tr>
<td>Kernel Functions Optimization</td>
</tr>
<tr>
<td>Runtime Framework</td>
</tr>
</tbody>
</table>

### Algebraic Transformations: Operation Minimization

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

- Requires \(4 \times N^{10}\) operations if indices \(a-l\) have range \(N\)
- Using associative, commutative, distributive laws acceptable
Algebraic Transformations: Operation Minimization

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

\[ S_{abij} = \sum_{c,d,e,f,k,l} A_{acik} C_{dfjk} B_{befl} D_{cdel} \]
Algebraic Transformations: Operation Minimization

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

- Requires \(4 \times N^{10}\) operations if indices \(a-l\) have range \(N\)
- Using associative, commutative, distributive laws acceptable
- Optimal formula sequence requires only \(6 \times N^6\) 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)
- \(V >> O\)

\[ V_{bcij} = t^b_i s^b_j \rightarrow O^2 V^2\] ops
\[ r^b_i = \sum_{i,j} V_{bcij} f^j_c \rightarrow 2O^2 V^2\] ops

\[ b^c_i = \sum_{c,j} t^c_i f^j_c s^b_j \rightarrow 3O^2 V^2\] ops

- Reduce the operation count from \(3O^2 V^2\) to \(4O^2 V\)
- Algorithms: dynamic programming (for small cases) and heuristic search (for large cases)
Multi-Term Optimization (Factorization)

- Unoptimized:
  \[ r_{ij}^{ab} = \sum_{c,d} t_{ij}^{cd} v_{cd}^{ab} + \sum_{c,d} u_{ij}^{cd} v_{cd}^{ab} \rightarrow 2O^2V^4 + 3O^2V^4 \text{ ops} \]

- Single-term optimization:
  \[ r_{ij}^{ab} = \left( \sum_{c,d} t_{ij}^{cd} v_{cd}^{ab} \right) s_{ij}^{cd} + \sum_{c,d} u_{ij}^{cd} v_{cd}^{ab} \rightarrow 2O^2V^4 + 2OV^4 + 2O^2V^3 \text{ ops} \]

- Factorization:
  \[ r_{ij}^{ab} = \sum_{c,d} \left( t_{ij}^{cd} s_{ij}^{cd} + u_{ij}^{cd} \right) v_{cd}^{ab} \rightarrow 2O^2V^4 + O^2V^2 \text{ ops} \]

- Improved operation count over single-term optimization.

---

Common Subexpression Elimination

- \( p, q : \text{range } M = O + V \)

\[ I_1^i = \sum_p a_p q_i p \quad v_j^i = \sum_p I_1^i p t_j^p \]
\[ \rightarrow 2OM^2 \text{ ops} \quad \rightarrow 2O^2M \text{ ops} \]

\[ I_2^i = \sum_q a_q t_i^q \quad w_j^i = \sum_p I_2^i p s_j^p \]
\[ \rightarrow 2OM^2 \text{ ops} \quad \rightarrow 2O^2M \text{ ops} \]

\[ w_b^i = \sum_p a_p t_i^p u_b^p \]
\[ \rightarrow 3OVM^2 \text{ ops} \quad \rightarrow 2O^2M \text{ ops} \]

- 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 \(4 \times N^{10}\) operations if indices \(a-l\) have range \(N\)
- Optimized form requires only \(6 \times N^{6}\) operations
  \[ T_1(b,c,d,f) = \sum_{e,l} B(b,e,f,l)D(c,d,e,l) \]
  \[ T_2(b,c,j,k) = \sum_{d,l} T_1(b,c,d,f)C(d,f,j,k) \]
  \[ S(a,b,i,j) = \sum_{c,k} T_2(b,c,j,k)A(a,c,i,k) \]

- 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)

\[ T_1_{bdf} = \sum_{c,l} B_{bdf}D_{c,dl} \]
\[ T_2_{bcjk} = \sum_{a,j} T_1_{bdf}C_{dfjk} \]
\[ S_{abij} = \sum_{c,k} T_2_{bcjk}A_{acik} \]

Formula sequence
Memory Minimization: Compute by Parts (Loop Fusion)

\[
T_{1bcdf} = \sum_{e, f} B_{befl} D_{cdef}
\]
\[
T_{2bcjk} = \sum_{d, f, j} T_{1bcdf} C_{dfjk}
\]
\[
S_{abij} = \sum_{c, k} T_{2bcjk} A_{acik}
\]

Formula sequence | Unfused code
---|---

\[
T_{1bcdf} = 0; T_{2bcdf} = 0; S = 0
\]
for \(b, c, d, e, f, l\)

\[
T_{1bcdf} += B_{befl} D_{cdef}
\]
for \(b, c, d, f, j, k\)

\[
T_{2bcjk} += T_{1bcdf} C_{dfjk}
\]
for \(a, b, c, i, j, k\)

\[
S_{abij} += T_{2bcjk} A_{acik}
\]

Memory Minimization: Compute by Parts (Loop Fusion)

\[
T_{1bcdf} = \sum_{e, f} B_{befl} D_{cdef}
\]
\[
T_{2bcjk} = \sum_{d, f, j} T_{1bcdf} C_{dfjk}
\]
\[
S_{abij} = \sum_{c, k} T_{2bcjk} A_{acik}
\]

Formula sequence | Unfused code (Partially) Fused code
---|---|---

\[
T_{1f} = 0; T_{2f} = 0
\]
for \(b, c\)

\[
T_{1f} += B_{befl} D_{cdef}
\]
for \(d, e, f, l\)

\[
T_{2f} += T_{1f} C_{dfjk}
\]
for \(d, f, j, k\)

\[
S_{abij} += T_{2f} A_{acik}
\]
for \(a, i, j, k\)
Memory Minimization: Loop Fusion

Unfused code

\[
S = 0 \\
\text{for } b, c \\
\text{ } \\
T1f = 0; T2f = 0 \\
\text{for } d, e, f, l \\
\text{ } \\
T1fdr += B_{befl} D_{cdel} \\
\text{for } d, e, f, j, k \\
\text{ } \\
T2_{bcjk} += T1_{bcedf} C_{dfjk} \\
\text{for } a, b, c, i, j, k \\
\text{ } \\
S_{abij} += T2_{bcjk} A_{acik}
\]

(Partially) Fused code

\[
S = 0 \\
\text{for } b, c \\
\text{ } \\
T1f = 0; T2f = 0 \\
\text{for } d, f \\
\text{ } \\
T1fdr += B_{befl} D_{cdel} \\
\text{for } e, l \\
\text{ } \\
T2fjkl += T1f_{dr} C_{dfjk} \\
\text{for } a, i, j, k \\
\text{ } \\
S_{abij} += T2f_{jkl} A_{acik}
\]

Fully Fused code

\[
S = 0 \\
\text{for } b, c \\
\text{ } \\
T1f = 0; T2f = 0 \\
\text{for } d, f \\
\text{ } \\
T1fdr += B_{befl} D_{cdel} \\
\text{for } e, l \\
\text{ } \\
T2fjkl += T1f_{dr} C_{dfjk} \\
\text{for } a, i, j, k \\
\text{ } \\
S_{abij} += T2f_{jkl} A_{acik}
\]

- 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

\[
\text{for } a, e, c, f \\
\text{for } i, j \\
\text{X}_{aefc} += T1_{iacj} T1_{ijcf}
\]

\[
\text{for } c, e, b, k \\
T1_{cbejk} = f1(c, e, b, k)
\]

\[
\text{for } a, f, b, k \\
T2_{afbk} = f2(a, f, b, k)
\]

\[
\text{for } b, k \\
Y_{cef} += T1_{cbejk} T2_{afbk}
\]

\[
\text{for } c, e, a, f \\
E += X_{aefc} Y_{cef}
\]

---

<table>
<thead>
<tr>
<th>array</th>
<th>space</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>V^4</td>
<td>V^4O^2</td>
</tr>
<tr>
<td>T1</td>
<td>V^3O</td>
<td>C_{fil} V^3O</td>
</tr>
<tr>
<td>T2</td>
<td>V^3O</td>
<td>C_{r2} V^3O</td>
</tr>
<tr>
<td>Y</td>
<td>V^4</td>
<td>V^2O</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>V^4</td>
</tr>
</tbody>
</table>

a .. f: range V = 1000 .. 3000
i .. k: range O = 30 .. 100
Memory-Minimal Form

for a, f, b, k
\[ T_{2abk} = f_2(a, f, b, k) \]
for c, e
\[ T_{1bk} = f_1(c, e, b, k) \]
for a, f
\[ X += T_{ija} T_{ijc} \]
for b, k
\[ Y += T_{1bk} T_{2abk} \]
\[ E += X Y \]

Array | Space | Time
--- | --- | ---
X | 1 | V^4O^2
T1 | VO | C_{f1}V^3O
T2 | V^3O | C_{f2}V^3O
Y | 1 | V^3O
E | 1 | V^4

Fusion of loops allows reduction of rank of arrays

Redundant Computation Allows Full Fusion

for a, e, c, f
\[ X += T_{ija} T_{ijc} \]
for b, k
\[ T1 = f_1(c, e, b, k) \]
\[ T2 = f_2(a, f, b, k) \]
\[ Y += T1 T2 \]
\[ E += X Y \]

Array | Space | Time
--- | --- | ---
X | 1 | V^4O^2
T1 | 1 | C_{f1}V^3O
T2 | 1 | C_{f2}V^3O
Y | 1 | V^3O
E | 1 | V^4
Tiling to Reduce Recomputation

for a!, e!, c!, f!

Loop over tiles

for a, e, c, f
for i, j

\[ X_{aef} := T_{iae} T_{ijcf} \]
for b, k
for c, e

\[ T_{1ce} = f_{1}(c, e, b, k) \]
for a, f

\[ T_{2af} = f_{2}(a, f, b, k) \]
for c, e, a, f

\[ Y_{ceaf} := T_{1ce} T_{2af} \]
for c, e, a, f

\[ E := X_{aef} Y_{ceaf} \]

Tiling further improves locality

<table>
<thead>
<tr>
<th>array</th>
<th>space</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>B^4</td>
<td>V^4O^2</td>
</tr>
<tr>
<td>T1</td>
<td>B^2</td>
<td>C_{f1}V^3O</td>
</tr>
<tr>
<td>T2</td>
<td>B^2</td>
<td>C_{f2}V^3O</td>
</tr>
<tr>
<td>Y</td>
<td>B^4</td>
<td>V^5O</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>V^4</td>
</tr>
</tbody>
</table>

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:** $(a,b,c) \rightarrow (c,b,a)$
- **GEMM:** $B(a,i) \times A(cb,a) \rightarrow T1(ic,b);$ with $(t,t)$
- **GEMM:** $C(b,j) \times T1(ic,b) \rightarrow E(j,ic);$ with $(t,t)$
- **Reshape E:** $(j,i,c) \rightarrow (i,j,c)$

2: 

- **GEMM:** $(a,bc)xB(a,i) \rightarrow T1(bc,i);$ with $(t,n)$
- **GEMM:** $T1(b,ci)xC(b,j) \rightarrow E(ci,j);$ with $(t,n)$
- **Reshape E:** $(c,i,j) \rightarrow (i,j,c)$

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
Considering CCSD Iterations

\[ v_{oo}v_{h}^{h_{1}k_{2}} = (c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}} + a_{m}mmm_{q_{4}^{l_{2}}}) \]

\[ v_{o}o_{v}v_{p}^{h_{1}p_{2}} = (c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}} + a_{m}mmm_{q_{4}^{l_{2}}}) \]

\[ v_{o}o_{v}v_{p}^{h_{1}p_{2}} = (c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}} + a_{m}mmm_{q_{4}^{l_{2}}}) \]

\[ v_{o}o_{v}v_{p}^{h_{1}p_{2}} = (c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}} + a_{m}mmm_{q_{4}^{l_{2}}}) \]

\[ \text{residual}_{h_{1}p_{2}} = 0.25 \times (t_{v}v_{h_{1}p_{2}} + f_{o}o_{h_{1}p_{2}}) - 0.25 \times (v_{o}o_{v}v_{h_{1}p_{2}} + t_{v}v_{h_{2}}) + 0.25 \times f_{v}v_{p_{1}} \]

\[ + 0.25 \times (t_{v}v_{h_{1}p_{2}} + f_{o}o_{h_{1}p_{2}}) - 0.25 \times (f_{o}o_{v}v_{h_{1}p_{2}} + t_{v}v_{h_{2}}) + 0.25 \times f_{v}v_{h_{1}} \]

\[ - 0.25 \times (t_{v}v_{h_{2}p_{3}} + f_{o}o_{h_{2}p_{3}} + v_{o}o_{v}v_{h_{2}p_{3}}) \]

\[ + 0.25 \times (t_{v}v_{h_{2}p_{3}} + f_{o}o_{h_{2}p_{3}} + v_{o}o_{v}v_{h_{2}p_{3}}) - 0.125 \times (v_{o}o_{v}v_{h_{2}p_{3}} + v_{o}o_{v}v_{h_{2}p_{3}}) - 0.125 \times (t_{v}v_{h_{2}p_{3}} + f_{o}o_{h_{2}p_{3}} + v_{o}o_{v}v_{h_{2}p_{3}}) \]

\[ - 0.125 \times (t_{v}v_{h_{2}p_{3}} + f_{o}o_{h_{2}p_{3}} + v_{o}o_{v}v_{h_{2}p_{3}}) - 0.25 \times t_{v}v_{h_{1}p_{2}} + t_{v}v_{h_{2}p_{3}} \]

\[ + 0.125 \times (t_{v}v_{h_{2}p_{3}} + f_{o}o_{h_{2}p_{3}} + v_{o}o_{v}v_{h_{2}p_{3}}) + 0.125 \times (v_{o}o_{v}v_{h_{2}p_{3}} + f_{o}o_{v}v_{h_{2}p_{3}}) \]

... Other computations that modify tensors t_{vo} etc.

Optimized CCSD T1

\[ it_{1, q_{1}^{l_{1}}} = (c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}}) \]

\[ it_{2, q_{1}^{l_{1}}} = (c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}}) \]

\[ v_{o}o_{v}v_{p}^{h_{1}p_{2}} = (c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}}) \]

\[ it_{2, q_{1}^{l_{1}}} = (c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}}) \]

\[ it_{3, q_{1}^{l_{1}}} = (c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}}) \]

\[ it_{4, q_{1}^{l_{1}}} = (c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}}) \]

\[ it_{5, q_{1}^{l_{1}}} = (c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}}) \]

\[ it_{6, q_{1}^{l_{1}}} = (c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}}) \]

\[ \text{residual}_{h_{1}p_{2}} = 0.25 \times f_{v}v_{h_{1}p_{2}} - 0.25 \times (f_{o}o_{h_{1}p_{2}} + t_{v}v_{p_{2}}) + 0.25 \times f_{v}v_{h_{1}p_{2}} \]

\[ + 0.125 \times (t_{v}v_{h_{1}p_{2}} + c_{m}q_{4}^{l_{1}} + c_{m}q_{4}^{l_{2}} + c_{m}q_{3}^{h_{1}} + c_{m}q_{3}^{h_{2}}) \]

\[ - 0.25 \times (t_{v}v_{h_{1}p_{2}} + f_{o}o_{h_{1}p_{2}} + t_{v}v_{h_{2}}) - 0.125 \times (t_{v}v_{h_{1}p_{2}} + f_{o}o_{h_{1}p_{2}} + t_{v}v_{h_{2}}) \]

\[ - 0.125 \times (t_{v}v_{h_{1}p_{2}} + f_{o}o_{h_{1}p_{2}} + t_{v}v_{h_{2}}) + 0.25 \times (t_{v}v_{h_{1}p_{2}} + f_{o}o_{h_{1}p_{2}} + t_{v}v_{h_{2}}) \]

\[ - 0.25 \times (t_{v}v_{h_{1}p_{2}} + f_{o}o_{h_{1}p_{2}} + t_{v}v_{h_{2}}) + 0.25 \times (t_{v}v_{h_{1}p_{2}} + f_{o}o_{h_{1}p_{2}} + t_{v}v_{h_{2}}) \]

... Other computations that modify tensors t_{vo} etc.
Impact of Optimizations

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

<table>
<thead>
<tr>
<th>Iteration Count</th>
<th>Operation Count</th>
<th>Reduction Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (Brueckner)</td>
<td>Separated steps</td>
<td>(5.36 \times 10^{12})</td>
</tr>
<tr>
<td></td>
<td>Combined Opt</td>
<td>(1.51 \times 10^{12})</td>
</tr>
<tr>
<td>10</td>
<td>Separated steps</td>
<td>(5.63 \times 10^{12})</td>
</tr>
<tr>
<td></td>
<td>Combined Opt</td>
<td>(2.26 \times 10^{12})</td>
</tr>
</tbody>
</table>

**CCSD T2**

<table>
<thead>
<tr>
<th>Iteration Count</th>
<th>Expanded MO Tensors</th>
<th>Operation Count</th>
<th>Reduction Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Separated Steps</td>
<td>(2.85 \times 10^{14})</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Combined Opt</td>
<td>(1.93 \times 10^{13})</td>
<td>14.75</td>
</tr>
<tr>
<td>10</td>
<td>Separated Steps</td>
<td>(4.22 \times 10^{14})</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Combined Opt</td>
<td>(1.67 \times 10^{14})</td>
<td>2.53</td>
</tr>
</tbody>
</table>

---

**Experiment 2**

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

\[
a_{rs}^{pq} = \sum_{z} u_{z}^{pq} u_{rs}^{z}
\]

<table>
<thead>
<tr>
<th>Equation</th>
<th>Number of terms</th>
<th>Expanded MO Integrals</th>
<th>AO Integrals</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCSD E</td>
<td>5</td>
<td>v_vvoo</td>
<td>a_mmmm</td>
</tr>
<tr>
<td>CCSD T1</td>
<td>26</td>
<td>v_vvov, v_ovvo, v_ovov, v_vvoo, v_ovoo</td>
<td>a_mmmm</td>
</tr>
<tr>
<td>CCSD T2</td>
<td>57</td>
<td>v_ooovo, v_ooov, v_ovvo, v_oovv, v_ovov, v_ovo0, v_vvvo, v_vvvv, v_vvoo</td>
<td>a_mmmm</td>
</tr>
</tbody>
</table>

- Index ranges \(O = 100, V = 5000, M = O + V, Z = 10 (O + V)\)
## Impact of Optimizations

<table>
<thead>
<tr>
<th>Iteration Count</th>
<th>Optimization</th>
<th>Operation Count</th>
<th>Reduction Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Separated Optimization</td>
<td>1.15e+20</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Combine AO-to-MO and CCSD</td>
<td>8.77e+19</td>
<td>1.31</td>
</tr>
<tr>
<td></td>
<td>Cholesky-AO and AO-to-MO</td>
<td>8.39e+19</td>
<td>1.37</td>
</tr>
<tr>
<td></td>
<td>Combining all three steps</td>
<td>4.87e+18</td>
<td>23.70</td>
</tr>
<tr>
<td>10</td>
<td>Separated Optimization</td>
<td>2.77e+20</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Combine AO-to-MO and CCSD</td>
<td>2.52e+20</td>
<td>1.10</td>
</tr>
<tr>
<td></td>
<td>Cholesky-AO and AO-to-MO</td>
<td>2.41e+20</td>
<td>1.15</td>
</tr>
<tr>
<td></td>
<td>Combining all three steps</td>
<td>4.75e+19</td>
<td>5.83</td>
</tr>
</tbody>
</table>

## Space-time Trade-offs

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

Hand-coded solution (single algorithm)

TCE explores many algorithms, selects best

\[
A^3 A = \frac{1}{2} \left( X_{\alpha,\alpha,\alpha} Y_{\alpha,\alpha,\alpha} + X_{\alpha,\alpha,\alpha} Y_{\alpha,\alpha,\alpha} + X_{\alpha,\alpha,\alpha} Y_{\alpha,\alpha,\alpha} \right) + X_{\alpha,\alpha,\alpha} Y_{\alpha,\alpha,\alpha} + X_{\alpha,\alpha,\alpha} Y_{\alpha,\alpha,\alpha} + X_{\alpha,\alpha,\alpha} Y_{\alpha,\alpha,\alpha}
\]

\[
X_{\alpha,\alpha,\alpha} = \epsilon_{ij}^{\alpha \alpha} \epsilon_{ij}^{\alpha \alpha} \\
Y_{\alpha,\alpha,\alpha} = \langle ab | ck \rangle \langle cb | \alpha \alpha \rangle
\]
Experiments: Index Permute + BLAS

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

- 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%

<table>
<thead>
<tr>
<th>Unoptimized (sec.)</th>
<th>Optimized (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEMM</td>
<td>Index Permutation</td>
</tr>
<tr>
<td></td>
<td>GEMM</td>
</tr>
<tr>
<td>10.06</td>
<td>2.58</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Unoptimized (sec.)</th>
<th>Optimized (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEMM</td>
<td>Index Permutation</td>
</tr>
<tr>
<td></td>
<td>GEMM</td>
</tr>
<tr>
<td>12.23</td>
<td>7.74</td>
</tr>
</tbody>
</table>
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:
Further Reading

• Stencils:


• Stencils (continued):


Further Reading

- Irregular codes (finite-elements code generation, runtime compilation, ...):

Further Reading

- Tensor Contraction Engine (TCE):
Further Reading

• Tensor Contraction Engine (TCE) -- continued:

