Abstracting Iteration- and Data-Space Transformations with High-Level Language Features

by David Wonnacott, Mary Glaser, Divesh Otwani, Sehyeok Park, Justin McHenry

Haverford College

IMPACT 2019 (Valencia, Spain)
High-Performance Computing (compiler-writer’s vision)

Programmer writes clean code about the *ideas* of their *algorithm*

```plaintext
for i in 0..n-1 {
    for j in 0..n-1 {
        x1[i] = x1[i] + A[i,j] * y1[j];
    }
}
for i in 0..n-1 {
    for j in 0..n-1 {
        x2[i] = x2[i] + A[j,i] * y2[j];
    }
}
```

or perhaps even

```plaintext
x1 += A * y1;
x2 += transpose(A) * y2;
```

and the compiler ensures it runs fast & cool on high-performance systems.
High-Performance Computing (actual practice, too often)

I’d love to be wrong about any of these:

- May use “-O”, may not, skeptical about specialized tools
  - “blind alley” tools, “fragile” performance benefits
  - “orphaned” tools, not promptly (or ever) updated w/compilers
- Not excited about unproven, over-specialized, or steep-learning-curve tools
- Producing C/C++/… hand-optimized for current hardware, may be messy
- Willing (and quite able) to learn new tools if clear good payoff/effort

¿ If this is the case, how to help loop transformation + polyhedral model catch on ?
Manual-yet-Clean Application of Loop Optimizations

Core idea: idiomatic source as common, user-editable I.R. for transformations

- **Iterators** can be used to abstract loop transformations
- **Classes** can be used to abstract data-space transformations
- can be created by software, students, etc., and traded with friends

But what about…

- fast execution?
- generality/fragility?
- gradual learning curve?
- clarity of resulting code?
- ease of use?
- correctness?
Iterators in the Chapel Language

- **Domain** and **Schedule** of executions/instances of a loop body
- Semantics: for each execution of “yield”, loop body is executed
- Can be implemented via coroutines, functions returning lists, *inlining*

```chapel
for i in 0..n-1 {
    for j in 0..n-1 {
        x1[i] = x1[i] + A[i,j] * y1[j];
    }
}

for i in 0..n-1 {
    for j in 0..n-1 {
        x2[i] = x2[i] + A[j,i] * y2[j];
    }
}

iter itersBasic(n: int): (stmts, int, int) {
    for i in 0..n-1 {
        for j in 0..n-1 {
            yield (colPass, i, j);
        }
    }
    for i in 0..n-1 {
        for j in 0..n-1 {
            yield (rowPass, i, j);
        }
    }
    for (s, i, j) in itersBasic(n) do {
        if (s == colPass) {
            x1[i] = x1[i] + A[i,j] * y1[j];
        } else {
            assert (s == rowPass);
            x2[i] = x2[i] + A[j,i] * y2[j];
        }
    }
}
```
Selecting an Iterator in Chapel: The good, the bad, and the ugly

// Fused (like Pluto w/o tiling or parallel)
iter itersOnePhase(n: int): (stmts, int, int) {
    for i in 0..n-1 {
        for j in 0..n-1 {
            yield (colPass, i, j);
            yield (rowPass, i, j);
        }
    }
}

// Based on "slice back, then forward" idea
iter itersBySlice(n: int): (stmts, int, int) {
    for i in 0..n-1 {
        for j in 0..n-1 {
            yield (colPass, i, j);
            yield (rowPass, j, i);
        }
    }
}

// also
// itersTiled
// itersTiledOnePhase
// consider...
// itersLimAndLam?
// itersGSAL18_Iterative?

// To choose iterator, define "itOpt"
// Simpler, but slower, to define iters...
iter iters(n: int): (stmts, int, int) {
    if (itOpt == basic) {
        for (s, i, j) in itersBasic(n)
            yield(s, i, j);
    } else if (itOpt == onePhase) {
        for (s, i, j) in itersOnePhase(n)
            yield(s, i, j);
    }
}

for (s, i, j) in iters(n) do {
    if (s == colPass) {
        assert (s == colPass);
        x1[i] = x1[i] + A[i,j] * y1[j];
    } else { assert (s == rowPass);
        x2[i] = x2[i] + A[j,i] * y2[j];
    }
}
Abstracting Data Transformations via Classes/Records

- Abstraction barrier between abstract object (e.g., 2-d image) and memory
- Can be implemented via dynamic dispatch, sometimes *inlining*
- Should be able to express (top-of-my-head list; should have citations)
  - Array expansion (adding a dimension)
  - Array un-expansion (contraction? removing a dimension)
  - Array privatization
  - Padding
  - Ghost cells
  - Array transpose/index transformation (is the latter ever useful?)
  - Piecewise-affine index transformation to multiple arrays
  - Adding/removing arrays (may need added statements)
  - Copy-in/copy-out for better local layout (w/added statements)
  - ¿ * Anything else we should be aiming for? *
The Deriche Benchmark

// sweep j dimension, compute matrices y1, y2
for (i=0; i<_PB_W; i++)
  for (j=0; j<_PB_H; j++)
    y1[i][j] = ... uses imgIn, y1[i,j-1] ...
    ...
for (i=0; i<_PB_W; i++)
  for (j=_PB_H-1; j>=0; j--)
    y2[i][j] = ... uses imgIn, y2[i,j+1] ...
    ...
for (i=0; i<_PB_W; i++) // combine y1, y2
  for (j=0; j<_PB_H; j++)
    imgOut[i][j] = c1 * (y1[i][j] + y2[i][j]);

// sweep i dimension
for (j=0; j<_PB_H; j++)
  for (i=0; i<_PB_W; i++)
    y1[i][j] = ... uses imgOut, y1[i-1,j] ...
    ...
for (j=0; j<_PB_H; j++)
  for (i=_PB_W-1; i>=0; i--)
    y2[i][j] = ... uses imgOut, y2[i+1,j] ...
    ...
for (i=0; i<_PB_W; i++) // combine y1, y2
  for (j=0; j<_PB_H; j++)
    imgOut[i][j] = c2*(y1[i][j] + y2[i][j]);

// sweep j dimension +, C approach:
for (i=0; i<_PB_W; i++) {
  ym1 = ym2 = xm1 = SCALAR_VAL(0.0);
  for (j=0; j<_PB_H; j++) {
    y1[i][j] =
      a1*imgIn[i][j] + a2*xm1 +
      b1*ym1 + b2*ym2;
    xm1 = imgIn[i][j];
    ym2 = ym1;
    ym1 = y1[i][j];
  }
}
Data-space Transformation for Deriche: introducing/removing scalars

record derray2 {
    const W: int; const H: int;
    const dom: domain(2);
    var Vals: [dom] DATA_TYPE_IS;
    // if we want to cache scalars, use these:
    // var mostRecentWrite: DATA_TYPE_IS; // ym1
    // var prevWrite: DATA_TYPE_IS; // ym2
}

proc derray2(wid: int, ht: int) {
    W = wid; H = ht;
    dom = {0..W-1,0..H-1};
}

proc set(i: int, j: int, val: DATA_TYPE_IS) {
    Vals[i,j] = val;
}

// prevWrite = mostRecentWrite;
// proc get(i: int, j: int) {
//    return Vals[i,j];
//}

proc jlow(i: int, j: int) {
    if (j == 0) return 0.0;
    else return Vals[i, j-1];
}

// else return mostRecentWrite;

proc jlowlow(i: int, j: int) {
    if (j <= 1) return 0.0;
    else return Vals[i, j-2];
    // else return prevWrite;
}

// sweep j+, original C approach
for (i=0; i<_PB_W; i++) {
    ym1 = ym2 = xm1 = SCALAR_VAL(0.0);
    for (j=0; j<_PB_H; j++) {
        a1*imgIn[i][j] + a2*xm1 + b1*ym1 + b2*ym2);
    }
    ym2 = ym1;
    ym1 = y1[i][j];
}

// ALTERNATIVE sweep j+, abstract (Chapel)
for (statement, i, j) in diter(w,h) do
    if (statement == upCol) {
        (a1*imgIn.get(i,j) + a2*imgIn.jlow(i,j) + b1*y1c.jlow(i,j) + b2*y1c.jlowlow(i,j)));
    }

// ALTERNATIVE sweep j+, array notation (Chapel)
for (statement, i, j) in diter(w,h) do
    if (statement == upCol) {
        a1*imgIn[i][j] + a2*imgIn[i,j-1] + b1*y1[i,j] + b2*y1[i,j-2]);
    }
Data-space Transformation for Deriche: reducing dimensionality

record derray2in1Col {
    const W: int; const H: int;
    const dom: domain(1);
    var Vals: [dom] DATA_TYPE_IS;
}

proc derray2in1Col(wid: int, ht: int)
    W = wid; H = ht;
    dom = {0..H-1};
}

proc set(i: int, j: int, val: DATA_TYPE_IS)
    Vals[j] = val;
}

proc get(i: int, j: int)
    return Vals[j];
}

proc jlow(i: int, j: int)
    if (j == 0) return 0.0;
    else return Vals[j-1];
}

proc jlowlow(i: int, j: int)
    if (j <= 1) return 0.0;
    else return Vals[j-2];
}

// sweep j dimension, compute matrices y1, y2
for (i=0; i<_PB_W; i++)
    for (j=0; j<_PB_H; j++)
        y1[i][j] = ... uses imgIn, y1[i,j-1] ...
...
for (i=0; i<_PB_W; i++)
    for (j=_PB_H-1; j>=0; j--)
        y2[i][j] = ... uses imgIn, y2[i,j+1] ...
...
for (i=0; i<_PB_W; i++) // combine y1, y2
    for (j=0; j<_PB_H; j++)
        imgOut[i][j] = c1 * (y1[i][j] + y2[i][j]);

// sweep i dimension
for (j=0; j<_PB_H; j++)
    for (i=0; i<_PB_W; i++)
        y1[i][j] = ... uses imgOut, y1[i-1,j] ...
...
for (j=0; j<_PB_H; j++)
    for (i=_PB_W-1; i>=0; i--)
        y2[i][j] = ... uses imgOut, y2[i+1,j] ...
...
for (i=0; i<_PB_W; i++) // combine y1, y2
    for (j=0; j<_PB_H; j++)
        imgOut[i][j] = c2*(y1[i][j] + y2[i][j]);
High-Level-Language-Based Loop Optimization (challenges)

- **Correctness**: A function of the tuple \((\text{loop-body, iterator, classes})\)
  - for polyhedral subset, auto. check vs. original dataflow
- **Generality/Fragility**: Can be mixed with/converted-to “regular” code
- **Learning Curve**: Idiomatic uses of standard language features
- **Clarity & Transparency**:
  - Core code still focuses on original statements (now labelled)
  - Optimization modules make tuning transparent to those who look
- **Ease of Use**: Between automatic and hacking … could we compose iter’s?
- **Performance/Value**: ∃ useful cases … but, how many?
High-Level-Language-Based Loop Optimization (next steps)

- Get some performance data for Nussinov, parallel Deriche!
- Exploring additional codes (syr2k, mvt, adi, Nussinov … bigger codes)
- Exploring interactions with other low-level optimizations (e.g., vectorizing)
- Exploring additional target architectures, e.g. cluster computing
- Automating correctness checks

Iff this can be achieved, idea/approach could be worthwhile.
Related Approaches

We don’t know of other methodical idiom-based work (*thoughts?*)

Alternatives:

- Automatic tools/compilers: Great when they work consistently
- DSL’s: Similar, and require support
- Script-based rewrites, pragma’s: similar, and require support
- Turning performance-tuners loose on the whole code base
  - can work, in principle
  - code clarity and/or correctness may be compromised
Conclusions

High-Level Language Optimization

- Prior work shows ∃ code: can compete with polyhedral good-case.
- Prior work shows ∃ code: can outperform sequential Pluto.
- We believe we can outperform parallel Pluto (e.g., Deriche)…
  - but want to get the experiments nicer before presenting them.
- Potentially useful for numerical scientists when tools fail?
- Potentially useful for compiler writers?
  - For exchanging transformations not widely supported?
  - For trying out ideas before writing a full compiler?
  - “Neutral territory” in which to exchange loop transformations?
  - Get users to accept/care-about instance-wise code transformation?
    → Transparency of our contribution
    → Ability for them to take control