Static Versioning in the Polyhedral Model
Adithya Dattatri & Benoit Meister, Reservoir Labs
Outline

● Who we are: Reservoir Labs
● Polyhedral versioning: background & motivation
● Approach
● Results
# Reservoir Labs

## Technology Expertise

<table>
<thead>
<tr>
<th>High Performance Computing</th>
<th>Networking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R-Stream</strong></td>
<td><strong>GradientGraph</strong></td>
</tr>
<tr>
<td>Automatic Parallelization and Mapping Through Polyhedral Model</td>
<td>Network Optimization</td>
</tr>
<tr>
<td><strong>LLVM</strong></td>
<td><strong>R-Core</strong></td>
</tr>
<tr>
<td>Customization for Advanced Supercomputers</td>
<td>Packet Path Accelerator</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cybersecurity</th>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R-Scope</strong></td>
<td><strong>ENSIGN: Cyber</strong></td>
</tr>
<tr>
<td>Network Sensor Visibility Enterprise Security</td>
<td>Spectral Hypergraph Analytics</td>
</tr>
<tr>
<td></td>
<td>Asymptotic Improvements to Physical Simulation, Optimization, and Inverse Problems</td>
</tr>
</tbody>
</table>

[https://www.reservoir.com/](https://www.reservoir.com/)
info@reservoir.com
Polyhedral Versioning
Background & Motivation
Versioning (a.k.a Multi-Versioning)
What is it?

- Observation: **different optimization opportunities** arise under **different run-time conditions**

- With versioning, compiler generates:
  - Multiple versions of a code region
  - Code to select the most appropriate version at run-time
Traditional example

- Suppose alias analysis cannot statically disambiguate two pointers

```c
int * p1, *p2;
...
if (mayAlias(p1, p2)) {
    // code optimized assuming aliasing
} else {
    // code optimized assuming non-aliasing
}
```

- If these pointers were not aliased, more instructions could be run in parallel [Sampaio17]
Motivation

Deep Learning (DL) Optimization

- DL networks can re-use layers with varied input tensor sizes
  - Explored this via our R-Stream TensorFlow [TF] front-end TFRCC [TFRCC]

- R-Stream maps differently for different fixed input sizes
  - Mapping refers to polyhedral compiler’s optimization phase

- More and more DL networks have variable-size inputs
  - Assume: sizes are parameters to the optimized function
  - We may not know anything about them
  - A single mapping cannot be optimal for all sizes
  - Need to be more adaptive to sizes
Polyhedral versioning

Our solution

Run-time defined parameters (e.g., tensor sizes)

Code (e.g., outlined NN code)

Constraints for parametric affine domain over $a_1, \ldots, a_n$

Call to a version of `func`

This function...

```c
func (...)a1,\ldots, an,\ldots) {
    ...
    ...
}
```

...is compiled to this

```c
versioned_func(...)a1,\ldots,an,\ldots) {
    if (PD1) {
        if (PD2) {
            func_1(a1,\ldots,an);
        } else {
            func_2(a1,\ldots,an);
        }
    } else {
        if (PD3) {
            func_3(a1,\ldots,an);
        } else {
            ...
        }
    }
}
```
Other approaches

- Pre-compilation: User incorporates knowledge of run-time parameter values into program logic (R-Stream allows this via #pragma)

```c
#pragma rstream map "context:N>=128,N<=1024"
void matmult(real_t A[N][N], real_t B[N][N], real_t C[N][N]) {
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = 0.0;
            for (k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

- Just-In-Time: use polyhedral model in non-polyhedral codes
  - PolyJIT: find run-time polyhedral cases, point-wise versioning
  - Apollo: calls Pluto at runtime to optimize code
    - Recent run-time versioning + mini-auto-tuning support
<table>
<thead>
<tr>
<th>Polyhedral terms</th>
<th>Functional terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Generalized dependence graph (GDG)</td>
<td>1. Program function</td>
</tr>
<tr>
<td>○ GDG parameters</td>
<td>○ Formal parameters</td>
</tr>
<tr>
<td>2. GDG hierarchy</td>
<td>2. Function call graph</td>
</tr>
<tr>
<td>○ Parent / child GDGs</td>
<td>○ Caller / callee</td>
</tr>
<tr>
<td>3. Specialized (aka versioned) GDG</td>
<td>3. Versioned function</td>
</tr>
<tr>
<td>○ Affine constraints over GDG parameters</td>
<td></td>
</tr>
<tr>
<td>○ Used in optimization decisions</td>
<td></td>
</tr>
</tbody>
</table>
Approach
Approach outline

Main steps:

1. (Auto) generate useful GDG parameter domains for versions
   ○ Illustration: processor placement

2. Incorporate and encode versioning decisions into the mapping process

3. Generate versioned code
Determine GDG version domains
Processor Placement (1/2)

- Placement pass: associate placement function to each polyhedral statement
  \[ \text{Pl: } \mathbb{Z}^{\text{param}} \times \mathbb{Z}^{\text{iterations}} \rightarrow \mathbb{Z}^{\text{grid\_dims}} \]

- Occupation test
  - loop trip count \( \geq c \times \) processor grid size
  - \( c \) : “occupancy”, factor we want to occupy (\( \frac{1}{2} \) of the grid, 3x the grid size, ...)
  - If true: place along the loop
  - Otherwise, try another loop

- When trip count involves unbounded GDG parameters, mapper assumes they are large enough
  - Unchecked assumption
Determining GDG version domains

Processor Placement (2/2)

- \( t(N) \): parametric loop trip count

- \( pg(k) \): grid size along targeted dim \( k \)

- When \( t(N) \) cannot be bounded by a constant
  - Schedule the mapping of a GDG version
  - “Tell the mapper” to consider the following affine range (i.e., not 
    large enough assumption)

\[
  t(N) \leq c \times pg(k)
\]
Mapping and encoding versions (1/2)

- Introduce polyhedral statement called a “SpecializeOp”
  - Maintains versions of a GDG (“specialized GDGs”)

- Introduce a specializer GDG to hold a SpecializeOp
  - At codegen: conditionally calls the versioned functions

- A specialized GDG comes from “specializing a GDG”, that is
  - Clone the GDG
  - Intersect cloned GDG’s context with given domain
  - Here, given domain will be $t(N) \leq c \times pg(k)$
Mapping and encoding versions (2/2)

- Insert specialized GDG into existing GDG hierarchy
  - D : “specializer GDG”
  - P : “parent GDG” of G
  - $C_1$ and $C_2$ : the “child GDGs” of G
- After insertion, the specialized GDG is scheduled for mapping
  - Start where mapping was at for the input GDG (e.g., right at placement pass)
Code Generation (1/2)

- Generate nested if/else for the specializer GDG that
  - avoids explicit polyhedral differences (ugly code, complexity)
  - executes only one version for any parameter value

- Note: extra degree of liberty when specialized domains overlap
  - Unexploited here

- Naive approach
  - $C_i = \text{specialized GDG } G_i \text{'s context}\n  - #(C_i) = \# \text{ of constraints in } C_i$
  - $N = \text{total } \# \text{ of contexts}\n  - Redundantly check constraints\n  - Nested constraints depth for $G_i$: $\sum_{j=1}^{i} #(C_j)$

```plaintext
if (C1) {
    call the function lowered for G1
} else if (C2) {
    call the function lowered for G2
}

.  
.  
.  
else if (Cn) {
    call the function lowered for Gn
}
```
Code Generation (2/2)

- Outermost conditions: pick a constraint that divides the contexts non-trivially into included/not included GDG contexts

- Following properties:
  - No constraint checked more than once for any parameter values
  - Total number of constraints to get to $G_i$ is $\leq N + \#(C_i)$
  - See paper for proof

- Dividing as evenly as possible helps drive $N$ to $\log_2(N)$ in upper bound
Evaluation
Evaluation
Specifications

- Test machine processor info:
  - 1 socket, 8 cores/socket and 2 threads/core
  - Processing grid size: [16]

- Three test programs
  - **fc**: a fully connected layer where input/output sizes are equal
  - **convolution_googlenet**: 1st convolution of GoogLeNet
  - **maxpool_resnet**: a residual NN layer that uses MaxPooling

- Test programs are functions that have one run-time defined parameter
  - Here, versioned code is branched on this parameter’s value
  - For small parameter values, versioned program executes further optimized code
  - For large parameter values, versioned and non-versioned programs execute virtually the same code
Evaluation
R-Stream mapping, OpenMP target

For each (layer, occupancy setting, param value):
1. Compile program w/ versioning and w/o versioning
2. Run versioned program with fixed param value for 5 trials
   a. Dampens OpenMP variability
3. Run the non-versioned with the fixed param value for 5 trials
4. Compute run time speed-up

- Occupancy values:
  ○ 100% (full) and 200% (double)
  ○ 200% is to leverage dynamic load-balancing of OpenMP
Results

Versioning speedup

Small param range, significant speed-up

Large param range, same performance

Small param range, significant speed-up

Large param range, same performance

fc, c=1

fc, c=2
Results

Versioning speedup

** convolution_googlenet**
- Small param range, significant speed-up
- Large param range, same performance

** maxpool_resnet**
- Small param range, significant speed-up
- Large param range, same performance

Parameter value (number of images handled per batch)

Parameter value (height of input image)
Results
Versioned GDG vs non-versioned

- Speedup was due to sequential version being faster than parallel version for small size parameters (expected)

- Example layers resulted in sequential vs. parallel
  - Want to find examples where different placement choices are made

- “Bump” between versions
  - Versioning domain inequality can be improved
  - Occupation test is very simple but not optimal

- Simple target (OpenMP) and pass (processor placement)
  - Useful to understand basic problematic
  - More tradeoffs and questions w/ other passes & targets
Results

Compilation time

Upshot: low overhead

Tradeoff between partial mapping (placement & onward) vs. full mapping
- Full: More optimization opportunities, but higher compilation time
Thank You

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC0017071, DE-SC0018480, and DE-SC0019522.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
References


