Applying Temporal Blocking with a Directive-based Approach

Shota Kuroda, Toshio Endo, Satoshi Matsuoka
Tokyo Institute of Technology

Supported by:
- JST-CREST, "Software Technology that Deals with Deeper Memory Hierarchy in Post-petascale Era”
- JST-CREST, "EBD: Extreme Big Data Convergence of Big Data and HPC for Yottabyte Processing"
Our Focus: Stencil Computations

• Important kernels for various simulations (CFD, material...)

• Regions to be simulated are expressed as multi-dimensional arrays

• In each temporal iteration, the value of each point is computed from “adjacent points” in previous iteration

\[
A[t+1][x] = (A[t][x-1] + A[t][x] + A[t][x+1]) \times c;
\]

→ **Memory bandwidth major.** The key for performance improvement is **locality improvement**
Temporal Blocking (TB)

- TB improves memory access locality by blocking: [Wolf91] [Wonnacott00] etc.
- When we pick up a sub-domain, we perform multiple (bt-step) updates at once, and then proceed to the next one
  - $bt$: temporal block size
- A simple “rectangle” blocking/tiling violates dependency!

→ A “skewed” block shape is needed. There are variations
Issues in Introducing TB

• Higher programming cost for introducing “skewed” blocks

Original simple 1D stencil

```c
for (t = 0; t < T; t++)
    for (x = 1; x < N-1; x++)
        A[t+1][x] = (A[t][x-1] + A[t][x] + A[t][x+1]) * c;
```

TB with Trapezoid shape

```c
for (t1=ceil(-N-29,32); t1<=floor(T-2,32); t1++)
    for (t2=max(t1,-t1-1); t2<=min(min(floor(-16*t1+T-1,16),floor(16*t1+N+13,16)),floor(T+N-3,32)); t2++)
        for (t3=max(max(max(0,16*t1+16*t2),32*t1+1),32*t2-N+2); t3<=min(min(T-1,32*t2+30),
            16*t1+16*t2+31), 32*t1+N+29); t3++)
            lbv=max(max(32*t2,t3+1),-32*t1+1+2*t3-31);
        ubv=min(min(-32*t1+2*t3,32*t2+31),t3+N-2);
    for (t4=lbv; t4<=ubv; t4++)
        A[t3+1][(-t3+t4)] = (A[t3][(-t3+t4)-1] + A[t3][(-t3+t4)] + A[t3][(-t3+t4)+1]) / 3;
```
Existing Project

• Pluto compiler [Bondhugula 08]
  • Polyhedral source to source compiler
  • The target loop is attached a #pragma directive
  • Users specify how such loops are transformed as command line options
  • Temporal blocking is supported!

• Issues (as far as we tested)
  • Block shape is fixed
  • Fails with pseudo multi-dimensional arrays (e.g. array[y * nx + x])
  • A single set of options (cf. block sizes) are applied to all target loops
    → Tuning per target loop is hard
Our Approach

Directive based introduction of temporal blocking

→ Blocking parameters (block shape, sizes) are customizable for each target loop

Based on Polly/LLVM by Tobias Grosser

→ Wider applications, especially with pseudo multi-dimensional (MD) arrays
<table>
<thead>
<tr>
<th></th>
<th>Pluto</th>
<th>Polly</th>
<th>Ours (Currently)</th>
<th>Ours (Planned)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block Shape</strong></td>
<td>diamond</td>
<td></td>
<td>trapezoid</td>
<td>none/trapezoid/wavefront</td>
</tr>
<tr>
<td><strong>Pseudo MD Arrays</strong></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Methods to Specify Block Sizes</strong></td>
<td>command line option</td>
<td>directive</td>
<td>directive</td>
<td></td>
</tr>
</tbody>
</table>
Compilation Flow in the Original LLVM & Polly

1. Source code is transformed to intermediate representation, LLVM-IR
2. Detect Static Control Parts (SCoP), which corresponds to loops to be transformed
3. Construct polyhedral model for each SCoP
4. The “Schedule” of loop iterations is modified
5. LLVM-IR is reconstructed by using original IR and modified model
Compilation Flow of Our Modified Tool Chain: Step 1

- Parses our new directives
- Embeds their information as metadata in LLVM-IR
Directive Design for Customizable Temporal Blocking

Programmers write directives that start with `#pragma tb`, before temporal loop of the target

- **tile_size(bt,b1,b2..) clause**
  - Specifies block sizes
  - For each loop dimension (including temporal)

- **radius(r1,r2...) clause**
  - Specifies radii of stencil
  - For each spatial dimension

- **scheme(s1,s2...) clause**
  - Specifies block shapes
  - For each spatial dimension
  - s1, s2 should be “none” or “trapezoid”
  - “wavefront”, “diamond” are to be implemented
An Example of Directives

```
#pragma tb tile_size(8,16,512)  // Block sizes for t, y, x
#pragma tb radius(1,2)  // Stencil radii for y, x
#pragma tb scheme(trapezoid,trapezoid)  // Shapes for y,x

for(t=0 ; t<nt ; ++t)  // Temporal loop (t-dim)
  for(y=1 ; y<ny-1 ; ++y)  // Spatial loop (y-dim)
    for(x=2 ; x<nx-2 ; ++x)  // Spatial loop (x-dim)
      a[t+1][y * disp + x] = alpha * ( a[t][(y - 1) * disp + x] + a[t][ y * disp + x - 2] + a[t][ y * disp + x ] + a[t][ y * disp + x + 2] + a[t][(y + 1) * disp + x ] );
```
Compilation Flow: Step 2

1. Detect SCoP, target of transformation
2. Construct **Polyhedral model** of the SCoP
SCoP conditions (simplified)

A program fragment is a SCoP if:

- Used control structures are “for” or “if”
- Each loop has a single inductive variable (IV), which is increased constantly from a lower bound to an upper bound
- Lower/upper bounds are affine expressions of parameters and IVs of outer loops
- The condition of “if” statement is a comparison of affine expressions
- Each statement is an assignment of expressions to a variable or an array element
- An expression consists of operators whose operands are array elements, parameters, constants
- An array index is an affine expression of IVs, parameters, constants

The following patterns frequently appear in stencil computations with “double buffering” technique

```c
void calc(float *a[2], const long nt, const long nx){
    for(long t=0; t<nt; ++t){
        const long s = t%2;
        const long d = (t+1)%2;
        for(long x=0; x<nx; ++x){
            a[d][x] = (1.f/3.f) *
                (a[s][x-1] + a[s][x] + a[s][x+1]);
        }
    }
}
```

Polly Error: Base address not invariant in current region 😞
This is A SCoP

```c
void calc(float *a[2], const long nt, const long nx){
    #pragma tb tile_size(8,16) radius(1) scheme(trapezoid)
    for(long t=0 ; t<nt ; ++t)
        if ( t % 2 == 0 )
            for(long x=0 ; x<nx ; ++x)
                a[1][x] = (1.f/3.f) *
                           (a[0][x-1] + a[0][x] + a[0][x+1]);
        else
            for(long x=0 ; x<nx ; ++x)
                a[0][x] = (1.f/3.f) *
                           (a[1][x-1] + a[1][x] + a[1][x+1]);
}
```

In this work, we modified the user source code by hand

⇒ Polly successfully detects this pattern as a SCoP

This modification should be automatically done in future
An Example of Polyhedral Model

Input Code fragment

```c
for ( t=0; t<nt; ++t)
    if (t % 2 == 0)
        for ( x=1; x<nx-1; ++x)
            a[1][x] = a[0][x-1] + a[0][x] + a[0][x+1];
    else
        for ( x=1; x<nx-1; ++x)
            a[0][x] = a[1][x-1] + a[1][x] + a[1][x+1];
```

Polyhedral model (simplified)

"statements" : [
    { "domain" : "[nt, nx] -> { Stmt[t, x] : 2*floor((t)/2) = t and 0 <= t < nt and 1 <= x < nx-1 }", "schedule" : "[nt, nx] -> { Stmt[t, x] -> [t, x] }" }, ...
]

**domain:**
The domain of loop iterations (t and x in this case)

**Schedule:**
Specifies the execution of loop iterations.
lexicographical order of timestamps are applied

![Temporal loop t](#)

![Spatial loop x](#)
Compilation Flow: Step 3

A new LLVM pass is developed
It applies temporal blocking by change of scheduling
• Blocking parameters in metadata are used
Iteration Schedule for 1D Temporal Blocking

Temporal loop \( t \)

Spatial loop \( x \)

Block kind = 0

Block kind = 1

Stmt\( [t, x] \rightarrow [T, 0(=\ block\_kind), X, t, x] \)

Stmt\( [t, x] \rightarrow [T, 1(=\ block\_kind), X, t, x] \)

Stmt\( [t, x] \rightarrow [t, x] \)
#pragma tb tile_size(13, 312) radius(1) scheme(trapezoid)

for(int t = 1; t < nt; t++){
    if(t % 2 == 1)
        for(int x = 1; x < nx - 1; x++)
            a[1][x] = (a[0][x - 1] + a[0][x] + a[0][x + 1]) * 0.333f;
    else
        for(int x = 1; x < nx - 1; x++)
            a[0][x] = (a[1][x - 1] + a[1][x] + a[1][x + 1]) * 0.333f;
}

[jt, jx] -> { Stmt[t, x] -> [t, x] }

[jt, jx] -> {
    Stmt[t, x] -> [T, 0, X, t, x] :
        ( T = floor(t / 13) and
            X = floor( ( x + 1 * 12 - ( t - 13 * T)) / 600 ) and
            floor( ( x + 1 * 12 - ( t - 13 * T)) / 600 )
            = floor( ( x - 312 + 1 * 12 + ( t - 13 * T)) + 600 ) / 600 ) )
    Stmt[t, x] -> [T, 1, X, t, x] :
        ( T = floor(t / 13) and
            X = floor( ( x + 1 * 12 - ( t - 13 * T)) / 600 ) and
            floor( ( x + 1 * 12 - ( t - 13 * T)) / 600 )
            != floor( ( x - 312 + 1 * 12 + ( t - 13 * T)) + 600 ) / 600 ) )
}

Directive parameters in IR metadata are used.
Compilation Flow: Step 4

By using the modified schedule, LLVM-IR with temporal blocking is generated.
Transformation Example

Source: 1D 3point Stencil

- Before:
  - Double loop of t and x
- After:
  - Quad loop of T, X, t, x
Coding Cost to Introduce Temporal Blocking

- The original codes are 1D 3point and 2D 5point stencils
- In “TB-auto” with our system, the main task of user programmer is to add the directives
- For comparison, we implemented temporal blocking by hand-coding (TB-manual)

<table>
<thead>
<tr>
<th></th>
<th>TB-auto</th>
<th>TB-manual</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1D</td>
<td>2D</td>
</tr>
<tr>
<td># of lines edited or added incl. directives</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td># of operations added</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Performance Evaluation

• 1D 3point stencil and 2D 5point stencil

• We compared the followings
  • Original code (original)
  • Temporal blocking by our system (TB-auto)
  • Temporal blocking by hand (TB-manual)
  • Spatial blocking in 2D stencil by hand (SB-manual)
    • Coding cost is smaller than TB-manual, but locality is not so good as TB
Measurement Conditions

- Measurements are done on Sandy-Bridge core i7 and Xeon Phi KNL

<table>
<thead>
<tr>
<th></th>
<th>Sandy</th>
<th>KNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel i7 3930K</td>
<td>Intel Xeon Phi CPU 7210</td>
</tr>
<tr>
<td># of cores</td>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>3.2GHz</td>
<td>1.3GHz</td>
</tr>
<tr>
<td>LL cache size</td>
<td>12MB</td>
<td>32MB</td>
</tr>
</tbody>
</table>

- OpenMP parallelization
  - In TB-Auto, Parallelization is done by mechanism of Polly (!)
  - In original, TB-manual, SB-manual, we attached "#pragma omp parallel for" to the outermost spatial loop

- Selection of spatial block size
  - We have obtained the optimal size for various temporal block sizes through preliminary experiments
  - We did do that with "tile_size" clause

- Other compiler setting
  - Our modified compiler based on clang 4.0
  - O3 optimization “after” the modified Polly phase
  - Auto-vectorization is not used
1D stencil on 6-core Sandy Bridge
(NX=16M, NT=2k)

TB-auto is 1.5x slower than TB-manual. Why?

Faster
Analysis of Lower Speed

We checked the output IR code of the innermost loop

```c
void calc(float *a[2], // ← For double buffering
         long nt, long nx){
    for(long x=0 ; x<nx ; ++x)
        a[1][x] = (1.f/3.f) * (a[0][x-1] + a[0][x] + a[0][x+1]);
}
```

- Loads from "a" should be placed out of the loop, since a[0] and a[1] are static
- Why this well-known optimization did not work?
  ← We did not place optimization phases before Polly pass, for Polly to transform loops successfully
- Why did not optimization passes after Polly work well? → Under investigation
Workaround: TB-auto-2

We forcibly moved redundant load operations out of the function

```c
void calc(float * restrict a0,
    float * restrict a1,
    const long nt,const long nx){
#pragma tb tile_size(8,16) radius(1) scheme(trapezoid)
    for(long t=0 ; t<nt ; ++t)
        if ( t % 2 == 0 )
            for(long x=0 ; x<nx ; ++x)
                a1[x] = (1.f/3.f) *
                    (a0[x-1] + a0[x] + a0[x+1]);
        else
            for(long x=0 ; x<nx ; ++x)
                a0[x] = (1.f/3.f) *
                    (a1[x-1] + a1[x] + a1[x+1]);
}
```

...Apparently we need better and automated method in future
1D stencil on Sandy Bridge \((\text{NX}=16M, \text{NT}=2k)\)

Largely improved by the workaround

---

Faster
The workaround (TB-auto-2) is working, but difference between TB-manual and TB-auto-2 is larger than on SandyBridge → Under investigation
2D on 6-core Sandy Bridge
(NX=NY=4k, NT=2k)

In this case, spatial blocking (SB) is meaningless → TB is needed!!
While TB-auto is disappointing, TB-auto-2 is comparable to TB-manual
2D on 64-core KNL (NX=NY=4k,NT=2k)

TB-auto-2 works well, but the difference from TB-manual is larger than on SandyBridge
Summary

• We are developing a compilation tool towards automatic temporal blocking
  • Based on Polly/LLVM
  • Blocking parameters are customizable with #pragma directives
  • Blocks with skewed shape are automatically introduced

• Evaluation with 1D/2D stencil showed large speed-up by better locality
  • Some workarounds are still needed, mainly due to “double-buffering” programming technique
Future Directions

• Automation of the abovementioned workarounds
• Implementation of various block shapes

• Supporting real-world stencil/CFD applications !!!
  • How can we support complex kernels with multiple functions, complex data structures?
  • How can we support complex boundary conditions?