Static Versioning in the Polyhedral Model

Adithya Dattatri
Reservoir Labs
New York, USA
dattatri@reservoir.com

Benoît Meister
Reservoir Labs
New York, USA
meister@reservoir.com

ABSTRACT

We present an approach to enhancing the optimization process in a polyhedral compiler by introducing compile-time versioning, i.e., the production of several versions of optimized code under varying assumptions on its run-time parameters. We illustrate this process by enabling versioning in the polyhedral processor placement pass. We propose an efficient code generation method and validate that versioning can be useful in a polyhedral compiler by performing benchmarking on a small set of deep learning layers defined for dynamically-sized tensors.

CCS CONCEPTS

• **Computing methodologies → Parallel computing methodologies**.

KEYWORDS

polyhedral model, versioning, compilers, high performance computing, deep learning

ACM Reference Format:
Adithya Dattatri and Benoît Meister. 2021. Static Versioning in the Polyhedral Model. In *IMPACT 2021: The 11th International Workshop on Polyhedral Compilation Techniques, January 16–18, 2021 (Virtual event).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Polyhedral compilers [Baghdadi et al. 2019; Bondhugula et al. 2008; Grosser et al. 2012; Meister et al. 2011; Strout et al. 2018; Verdoolaege et al. 2013] can realize powerful optimizations and parallelization of loop-based programs by deriving a polyhedral representation of the loop code, performing intricate mathematical operation on that representation, and then rendering the transformed polyhedral representation as code. The final code can be rendered in a parallel programming language (e.g., C with OpenMP or pthreads, CUDA) or in a compiler's internal representation, from which scalar, single-thread optimization is performed.

Modern compilers are able to leverage optimization opportunities arising under specific run-time conditions by generating a variety of optimized code variants, and informing the runtime of when to use each variant. This process is called *versioning* (or sometimes *multi-versioning*) [A. Jimborean 2011; Chen and Long 2009; Luo et al. 2009]. A typical example is when alias analysis is unable to statically disambiguate two pointers. The compiler may detect that if said pointers were not aliased, more instructions could be run in parallel. If this is the case, the compiler may insert code that performs a run-time non-aliasing test, uses the optimized code when the test holds, and uses the unoptimized code otherwise [Sampaio et al. 2017].

Versioning has been used in many ways by compilers, including in the polyhedral raising phase mentioned previously. We discuss other interesting uses, such as disambiguating array references, extracting affine access functions out of polynomial-looking ones, and enabling just-in-time polyhedral optimizations for otherwise non-polyhedral programs, in Section 6. Here, we are interested in leveraging the idea of versioning in the polyhedral representation. Polyhedral mapping is the process of reordering operations and data in a loop-based computation to produce an optimized version of said computation, targeting a particular computer hardware platform. We describe an implementation of versioning in the R-Stream polyhedral compiler [Meister et al. 2011] and discuss how we enabled the processor placement pass to use it.

1.1 Application domain

The need for versioning appeared important to us while mapping deep learning codes. Tensor sizes are dynamic in some neural networks (as for instance [Wu et al. 2016], where a variable number of objects can be detected), and it seems worthwhile to adapt the polyhedral optimization strategy for layers that access these tensors as a function of the run-time tensor sizes. Hence, while versioning may be useful in other application domains, we use deep learning layers to illustrate the utility of versioning in a polyhedral compiler.

1.2 Model of an optimized code region

To simplify the discussion, we assume, without loss of generality, that the loop code region modeled by a polyhedral representation is outlined [Zhao and Amaral 2005] into a function. This way, we can refer to the optimized code region as a function and its live-in and live-out values as the function's formal parameters. By the same token, we are looking at applying versioning to the values of the function's formal parameters.

After outlining is done, we consider programs with at least one function with numerical (integer or rational) arguments that satisfy either of the following conditions:

(1) At least one numerical argument of the function is defined by a run-time value.
(2) The function is called with varied values for at least one numerical argument.

We note that recursive functions, which can be generated from polyhedral programs [Vasilache et al. 2013], almost always satisfy both conditions. However, we choose to focus on deep learning layers, which are, in our experience, rendered as non-recursive.

1.3 Versioning approach

Typically, versioning occurs in (at least) one of three places:

(1) Prior to compilation, the user can incorporate knowledge about the run-time values of the function arguments into the program logic for consideration by the polyhedral compiler. In R-Stream, this is explicitly supported for users through a special pragma annotation.
(2) Just-in-time (JIT) compilation creates versions of code regions, in which the function arguments are fixed to frequently-used values. However, the type of versioning performed at run-time compilation is limited by the need to minimize compilation time. Hence, versions would be determined by the run-time values of the function arguments. Furthermore, polyhedral compilation is generally considered too slow to be used in a JIT context.
(3) In ahead-of-time compilation, the compiler generates code for a function with numerical arguments that conditionally executes optimized and parallelized code upon checking the run-time argument values.

1.4 Overview

In this paper, we provide an ahead-of-time approach to versioning programs in the polyhedral model. In our approach, we attempt to make minimal assumptions about the implementation and design details of the underlying polyhedral compiler infrastructure. Hence, we anticipate that any polyhedral compiler can be reasonably extended to support our approach to versioning. Having successfully implemented our versioning approach in R-Stream, we describe the salient issues and suggest how to address these in the relevant sections.

We first provide enough background to understand the rest of the paper in Section 2. Then, we detail our approach to polyhedral versioning in Section 4. We validate the need for versioning experimentally and its impact on compilation time in Section 5, using a few examples from deep learning. Finally, we discuss related work in Section 6 and provide further direction in Section 7.

2 BACKGROUND

This section offers an overview of the main concepts needed to understand the technical description of the polyhedral versioning approach presented here. First, we provide a brief introduction to polyhedra in Section 2.1, which is the foundation for polyhedral compilation. Then we state and name the main phases of polyhedral compilation in Section 2.2; our approach will introduce modifications to some these phases. Lastly, we describe the polyhedral intermediate representation (IR) and introduce the conventional terms accordingly in Section 2.3; our approach will be defined over these terms.

2.1 Polyhedra

In a vector space V, a polyhedron $P \subseteq V$ is the set of solutions to a set of linear inequalities

$$P : Ax + b \geq 0, x \in V \tag{1}$$

Geometrically, Equation (1) defines P as the intersection of finitely many half-spaces in V given by the rows of A. A finite polyhedron is called a polytope.

It is possible to consider a subset of the dimensions of V as special variables, which do not get instantiated. Such variables are called the *parameters* of the polyhedron. For example, let us consider a parametric polytope example presented in [Clauss and Loechner 1998], which has two variables (i, j) and two parameters (n, m):

Example 2.1.

$$Q(i, j) = \{(i, j) \in \mathbb{Z}^2 : 0 \leq i \leq n; 0 \leq j \leq m\} \tag{2}$$

Q is the set of lattice points of the rectangle whose lower left and top right corners are at $(0, 0)$ and (n, m), respectively.

As hinted in Equation (2), in the polyhedral model of loops, we are often interested in integer-valued points inside polyhedra [Schrijver 1998; Verdoolaege 2010].

2.2 Automatic parallelization flow

In the polyhedral model of compilation, there are three main phases: raising, mapping and lowering. The raising phase translates the program from the input form to the polyhedral IR. In the case of R-Stream, the input source program is first translated to Sprig, a sea-of-nodes compiler IR. Raising is then performed from Sprig. The mapping phase performs the optimizations and parallelizations, termed *mapping decisions*, on the polyhedral IR of the program. The part of the polyhedral compiler that performs the mapping phase is termed the *(polyhedral) mapper*. Finally, the lowering phase translates the *mapped* program from the polyhedral IR to the output language.

2.3 Polyhedral IR

The polyhedral IR represents an approximation of the input code that uses affine expressions and leads to safe dependencies. For instance, code that writes to a data region that is not a polyhedron but that can be bounded by a polyhedron can be represented as accessing the bounding polyhedral region as "may write." The polyhedral IR used by R-Stream is based on Feautrier's Generalized Dependence Graph (GDG) [Darte et al. 2000; Feautrier 1992]. The polyhedral model focuses on the optimization of nested loop code that operate on multi-dimensional arrays. Hence, loop iteration domains and array access functions are first-class elements of polyhedral representations. GDG vertices represent polyhedral statements, which define an iteration domain, an operation performed for each iteration of the iteration domain, and a set of functions used to access data represented as multi-dimensional arrays. GDG edges represent pairwise dependence relationships between polyhedral statements (vertices).

We distinguish two types of polyhedral statements here:

- ClientOps represent operations in the input function, associated with their polyhedral iteration domain and the array

access functions involved in said operations. The semantics of a function raised into a GDG are fully captured by a set of ClientOps.
- PseudoOps are operations introduced by the mapper to express a parallel mapping of code. Examples include DMA transfers, barriers, thread spawning, asynchronous scheduling of a task, function calls and others.

Each raised function in the input IR is initially represented in the polyhedral IR as one GDG. The mapping process transforms the GDG, often with relation to a hierarchy of GDGs, where each GDG is analogous to a function. The GDG hierarchy can take the form of a general graph (since recursive calls form cycles), but there is always a *root GDG*, without predecessors, for each input function. Calling the input function is equivalent to calling the root GDG.

Since the GDG hierarchy shape is largely that of a tree, we refer to the source of an edge in the GDG hierarchy as a *parent GDG*. The destination of an edge in the GDG hierarchy is called a *sub-GDG*.

The numerical function arguments (if any) of an input function become the GDG parameters. The iteration domain and array access functions of polyhedral statements may be functions of the GDG parameters. Each GDG defines a polyhedral domain of the GDG parameter values for which the GDG is “valid”. This validity domain can represent preconditions to the function or simply the set of values for which the polyhedral statements’ iteration domains are not all empty. We refer to a GDG’s validity domain as the GDG’s *context* throughout this paper.

3 MOTIVATION

TensorFlow [Abadi et al. 2016] is one of the leading Deep Learning research frameworks out there. Users can use it to compose Deep Learning models, train them, and finally deploy them for inference. As part of an effort to optimize inference for these models, we built the very first polyhedral deep learning optimizer by creating a TensorFlow front-end to R-Stream [Pradelle et al. 2017] a few years ago. This exposed us to models that repeat the use of some layers with varying input tensor sizes (a good example of this is ResNet [He et al. 2015], a family of residual neural nets). Very often, these sizes are fixed, which allows the mapper to know everything about the iteration domains and access functions of the polyhedral statements representing the layers at compile time. We observed that R-Stream’s polyhedral mapper made different mapping decisions for different input sizes.

However, another set of neural networks use and produce tensors whose sizes are only known dynamically (e.g., [Wu et al. 2016], an object-detection net meant for autonomous driving). For example, detection networks may detect a variable set of objects, depending upon the input image. In these cases, some of the tensor sizes will be unknown at compilation time, and they have to be treated as parameters in the polyhedral IR.

Still, we want the mapping decisions to internalize the tensor sizes, even though the mapper may not know much about the particular run-time tensor sizes. The next section presents our approach to solving this issue, based on versioning.

4 APPROACH

A naive approach would be to enumerate all possible values of the parameters and optimize the code for each of them. Without prior knowledge about the run-time values, this is not efficient let alone practical, since this approach could generate unreasonably large amounts of code or a GDG’s context can be unbounded.

Instead, we divide the space of all collections of parameter values into finitely many ranges. Then we let the mapper generate mapping decisions for each range. Because the context of a GDG is defined to be a polyhedron, we restrict our focus to ranges that are polyhedral domains. Our approach to versioning can be realized as the answers to the following questions:

(1) How to inform the mapper to incorporate a given context sub-domain into its mapping process? (Section 4.1)
(2) How to auto-generate the useful context sub-domains? (Section 4.2)
(3) How to generate the subsequent versioned code ? (Section 4.3)

The polyhedral mapping process in R-Stream is organized by a mapping pass scheduler. An optimal list of polyhedral passes (called a "strategy") is determined for the targeted architecture and associated with the GDG to be mapped. Some polyhedral passes created new GDGs. These GDGs are usually called by the GDG being mapped, making them their "child GDG" in the GDG hierarchy. For instance, the thread generation pass splits a GDG into a master GDG (to be executed by the master thread) and one or more slave GDGs. The input GDG becomes the master GDG, and the slave GDGs are now children to the master GDG. These new GDGs are then themselves associated with a mapping strategy and queued up to the mapping pass scheduler. The mapping process ends when all the GDGs have been mapped.

With the versioning feature, polyhedral passes can now also define extra versions of the GDG being mapped, by defining extra context constraints (i.e., a sub-domain of the context)

4.1 Informing the mapper

In this section, we define how a polyhedral pass can let the mapping scheduler know that it decided to create a new version of a GDG, specialized to a sub-domain of the GDG’s context, and what happens to the GDGs and mapping process. We introduce the notions of a *specialized GDG*, `SpecializeOp` and *specializer GDG*. At code generation, a specialized GDG would correspond to a version of the input function. A `SpecializeOp` is a PseudoOp that keeps track of a family of specialized GDGs, that is (in terms of code generation) a family of versions for one input function. More formally, a family F of specialized GDGs is a maximal set of GDGs where either of the following conditions exclusively holds for $G' \in F$:

(1) $\exists G \in F$ where $G \neq G'$ and G' is a specialized GDG generated by specializing G
(2) G' is the GDG from which all the other GDGs in F were directly or indirectly specialized

In the versioning process presented here, the members of a family are created by applying the SPECIALIZE procedure described below in any of the polyhedral passes run during the mapping process.

The `SpecializeOp` is lowered into code that calls the function lowered for a specialized GDG, upon checking that its context is satisfied by run-time parameter values. A specializer GDG is created by the mapper to contain a `SpecializeOp`. This GDG enables the generation of a wrapper function in the lowered program that contains the code of the `SpecializeOp`. This wrapper function replaces the input function in the lowered program.

Now, a specialized GDG arises from the following process. When the mapper is provided a GDG and a polyhedral domain over GDG parameter values to specialize towards, the mapper clones the GDG and adds the constraints to the clone GDG's context, which will be used to make mapping decisions. Specializing a specialized GDG is quite possible and well-defined; this would result in adding another GDG to the family of specialized GDGs.

For the mapper to keep track of the specialized GDGs and the specializer GDG created, they must be carefully inserted into the existing GDG hierarchy. A specializer GDG will be the parent GDG (in the hierarchy) of all the GDGs in the family of specialized GDGs that is given by the `SpecializeOp` contained in the specializer GDG. Furthermore, the hierarchy must re-structured so that for a family F of specialized GDGs, only the corresponding specializer GDG is "visible" to GDGs outside of F in the hierarchy; said in terms of code generation, the different function versions should only be called by the wrapper function in the lowered program. Creating a specializer GDG is a convenient way to handle all cases of GDG hierarchy, including recursive calls, and GDGs with more than one parent. In the simpler cases, the specializer GDG can be later inlined to remove the associated extra function call.

We provide the SPECIALIZE procedure that precisely gives our approach. SPECIALIZE takes as input a GDG G and a set of affine constraints C and is defined in Algorithm 1.

A specialized GDG is a bad specialization if its context is empty or if there already exists a version of this GDG with the same context. Mapping an extra GDG G' obviously requires extra work from the mapper. Since polyhedral mapping time can be non-trivial, an important trade-off between compilation time and optimization potential exists. If some mapping passes have already occurred on G when SPECIALIZE is called, two options are available. Repeating all the mapping steps on G' can enable more optimization, especially if the behavior of these previous steps is conditioned by the context, but it also likely doubles the compilation time of G. Conversely, starting the mapping process for G' at the beginning of the step that called SPECIALIZE introduces only a fraction of the total mapping time for G', but may miss some optimization opportunities.

Since each polyhedral pass that uses versioning can split the context of the input GDG in two or more sub-domains, the number of specialized GDGs can grow exponentially with the number of such passes. To limit the risk of an exponential compilation time blowup, the default behavior of SPECIALIZE is the latter.

We note that it is not necessary to create a custom-polyhedral statement if it is not supported by the polyhedral compiler infrastructure. The families of specialized GDGs can be maintained in other parts of the mapper state. Moreover, in our work, C is generated as per the procedure in Section 4.2.

Algorithm 1 Informing the mapper

```
procedure SPECIALIZE($G$, $C$)
    Let $G'$ be a clone of $G$
    Intersect the context of $G'$ with $C$
    if $G'$ is a bad specialization then
        return
    else if parent $P$ of $G$ exists and is a specializer GDG then
        Do nothing
    else
        Create a specializer GDG $D$
        Create a new SpecializeOp $s$ and add it to $D$
        Add $G$ to the family of specialized GDGs in $s$
        if parent $P$ of $G$ exists then
            Set parent of $D$ to $P$
            Remove $G$ as a subGDG of $P$
            Add $D$ as a subGDG of $P$
        end if
        Set parent of $G$ to $D$
    end if
    $D \leftarrow$ parent GDG (i.e., specializer GDG) for $G$
    $s \leftarrow$ SpecializeOp of $D$
    Add $G'$ to the family of specialized GDGs in $s$
    Set parent of $G'$ to $D$
    Tell mapper to map $G'$
end procedure
```

4.2 Generating versioning constraints

Each polyhedral pass whose behavior is determined by parameters, e.g., the size of iteration domains along certain dimensions, or the way array access references relate to each other, is a candidate for versioning. R-Stream has many passes, and an exhaustive review of them in relation to versioning is outside the scope of this paper.

We choose to illustrate polyhedral versioning on the placement pass, because its behavior varies strongly as a function of the iteration domain sizes. Moreover, because it is a fairly unsophisticated pass, our discussion can remain centered on versioning itself. The goal of placement is to define, for each polyhedral statement, a function from its iterations (i.e., any point of its iteration domain) to processor coordinates. The R-Stream machine model represents processing entities in a multi-dimensional hyper-rectangular grid. Hence, placement functions typically have as many dimensions as the targeted processor grid. Let Pl be the placement function of statement op. For any value of the parameters $N \in \mathbb{Z}^p$, iteration $I \in \mathbb{Z}^n$ of op gets executed by processor $x = Pl(I, N)$.

With the OpenMP[Board 2020] target, the default placement heuristic in R-Stream enumerates the loop dimensions of the polyhedral statements and tries to select the outermost loop dimension. A major test that determines the behavior of placement is checking whether a placement function would occupy the whole processor grid. We call this the *occupation test*. The test holds when the loop dimension considered for placement to a given processor grid dimension is large enough, that is when its trip count is at least the size of the targeted processor grid dimension. When this test fails, the pass declines to distribute the loop across the targeted processor grid dimension and tries the next inner eligible loop, by default.

Unfortunately, when a loop's trip count depends upon GDG parameters and when the context does not bound these parameters, the occupation test is undecidable. Before versioning was introduced, our placement pass made the unchecked assumption that the parameters are large enough for the occupation test to be true. With versioning, we make this assumption explicit, by creating a version where the assumption is *not* met.

Because the trip count of a loop often varies as a function of outer loop dimensions, occupation tests can be defined and used alternatively. For instance, we could decide that the average trip count must occupy the grid dimension, or weighted averages among the statements sharing the loop, or the maximum, etc. While several of these tests are available from the placement pass, we chose to use the maximum. The maximum trip count is obtained by computing parametric bounding box of the loop's trip count domain and taking the difference (plus one) between the upper and lower bound of the bounding box.

Another key parameter of the placement pass is its "occupancy" (let us denote it with c). Occupancy defines the number of loop iterations per processor (along a given processor grid dimension). Occupancy can play a different role depending upon the targeted architecture. An occupancy greater than one is often used with programming models supporting dynamic scheduling and load balancing, reflecting a desire to over-provision work. An occupancy less than one is often used with statically scheduled targets, when these have a large amount of processing elements along the considered dimension, indicating that using a fraction of the grid is still profitable.

Placement declines to distribute a loop if its trip count is less than c times the targeted processor grid size. The user might set the occupancy to $\frac{1}{2}$ to use only half the processors (using a command-line parameter). On the other hand, the user may require at least two iterations of the given loop per processing element by setting the occupancy to two.

When placement selects a loop for placement along dimension k of the processing grid, and its trip count is a parametric function $t(N)$, we let placement trigger the mapping of a specialized GDG by calling SPECIALIZE on the current GDG in the placement pass and the following affine constraint

$$t(N) \leq c.pg(k)$$

where $pg(k)$ is the size of the processor grid along dimension k. This constraint informs the mapper that $t(N)$ is not large enough when mapping the specialized GDG.

4.3 Versioned code generation

Code generation of the specialized GDGs does not require any modification to support versioning. Specialized GDGs do not have a special status in the lowering phase. Hence, this section of the paper focuses on modifications made to the lowering phase to generate code for a specializer GDG and in particular its `SpecializeOp`.

Consider a specializer GDG D with `SpecializeOp` s. Let $n(s)$ denote the size of the family of specialized GDGs contained in s. Let $\{G_i\}_{i\in[n(s)]}$ denote the family of specialized GDGs in s and let $\{C_i\}_{i\in[n(s)]}$ be the contexts where C_i is the context of G_i with $\#(C_i)$ many constraints. A specializer GDG will correspond to a function in the lowered program that checks for a specialized GDG's context whose constraints are satisfied by the run-time argument values and calls that function lowered for that specialized GDG. While the context of multiple specialized GDGs G and G' may be satisfied by a given run-time value, even when G' is not (transitively) a specialization of G. If not, the first GDG in the SpecializeOp's list is selected. Our code generation algorithm only enforces that if G' is (transitively) a specialization of G, G' is selected. We note that this design choice does not affect correctness. Allowing overlapping contexts prevents us from computing complicated non-convex contexts, which would result in an explosion of conditionals. Instead, we enforce that only one GDG of a given family is executed for any valid value of the original GDG's parameters using `if/else` constructs. Here is a simple approach to generate code for s:

```
if ($C_1$) {
    call the function lowered for $G_1$
}
else if ($C_2$) {
    call the function lowered for $G_2$
}
    ⋮
else if ($C_n$) {
    call the function lowered for $G_n$
}
```

With this code, there is the possibility of re-evaluating the same constraint more than once across different if-statements when the contexts share constraints. Furthermore, when G_i is called, all contexts C_j for $j \leq i$ need to be checked, which can create unnecessary overhead. We provide a heuristic that generates code for a specializer GDG, and which does not check any constraint more than once for a given set of run-time values, but might check some extra constraints, relative to the constraints of the context of the GDG that the run-time values satisfy. We proceed to provide details of this heuristic, which is run after mapping and in the beginning of the lowering phase.

4.3.1 Specialization tree. Our code generation heuristic involves constructing a *specialization tree* for each `SpecializeOp`, which mirrors the structure of a conditionally branched code. We note this tree is not to be confused with the GDG hierarchy. We use this rooted-tree directly to generate the code for a single specializer GDG. We define two types of nodes in the specialization tree, namely `Cnd` and `FnCall`. We let each `Cnd` node maintain a set of constraints over the GDG parameters to be lowered into checking a condition over the numerical function arguments and each `FnCall` node maintain a reference to a specialized GDG, whose corresponding function is to be called. The leaves of the tree will be of type `FnCall` and all other nodes will be of type `Cnd`. Each `Cnd` node will have between one and two children. If a `Cnd` has one child, then the child corresponds to the true branch of the conditional. Otherwise, if a `Cnd` node has two children, there will be a distinguished left and right child, which will correspond to the true and false branches of the conditional, respectively. Both types of nodes maintain a Boolean flag indicating whether it is in a true (nested if) or false branch (same-level else).

4.3.2 Tree generation. In this phase where the tree is first generated, each of the Cnd nodes of the tree will have only one constraint. We require the following pre-conditions to hold for the SpecializeOp s prior to tree generation:

(1) No two contexts of specialized GDGs in s are equal.
(2) No specialized GDG in s has an empty context.

To ensure the first condition, for every pair of GDGs that have the same context, we remove one of them from s. To ensure the second condition, we remove GDGs with empty contexts from s. After the pre-conditions are ensured to hold and prior to tree generation, we also assert that the family of specialized GDGs has at least two specialized GDGs. These steps form our pre-processing. Due to the first condition, each GDG uniquely corresponds to a context.

We now define a recursive procedure TREE-GEN to generate the tree. The goal is to select the available constraints from all the GDG contexts and turn them into nested if/else conditionals. Some constraints are shared among contexts, other constraints include, exclude or intersect GDG contexts. TREE-GEN takes in four arguments: *activeCtxs*, *availCstrs*, *isTrue* and *ptNode*. *activeCtxs* is a set of contexts (and thereby their corresponding GDGs) that are left to be captured by FnCall nodes. *availCstrs* is a set of constraints that remain available for use by Cnd nodes; here we treat two constraints as equal if and only if they include the same integer points. *isTrue* is a Boolean flag that indicates whether the current node being constructed is directly within a true or false branch. Lastly, *ptNode* is the Cnd node that will be the parent of the current node being constructed. The procedure returns the root node of the specialization tree, which is of type Cnd. In the first call to TREE-GEN (after pre-processing), we set the argument values as follows:

(1) *activeCtxs*: union of the specialized GDG contexts
(2) *availCstrs*: union of all specialized GDG context constraints
(3) *isTrue*: true in our convention, but does not matter for root
(4) *ptNode*: null

On a high-level, the TREE-GEN proceeds as follows:

(1) When $activeCtxs.size > 1$, pick a *differentiating* constraint c from *availCstrs* that differentiates two contexts C_1 and C_2 in *activeCtxs*; in other words, c includes either C_1 or C_2 and does not include the other. Such a c must exist when $activeCtxs.size > 1$ (proved in Lemma 4.1).
(2) When $activeCtxs.size = 1$, pick any c from *availCstrs* that includes a context in *activeCtxs*. If no such c exists, then bind the specialized GDG corresponding to the one remaining context in *activeCtxs* to a FnCall node and return the node. Otherwise, proceed with the next steps.
(3) Create a Cnd node and add c to the node's set of constraints.
(4) Partition *activeCtxs* into those included (true branch) and not included (false branch) by c. Recursively call TREE-GEN on both of these partitions to build the rest of the specialization tree. We remove c from *availCstrs* before these sub-calls as it will not be used in the false branch sub-call and should not be chosen again in the true branch sub-call. We add back c after the sub-calls, as it can be used in other parts of the specialization tree. Return the Cnd node created in this call.

We provide representative pseudocode for the specialization tree generation in Algorithm 2.

Algorithm 2 Tree generation

```
procedure TREE-GEN($activeCtxs$, $availCstrs$, $isTrue$, $ptNode$)
    if $activeCtxs.size > 1$ then
        $c \leftarrow diffCstr(availCstrs, activeCtxs)$
        assert $c$ exists
    else
        if exists $c$ that includes context in $activeCtxs$ then
            $c \leftarrow cstrIncludeCtx(availCstrs, activeCtxs)$
        else                                        ▷ FnCall node case
            $g \leftarrow ctxToGDG(activeCtxs.get())$
            $fnCallNode \leftarrow$ new $FnCall(isTrue, g)$
            assert $ptNode$ ! = $null$
            $ptNode.addChild(fnCallNode)$
            return $fnCallNode$
        end if
    end if
    $cndNode \leftarrow$ new $Cnd(isTrue)$                    ▷ Cnd node case
    $cndNode.addCstr(c)$
    if $ptNode$ ! = $null$ then
        $ptNode.addChild(cndNode)$
    end if
    $includedCtxs \leftarrow ctxsIncludedBy(c)$    ▷ subset of $activeCtxs$
    $availCstrs.remove(c)$
    $activeCtxs.remove(includedCtxs)$
    TREE-GEN($includedCtxs$, $availCstrs$, $True$, $cndNode$)
    if $activeCtxs$ is not empty then
        TREE-GEN($activeCtxs$, $availCstrs$, $False$, $cndNode$)
    end if
    $availCstrs.add(c)$
    return $cndNode$
end procedure
```

LEMMA 4.1. *If $activeCtxs.size > 1$, there exists a constraint in availCstrs that differentiates between two contexts in activeCtxs.*

PROOF. Suppose $activeCtxs.size > 1$, but none of the constraints in *availCstrs* differentiates between two contexts in *activeCtxs*. Consider two contexts C_1 and C_2 in *activeCtxs*, which must be distinct by the pre-processing. There must be a differentiating constraint c that includes either C_1 or C_2 and does not include the other. c must have been removed in a previous call for which the current call is (transitively) a sub-call of, for otherwise c would be in *availCstrs*. This implies that c was added to the set of constraints of the Cnd node created in this previous call. However, if this were the case, C_1 and C_2 would not appear in the same *activeCtxs*, a contradiction. ◻

Lemma 4.1 shows that the claim made in the first high-level step is well-defined. We now prove Lemma 4.2, which implies that for each FnCall node, the corresponding GDG context is equivalent to the intersection of the conditions on the path from the root to the node.

LEMMA 4.2. *Given a* FnCall *node x, for each constraint c of the corresponding GDG context C, there will exist an ancestor* Cnd *node a that contains c in its set of constraints. Furthermore, if a has a* Cnd *child node w that is an ancestor of x, then $isTrue$ must be set for w.*

Proof. Suppose that for a `FnCall` node x, there is some constraint c of the corresponding GDG context C such that no `Cnd` node ancestor of x contains c in its set of constraints. Now consider the call to TREE-GEN that generates x. c would be in the *availCstrs* of this call. However, creating a `FnCall` node only occurs when there are no constraints in *availCstrs* that cover the one remaining context in *activeCtxs*, a contradiction. This implies the existence of a `Cnd` node ancestor a that contains c. Furthermore, if a has a `Cnd` child node w that is an ancestor of x, the *includedCtxs* of the call that generates a would contain C and w would be generated in the first sub-call, that is with the *isTrue* argument set to $True$. □

In Lemma 4.3, we show that we do not need to check too many constraints in addition to the constraints of a specialized GDG's context to get to the corresponding function call.

Lemma 4.3. *Let s be a* `SpecializeOp` *with family of specialized GDGs* $\{G_i\}_{i\in[n(s)]}$ *and contexts* $\{C_i\}_{i\in[n(s)]}$ *where* C_i *is the context for* G_i. *In the specialization tree for s, the path length from the root to the* `FnCall` *node that is associated to* G_i *is* $\leq n(s) + \#(C_i)$.

Proof. When $activeCtxs.size > 1$, a call partitions *activeCtxs* into two sets of size at most $activeCtxs.size - 1$. In this way, $\leq n(s)$ calls are required to get C_i to be the only remaining context in *activeCtxs*. Then we need to generate $\leq \#(C_i)$ many `Cnd` nodes for the remaining constraints of C_i. □

Lemma 4.3 also implies that the depth of a specialization tree for s is $\leq n(s) + \max_{i\in[n(s)]} \#(C_i)$. When calling the lowered function for G_i, the conditions when our heuristic is guaranteed to beat the simple approach (as far as checking fewer constraints) is given by the following inequality:

$$n(s) + \#(C_i) \leq \sum_{j=1}^{i} \#(C_j) \Rightarrow n(s) \leq \sum_{j=1}^{i-1} \#(C_j)$$

We sum over all $i \in [n(s)]$ to arrive at the following inequality:

$$n(s)^2 \leq \sum_{i=1}^{n(s)} \sum_{j=1}^{i-1} \#(C_j) = \sum_{i=1}^{n(s)-1} \#(C_i) \cdot (n(s) - i)$$

When this inequality holds, we use the heuristic over the simple approach. Furthermore, to make the heuristic better, in the first high-level step, we pick the constraint that results in a partition of *activeCtxs* into sets that are as close to being equal in size as possible. Ideally, if we are able to select a constraint that exactly partitions *activeCtxs* into equal sized sets in every call to TREE-GEN, then the $n(s)$ in the upper bound of Lemma 4.3 becomes $\log_2(n(s))$, which justifies this additional optimization.

4.3.3 Tree collapsing. To render the output code more readable and compact, nested if statements (without same-level else statements) are collapsed into one if statement that uses a conjunction of the conditionals. While several related simplifications or collapses could be applied, it is not clear that they would actually improve readability. We are not expecting to improve performance here, since the backend compiler will presumably generate equivalent CFGs regardless of whether these extra transformations are performed.

5 RESULTS

In this section, we first describe the infrastructure we use for testing as well as the the neural network layers we benchmark. Then we describe our benchmarking procedure. Lastly, we show and analyze the performance benefits of versioned code and display the compilation time overhead for our versioning method. We want to compare the behavior of versioned programs with non-versioned ones, for a varying problem size.

5.1 Benchmarking infrastructure

In Section 5.3, we evaluate the performance benefits of using versioning in the placement pass on three neural network (NN) layers:

(1) fc: a fully connected layer in which the input and output sizes are equal (i.e., square matrix multiplication)
(2) convolution_googlenet: GoogLeNet's [Szegedy et al. 2014] first convolution
(3) maxpool_resnet: a residual NN that uses MaxPooling

We use parametric versions of these codes, in which one or more loop bounds are given by a numerical parameter of the layer function. In fc, we parameterize on the length of the input vector; we refer to this parameter as Q. In convolution_googlenet, we parameterize on the number of images handled in a single batch; we refer to this parameter as *batch*. In maxpool_resnet, we parameterize on the height of the images; we refer to this parameter as *height*. We note that Q, *batch* and *height* are positive integer parameters. We use an empirical upper bound for these parameters, which we derive from bounds used in constant-size versions of the layers.

To benchmark these codes, we embed them into a program with *microkernel* structure. Here, a microkernel structure consists of the code that is embedded in it as well as the following functions: initialize_once, initialize, kernel, check. The kernel method is the main source of program behavior, and in our case, calls the NN code. Furthermore, a microkernel supports running the kernel method for any specified fixed number of trials and specifying run-time parameter values to be used by the kernel method. Both the number of trials and parameter values may be provided via command-line arguments to the microkernel. The execution of a microkernel consists of the following steps:

(1) Call the initialize_once function
(2) For each trial, do the following:
    (a) Call the initialize function
    (b) Call the kernel function
    (c) Call the check function, which checks if the kernel correctly performed its computation

In each trial, the cache is flushed right after the initialize function, but before the kernel function. Upon finishing a run, the microkernel displays the execution time of the kernel method totaled across all trials. The microkernels are written in C. Our test machine is a Ubuntu 18.04.5 64-bit server that has a Intel Xeon W-2245 CPU @ 3.90GHz processor, which features one socket, eight cores per socket and two threads per core.

5.2 Benchmarking procedure

For benchmarking, we restrict our focus to R-Stream's OpenMP backend, which generates C code that features R-Stream-generated optimizations and parallelizations (i.e., OpenMP constructs). In an attempt to make full utilization of our test machine's compute resources, we set OMP_NUM_THREADS to 16. R-Stream dynamically chooses among a collection of LP solvers at compile-time, which is used to perform various polyhedral tasks. To remove any variability that might arise here, we fix the LP solver prior to compilation to COIN-OR [cbc 2008]. We fed R-Stream with a machine model that represents the machine with the parameters given by the `lscpu` tool. `gcc` is used as the backend compiler to R-Stream with the `-march=native -O3 -fno-trapping-math` options.

For our specific selection of NN codes, R-Stream with versioning enabled generates the following simple conditional branching:

```
if ($param \leq limit$) {
    call 1st version of NN code
} else {
    call 2nd version of NN code
}
```

Here, *param* is a placeholder for Q, *batch*, and *height* and *limit* is a placeholder for a constant that the placement pass chooses based on the processor grid size and the occupancy option (see Section 4.2). Our test machine has a processor grid with 16 processing elements. For performance benchmarking, we use the following procedure given a fixed NN code's microkernel, occupancy setting and value for *param*:

(1) Compile the microkernel with the fixed occupancy setting using R-Stream w/ versioning and R-Stream w/o versioning
(2) Run the versioned microkernel with the fixed value for *param* for five trials (to dampen OpenMP variability)
(3) Run the non-versioned microkernel with the fixed value for *param* for five trials
(4) Compute the execution time speed-up

Regarding occupancy, we toggle between 1x and 2x, which results in 16 and 32 (respectively) for the values attained in place of *limit*. While the reason for setting 1x occupancy is to maximize processor utilization, the reason for setting 2x occupancy is to leverage the dynamic load-balancing provided by OpenMP. The set of values for *param* that we use for a microkernel is given by the following ranges: $[1, .., limit]$, eight equally-spaced points in $[limit + 1, .., param]$ and $[limit + 1, .., 2 \cdot limit]$. We choose these set of values for *param* to provide an equal-sized window for running both versions of code, to show how the versioning scales with respect to no-versioning, and to keep the visualization of the results simple. Here is the formula we use to calculate speed-up:

$$\frac{\text{run time of optimized code w/o versioning}}{\text{run time of optimized code w/ versioning}}$$

5.3 Performance benchmarking

5.3.1 fc. To increase legibility of the speedup results for fc, we have split their representation between the $Q \leq limit$ and $Q > limit$ ranges. The fully-connected layer (Figures 1 and 2) is the most dramatic of our examples, because placement chooses to *not* parallelize the code for low values of Q (i.e., $Q \leq limit$) in the specialized version. It is a sobering reminder that parallelizing small matrix multiplications across cores on a cache machine is not advisable. The random-looking speedups are due to the low absolute computation time, leading to higher performance variability, and to penalties incurred by false sharing. Setting the threshold to 32, as in Figure 2 (top) confirms this observation, with a smoother behavior as the absolute computation time grows. In the bottom plots of Figures 1 and 2, the speedup revolves around 1x, which confirms that the versioned code behaves like the non-versioned code (up to the remaining variability due to OpenMP), when the non-specialized GDG is selected. This is true for all the benchmarks (as seen in the right halves of Figures 3, 4, 5 and 6). This is not surprising, since the optimization of the non-specialized GDG is unchanged by the versioning process.

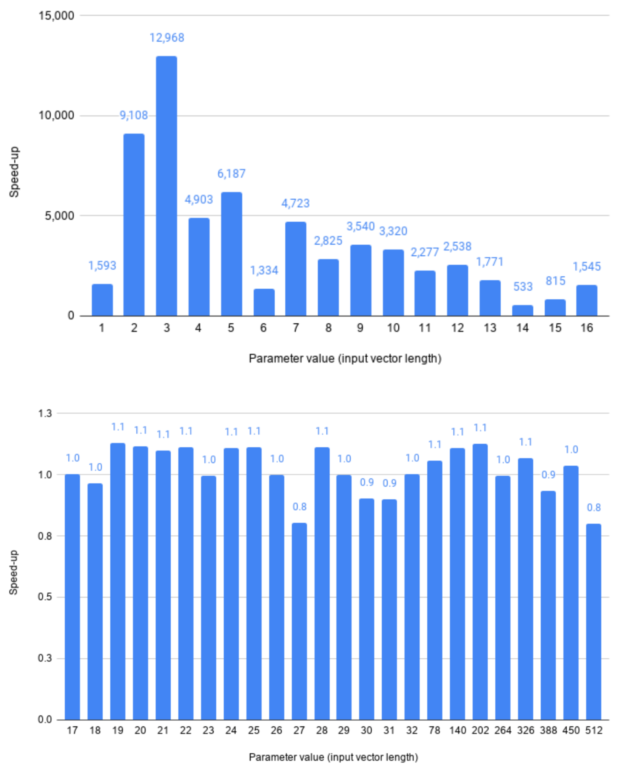

Figure 1: Speed-ups for fc, 1x occupancy

5.3.2 convolution_googlenet. The convolution example exposes a side-effect of choosing to restart the mapping process at the placement pass for the specialized GDG. At this stage, tiling decisions have been made, on the basis of a large number of iterations along the batch dimension. This resulted in "fat" tiles, leaving only a few iterations in the inter-tile dimensions resulting from the height and width dimensions, which are fixed in the input program. As a result, these inter-tile dimensions do not pass the occupation test and the sequential code is used.

Starting a full mapping for the specialized version would have had the potential to use these loop dimensions for placement, at the expense of a doubled mapping time. We plan to evaluate the exact impact of a full mapping in further work.

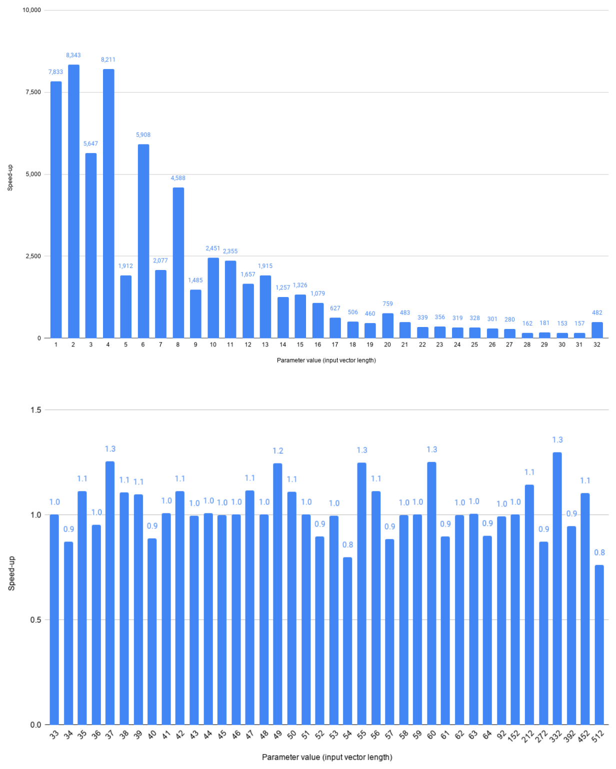

Figure 2: Speed-ups for fc, 2x occupancy

Still here, a sequential mapping performs two to three times better than a parallelization along the batch dimension for small values of the batch size.

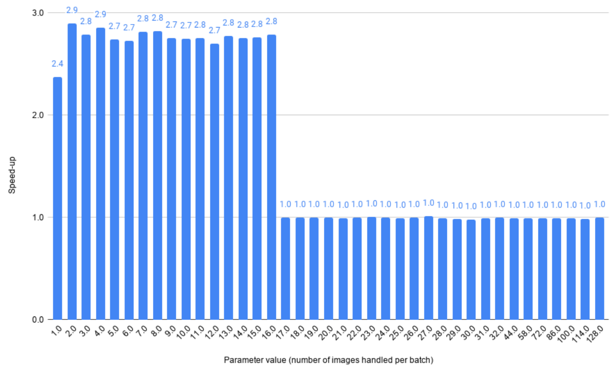

Figure 3: Speed-ups for convolution_googlenet, 1x occupancy

5.3.3 maxpool_resnet. The scenario is slightly different for maxpool, in that the bound on the second loop (OUT_H) is parametric but linked to the value of the outermost loop (HEIGHT). In this example, the other dimensions are fixed and did not pass the occupation

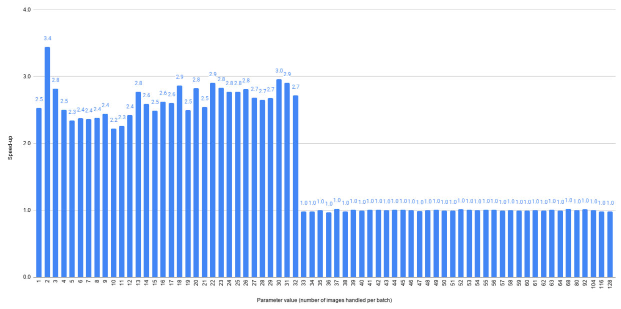

Figure 4: Speed-ups for convolution_googlenet, 2x occupancy

test. Again, we are comparing a sequential version with parallelized ones on small versions of the kernel, and they win.

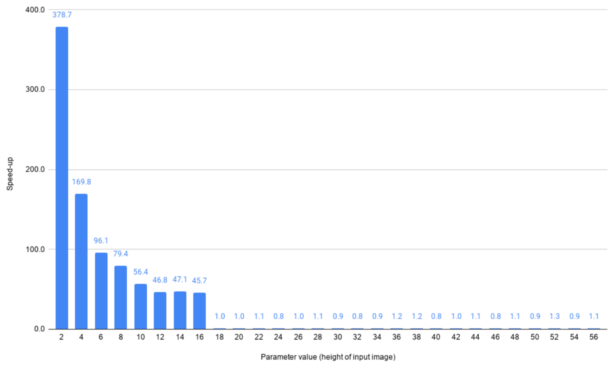

Figure 5: Speed-ups for maxpool_resnet, 1x occupancy

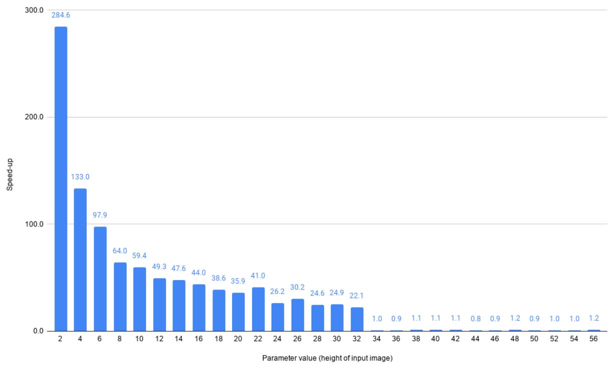

Figure 6: Speed-ups for maxpool_resnet, 2x occupancy

In all cases, the parallel branch had a self-contained #PRAGMA OMP PARALLEL FOR, while the sequential branch had none. We did not check if the sequential path avoided some of the OpenMP runtime overhead, such as starting threads. Such an overhead avoidance would contribute to the speedups.

While there is a clear benefit to using versioning in the placement pass, here we have only shown cases where a sequential (but optimized for cache locality and vectorized) version can be faster than a parallel one. In the future, we would like to explore different parameterizations of kernels and find out cases where a different parallelization is happening.

It could be argued that the absolute run time of the small-size examples is not significant enough to warrant paying the extra compilation cost, or the extra code size. The answer depends upon how often such small-size occurrences appear in the execution of the targeted program. This could be determined by profiling, i.e., instrumenting the program, running it on representative inputs, and counting the proportion of "small" cases. The result would be a decision to use the versioning option of the placement pass or not. Also, some hardware platforms have tight instruction memories, and polyhedral versioning may not be reasonable. This issue is more generally applicable to versioning, and can be tackled by exploring configurations with and without versioning through auto-tuning.

Finally, with this selection of benchmarks targeted to OpenMP, we have considered that the cost of the added conditional code was negligible. This may not be the case for more complicated specializations, which have more specialization constraints.

5.4 Compilation cost for versioning

For measuring the compilation time overhead that results from versioning, for a fixed occupancy setting and microkernel, we compiled the microkernel using R-Stream + versioning and R-Stream for ten iterations and computed the geometric mean slowdown over the ten iterations. Here is the formula we use for slowdown:

$$\frac{\text{R-Stream compilation time w/ versioning}}{\text{R-Stream compilation time of w/o versioning}}$$

As demonstrated by Figure 7, the cost of compiling code with versioning turned on is not much more than compiling with just R-Stream.

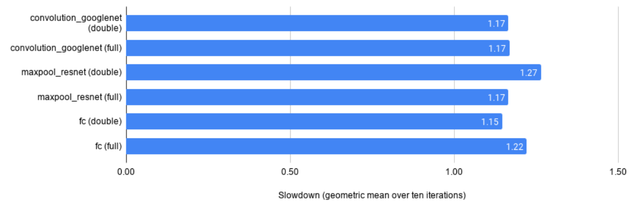

Figure 7: Versioning compilation time overhead

A method that has been used in other compilers is to use polyhedral enumerators [Clauss et al. 1997; Verdoolaege 2007] to estimate the distributed loop trip count, and dynamically set the number of threads to one if the trip count is too small. This method is specific to placement and OpenMP (not as easy to implement in other parallel programming languages), and since we are addressing a more general issue, we did not think it was worth performing a formal comparison in this paper. However, the difference is clear. Code size and compilation time would be smaller, but optimization opportunities could be lost. Also, openmp is not used in our sequential version, saving some overhead as compared to that method.

6 RELATED WORK

Versioning has been used before in the context of polyhedral compilation, although not with the same objectives.

6.1 Raising

Grosser's work [Grosser et al. 2015] de-flattens one-dimensional arrays that are accessed through polynomial functions of the loop counters and size parameters into multi-dimensional arrays with affine access functions. This process generates affine conditions on the parameters for the de-flatttening to be applicable. When these conditions are not met at run-time, an unoptimized version of the function is selected. Practically speaking, the affine conditions become part of the context of the GDG associated with the optimized function. This context is then unmodified by the mapping process.

6.2 Polyhedral JIT

PolyJIT [Simbürger et al. 2019] takes a more direct approach to polyhedral Just-In-Time (JIT) compilation, by detecting non-polyhedral functions that become polyhedral when some of its parameters are instantiated. A polyhedral compilation (using Polly [Grosser et al. 2012]) is then triggered at run time for new instances of these parameters. Metaphorically, versions produced by the PolyJIT mechanism correspond to equalities in the context of the GDG associated with the function. This system works well when a limited number of combinations of the versioned parameters are used at run time, which amortizes polyhedral compilation times over the number of executions of the function. In contrast, in our approach the number of polyhedral compilations is not dependent upon the number of dynamic instances of a GDG's parameters.

The Apollo project [Caamano et al. 2017] is a daring approach to using the polyhedral model in a Just-In-Time manner. To reduce JIT code generation time, code fragments called *code bones* are generated at compile-time and assembled into code at runtime. Run-time polyhedral dependence tests inform the code fragments assembly into parallel, speculative code, which gets JITed. Recent work on Apollo adds true runtime multi-versioning to this process [R. Lazcano 2020]. In contrast with PolyJIT, this work introduces a form of mini-auto-tuning, where the best code version for a given value of the parameters is selected from a fixed set of mappings generated at runtime, and then memoized to avoid redundant assembly overhead.

The true power of these JIT approaches is that they make the polyhedral model available to programs that cannot leverage it at compile time. In that sense, they enlarge the application domain of polyhedral optimization.

6.3 Nested conditional optimizations

We have explored some heuristics to reduce the overall number of conditionals being tested in the nested conditional code that defines which version is to be executed.

This question has been tackled in the more general context of control flow optimization of arbitrary programs. A full review of this field would not be useful in this paper, although we note that this question is particularly critical in the problem of reverse if-condition [Warter et al. 1993].

Our work differs in that we have the advantage of knowing that all our conditionals are affine relationships and that conjunctions thereof form a polyhedral context. This allows us to drive code generation based on loose and tight inclusion relationships.

Optimal ordering of conditionals can also be informed by execution traces, as in profile guided optimizations [Pettis and Hansen 1990]. However, since we are generating these conditionals from a partition of a polyhedral context, it can be more effective to compute the importance of each context at compile-time, either by using polyhedral counting methods [Clauss et al. 1997; Verdoolaege et al. 2007] or through polyhedral sampling [Meister and Clauss 2020] of the context.

7 CONCLUSION

While this paper focuses on the logistics and impact on performance of applying versioning within a polyhedral mapper, there is much to explore. We touched upon trade-offs made to avoid paying for improved run-time performance with an explosion of versions and a subsequently long compilation time. While we focused on placement, several other polyhedral passes can benefit from specialization, raising new questions. How should the number of created versions be controlled, in order to get the best trade-off between compilation time with run-time performance? We are also curious about which polyhedral passes best benefit from versioning for a given target machine. Comprehensive uses of versioning will also better leverage the qualities of our conditional code generation heuristics.

Even though the placement use case could be explored further, by identifying parametric kernels that trigger more complex mapping behaviors than the ones observed in this paper, we believe this work successfully demonstrates the usefulness of compile-time versioning in the polyhedral model.

ACKNOWLEDGMENTS

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

2008. Cbc User's Guide: `http://www.coin-or.org/Cbc/index.html`.

Ph. Clauss A. Jimborean, V. Loechner. 2011. Handling Multi-Versioning in LLVM: Code Tracking and Cloning. In *WIR 2011: Workshop on Intermediate Representations, in conjunction with CGO.*

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

Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code.

OpenMP Architecture Review Board. 2020. OpenMP Application Programming Interface, Version 5.1.

U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI '08).* Tucson, Arizona. http://www.cse.ohio-state.edu/~bondhugu/publications/uday-pldi08.pdf

Juan Manuel Martinez Caamano, Aravind Sukumaran-Rajam, Artiom Baloian, Manuel Selva, and Philippe Clauss. 2017. APOLLO: Automatic speculative POLyhedral Loop Optimizer. In *7th International Workshop on Polyhedral Compilation Techniques (IMPACT)* (Stockholm, Sweden).

X. Chen and S. Long. 2009. Adaptive multi-versioning for OpenMP par-allelization via machine learning. In *15th Int. Conf. on Parallel andDistributed Systems (ICPADS).*

Philippe Clauss and V. Loechner. 1998. Parametric Analysis of Polyhedral Iteration Spaces. *J. VLSI Signal Process. Syst.* 19, 2 (1998), 179–194. https://doi.org/10.1023/A:1008069920230

P. Clauss, V. Loechner, and D. Wilde. 1997. Deriving formulae to count solutions to parameterized linear systems using Ehrhart polynomials: Applications to the analysis of nested-loop programs. citeseer.ist.psu.edu/clauss97deriving.html

A. Darte, Y. R., and F. Vivien. 2000. *Scheduling and Automatic Parallelization.* Birkhäuser.

Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time. *International Journal of Parallel Programming* 21, 5 (Oct. 1992), 313–348. citeseer.ist.psu.edu/feautrier92some.html

Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 4 (2012). http://dblp.uni-trier.de/db/journals/ppl/ppl22.html#GrosserGL12

Tobias Grosser, J. Ramanujam, Louis-Noel Pouchet, P. Sadayappan, and Sebastian Pop. 2015. Optimistic Delinearization of Parametrically Sized Arrays. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15).* Association for Computing Machinery, New York, NY, USA, 351–360. https://doi.org/10.1145/2751205.2751248

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

L. Luo, Y. Chen, C. Wu, S. Long, and G. Fursin. 2009. Finding representative sets of optimizations for adaptive multiversioning applications. In *International Workshop on Statistical and Machine learning ap-proaches to ARchitectures and compilaTion* (Paphos Chypre).

Benoît Meister and Philippe Clauss. 2020. Uniform Random Sampling in Polyhedra. In *IMPACT 2020 - 10th International Workshop on Polyhedral Compilation Techniques.* Bologna, Italy. https://hal.inria.fr/hal-02425752

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.

Karl Pettis and Robert C. Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'SO Conference on Programming Language Design and Implementation* (White Plains, New York).

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).*

E. Juárez Ph. Clauss R. Lazcano, D. Madroñal. 2020. Runtime multi-versioning and specialization inside a memoized speculative loop optimizer. In *Compiler Construction* (San Diego, CA, USA). 96–107.

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

A. Schrijver. 1998. *Theory of Linear and Integer Programming.* Wiley. https://books.google.com/books?id=zEzW5mhppB8C

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

M. M. Strout, M. Hall, and C. Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934. https://doi.org/10.1109/JPROC.2018.2857721

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

Nicolas Vasilache, Muthu Baskaran, Tom Henretty, Benoît Meister, M. Harper Langston, Sanket Tavarageri, and Richard Lethin. 2013. A Tale Of Three Runtimes. arXiv:1409.1914.

S. Verdoolaege. 2007. `barvinok`*: User Guide (version 0.25)*. http://www.kotnet.org/~skimo/barvinok/barvinok.pdf

Sven Verdoolaege. 2010. isl: an integer set library for the polyhedral model. In *Proceedings of the Third international congress conference on Mathematical software (ICMS'10)*. ACM Press, 299–302.

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and Maurice Bruynooghe. 2007. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica* 48, 1 (2007), 37–66. https://doi.org/10.1007/s00453-006-1231-0

Nancy Warter, Scott Mahlke, Wen mei Hwu, and B. Rau. 1993. Reverse If-Conversion. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. 290–299. http://citeseer.ist.psu.edu/warter93reverse.html

Bichen Wu, Forrest N. Iandola, Peter H. Jin, and Kurt Keutzer. 2016. SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving. *CoRR* abs/1612.01051 (2016). arXiv:1612.01051 http://arxiv.org/abs/1612.01051

P. Zhao and J. N. Amaral. 2005. Function outlining and partial inlining. In *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*. 101–108. https://doi.org/10.1109/CAHPC.2005.26