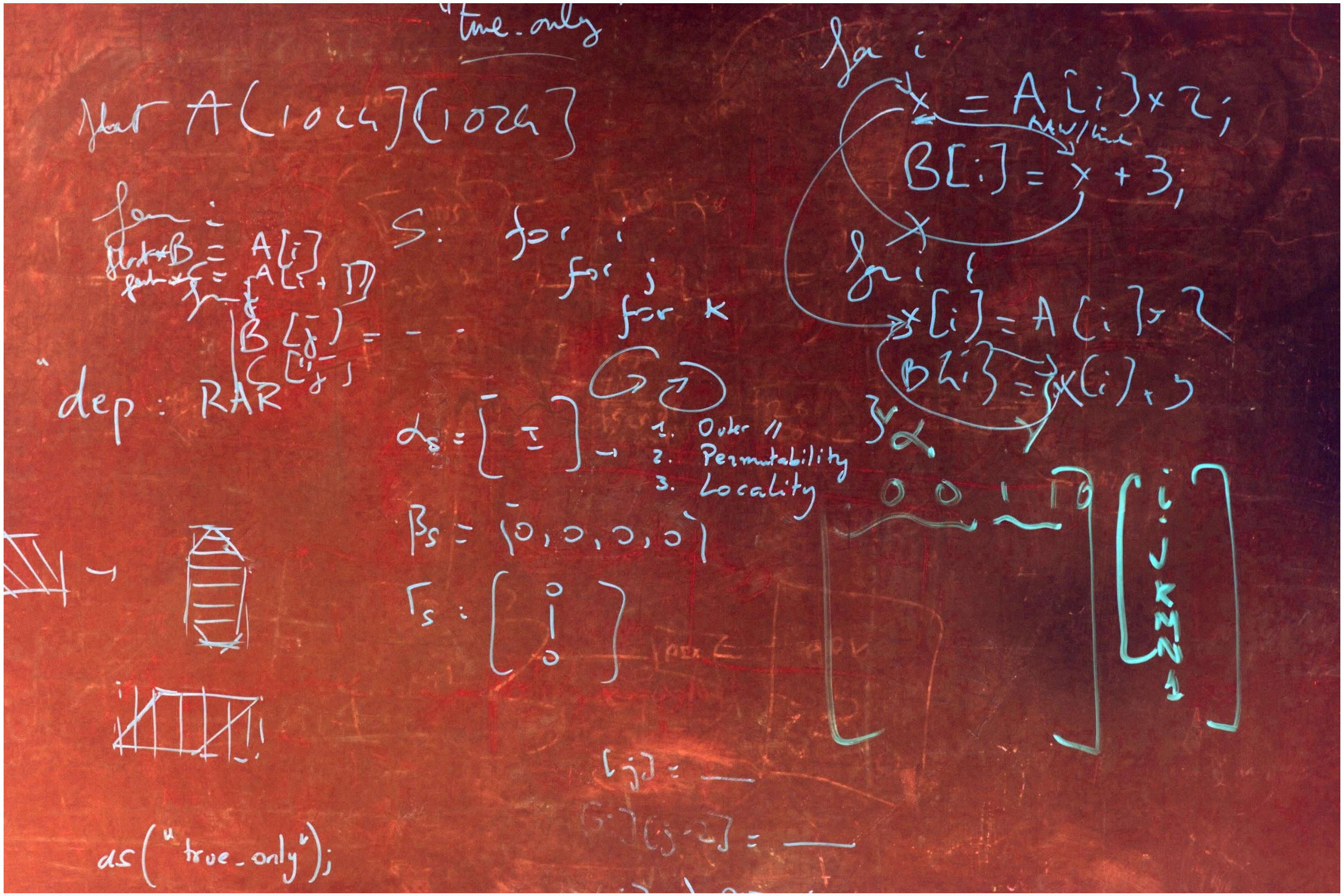
Static Versioning in the Polyhedral Model

Adithya Dattatri & Benoit Meister, Reservoir Labs



Outline

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

Reservoir Labs

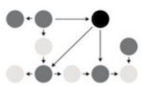
Technology Expertise

High Performance Computing

R-Stream
Automatic Parallelization and Mapping Through Polyhedral Model

LLVM
Customization for Advanced Supercomputers

Networking



GradientGraph
Network Optimization

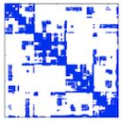
R-Core
Packet Path Accelerator

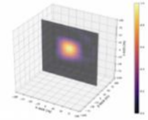
Cybersecurity

R-Scope
Network Sensor Visibility Enterprise Security

ENSIGN: Cyber
Spectral Hypergraph Analytics

Algorithms





Asymptotic Improvements to Physical Simulation, Optimization, and Inverse Problems

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

```
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 a1,...,an

Call to a version of func

This function...

```
func (...a1,...,an,...) {
        ...
        ...
}
```

...is compiled to this

```
versioned_func(...,a1,...,an,...) {
     if (PD1) {
          if (PD2) {
               func_1(a1,...,an);
          } else {
               func_2(a1,...,an);
          }
     } else {
          if (PD3) {
               func_3(a1,...,an);
          } else {
               ...
          }
     }
}
```

Other approaches

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

```
#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

Polyhedral Intermediate Representation (IR)

Correspondences

Polyhedral terms	Functional terms
1. Generalized dependence graph (GDG) ○ GDG parameters	1. Program function ○ Formal parameters
2. GDG hierarchy ○ Parent / child GDGs	2. Function call graph ○ Caller / callee
3. Specialized (aka versioned) GDG	3. Versioned function
4. Context of a GDG ○ Affine constraints over GDG parameters ○ Used in optimization decisions	4. Function domain / preconditions

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
  Pl: $\mathbb{Z}^{\text{param}} \times \mathbb{Z}^{\text{iterations}} \rightarrow \mathbb{Z}^{\text{grid\_dims}}$

- Occupation test
  - loop trip count >= c x processor grid size
  - c : "occupancy", factor we want to occupy (½ 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)

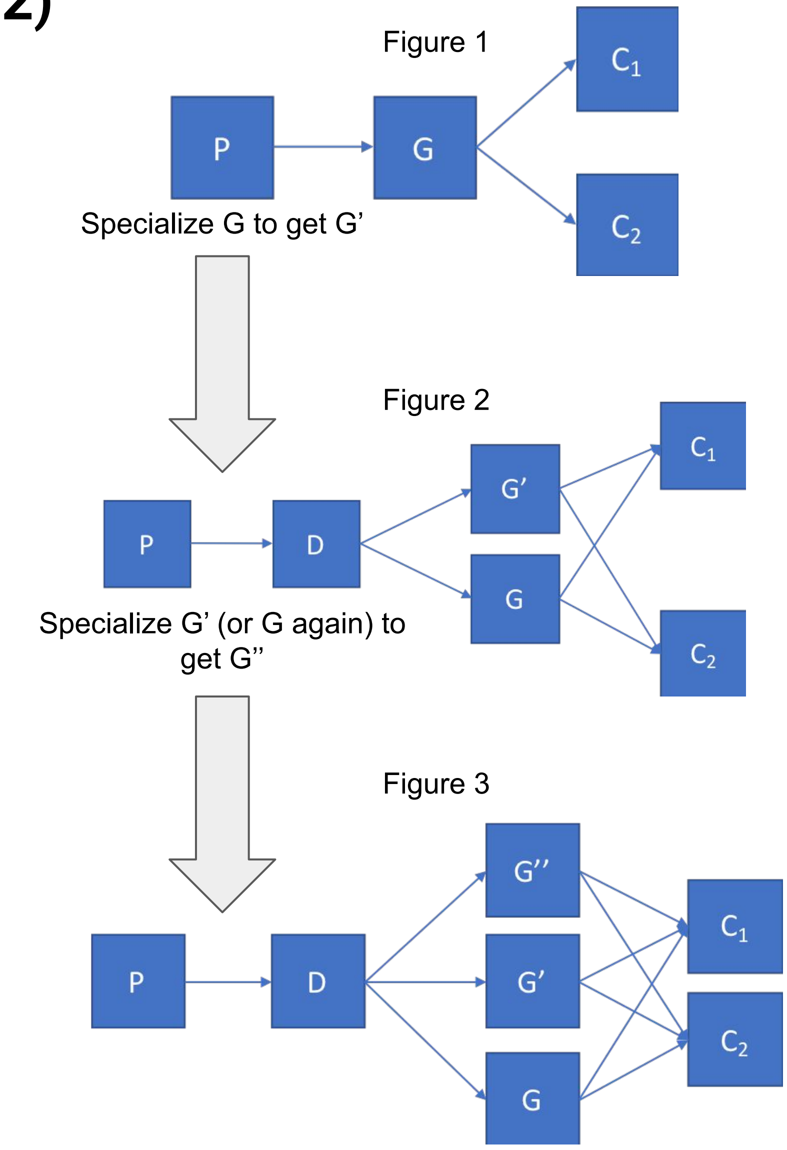
$$t(N) \leqslant c * pg(k)$$

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 $t(N) \leqslant c * pg(k)$

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

```
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 $\leqslant N + \#(C_i)$
  - 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

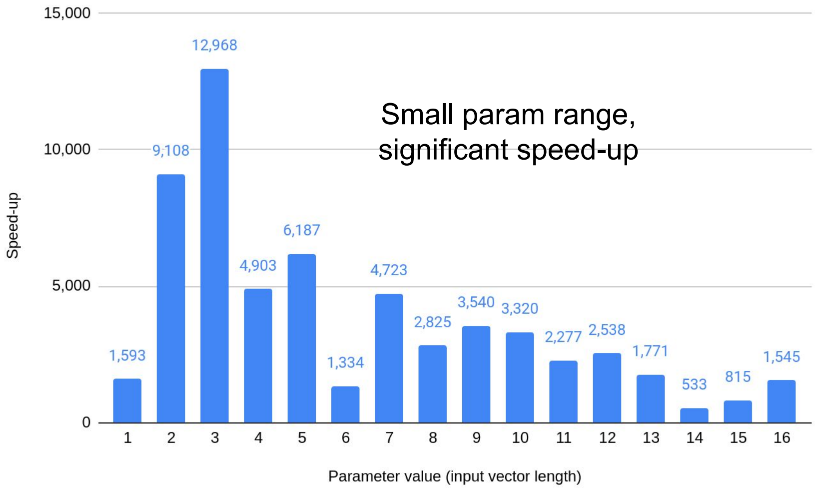
3. Run the non-versioned with the fixed param value for 5 trials

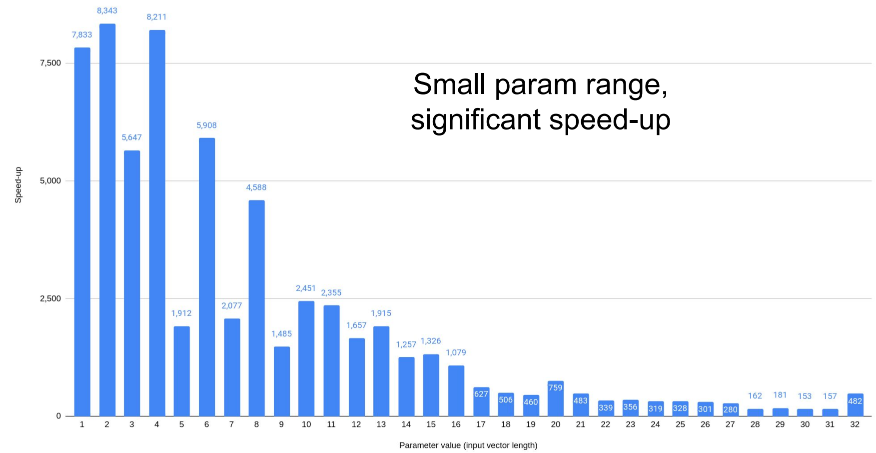
4. Compute run time speed-up

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

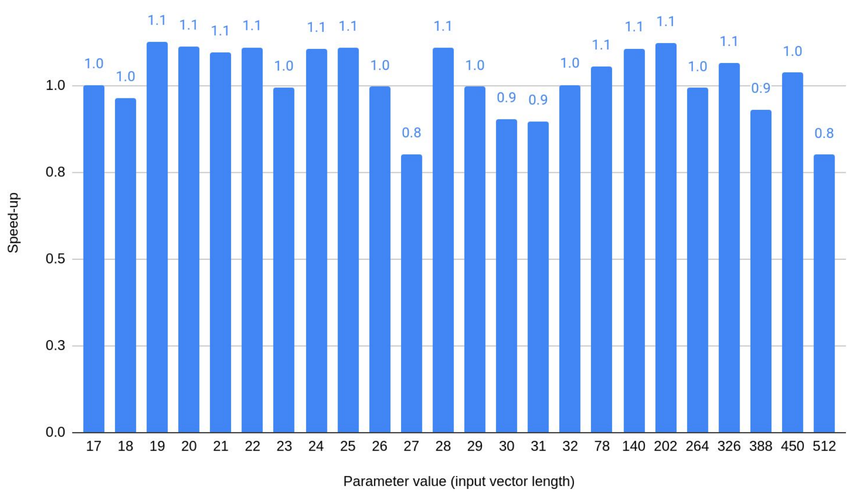
Results

Versioning speedup



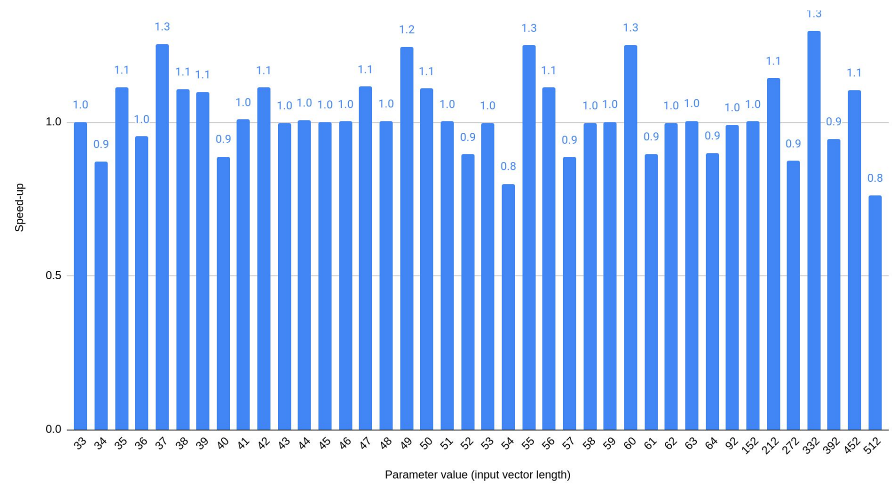


Large param range, same performance



fc, c=1

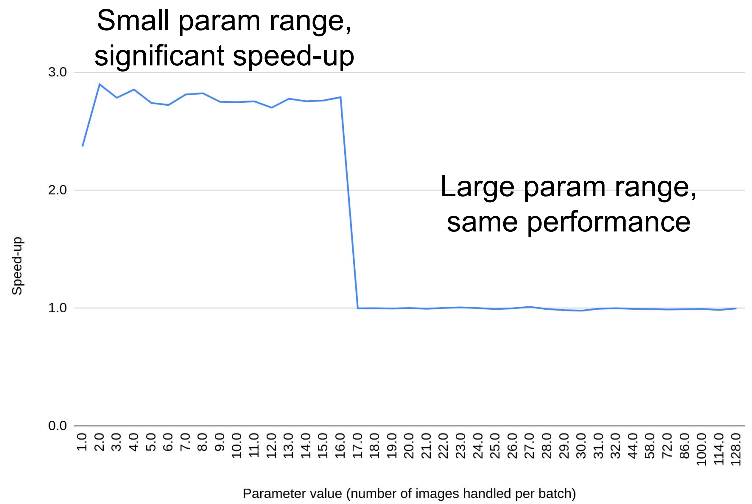
Large param range, same performance

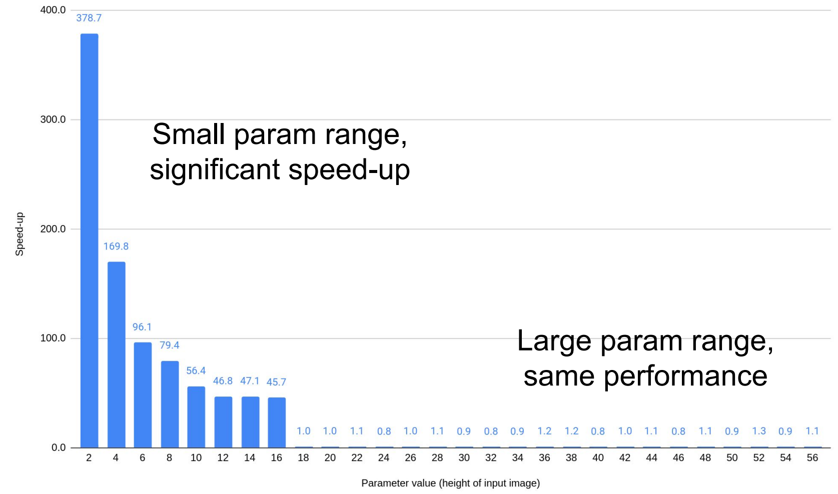


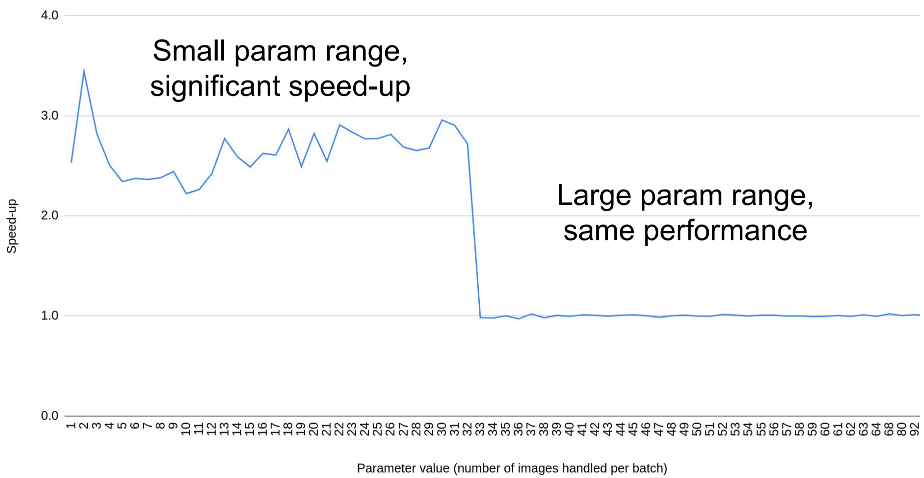
fc, c=2

Results

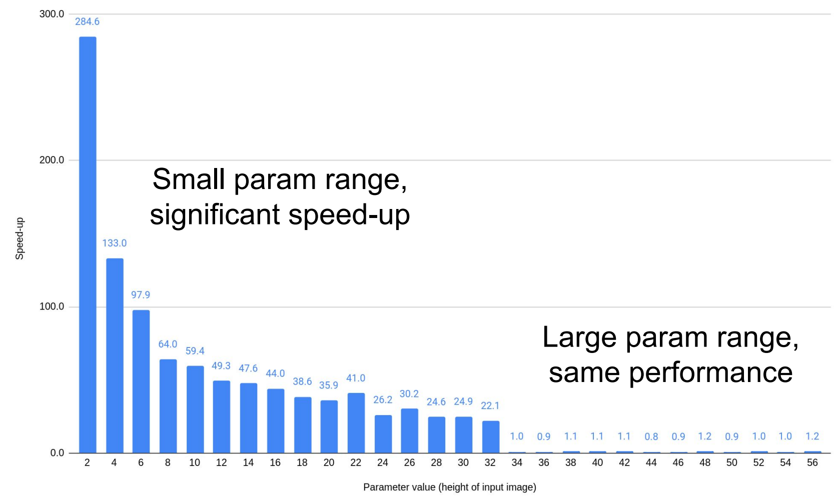
Versioning speedup







convolution_googlenet



maxpool_resnet

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

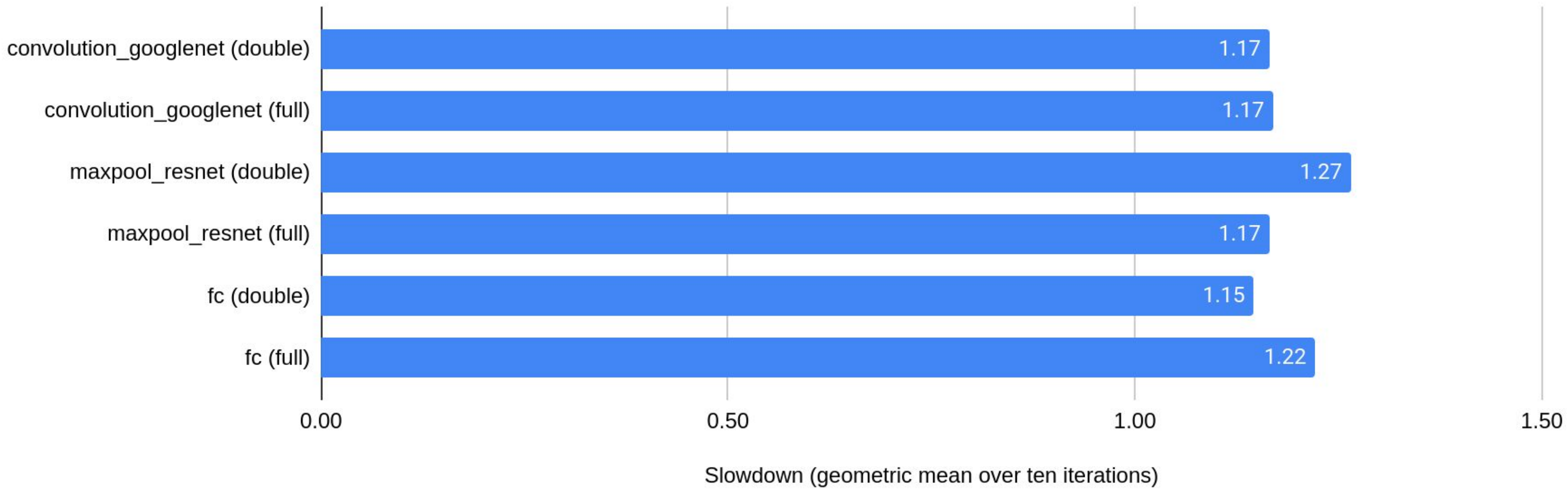
- “Bump” between versions
  - Versioning domain inequality can be improved
  - Occupation test is very simple but not optimal

- 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.

References

[Apollo] Lazcano, R., Madroñal, D., Juarez, E., & Clauss, P. (2020, February). Runtime multi-versioning and specialization inside a memoized speculative loop optimizer. In Proceedings of the 29th International Conference on Compiler Construction (pp. 96-107).

[GoogLeNet] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. CoRR abs/1409.4842 (2014). arXiv:1409.4842 http://arxiv.org/abs/1409.4842

[PolyJIT] Andreas Simbürger, Sven Apel, Armin Größlinger, and Christian Lengauer. 2019. PolyJIT: Polyhedral Optimization Just in Time. Int J Parallel Prog 47 (2019), 874–- 906. https://doi.org/10.1007/s10766-018-0597-3

[R-Stream] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Baskaran, Allen Leung, and Richard Lethin. 2011. R-Stream Compiler. In Encyclopedia of Parallel Computing, David Padua (Ed.). Springer Reference

[Sampaio17] Sampaio, Diogo N., Louis-Noël Pouchet, and Fabrice Rastello. "Simplification and runtime resolution of data dependence constraints for loop transformations." In Proceedings of the International Conference on Supercomputing, pp. 1-11. 2017.

[TF] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[TFRCC] Benoıt Pradelle, Benoıt Meister, Muthu Baskaran, Jonathan Springer, and Richard Lethin. 2017. Polyhedral Optimization of TensorFlow Computation Graphs. In 6th Workshop on Extreme-scale Programming Tools (ESPT) at The International Conference for High Performance Computing, Networking, Storage and Analysis (SC17).