Domain-Specific Abstractions and Compiler Transformations

P. (Saday) Sadayappan
The Ohio State University
## Acknowledgements

<table>
<thead>
<tr>
<th>Collaborators</th>
<th>Ph.D. Students</th>
<th>Funding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gerald Baumgartner (LSU)</td>
<td>Muthu Baskaran</td>
<td>DoE</td>
</tr>
<tr>
<td>Jason Cong (UCLA)</td>
<td>Uday Bondhugula</td>
<td>NSF</td>
</tr>
<tr>
<td>Franz Franchetti (CMU)</td>
<td>Jim Dinan</td>
<td>DARPA</td>
</tr>
<tr>
<td>Robert Harrison (Stony Brook)</td>
<td>Xiaoyang Gao</td>
<td></td>
</tr>
<tr>
<td>So Hirata (U. Illinois)</td>
<td>Albert Hartono</td>
<td></td>
</tr>
<tr>
<td>Jarek Nieploha (PNNL)</td>
<td>Justin Holewinski</td>
<td></td>
</tr>
<tr>
<td>Marcel Nooijen (Waterloo)</td>
<td>Sriram Krishnamoorthy</td>
<td></td>
</tr>
<tr>
<td>Srin Parthasarathy (OSU)</td>
<td>Qingda Lu</td>
<td></td>
</tr>
<tr>
<td>Louis-Noel Pouchet (UCLA)</td>
<td>Mohammad Arafat</td>
<td></td>
</tr>
<tr>
<td>Russ Pitzer (OSU, Chem)</td>
<td>Tom Henretty</td>
<td></td>
</tr>
<tr>
<td>Dan Quinlan (LLNL)</td>
<td>Pai-Wei Lai</td>
<td></td>
</tr>
<tr>
<td>J. Ramanujam (LSU)</td>
<td>Qingpeng Niu</td>
<td></td>
</tr>
<tr>
<td>Nasko Routtev (OSU)</td>
<td>Samyam Rajbhandari</td>
<td></td>
</tr>
<tr>
<td>Vivek Sarkar (Rice)</td>
<td>Mahesh Ravishankar</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Kevin Stock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sanket Tavarageri</td>
<td></td>
</tr>
</tbody>
</table>
Why Domain-Specific Frameworks?

• Heterogeneity creates a huge software challenge
  – Multiple implementations for different targets, e.g., OpenMP (multi-core), OpenCL (GPU), VHDL (FPGA)
  – Future energy-limited systems even more complex to program: physical 2D/3D locality for data movement

• Is Write-Once-Execute-Anywhere feasible?
  – Too daunting a challenge for arbitrary computations expressed in general-purpose languages
  – With domain-specific abstractions, it is much more hopeful that such a goal can be achieved
Domain-Specific Abstractions

- Stencil computations
- Tensor expressions
- Affine computations
Embedded DSL 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
Related Work

- 10+ publications over the last few years on optimizing stencil computations
- Some stencil DSLs and stencil compilers
  - Pochoir (MIT), PATUS (Univ. Basel)
  - Generate code for multi-core CPUs only; cannot handle multi-stencil computations, e.g. CDSC imaging pipeline
- Frameworks for building DSLs
  - SEJITS (LBL); Liszt, OptiML, OptiQL, ... (Stanford)
  - Our focus is complementary: developing abstraction-specific compiler transformations matched to performance-critical characteristics of target architecture
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

**Inefficiency: Each element of b is loaded twice**
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

for (i = 0; i < N; ++i)
a[i]=b[i-1]+b[i]+b[i+1];
Stencil-Specific Transformation: Evaluation

The chart compares the performance of different benchmarks on various microarchitectures, measured in Gflops. The benchmarks include:

- Phenom
- Core2Quad
- Core i7
- J-1D
- J-2D-5pt
- J-2D-9pt
- J-3D
- Heattut-3D
- FDTD-2D
- Rician-2D

The performance is measured in Gflops for each benchmark on Phenom, Core2Quad, and Core i7 microarchitectures.

Legend:
- Ref.
- DLT
- DLTi
Stencil Compiler for GPUs

• Very different optimization challenges than SIMD
  – Vector-SIMD alignment problem is non-issue for GPUs
  – But constraints on thread block synchronization, thread divergence and limited cache/scratchpad memory

• Overlapped tiling
  – Redundantly compute neighboring cells to avoid inter-thread-block sync, lower communication, and avoid thread divergence

• Details in poster session (and demo of compiler)
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
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
}
Stencil Compiler for GPU: Performance

The diagram shows the performance comparison of different stencil compiler methods for GPU computations. The x-axis represents various tasks such as Rician Denoise 2D, Rician Denoise 3D, Segmentation 3D, TV Update 2D, and TV Update 3D. The y-axis represents the performance in GStencils/sec.

Legend:
- Intel C
- OverTile (Tesla C2050)
- Intel C (Auto-Par)
- OverTile (Tesla K10)
The Tensor Contraction Engine
A Domain-Specific Compiler for Many-Body Methods in Quantum Chemistry

Oak Ridge National Laboratory
David E. Bernholdt, Robert Harrison

Pacific Northwest National Laboratory
Jarek Nieplocha

Louisiana State University
Gerald Baumgartner, J. Ramanujam

Ohio State University
Xiaoyang Gao, Albert Hartono, Sriram Krishnamoorthy, Qingda Lu, Alex Sibiryakov, Russell Pitzer, P. Sadayappan

University of Florida
So Hirata

University of Waterloo
Marcel Nooijen

Supported by NSF and DOE
Time Crunch in Quantum Chemistry

Two major bottlenecks in computational chemistry:
• Very 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 low-level programming for performance, and less time doing science :-(

• Coupled Cluster family of models in electronic structure theory
• Increasing number of terms => explosive increase in code complexity
• Theory is the same, but efficient implementations of higher order models 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>
CCSD Doubles Equation
(Quantum Chemist’s Eye Test Chart :)

\[ \hbar[a,b,i,j] = \text{sum}[t[b,c]*t[i,j,a,c], \{c\}] - \text{sum}[t[k,c]*t[k,b]*t[i,j,a,c], \{c\}] + \text{sum}[t[a,c]*t[i,j,c,b], \{c\}] - \text{sum}[t[k,c]*t[k,a]*t[i,j,c,b], \{c\}] - \text{sum}[t[k,j]*t[i,k,a,b], \{k\}] - \text{sum}[t[k,c]*t[j,i]*t[k,a,b], \{k\}] - \text{sum}[t[k,i]*t[j,k,b,a], \{k\}] - \text{sum}[t[k,c]*t[i,c]*t[j,k,b,a], \{k\}] - \text{sum}[t[i,c]*t[j,d]*v[a,b,c,d], \{c,d\}] + \text{sum}[t[i,c]*t[j,c]*v[a,b,c,d], \{c,d\}] - \text{sum}[t[j,c]*t[i,c]*v[a,b,c,d], \{c,d\}] - \text{sum}[t[j,c]*t[j,i]*v[a,b,c,d], \{c,d\}] + \text{sum}[t[j,c]*t[j,d]*v[a,b,c,d], \{c,d\}] + \text{sum}[t[i,c]*t[j,k,b]*v[a,b,c,d], \{c,d\}] - \text{sum}[t[i,c]*t[j,d]*v[a,b,c,d], \{c,d\}] + \text{sum}[t[i,c]*t[j,i]*v[a,b,c,d], \{c,d\}] - 2*\text{sum}[t[i,c]*t[j,i]*v[a,b,c,d], \{c,d\}] + \text{sum}[t[j,c]*t[i,k,b]*v[a,b,c,d], \{c,d\}] - \text{sum}[t[j,c]*t[j,k,b]*v[a,b,c,d], \{c,d\}] + \text{sum}[t[j,c]*t[j,k,b]*v[a,b,c,d], \{c,d\}] - \text{sum}[t[j,c]*t[j,k,b]*v[a,b,c,d], \{c,d\}]
\]
Tensor Contraction Engine

- Automatic transformation from high-level specification
  - Chemist specifies computation in high-level mathematical form
  - Synthesis system transforms it to efficient parallel program
  - Code is tailored to target machine
  - Code can be optimized for specific molecules being modeled

- Multi-institutional collaboration (OSU, LSU, Waterloo, ORNL, PNNL, U. Florida)

- Two versions of TCE developed
  - a) Full chemistry, but fewer optimizations (Hirata)
  - b) Excluded some details, but sophisticated optimizations
  - Used to implement over 20 models, in latest release of NWChem (a few million lines of synthesized code)
  - First parallel implementation for many of the methods
  - New improved TCE-2 planned

\[
A3A = \frac{1}{2} (\sum_{ce,af} X_{ce,af} Y_{ae,cf} + \sum_{ae,cf} X_{ae,cf} Y_{ae,cf} + \sum_{ce,af} X_{ce,af} Y_{ae,cf})
\]

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

\[
\text{range } V = 3000; \\
\text{range } O = 100; \\
\text{index } a,b,c,d,e,f : V; \\
\text{index } i,j,k : O; \\
\text{mlimit} = 10000000; \\
\text{function } F1(V,V,V,O); \\
\text{function } F2(V,V,V,O); \\
\text{procedure } P(\text{in } T1[O,O,V,V], \text{in } T2[O,O,V,V], \text{out } X) = \\
\text{begin} \\
A3A \leftarrow \sum \sum \{ \text{F1}(a,b,e,k) \times \text{F2}(c,f,b,k), \{b,k\} \} \\
\times \sum \sum \{ \text{T1}[i,j,c,e] \times \text{T2}[i,j,a,f], \{i,j\} \}, \\
\{a,e,c,f\} \times 0.5 + ...; \\
\text{end}
\]
for(t=0; t<tmax; t++)  {
    for (j=0; j<ny; j++)
        ey[0][j] = init_f[t];
    for (i=1; i<nx; i++)
        for (j=0; j<ny; j++)
            ey[i][j]=ey[i][j]-0.5*(hz[i][j]-hz[i-1][j]);
    for (i=1; i<nx; i++)
        for (j=0; j<ny; j++)
            ex[i][j]=ex[i][j]-0.5*(hz[i][j]-hz[i][j-1]);
    for (i=0; i<nx; i++)
        for (j=0; j<ny; j++)
            hz[i][j]=hz[i][j]-0.7*(ex[i+1][j]-ex[i][j] + ey[i+1][j]-ey[i][j]);
}

for (t1=0; t1<=floord(2*tmax+ny-2,32); t1++) {
    lb1=max(ceild(t1,2),ceild(32*t1-tmax+1,32));
    ub1=min(min(floord(tmax+ny-1,32),floord(32*t1+ny+31,64)),t1);
    #pragma omp parallel for shared(t1,lb1,ub1) private(t2,t3,t4,t5,t6)
    for (t2=lb1; t2<=ub1; t2++) {
        for (t3=max(ceild(32*t2-ny-30,32),t1-t2); t3<=min(min(floord(32*t2+nx+30,32),
            floord(tmax+nx-1,32)), floord(32*t1-32*t2+nx+31,32));t3++ ) {
            if ((t1 == t2+t3) && (t1 <= floord(64*t2-ny,32))) {
                for (t6=32*t2-ny+1;t6<=min(32*t1-32*t2+31,32*t2-ny+nx);t6++) {
                    hz[-32*t2+t6+ny-1][ny-1]=hz[-32*t2+t6+ny-1][ny-1]- 0.7*(ex[-32*t2+t6+ny-1][ny-1] +1] -
                    ex[-32*t2+t6+ny-1][ny-1]+ey[-32*t2+t6+ny-1 +1][ny-1]-ey[-32*t2+t6+ny-1][ny-1]);
                }
            }
        }
    }
}
Customized Code Generation for Tensor Contractions

• Effective SIMD utilization is increasingly important for high performance on current/emerging processors
• Automatic vectorization by production compilers (even with manual unrolling) often results in performance well under 50% of machine peak
• Customized code generator (using vector intrinsics) for tensor contractions
Approach to Code Generation

Overall Approach
- Explicitly generate vector-intrinsics based code for TC
- Compile generated code using icc/gcc
- Use Machine Learning model to predict performance of generated assembly code
- Explore space of code variants and choose the one with highest predicted performance

Search Space
Vectorized Dimension
Determine how memory is accessed
Loop Permutation
Enable hoisting loads/stores outside inner loops
Unroll-and-Jam
Enable register reuse
Example: Multi-resolution Kernel

\[ R_{ijk} = \sum_{i'j'k'} S_{i'j'k'} X_{i'i} Y_{j'j} Z_{k'k} \]

- Used extensively in MADNESS (Multi-resolution Adaptive Numerical Environment for Scientific Simulation)
- Tensors are small, frequently fitting completely within L1-cache

Low Rank Decomposition

\[ X_{i'i} = \sum_l X_{il} X_{li}' \]
\[ Y_{j'j} = \sum_m Y_{jm} Y_{mj}' \]
\[ Z_{k'k} = \sum_n Z_{kn} Z_{nk}' \]
Example: Multi-resolution Kernel

Kernel

\[ R_{ijk} = \sum_{i,j,k'} S_{i'j'k'} X_{i'i} Y_{j'j} Z_{k'k} \]

- Implemented as a series of six tensor contractions

\[ A_{ljk} = \sum_i S_{ijk} \cdot X_{il}^L \]
\[ B_{lmk} = \sum_j A_{ljk} \cdot Y_{jm}^L \]
\[ C_{lmn} = \sum_k B_{lmk} \cdot Z_{kn}^L \]
\[ D_{lmk'} = \sum_n C_{lmn} \cdot Z_{nk'}^R \]
\[ E_{lj'k'} = \sum_mD_{lmk'} \cdot Y_{mj'}^R \]
\[ R_{i'j'k'} = \sum_l E_{lj'k'} \cdot X_{li'}^R \]
Vectorization Dimensions

Inner loop “j” is good for Vectorization (stride is 0 or 1)

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

Inner loop “i” is bad for vectorization (access stride is N)

```
for k=0; k<N; k++
for j=0; j<N; j++
for i=0; i<N; i++
    C[i][j]+=A[i][k]*B[k][j];
```
Vectorization via Register Transpose

We can vectorize along “i” via use of register transpose

Cost of register transpose often amortizable

for k=0; k<N; k++
for j=0; j<N; i++
for i=0; i<N; j++
C[i][j]+=A[i][k]*B[k][j];
1: procedure IKJ(A_{ki}, B_{jk}, C_{ij})
2:  
3:     for (i ← 0; i < M; i++) do
4:         for (k ← 0; k < K; k+= 4) do
5:             a_0[0 : 3] ← SPLAT(A[k + 0][i])
6:             a_1[0 : 3] ← SPLAT(A[k + 1][i])
7:             a_2[0 : 3] ← SPLAT(A[k + 2][i])
8:             a_3[0 : 3] ← SPLAT(A[k + 3][i])
9:     for (j ← 0; j < N; j+= 4) do
10:         b_0[0 : 3] ← B[j + 0][k : k + 3]
11:         b_1[0 : 3] ← B[j + 1][k : k + 3]
12:         b_2[0 : 3] ← B[j + 2][k : k + 3]
13:         b_3[0 : 3] ← B[j + 3][k : k + 3]
14:         TRANSPOSE(b_0, b_1, b_2, b_3)
15:         c[0 : 3] ← C[i][j : j + 3]
16:         c[0 : 3]+ = a_0[0 : 3] * b_0[0 : 3]
17:         c[0 : 3]+ = a_1[0 : 3] * b_1[0 : 3]
18:         c[0 : 3]+ = a_2[0 : 3] * b_2[0 : 3]
19:         c[0 : 3]+ = a_3[0 : 3] * b_3[0 : 3]
20:         C[i][j : j + 3] ← c[0 : 3]
21:     end for
22: end for
23: end procedure

Contraction
\[ C_{ij} = \sum_k A_{ki} \cdot B_{jk} \]

- Vectorized along \( j \)
- \( B_{jk} \) transposed
- Each element of \( A_{ki} \) is splatted (broadcast) to all elements of a vector register
Multiresolution Kernel Performance

Graphs showing performance metrics for different low ranks and kernels, including ICC, BLAS, and Generated, measured in GFLOPS.
Example: A CCSD Tensor Contraction

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

for (i=0; i<P; i++)
for (j=0; j<Q; j++)
for (k=0; k<R; k++)
for (l=0; l<S; l++)
for (m=0; m<T; m++)
for (n=0; n<U; n++)
\[ C[i][j][k][l] += A[i][m][k][n]*B[j][n][l][m]; \]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>GCC</th>
<th>ICC</th>
<th>Machine Peak</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem double sse</td>
<td>1.406</td>
<td>2.740</td>
<td>10.64</td>
<td>7.579</td>
</tr>
<tr>
<td>Nehalem single sse</td>
<td>1.405</td>
<td>2.642</td>
<td>21.28</td>
<td>13.428</td>
</tr>
<tr>
<td>Sandy Bridge double avx</td>
<td>2.231</td>
<td>4.361</td>
<td>27.2</td>
<td>16.768</td>
</tr>
<tr>
<td>Sandy Bridge single avx</td>
<td>2.255</td>
<td>5.075</td>
<td>54.4</td>
<td>36.937</td>
</tr>
</tbody>
</table>

Performance in GFLOP/s
Domain-Specific Abstractions

- Stencil computations
- Tensor expressions
- Polyhedral framework for affine computations and
Why Polyhedral Compiler Framework?

• Conventional AST-based compiler frameworks not powerful enough to transform imperfectly nested loops effectively
  – Intel icc, PGI pgcc, gcc

• Polyhedral model: mathematical abstraction of affine computations enables powerful transformations
  – Imperfectly nested loops modeled as collection of polyhedra
  – Instance-wise data dependences also modeled as polyhedra
  – Loop transformation equivalent to affine hyperplane schedules

• Automatic parallelization and data locality optimization

```c
for ( t =0; t <=T-1;t++)
{
    for ( i = 1; i < N-2;i++)
    for ( i = 1; i < N-2;i++)
        A[i]=B[i];
}
```

```c
for ( t =0; t <=T-1;t++)
{
    for ( i =2*t+2; i <=2*t+N-2;i++)
    {
        B[-2*t+i]=(A[-2*t+i+1]+A[-2*t+i]+A[-2*t+i-1])/3;
        A[-2*t+i-1]=B[-2*t+i-1];
    }
}
```
Data Dependences and Tiling

for (t = 0; t < T; t++) {
    for (i = 2; i < N - 1; i++)
        b[i] = (a[i-1] + a[i] + a[i+1]) / 3;
    for (j = 2; j < N - 1; j++)
        a[j] = b[j];
}

1D-Jacobi code

Tiling code as-is not legal due to cyclic inter-tile dependences

If N>cachesize, #misses:
no tiling: \(O(N*T/L)\)
tilesize \(B\): \(O(N*T/(L*B))\)
Transformation to Enable Tiling

Original execution ordering makes tiling illegal.

Dependence-preserving reordering of iteration space makes tiling legal.
Transformation to Enable Tiling

```c
for (t0=0;t0<=T-1;t0++) {
    for (t1=2*t0+3;t1<=2*t0+N-2;t1++) {
        b[-2*t0+t1] = (a[-2*t0+t1-1] +
                       a[-2*t0+t1] + a[-2*t0+t1+1])/3;
        a[-2*t0+t1-1] = b[-2*t0+t1-1];
    }
    a[N-2] = b[N-2];
}
```

Peeling, skewing and fusion needed to make tiling legal, i.e. eliminate all cyclic inter-tile dependence
Polyhedral Compiler Transformation

for (i=0; i<N; i++)
{
  for (j=0; j<N; j++)
    for (k=0; k<N; k++) S1;
  for (p=0; p<M; p++) S2;
}

N=4
M=3

- Uniform, powerful abstraction for imperfect loop nests
- Uniform, powerful handling of parametric loop bounds
- Loop transform == Affine hyperplane schedule

=> Arbitrary sequence of transforms == change of affine coeffs.
for(t=0; t<max; t++) {
  for (j=0; j<ny; j++)
    ey[0][j] = init_f[t];
  for (i=1; i<nx; i++)
    for (j=0; j<ny; j++)
      ey[i][j] = ey[i][j] - 0.5 * (hz[i][j] - hz[i-1][j]);
  for (i=1; i<nx; i++)
    for (j=0; j<ny; j++)
      ex[i][j] = ex[i][j] - 0.5 * (hz[i][j] - hz[i][j-1]);
  for (i=0; i<nx; i++)
    for (j=0; j<ny; j++)
      hz[i][j] = hz[i][j] - 0.7 * (ex[i][j+1] - ex[i][j] + ey[i+1][j] - ey[i][j]);
}

for (t1=0; t1<=floor(2*max+ny-2,32); t1++) {
  lb1 = max(ceil(t1,2), ceil(32*t1-max+1,32));
  ub1 = min(min(floor(max+ny-1,32), floor(32*t1+ny+31,64)), t1);
  #pragma omp parallel for shared(t1, lb1, ub1) private(t2,t3,t4,t5,t6)
  for (t2=lb1; t2<=ub1; t2++) {
    for (t3=max(ceil(32*t2-ny-30,32), t1-t2); t3<=min(min(floor(32*t2+nx+30,32),
      floor(max+nx-1,32)), floor(32*t1-32*t2+nx+31,32)); t3++) {
      if ((t1 == t2+t3) && (t1 <= floor(64*t2-ny,32))) {
        for (t6=32*t2-ny+1; t6<=min(32*t1-32*t2+31,32*t2-ny+nx); t6++) {
          hz[-32*t2+t6+ny-1][ny-1] = hz[-32*t2+t6+ny-1][ny-1] - 0.7 * (ex[-32*t2+t6+ny-1][ny-1] +
            ex[-32*t2+t6+ny-1][ny-1] + ey[-32*t2+t6+ny-1 +1][ny-1] - ey[-32*t2+t6+ny-1][ny-1]);
        }
      }
    }
  }
}

... Few hundred lines of output code omitted ...
Challenges and Opportunities

• Domain-specific abstractions can greatly benefit programmers and compilers, but
  – Can a general framework be developed that can effectively transform multiple DSL abstractions for different architectures?
  – What about irregular computations?

• Close collaboration between computer scientists and computational scientists will be critical
  – Mini-Apps from DOE Co-Design; Berkeley Dwarfs

• Interoperability of domain-specific languages and standard programming models is critical
  – Embedded DSLs
Summary

• The ‘power wall’ has made heterogeneous computing essential

• Using domain-specific abstractions is a promising approach to compiler optimization for effective heterogeneous computing
  – Productivity, portability, performance
  – Write once, execute anywhere

• Many challenges remain, but there is good momentum in the high-performance computing community in developing domain-specific approaches
Backup Slides
# Multi-Level Optimization Framework

<table>
<thead>
<tr>
<th>Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-Level Algebraic Transformations</td>
</tr>
<tr>
<td>Parallelization &amp; Data Locality Optimizations</td>
</tr>
<tr>
<td>Kernel Functions Optimization</td>
</tr>
<tr>
<td>Runtime Framework</td>
</tr>
</tbody>
</table>
Algebraic Transformation: Example

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

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

\[ S(a,b,i,j) = \sum_{c,d,f,k} A(a,c,i,k) \left( \sum_{e,l} B(b,e,f,l)D(c,d,e,l) \right) \]

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

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

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

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

\[ T1(b,c,d,f) = \sum_{e,l} B(b,e,f,l)D(c,d,e,l) \]
\[ T2(b,c,j,k) = \sum_{d,f} T1(b,c,d,f)C(d,f,j,k) \]
\[ S(a,b,i,j) = \sum_{c,k} T2(b,c,j,k)A(a,c,i,k) \]

- 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)

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

\[
T_1 = 0; T_2 = 0; S = 0
\]
for \(b, c, d, e, f, l\)

\[
T_1 = 0; T_2 = 0; S = 0
\]
for \(b, c\)

\[
S = 0
\]
for \(b, c\)

Unfused code

(Partially) Fused code
Memory Minimization: Loop Fusion

Unfused code

\[
T_1 = 0; \; T_2 = 0; \; S = 0
\]

for b, c, d, e, f, l

\[
T_{1_{bcdf}} += B_{befl} \; D_{cdel}
\]

for b, c, d, f, j, k

\[
T_{2_{bcjk}} += T_{1_{bcdf}} \; C_{dfjk}
\]

for a, b, c, i, j, k

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

(Partially) Fused code

\[
S = 0
\]

for b, c

\[
T_{1f} = 0; \; T_{2f} = 0
\]

for d, e, f, l

\[
T_{1f_{df}} += B_{befl} \; D_{cdel}
\]

for d, f, j, k

\[
T_{2f_{jk}} += T_{1f_{df}} \; C_{dfjk}
\]

for a, i, j, k

\[
S_{abij} += T_{2f_{jk}} \; A_{acik}
\]

Fully Fused code

\[
S = 0
\]

for b, c

\[
T_{1f} = 0; \; T_{2f} = 0
\]

for d, f

\[
T_{1f} += B_{befl} \; D_{cdel}
\]

for e, l

\[
T_{2f_{jk}} += T_{1f} \; C_{dfjk}
\]

for j, k

\[
S_{abij} += T_{2f_{jk}} \; A_{acik}
\]

- **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