Coarsening Optimization for Differentiable Programming

XIPENG SHEN* and GUOQIANG ZHANG, Facebook & North Carolina State Univ., United States
IRENE DEA, SAMANTHA ANDOW, and EMILIO ARROYO-FANG, Facebook, Inc., United States
NEAL GAFTER, JOHANN GEORGE, and MELISSA GRUETER, Facebook, Inc., United States
ERIK MEIJER, OLIN GRIGSBY SHIVERS, and STEFFI STUMPOS, Facebook, Inc., United States
ALANNA TEMPEST, CHRISTY WARDEN, and SHANNON YANG, Facebook, Inc., United States

This paper presents a novel optimization for differentiable programming named *coarsening optimization*. It offers a systematic way to synergize symbolic differentiation and algorithmic differentiation (AD). Through it, the granularity of the computations differentiated by each step in AD can become much larger than a single operation, and hence lead to much reduced runtime computations and data allocations in AD. To circumvent the difficulties that control flow creates to symbolic differentiation in coarsening, this work introduces ϕ-*calculus*, a novel method to allow symbolic reasoning and differentiation of computations that involve branches and loops. It further avoids "expression swell" in symbolic differentiation and balance reuse and coarsening through the design of *reuse-centric segment of interest identification*. Experiments on a collection of real-world applications show that *coarsening optimization* is effective in speeding up AD, producing several times to two orders of magnitude speedups.

CCS Concepts: • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: differentiable programming, compiler, program optimizations, SSA, Calculus

ACM Reference Format:
Xipeng Shen, Guoqiang Zhang, Irene Dea, Samantha Andow, Emilio Arroyo-Fang, Neal Gafter, Johann George, Melissa Grueter, Erik Meijer, Olin Grigsby Shivers, Steffi Stumpos, Alanna Tempest, Christy Warden, and Shannon Yang. 2021. Coarsening Optimization for Differentiable Programming. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 130 (October 2021), 28 pages. https://doi.org/10.1145/3485507

1 INTRODUCTION

A program written with differentiable programming can be differentiated automatically. The differentiation results can then be used for gradient-based optimization (e.g., gradient descent) of the parameters in the program.

Differentiable programming have been used in scientific computing, physics simulations, and other domains to help mitigate the burden of manual error-prone coding of derivative computations. Recent several years have witnessed a growing interest of differentiable programming

*Contact author: Xipeng Shen (xshen5@ncsu.edu). Work done while the first two authors were with Facebook Inc.

Authors' addresses: Xipeng Shen, xshen5@ncsu.edu; Guoqiang Zhang, gzhang9@ncsu.edu, Facebook & North Carolina State Univ., Raleigh, United States; Irene Dea, irenedea@fb.com; Samantha Andow, samdow@fb.com; Emilio Arroyo-Fang, earroyof@fb.com, Facebook, Inc., Menlo Park, United States; Neal Gafter, nmgafter@fb.com; Johann George, jog@fb.com; Melissa Grueter, melissagrueter@fb.com, Facebook, Inc., Menlo Park, United States; Erik Meijer, erikm@fb.com; Olin Grigsby Shivers, olinshivers@fb.com; Steffi Stumpos, stumpos@fb.com, Facebook, Inc., Menlo Park, United States; Alanna Tempest, atem@fb.com; Christy Warden, christywarden@fb.com; Shannon Yang, shannony@fb.com, Facebook, Inc., Menlo Park, United States.

2475-1421/2021/10-ART130
https://doi.org/10.1145/3485507

in machine learning (ML) [Baydin et al. 2018; van Merriënboer et al. 2018] and Probabilistic Programming [Tehrani et al. 2020], to accommodate the needs of various customized ML operators, user-defined operations in the learning targets (e.g., the physical environment of reinforcement learning) and statistical sampling.

The key technique in differentiable programming is *automatic differentiation*. For a program (P) that produces output (y) from some given values (X), automatic differentiation automatically computes the derivatives ($\partial y/\partial x$) ($x \in X$) without the need for users to write the differentiation code. The given program P is called the *primal* code, and x is called an *active input variable*.

Existing approaches of automatic differentiation fall into two categories: (i) *Symbolic differentiation*, which uses expression manipulation in computer algebra systems, (ii) *Algorithmic differentiation*, which performs a non-standard interpretation of a given computer program by replacing the domain of the variables to incorporate derivative values and redefining the semantics of the operators to propagate derivatives per the chain rule of differential calculus (elaborated in Section 2).

Symbolic differentiation has been commonly regarded inappropriate for differentiable programming, for several reasons: (i) It results in complex and cryptic expressions plagued with the problem of "expression swell" [Cor 1988]. (ii) It requires models to be defined as closed-form expressions, limiting the use of control flow and other features that are common in computer programs.

Consequently, existing differentiable programming systems are all based on *algorithmic differentiation (AD)*. Algorithmic differentiation computes derivatives through accumulation of values during code execution to generate numerical derivative evaluations. In contrast with the effort involved in arranging code as closed-form expressions under the syntactic and semantic constraints of symbolic differentiation, algorithmic differentiation can be applied to regular code, allowing branching, loops, and other language features. Some examples are Autograd [Maclaurin 2016], PyTorch [Paszke et al. 2017], JAX [Bradbury et al. 2018], and Zygote [Innes 2020].

In this work, we advocate for a hybrid approach for differentiable programming. This new approach seamlessly integrates symbolic differentiation with algorithmic differentiation through *coarsening*, a compiler-based technique we introduce in this work.

The motivation of the new approach is to eliminate the large overhead in AD incurred by its fine-grained differentiation and operation overloading. Rather than differentiation at each operation, this new approach tries to enlarge the granularity to a sequence of operations, hence the name "coarsening optimization". It identifies a part of the to-be-differentiated computations that are amenable for symbolic differentiation, elevates it to a high-level symbolic representation, applies symbolic differentiation on it, generates the code, and then integrates it back into the computation flow of AD.

By doing that, the coarsening optimization gives four-fold benefits: (i) It avoids many calls to the fine-grained differentiation functions and the creations of many intermediate results; (ii) the symbolic representation makes it easy to directly benefit from expression simplifications by existing symbolic engines and hence leads to more efficient code being generated; (iii) it can form a synergy with computation reuse and hence amplify the benefits; (iv) it can sometimes remove the unnecessary primal computations. If what users want is only the derivative of a function, current AD still needs to run the primal computation because of the nature of its differentiation process (Sec 2). But if coarsening can be applied to the entire function, then only its generated differentiation function needs to run, foregoing the executions of the primal function.

In addition to those benefits, coarsening features several appealing properties: (i) As the coarsening optimization typically happens at compile time, it trades a slight increase of compile time for significant runtime savings; (ii) functioning as a way to add "shortcuts" to AD, it can be seamlessly integrated into both forward and backward differentiation; (iii) it applies regardless whether the gradients are for first or higher order optimizations.

To materialize the optimization, there are several major challenges.

Challenge I: Complexities from control flow (e.g., branches, loops). Symbolic differentiation requires a closed form of the computation, which has been regarded as difficult for code involving complex control flow. Limiting coarsening to the code segments between the appearances of such complexities in a program would result in many short code segments, leaving many optimization opportunities submerged and much power of coarsening untapped.

Challenge II: "Expression swell". "Expression swell" is a criticism to symbolic differentiation mentioned in some literature, which refers to the observation that the derivative often has a much larger representation than the original function has [Cor 1988]. For instance, in a straightforward implementation, the derivative of the multiplication of n terms becomes an expression with n^2 terms: $d(f_1 f_2 \cdots f_n)/dx = \sum_i d(f_i)/dx \prod_{j \neq i} f_j$.

Challenge III: Tension with computation reuse. Some calculations in the primal computation may be also required in the differentiation (e.g., $e^{X \cdot \beta}$ is part of both $(1 + e^{X \cdot \beta})$ and its differentiation over β, $Xe^{X \cdot \beta}$). Reuse opportunities can also exist between different parts of differentiation. The fine-grained operations in algorithmic differentiation already build on such reuses. But in coarsened differentiation, without a careful design, such reuse opportunities can get lost as the symbolic transformation reorders and reorganizes the involved calculations. On the other hand, naively maximizing computation reuse would limit the granularity of coarsening. So there is a challenge in reconciling the tension between reuse and coarsening.

We address the challenges through two major innovations. (i) For challenge I, we introduce *ϕ-calculus*, a novel method that allows symbolic reasoning and differentiation of computations with complex control flow. Building on the *ϕ-function* in single static assignment (SSA), *ϕ-calculus* makes the derivation of a closed form possible for computations involving complicated control flow. It further offers a set of formulae for symbolically reasoning about and differentiating the closed forms that involve ϕ-functions. (ii) For challenges II and III, we propose *reuse-aware SOI identification* as a way to identify the code segments of interest (SOI) for coarsening. It can strike a good tradeoff between coarsening and reuse, and at the same time keep the effects of "expression swell" under control.

Based on an AD tool for Kotlin (DiffKt), we evaluated coarsening on 18 settings of six applications on two machines. The results show that coarsening is effective in significantly expanding the applicable scope of symbolic differentiation, and hence dramatically reducing the runtime overhead of AD. The performance improvement is substantial, 1.03×-27× speedups of the differentiation and 1.08×-11× speedups of the end-to-end application execution. We further examined the potential of coarsening on several other AD tools (Zygote [Innes 2020] for Julia, Jax [Bradbury et al. 2018] for Python, Adept [Hog 2014] for C++) by experimenting with the implementations of the symbolic differentiation results in their corresponding languages. The speedups from the coarsening results are even greater, 66×-335×, indicating the potential of coarsening in serving as a general optimization technique for AD.

To the best of our knowledge, this is the first work that proposes a systematic approach to integrating symbolic differentiation with algorithmic differentiation for differentiable programming. The developed $\phi-$calculus offers the first method to enable symbolic differentiation of computations spanning over complex control flow. The resulting hybrid differentiation approach gets the best of both worlds, that is, the efficiency from the compile-time symbolic differentiation and the generality of AD.

In summary, this work makes the following contributions:

- It introduces *coarsening optimization*, the first approach to systematical integration of symbolic differentiation into algorithmic differentiation for general programs.

- It develops *ϕ-calculus* that eliminates the barriers of control flow to symbolic differentiation.
- It proposes *reuse-aware SOI identification* to balance reuse and coarsening.
- It validates the benefits of coarsening, confirming its potential for significantly improving AD efficiency.

2 BACKGROUND AND TERMINOLOGY

At the foundation of AD is the chain rule. We explain it in a simple setting. Suppose y is the output of a sequence computations on input x, and y_i ($i = 1, 2, \cdots, k$) are the intermediate results produced during the sequence of computations from x to y, that is, $y_1 = f_1(x), y_2 = f_2(y_1), \cdots, y_k = f_k(y_{k-1}), y = f(y_k)$. The chain rule says that the derivative of y on x (or called *x's gradient* regarding y) can be computed as follows:

$$dy/dx = dy/dy_k * dy_k/dy_{k-1} * \cdots * dy_1/dx$$

In a program, besides the variables relevant to the deriatives of interest, there can be many other variables. To distinguish them, we call the relevant output variables like y *active output variables*, relevant input variables like x *active input variables*, and other relevant variables simply *active variables*. Further, we call the computations in the original program from active input variables to active output variables *primal computations*, and the computations to compute the derivatives *gradient computations*.

There are two ways to interpret the chain rule, which lead to the *forward* and *backward* AD respectively. We explain them by assuming an implementation of AD via operator overloading, the most common way of implementation of AD.

The first is to regard the rhs of the chain rule a sequence of computations from the rightmost term to the leftmost term. Corresponding to AD implementation, the derivatives of dy_1/dx is computed as a side step of operator overloading when y_1 is computed from x in the primal computation, and the result is then passed to the next step of primal computation, which computes y_2 and dy_2/dy_1 and then multiplies it with the received value of dy_1/dx. The process continues and produces dy/dx eventually. This implementation is called *forward AD*.

The second way is to regard the rhs of the chain rule a sequence of computations from the leftmost term to the rightmost term. In this case, at each step in the *primal* computation (which is still right-to-left), some operations needed for differentiation are recorded in a data structure, such as a stack [Hog 2014], as part of the operations of the overloaded operators. When the primal computations reach the last step and the gradient computation actually starts, the operations on the stack are executed in a backward order, starting from those of the leftmost term in the rhs of the chain rule. After dy/dy_k is computed, the result is passed to the next step, which computes dy_k/dy_{k-1} and then multiples it with the received value of dy/dy_k. The process continues until the gradient of x is computed. This implementation is called *backward AD*.

Backward AD is a more popular choice in existing AD tools because it is overall more efficient in general settings [Margossian 2019]. The two methods are sometimes used together. Note that in both of them, primal computations are necessary to run so that the overloaded operators can take place, even if what the user wants are just the gradients. Coarsening optimization can lift such a requirement as shown later in this paper.

In cases that are not differentiable (e.g., x=0 in $relu(x)$), AD tools approximate the gradients (e.g., using 0 at $relu(0)$); coarsening optimization preserves the same behavior.

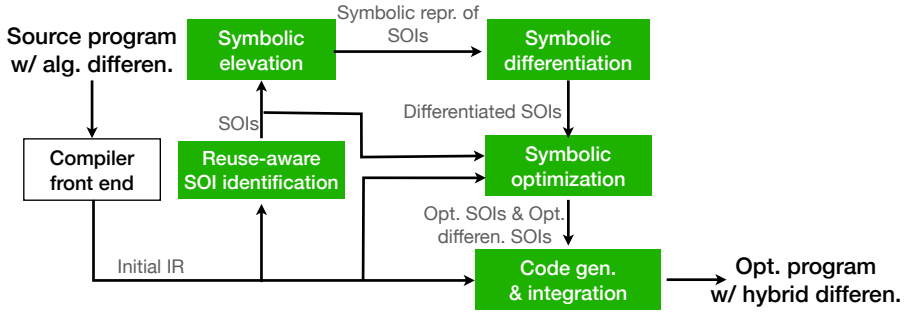


Fig. 1. The overall workflow of coarsening for AD. Solid boxes are the main components in coarsening.

3 OVERVIEW OF COARSENING OPTIMIZATION

The basic definition of *coarsening optimization* for AD is as follows:

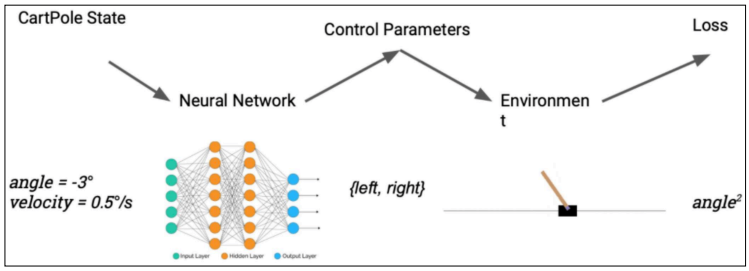
Definition 1. *Let S be a sequence of program statements that implement the computations from active input I to active output P.* Coarsening optimized AD *is applied to S if a closed form F is produced that captures the computations in S and F goes through a symbolic differentiation with the results integrated into the AD process.*

Figure 1 shows the high-level workflow of coarsening for AD. The input is a program written in a certain AD-based differentiable programming language. Coarsening works on the intermediate representation (IR) output from the front end of the default compiler. From it, the *reuse-aware SOI identification* component identifies the code segments of interest (SOIs), which are sent to the *symbolic elevation* component to produce a symbolic representation of the SOIs. The *symbolic differentiation* component takes them in and outputs the symbolic form of the differentiated SOIs. The *symbolic optimization* component optimizes both the SOIs and the differentiated SOIs while drawing on their contexts captured in the original IR. The optimizations include simplifications via algebra systems, as well as identifying the places for profitable computation reuses between SOIs and the differentiated SOIs.

Coarsening optimization can be applied for AD at both compilation and runtime. We take compile-time optimization as the context of discussion.

Example. To help convey the intuition of coarsening optimization, we use CartPole as an example. CartPole is an example that uses deep reinforcement learning (DRL). As illustrated in Figure 2 (a), a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The cart is controlled by applying a force of +1 (to the right) or -1 (to the left) on the cart. The pendulum starts upright, and the goal is to learn a cart control policy to prevent the pendulum from falling over. The learning system consists of a Neural Network and a simulator of the cart and pole. At each time step, the system goes through the computation outlined in the inner loop body in Figure 2 (b), that is, calculating the output of the Neural Networks from the current cart and pole's states to decide the action for the cart to take, based on which, it then updates the states of the cart and pole according to the physics model. This process continues for another two time steps. The resulting total loss is then used in updating the weights of the Neural Networks via gradient descent. For each weight w in the Neural Networks, (i) the program obtains the value d_w for the derivative of w w.r.t. the loss, (ii) d_w is then used to create the differential $\lambda z.d_w * z$, and (iii) the value of weight w is updated by the result of evaluating the differential w.r.t. learning rate η:

$$w = w - (\lambda z.d_w * z)(\eta).$$

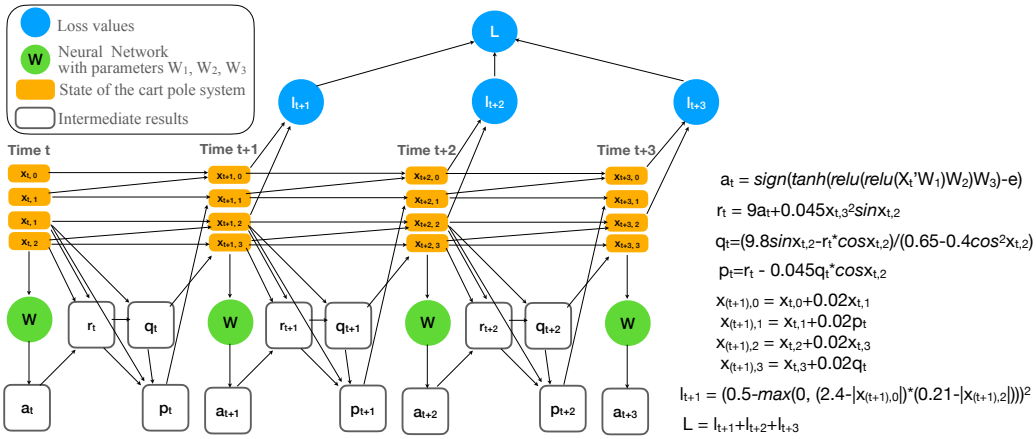


(a) Illustration of the CartPole problem. The goal is to learn a cart control policy to prevent the pendulum from falling over

```
Input:
  X0: initial states
  W0: initial Neural Networks parameters
Output:
  W: final parameters of Neural Networks

t = 0
while (loss > threshold)
    loss = 0
    for (i=0; i< B; i++)
        a = getAction(Xt+i, W)
        Xt+i+1 = updateState(Xt+i, a)
        loss += getLoss(Xt+i+1)
    backpropagate(loss, W, X)
    t += B
```

(b) Pseudo-code of the training harness of CartPole



(c) Computation flow of the forward pass of CartPole with the main computations shown on the right.

Fig. 2. A running example CartPole. (a) Problem illustration (artwork source: fluxml.ai); (b) Pseudo-code of the training harness; (c) The simplified computation graph of the forward pass and the core computations.

The learning then continues until the Neural Networks converge. The right part of Figure 2 (c) shows the core computations of states update in each iteration.

Intuition of Benefits. We can get some intuition about the potential benefits of coarsening by checking the differentiation of a state at time $t+2$, $x_{t+2,0}$, over the action at time t, a_t, in Figure 2. The result is needed in the computation of the gradient of a_t regarding the loss L_{t+2} and the total loss L. (The computation from the weight W to a is through Neural Networks, which have a standard structure; the gradient calculation of that part is through a highly polished vendor-provided library rather than the AD tool. Therefore, the actual effects of the AD on that benchmark is from the loss to a.)

If we put down the entire computation from a_t to $x_{t+2,0}$, we get the following (with constants already folded):

$$x_{t+2,0} = x_{t,0} + 0.04x_{t,1} + 0.003636a_t + 0.000018{x_{t,3}}^2 sinx_{t,2} - \frac{0.00016sinx_{t,2} - 0.00016a_t cos^2 x_{t,2} - 0.000008{x_{t,3}}^2 sinx_{t,2}cos^2 x_{t,2}}{(0.65 - 0.405cos^2 x_{t,2})} \tag{3.1}$$

It consists of 29 operations. If a (backward) AD library is used to get its differentiation over a_t, during the primal computation, at each of the 29 operations, a pullback function is generated for the differentiation of that operation, along with the closure and some intermediate objects allocated to hold the intermediate results that the differentiation would need to use.

In contrast, if symbolic differentiation is applied to the expression in Equation 3.1, the result is much simpler as shown as follows. The differentiation would then need to just make an invocation to one function that consists of only several straight-line calculations. Besides saving computations, it also saves the allocations of many intermediate objects.

$$\frac{d(x_{t+2,0})}{d(a_t)} = 0.003636 + \frac{0.00016cos^2 x_{t,2}}{0.65 - 0.405cos^2 x_{t,2}} \tag{3.2}$$

Besides the benefits demonstrated by the CartPole example, two other benefits are worth mentioning. First, because AD libraries are typically implemented via operator overloading, they have to wrap data objects in a special type (e.g., Tensor in PyTorch) so that customized operations can be invoked during the primal computations to implement the needed AD operations. Accesses to the data objects are therefore subject to the boxing overhead. Inside the code generated by the symbolic differentiation, as no operator overloading is needed, unboxed data objects can be directly used, reducing the boxing overhead. This benefit is especially prominent when the data involved are a collection of scalars or small vectors as in many physical simulations or Probablistic Programming applications (examples in Sec 7).

The other benefit of coarsening not captured by the CartPole example is the cancellation of terms or other simplifications that symbolic transformation can often harness. We explain it with a simple expression involving two matrices (X_1, X_2) and one vector (v): $(X_2 v)^T (X_1 X_2 v)$. Its symbolic differentiation over v can easily combine terms with common multipliers, yielding a form $X_2^T (X_1 + X_1^T) X_2 v$, significantly simpler than what the default AD would compute, $X_2^T (X_1 X_2 v) + ((X_2 v)^T X_1 X_2)^T$. Simplification of symbolic expressions is a common feature in symbolic engines; for more examples, please refer to the simplification module in Sympy [sym [n.d.]].

As mentioned in Section 1, to make *coarsening optimization* effective, there are three main challenges: control flows, "expression swell", and tension between coarsening and reuse. We next explain our solutions to these challenges.

4 ADDRESSING CONTROL FLOW: ϕ-CALCULUS

Control flow complexities are commonly perceived obstacles for symbolic differentiation. For straight-line code, it is easy to derive a closed form for the computations by symbolically substituting later references with their earlier definitions in the code. In the presence of control flow branches, the complexity increases exponentially: If we build a closed form for each possible path, in the worst case, there would be $O(2^B)$ closed forms for B conditional statements. That would not only increase the amount of work and execution time of symbolic differentiation, but also complicate compiler-based code generation from the differentiation results. The problem worsens when there are loops mixed with if-else. Without a closed form, symbolic differentiation cannot apply.

Without an effective way to handle control flow, coarsening can apply only to small pieces in a program, with each piece consisting of the code between two adjacent control flow branching or merging points. For some programs, that could lead to only small optimization scopes, leaving the power of coarsening optimization untapped.

We address the problem by proposing ϕ-calculus. It consists of a set of notations for symbolically representing loops and conditional statements, and introduces a series of formulae to facilitate

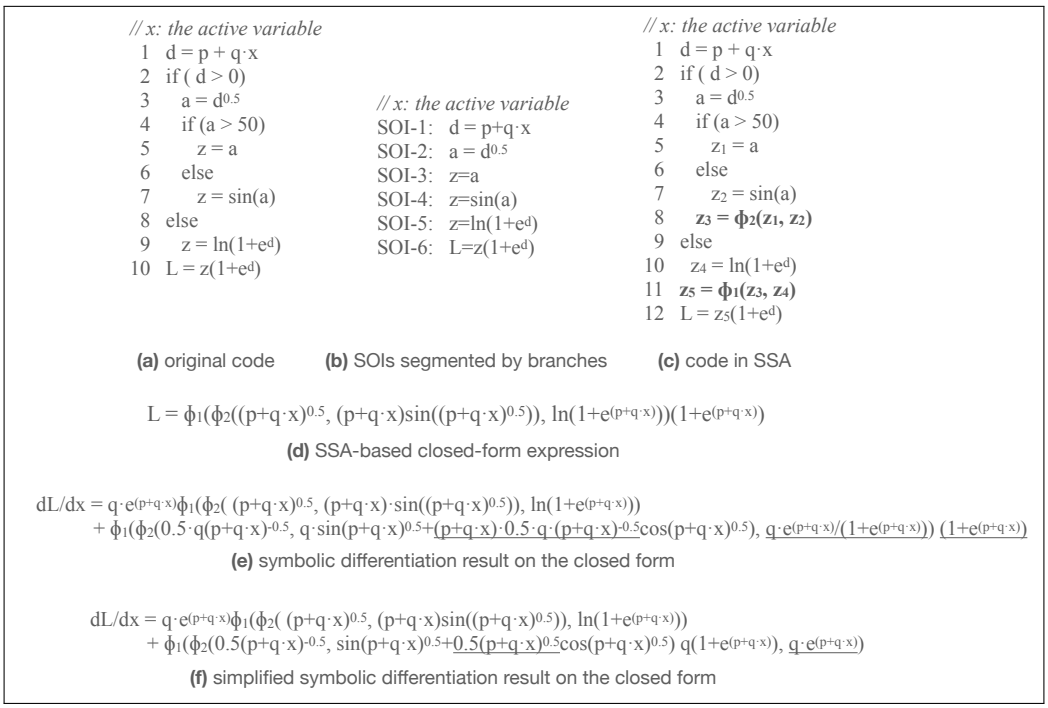


Fig. 3. Illustration of SSA and ϕ-calculus on an example with conditional statements

the reasoning and differentiation on the extended symbolic form. As ϕ-calculus is inspired by the concept of ϕ functions in SSA, we first give a quick review of SSA.

4.1 Background on SSA and ϕ Functions

Static Single Assignment form (SSA) is a kind of code representation widely used in modern compilers [Aho et al. 2006; Cytron et al. 1989]. Code in SSA has two properties: (i) no two static assignments assign values to the same variable; (ii) every reference refers to the value defined by a single static assignment. It uses a special ϕ function to resolve name ambiguities. A ϕ function "chooses" the right name among its (two or more) arguments based on the actual control flow.

Figure 3(c) shows the SSA form of the code in Figure 3(a). There are three assignments to z in the original code. They are all replaced with different names z_1, z_2, z_4. Meanwhile, two ϕ functions are inserted in Figure 3(c), with the one on line 8 resolving the name ambiguity caused by the inner if-else, and the one on line 11 for the outer if-else.

Figure 4(b) shows the SSA form of the loop in Figure 4(a). The loop structure involves two merging points, with one at the entry (L1), the other at the exit (L2). There are two ϕ functions at the entry, respectively for variables s and i; there is one ϕ function at the exit, for variable s. The former are called *entry ϕ-functions* and the latter is called *exit ϕ-function* of the loop [Ottenstein et al. 1990]. All loops, either regular or irregular (e.g., while loops with breaks), have such pairs of *ϕ-functions*.

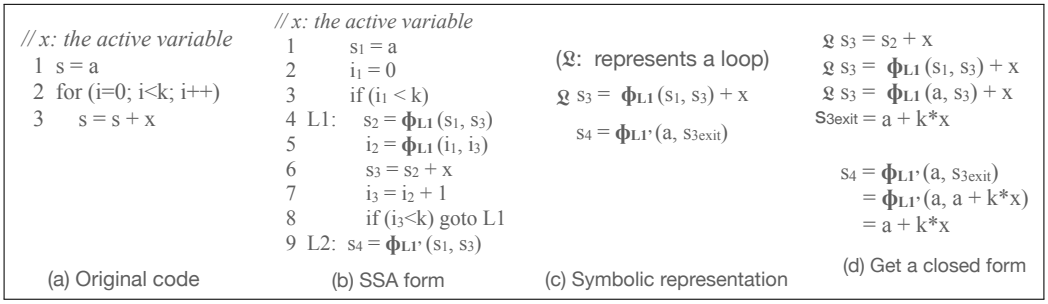


Fig. 4. Illustration of SSA and ϕ-calculus on a simple loop

4.2 Notations in ϕ-Calculus

Inspired by SSA, ϕ-calculus uses ϕ functions and other notations to symbolically represent conditional statements and loops:

- $\phi(a_1, a_2, \cdots, a_k)$: the standard ϕ function in SSA. Numerical subscripts are sometimes added to a standard ϕ function to distinguish ϕ functions that have different conditions.
- $\phi_{Li}(a_1, a_2, \cdots, a_k)$ & $\phi_{Li'}(a_1, a_2, \cdots, a_k)$: the entry and exit ϕ functions of loop i.
- ${\mathfrak{L}_i}^u \ < S >$: the computations in a statement S are surrounded by loop i with u iterations. Sometimes, the loop ID (i) is used in S to also denote the iteration number, which, by default, goes from 0 to $u-1$. In the representation of a loop for symbolic differentiation in coarsening, u can be a constant, an expression, or a symbol. For irregular loops (e.g., the `while` loop in Figure 6), for instance, u is a symbol in the expression that is symbolically differentiated, and its value is recorded in the execution of the primal code. In the following discussion, unless necessary, we omit u and/or i in the loop notations for better readability.
- $\sum$, $\prod$: the standard math notations of summation and product.
- ${\phi_{Li}}^{(j)}$: the instance of ϕ_{Li} in the jth iteration of loop i
- $a^{(j)}$: if a is an expression in the argument list of ϕ_{Li}, $a^{(j)}$ represents the value of a after j iterations of loop i.
- $a_{exit(L)}$: the value(s) of a at the exit of loop L.
- $f^{[n]}$: recursively apply function f for n times.

This set of notations are simple extensions of the standard ϕ function. But with them, code with complex control flow can now be symbolically expressed. Figure 3(d), for instance, shows the symbolic form of the computation of L by the code in Figure 3(a). Figure 4(c) shows the computation of s by the loop in Figure 4(a). The derivation of them involve just direct substitutions of names with their corresponding expressions; the ϕ-notations offer ways to symbolically represent the effects of loops and conditional statements.

Note that unlike the form in Figure 3(d), in Figure 4(c) is not yet a closed form: There is still the presence of $\mathfrak{L}$. Even for the closed form in Figure 3(d), it still contains ϕ functions. The other component of ϕ-calculus, *formulae in ϕ-calculus*, offers the facilities for (i) getting closed forms by removing $\mathfrak{L}$ and (ii) differentiating expressions involving ϕ functions.

4.3 Formulae in ϕ-Calculus

Figure 5 provides the core set of formulae in ϕ-calculus; at the top of the figure are five fundamental formulae and at the bottom are nine useful corollaries derived from the fundamental formulae. Most of the formulae are quite straightforward, but when being used together, they are powerful in

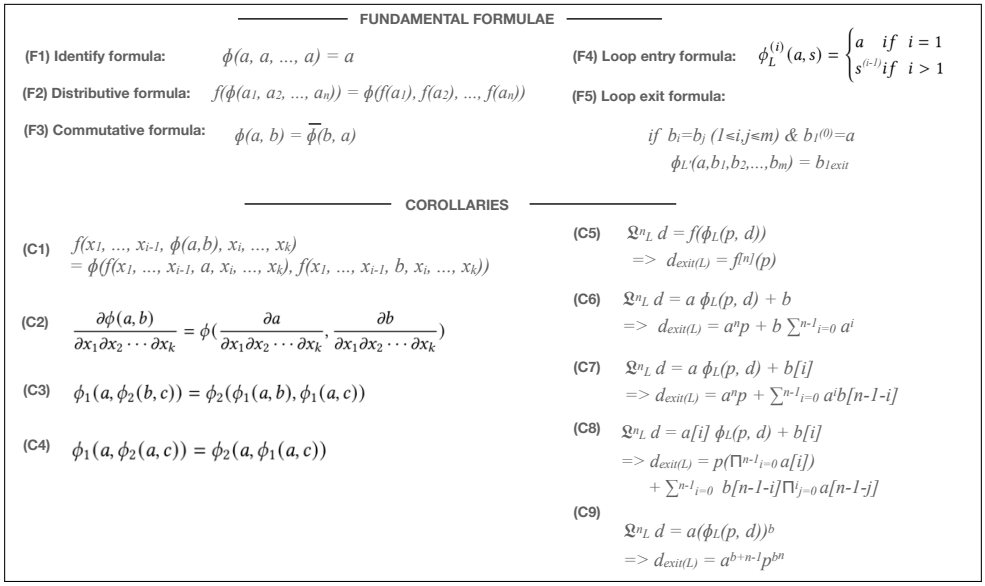


Fig. 5. Main formulae in ϕ-calculus.

getting rid of ϕ and loop notations from the symbolic representation of code. We next explain each of the formulae and provide brief proofs.

4.3.1 *Fundamental ϕ Formulae.*

1) **Identity Formula (F1 in Figure 5)**. The identity formula says that if all the arguments in a ϕ function equal to one another, the ϕ function can be replaced with any of its argument. It immediately follows the definition of the ϕ function.

2) **Distributive Formula (F2 in Figure 5)**. This formula says that a function that applies to a ϕ function can be distributed to each of the arguments of that ϕ function. It can be easily proved based on the definition of ϕ function.

3) **Commutative Formula (F3 in Figure 5)**. This formula shows the relationship between a ϕ function and its complement. In the formula, $\overline{\phi}$ is the *complement* of ϕ, that is, it chooses the first argument when ϕ chooses the second, and the second when ϕ chooses the first. The correctness of this formula immediately follows the definition.

4) **Loop entry formula (F4 in Figure 5)**. This formula shows the inherent property of a *loop-entry ϕ function*. For the definition of a *loop-entry ϕ function*, ϕ_L is reached always through the back edge of loop L except for its first instance in that loop. The formula hence follows. This simple formula is essential for ϕ-calculus to deal with loops as shown later.

5) **Loop exit formula (F5 in Figure 5)**. This formula says that $\phi_{L'}(a, b_1, b_2, \cdots, b_m)$ equals the value of b_1 at the exit of loop L if (i) the value of all arguments, except the first, of $\phi_{L'}$ are the same at $\phi_{L'}$, and (ii) those arguments before the entry point of the loop have the value equaling the first argument's value a. Its correctness can be easily proved with the *identity formula*. Notice that the only time when $\phi_{L'}$ takes its first argument is when the entire loop is skipped, in which condition, according to (ii), b_{1exit} equals a; in any other condition, $\phi_{L'}$ must take one of the other arguments, the value of which at the exit of the loop, according to (i), must equal b_{1exit}. This formula is useful for removing loop exit ϕ functions in the application of ϕ-calculus as shown later.

4.3.2 Corollaries. At the bottom of Figure 5 are some of the corollaries attained from the fundamental formulae. They provide facilities for transforming and simplifying ϕ expressions.

The corollaries are in two groups. The first group consists of C1 to C4. These corollaries offer conveniences for symbolic differentiation, optimizations, and code generations, reducing computations and code size. Corollary C1 can be easily derived from the *distributed formula* through currying. Corollary C2 follows corollary C1 when we substitute f with partial derivative. Corollary C3 follows corollary C1 when we substitute f with the ϕ function. Corollary C4 is attained when we apply C1 and then the *identity formula* (F1).

The other group consists of corollaries C5 to C9, which offer conveniences for transforming ϕ expressions into closed forms for symbolic differentiation.

Corollary C5 is proved as follows.

Proof. Because of F4, we have the following relations:

$$d^{(1)} = f(\phi_L^{(1)}(p, d)) = f(p) \tag{4.3}$$

$$d^{(2)} = f(\phi_L^{(2)}(p, d)) = f(d^{(1)}) = f(f(p)) = f^{[2]}(p) \tag{4.4}$$

$$d^{(3)} = f(\phi_L^{(3)}(p, d)) = f(d^{(2)}) = f(f^{[2]}(p)) = f^{[3]}(p) \tag{4.5}$$

$$\ldots \tag{4.6}$$

$$d^{(n)} = f(\phi_L^{(n-1)}(p, d)) = f(d^{(n-1)}) = f(f^{[n-1]}(p)) = f^{[n]}(p) \tag{4.7}$$

Because of the definition of SSA, after the loop entry function ϕ_L in the final iteration of loop L, there shall be no other assignment to d before the exit of the loop. Hence, $d_{exit(L)} = d^{(n)} = f^{[n]}(p)$. □

Corollaries C6 to C9 are variants of C5 with function f instantiated in several forms. They can be proved in a way similar to C5.

4.4 Examples

We now use several examples to show how ϕ-calculus helps symbolic differentiation. We start with two simple ones and end with a more complicated case with nested loops, breaks, if-else, and arrays.

(I) If-Else Example. We first look at the example in Figure 3. The ϕ functions resolve the difficulty for getting a closed form for the code. With the code in SSA, the derivation of the closed form for the code can simply ignore the conditional statements. What it needs to do is only to apply simple substitution of names with corresponding expressions based on the data flow. Figure 3(d) shows the closed form obtained from the SSA form in Figure 3(c). We add subscripts to the ϕ functions to help tell different ϕ functions apart.

According to corollary C2, we can apply derivation on x on the closed form and distribute the operation to the arguments of the ϕ functions. The result is shown in Figure 3(e). The underlines indicate two simplification opportunities. (i) The first underlined expression, $(p + qx) * 0.5 * q * (p + qx)^{-0.5}$, can be easily simplified by symbolic engines into $0.5q(p + qx)^{0.5}$. (ii) The second simplification opportunity appears after the *distributive formula* (F2) is applied such that the final term $(1+e^{p+qx})$ in the expression in Figure 3 (e) is distributed into the ϕ functions. That term cancels the denominator of the second underlined expression. Figure 3 (f) shows the result after symbolic simplication. It is worth noting that such optimization opportunities appear because of ϕ-calculus: They are both about interactions of the codelets across the boundaries of conditional branches, and hence would need the involved computations to be treated together. If each straight-line section of

the codelet is symbolically differentiated individually as shown by the segments of interest (SOIs) in Figure 3, those simplifications cannot get exposed. From Figure 3(f), code can then be generated with the ϕ functions materialized with conditional statements that check the corresponding branching decisions recorded during the primal computation.

(II) Simple Loop Example. Figure 4 (d) shows how ϕ-calculus helps produce a closed form for the loop in Figure 4 (a). With ϕ-calculus notations, the loop is symbolically represented in Figure 4 (c), on which, corollary C6 removes the loop notation and the loop entry ϕ, and produces the closed form of s_3 at the exit of the loop: $a + k \times x$ (k is the loop trip count). The application of *loop exit formula* F5 to the expression of s_4 removes the loop exit ϕ function, producing the simple expression $a + k \times x$. Symbolic differentiation can then be applied easily. For illustration purpose, this loop is made simple and the derivation of the closed form may resemble the recognition of induction variables in loop parallelizations [Tu and Padua 1995]. The next example gives a more thorough demonstration of the power of ϕ calculus.

(III) Complex Example (BGDHyperOpt). Figure 6 shows a more complex example. The code in Figure 6 (a) implements the use of batch gradient descent to determine the linear model on a dataset (x for inputs, y for response). The differentiation of interest is $d(err)/dr$, where r is the learning rate; this gradient can be used in finding out the best learning rate—a so-called *meta learning* problem that optimizes hyperparameters of a machine learning process.

The code consists of a `for` loop nested within a `while` loop; the `while` loop has a `break` in an `if-else` statement; there is another `for` loop following the `while` loop. To our best knowledge, no prior work can compute $d(err)/dr$ symbolically due to the control flow complexities.

Figure 6(b) shows the SSA form of the codelet. It includes nine ϕ functions; one of the loop exit ϕ functions ($\phi_{k'}$) has three arguments because of the `break` statement in the `while` loop.

Figure 6(c) shows the application of ϕ-calculus with the text boxes indicating the formulae or corollaries used at the important steps. We explain the process as follows.

Lines 1-5: Corollary C7 helps attain the closed-form expression of the value of $d3$ at the exit of the inner `for` loop; Formula F5 then resolves the ϕ'_i function and leads to the closed-form expression of $d5$.

Lines 7-10: This part tries to get the closed-form expression for the third argument ($w3$) of the $\phi_{k'}$ function on Line L6 in Figure 6(b). The part starts with a series of substitutions based on the results from Lines 1-5 in Figure 6(c), and then uses corollary C6 to get the closed-form expression of the value that $w3$ has at the normal (rather than via `break`) exit of the while loop.

Lines 12-16: This part tries to get the closed-form expression for the second argument ($w2$) of $\phi_{k'}$. It starts with substitutions with the results obtained already. It then uses the *distributive formula* F2 to transform the ϕ_k function on Line 14 in Figure 6(c) to a form matching the lhs form in corollary C6. The transformation is to factor out the terms in the second argument of ϕ_k such that the second arguement turns into pure $w2$ as shown on Line 15. Then corollary C6 can be applied, resolving the loop notation and also the ϕ_k function and producing the closed-form expression of $w2$ at the exit of the while loop as shown on Line 16 in Figure 6(c) ($\underline{\text{K}}$ stands for the trip count of the while loop).

Lines 17-18: simple substitutions of the three arguments in ϕ'_k.

Lines 20-24: This part tries to get the closed-form expression for $e4$. It first applies corollary C7 to the expression on Line 20 in Figure 6(c) to resolve ϕ_j and the loop notation, producing the closed-form expression for $e3_{exit}$ on Line 21. It then applies formula F5 to resolve ϕ'_j on Line 22, producing the closed-form expression on Line 24 for $e4$.

Line 26: A simple substitution with the results produced so far gives the closed-form expression of the final variable err. (For the sake of readability, we leave out the substitution of $w4$.) Symbolic differentiation can then be applied to err on r.

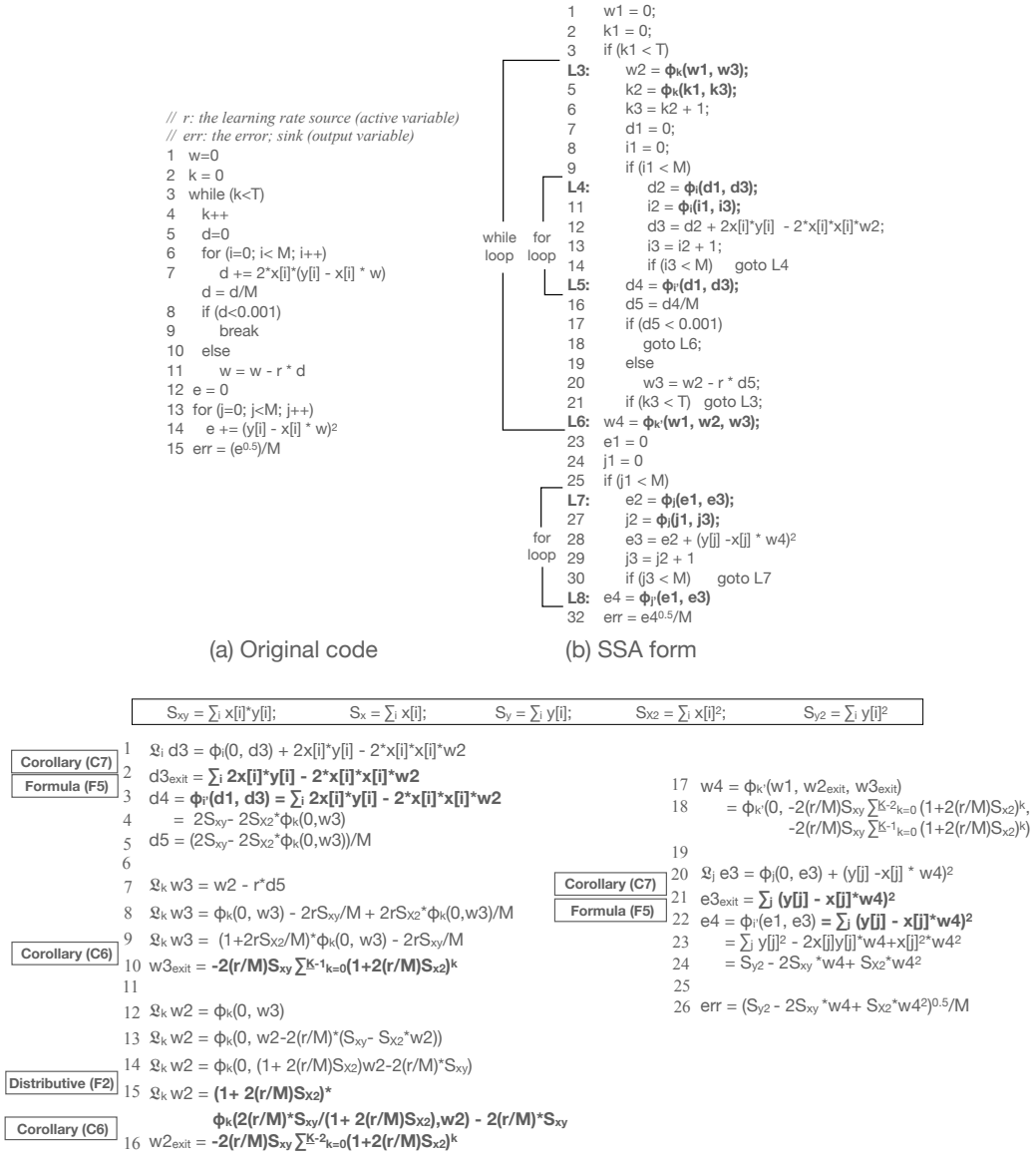


Fig. 6. Application of ϕ-calculus on hyperparameter optimizations, showing the treatment of loops and branches.

This part has demonstrated the applications of ϕ-calculus to several concrete examples. We will present the general use of ϕ-calculus in the overall algorithm of coarsening in the next section.

5 SOI IDENTIFICATION

With ϕ-calculus removing the barriers of control flow for coarsening, a *segment of interest (SOI)*—that is, the segment of code for symbolic differentiation—of a program can be much larger than a basic block. A larger SOI often offers more opportunities for optimizations, but it is not always better due to a tradeoff caused by two factors.

The first is "expression swell". As aforementioned, "expression swell" refers to the phenomenon that the derivative often has a much larger (in the worst case, quadratically larger) representation than the original function has [Cor 1988]. As a result, a very long expression can cause large memory usage and long running time of symbolic engines.

The second factor is the tension between coarsening and computation reuse. In coarsened differentiation, without a careful design, some computation reuse opportunities could get lost due to computation reordering caused by coarsening transformations, a phenomenon we call *reuse deprivation*.

Deprivation Example. The impact of reuse deprivation can be seen on $x_{t+2,0}$ and $x_{t+2,1}$ in the `CartPole` example in Figure 2. As Figure 2(c) shows, they are both computed from $x_{t+1,1}$. So potentially, if $d(x_{t+2,0}/d(a_t)$ has been computed, $d(x_{t+1,1})/d(a_t)$ could be known and could be reused in computing $d(x_{t+2,1}/d(a_t)$. Coarsening the computations from a_t and X_t (i.e., $[x_{t,0}, x_{t,1}, x_{t,2}, x_{t,3}]$) to $x_{t+2,0}$, however, deprives that reuse opportunity. The coarsening result has been shown in Equation 3.1, in which the holders of intermediate results, such as $x_{t+1,1}$, disappear. The differentiation over a_t is shown in Equation 3.2, which has no $d(x_{t+1,1})/d(a_t)$ or the derivatives of any other intermediate variables over a_t. As a result, when we need to compute the derivative of $x_{t+2,1}$ over a_t, we cannot reuse those intermediate derivatives.

The example illustrates a tension between reuse and coarsening. The larger is the coarsening granularity, the more opportunities there are for the enabled symbolic differentiation and optimization to take effect, but at the same time, it could incur deprivation of computation reuse opportunities.

What adds subtly to the relation is that reuse deprivation does not always lead to fewer reuse opportunities. Some reuse deprivations transform the reuse opportunities to another form. For instance, suppose that we have a way to get a closed form for the entire computation from X_t to L. All reuses, including $x_{t+1,1}$ for $x_{t+2,0}$ and $x_{t+2,1}$, turn into explicit sub-expressions in the closed form of L. There can hence be reuse opportunities exposed between them in the differentiation of the closed-form expression. The condition for such a transformation of reuse to occur is that the closed-form expression must subsume both parties that contain the reusable computations.

A single solution, reuse-aware SOI identification, addresses both factors. *Reuse-aware SOI identification* refers to an algorithm that solves the following optimization problem:

Definition 2. **Optimal SOI segmentation problem:** *Let G be a series of computations, l be the upper limit of the allowed sizes of an SOI, P be the set of valid partitions of G, that is, for any partition S in P, no element in S is larger than l. The problem is to find the optimal partition $S^* \in P$ such that the total running time is minimized, that is,*

$$\forall Q \in P, ad(S^*) + \sum_{s \in S^*} compute(dif(s)) \leq ad(Q) + \sum_{q \in Q} compute(dif(q)),$$

where $compute(dif(x))$ is the amount of computation involved in running the symbolically differentiated code segment for code x, and $ad(X)$ is the cost of the remaining AD differentiation of X after symbolic differentiation.

The upper bound of SOI size l in the problem description ensures that the symbolic engine works well even in the presence of the "expression swell" effects. The problem description indicates three factors relevant to SOI definitions.

1) The cost $ad(X)$. This cost is incurred at the boundaries of SOIs. Symbolic differentiation of an SOI computes only the derivatives of the active output variables of this SOI on the active input variables of this SOI. These derivatives have to be connected into a chain by AD to compute the derivatives of the ultimate active output variables on the ultimate active input variables. As a result, the more SOIs there are, the more AD overhead is there, and the larger is $ad(X)$. In the extreme case where each operation is an SOI, $ad(X)$ would equal to the cost taken by the default AD without coarsening. So this factor calls for larger SOIs.

2) Computation simplifications. The cost $\sum_{x \in X} compute(dif(x))$ is smaller if more computations are simplified. As two cancellable computations falling into two separate SOIs are not going to get cancelled by the symbolic engine, maximization of simplified computations also calls for larger SOIs.

3) Computation reuse. Maximizing computation reuse helps reduce the cost $\sum_{x \in X} compute(dif(x))$ as well. Unlike computation simplification, computation reuse exists both within and across SOIs. Exploiting reuse within an SOI happens in the default symbolic optimization and code optimization (e.g., common subexpression elimination (CSE) [Aho et al. 2006]). Reuse across SOIs is the natural result of the chain rules of differentiation, as shown by the potential reuse of $d(x_{t+1,1})/d(a_t)$ in computing $d(x_{t+2,0})/d(a_t)$ and $d(x_{t+2,1})/d(a_t)$ in Figure 2. In general, if y is an active output variable of both SOI_a and SOI_b and it is also an active input variable of SOI_c, then the derivative of the ultimate active output z on y (dz/dy) can be used in computing the derivatives of z on the active inputs of both SOI_a and SOI_b. Besides saving computations, reuses are also helpful for mitigating the "expression swell" problem as they split a long expression into shorter ones as noted in some previous work [Laue 2019; Wang et al. 2018]. Because expanding SOIs could lose inter-SOI reuses as the *deprivation example* has shown, this factor suggests that simply maximizing SOIs to the upper limit l cannot always give the best SOIs.

Finding the optimal SOI segmentation is difficult. For an SSA representation with N instructions, assuming every instruction can be put into an SOI, the number of possible partitions is between $l^{N/l}$ and l^N, where, l is the upper limit of the allowed size of an SOI. The reason is that the number of SOIs is between N/l and N, while the size of an SOI has l possibilities in the lower-bound case and up to l possibilities in the upper-bound case. Besides the exponential space, it would require detailed performance and overhead modeling, which is hard to be precise at static compile time.

It is however important for a viable solution to take all these factors into consideration. Figure 7(a) outlines our designed algorithm. For a given function f, for each of its active variable s that outlives f, the algorithm gets its def-use chain, which captures all the definitions in f that lead to the value of s. It then builds a *def-use region tree* out of all the definitions on the def-use chain. *Def-use region tree* is a data structure inspired by the classic *code region hierarchy* in compilers [Aho et al. 2006]. Traditionally, a code region is defined as a collection of nodes N and edges E such that (i) a header node h in N dominates all other nodes in the collection; (ii) if p is in N, then m must be in N if m reaches p without going through h; (iii) E includes all edges between nodes in N, except for those that enter h. In the code region hierarchy of a program, each code region is represented by a node in the hierarchy subsumed under the nodes that represent its enclosing code regions. *Def-use region tree* has two major differences from *code region hierarchy*: (i) only relevant variable definitions are considered; (ii) every loop-exit ϕ function is put as part of the region of the associated loop. The second property is for convenience in the derivation of symbolic expressions for loops. As an example, Figure 7(b) shows the def-use region tree of all the definitions on the def-use chain of *err*

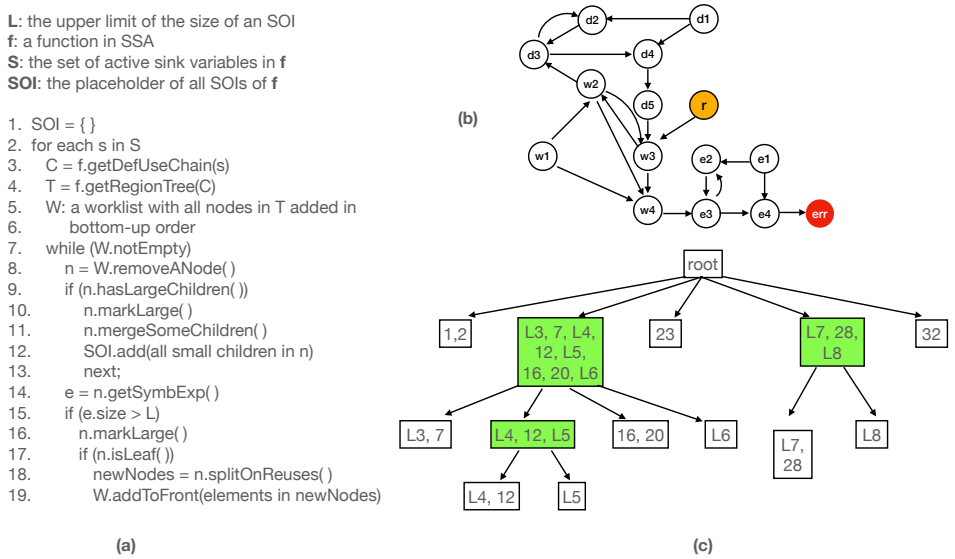


Fig. 7. (a) Algorithm of reuse-aware identification of SOIs for coarsening. (b) The def-use chain of *err* in the example shown in Figure 6(b) with r as the active input; inactive inputs (x, y, M) are omitted. (c) The *def-use region tree* for *err* in Figure 6(b); each element in a box is a line number in Figure 6(b); solid (green) boxes stand for loops.

in Figure 6(b); each element in the boxes is a line number in Figure 6(b). The three solid (green) boxes are the three loops. The three loop-exit ϕ functions (Lines L5, L6, L8 in Figure 6(b)) are put together with the three loops respectively.

The SOI identification algorithm then traverses the region tree in a bottom-up order, as outlined in Figure 7(a). For each node, if it has no large child node (i.e., exceeding the SOI size limit), its symbolic expression is derived through the ϕ-calculus. If the size of the derived expression exceeds the SOI size limit, that node is marked as a large node, and if it is a leaf node, it is split into two smaller nodes; the splitting point is chosen to be the variable that is contained in that node and has the largest number of references (and hence reuses) in f. The two new nodes created by the split are added to the front of `worklist`. If the current node has large children, there is no need to go up further in the *def-use region tree* as the upper nodes can only become even larger. In that case, the algorithm examines the immediate children of this node, merge consecutive small children nodes (up to the SOI size limit); after that, it puts each of the children nodes smaller than the limit as an SOI. The algorithm continues until *worklist* becomes empty. The size limit L can be empirically selected based on the machine and the symbolic engine.

The algorithm follows the principle of maximizing the size of SOIs within the SOI size limit while respecting reuses when it is necessary to split a def-use chain into multiple SOIs.

6 IMPLEMENTATION

We implement the coarsening optimization on an in-house AD tool named DiffKt. The tool was developed for Kotlin, a cross-platform, statically typed, general-purpose programming language with type inference. The tool itself is also written in Kotlin. We choose it as the basis mainly because of its availability and the statically typed nature of Kotlin which offers conveniences for static code analysis and transformations. But as a general optimization technique, coarsening can be

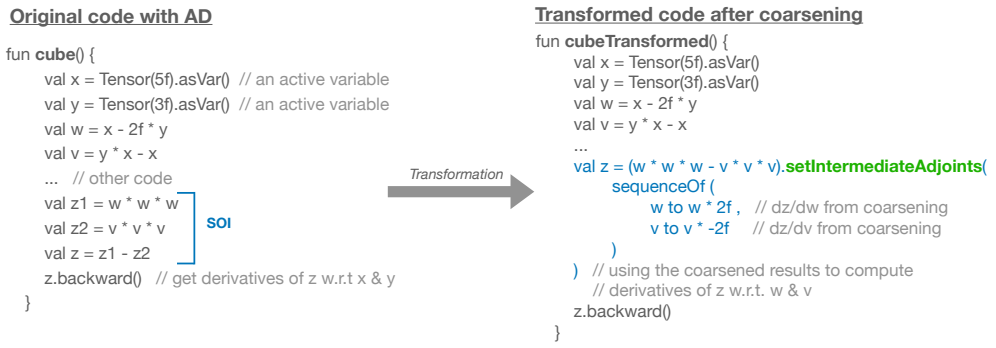


Fig. 8. An example showing how the coarsened results are integrated into the original program. The original primal code (left) is transformed to a form (right) such that the calculations of z in the SOI are put into one expression, and an adjoint is appended to that expression, which, at runtime, makes the AD take its content as the shortcuts for calculating the derivatives of z on the two active variables of the SOI, w and v. The results are used by the default AD in differentiating the rest of the code (i.e., from w and v to x and y.

potentially applied to many other AD tools; for some (e.g., Python AD tools), it may need to be done dynamically.

DiffKt was developed and optimized by 10+ engineers in industry in over a year. It supports both CPU and GPU, with high-performance native math/DNN libraries for Tensor computations. The tool is planned to open source in the near future. A systematic benchmarking of the tool over other AD tools is yet to be done, but preliminary measurements show that it outperforms PyTorch AD [Paszke et al. 2017] by over 10× on scalar-intensive cases (e.g., `HookeanSpring` in Section 7), and achieves comparable speeds on common deep learning models where pre-existing libraries are called for gradients calculations of the standard DNN layers under the hood of both of them. (As PyTorch AD and the Kotlin AD tool are in different programming languages, the comparison is only to give readers a sense about the industrial quality of the default tool.)

Similar to many other AD tools (e.g., PyTorch [Paszke et al. 2017], JAX [Bradbury et al. 2018]), DiffKt is a library-based implementation, enabling AD through operator overloading via a generic class `Tensor`. Implemented in 250K lines of code, it supports backward AD and includes a Tensor typing module as well.

As with other Automatic Differentiation (AD) tools (e.g., Zygote [Innes 2020]), DiffKt also allows the use of adjoints for custom differentiation. For a given expression e, if a custom differentiation of e is provided, the AD process will automatically invoke it rather than conduct the default operation-by-operation differentiation. The differentiations of the SOIs generated by coarsening are integrated into the original program as custom adjoints. An example is shown in Figure 8. The coarsening results form the content of the adjoint, which is associated with the coarsened primal expression through the call "setIntermediateAdjoints". If coarsening produces code for differentiating the entire primal code, the compiler simply replaces the calls of the corresponding `backward` function with the generated code; if the compiler in addition determines that the primal results are used in the program only for getting the derivatives, the compiler removes the invocations of the primal code.

Our implementation of coarsening on DiffKt is based on a Kotlin compiler. Our experiments focus on first-order backward AD, but it is worth noting that coarsening, as a way to offer shortcuts in AD, can in principle help higher-order differentiation and forward or mixed-direction AD as well. In addition to the ϕ-calculus and the SOI identification as presented earlier, our implementation also includes loop unrolling and the use of the primal computation results in the generated adjoint

functions when possible. As a side benefit of coarsening, our implementation of coarsening optimizes the primal in addition to the differentiation: After getting the closed form of the primal computation, it applies symbolic optimization to the primal and regenerates the code; an example is shown in the BGDHyperOpt benchmark in Section 7.2. In the case where only differentiation is needed and the entire primal can be symbolically differentiated, the optimizer removes the primal from the program when possible—an optimization elusive to existing AD. In the current implementation, for proof of concept, we use a symbolic engine extended from Sympy [sym [n.d.]], an open-source symbolic manipulation tool.

Like many other compiler-based optimizations that change the order of computations, coarsening could affect the numerical precision. A convenience offered by coarsening is that measures to avoid numerical unstableness can be seamlessly integrated into the code generation in coarsening. The code generator is equipped with the patterns for dealing with common numerically unstable expressions. Before it generates the code for a symbolic expression, it examines it to identify the numerical unstable expressions through pattern matching, and generates the code corresponding to their numerically stable forms. For expression $log(1 + e^{-x\beta})$, for example, as the code generator finds out that the expression matches one of the patterns in its unstable list, $log(1+e^{n})$, it generates the code to discern the value of the exponent, as illustrated as follows (MAXEXP is set to 40 in our implementation):

temp0 = -xβ
temp1 = (temp0 > MAXEXP)? MAXEXP : log(1+e^{temp0})

When the expression operates on tensors, the generated code uses masking functions (like `where` in PyTorch) for efficiency. A concrete example of numerically stable code generation in coarsening is the `HMC` benchmark detailed in Section 7.2.

7 EVALUATION

To evaluate the efficacy of the proposed techniques, we test coarsening on six applications in 18 total configurations on two different machines. *Backward AD is used.* The results show that the optimization improves the differentiation speed by 1.03-27×, and the whole application execution speed by 1.08-11×.

7.1 Methodology

Benchmarks. There are no common benchmark suites designed for evaluating AD. We collected six applications from several domains where AD is important, and implemented them in Kotlin with the Kotlin AD library. Table 1 lists the set of benchmarks used in the experiments. These benchmarks come from several domains, from physical simulation to statistical sampling, deep reinforcement learning, gaming, and meta learning. They also show a range of code complexities, with BGDHyperOpt featuring control flow complexities as Figure 6 has shown, Brachist. featuring a case where primal computation could be potentially removed, CartPole featuring a combination of matrix-based Deep Neural Network and scalar-based environment simulations, HMC featuring potential value overflow incurred by exponential computations, HookeanSpring featuring a sequence of regular vector operations, and QWOP featuring a long function with many small loops and if-else statements. For each benchmark, we include three configurations as listed in the right column of Table 1, which will be explained later in the discussion of the results of each application. We repeat the performance measurements multiple times and report both the mean and standard deviation of the timing results. Kotlin runs on Java virtual machines. For both the baseline and the optimized versions, the JRE went through a warm-up phase before timing starts to get the stable performance.

Table 1. Benchmarks and Configurations

Name	Domain	Description	Configs	
BGDHyperOpt	Meta-Learning	Optimizing the learning rate of batch gradient descent based linear regression	1 2 3	200 data records 1000 data records 2000 data records
Brachist.	Math. Physics	Brachistochrone curve calculation	1 2 3	200 data points 400 data points 1000 data points
CartPole	Deep Reinforcement Learning	Training a CartPole system	1 2 3	an update every 6 steps an update every 8 steps an update every 10 steps
HMC	Statistic Sampling for Prob. Prog.	Hamiltonian Monte Carlo Sampling for logistic regression	1 2 3	100 one-dim records 1000 two-dim records 800 three-dim records
HookeanSpring	Physical Simulation	Simulating the dynamics of a Hookean Springs system	1 2 3	10 vertices 20 vertices 40 vertices
QWOP	Gaming	An avatar learns walking via motion optimization	1 2 3	light-weight figure medium-weight figure heavy-weight figure

Table 2. Machines

Machine	Configuration
devServer	Intel(R) Xeon(R) Gold 6138 40-core CPU 2.00GHz, 250GB, CentOS Stream 8, Kotlin 1.4.20-M1, Java HotSpot 64-Bit Server VM, Java 1.8.0_192
Macbook	MacBook Pro, 2.4GHz 8-core Intel Core i9, 32GB 2667MHz DDR4, MacOS Catalina (v. 10.15.7), Kotlin 1.4.20-M1, Graalvm 20.3.0, Java 11.0.9

Machines. Because in practical scenarios, those AD-based applications may run on both servers and personal computers/laptops, we have measured the performance of the applications on both kinds of machines. Table 2 provides the machine details.

7.2 Results

In this part, we first present an overview of the performance, and then provide detailed discussions on each benchmark.

Table 3 reports the overall performance, where "baseline" represents executions of the default Kotlin AD tool and "opt" represents executions after coarsening is applied. The "Differentiation Time" column reports the time taken by differentiation in one iteration of each benchmark, while the "Overall Time" column reports the overall time of an iteration. We make two observations.

(1) Coarsening brings 1.03–27× speedups to the differentiation of the benchmarks, and 1.08-11× speedups to the overall execution. In most cases, the overall speedups are smaller than the differentiation speedups as there are some parts of the computation in the programs outside the part of the code targeted by the coarsening optimization (i.e., the part involved in differentiation). Exceptions are `Brachist.` and `HookeanSpring`; it is because in those two original programs, the only purpose of the primal computations are to let the AD to compute the gradients. Because coarsening generates code that can directly computes the gradients, the optimization removes the primal computations completely; the overall time is hence shortened even more than the time savings on the differentiation. (A side observation is that in all cases, the laptop runs faster than the server, probably due to its faster CPUs and the use of a more recent version of Java Runtime.)

Table 3. Experimental Results: Time per iteration and Speedups

Machine	Benchmark	Config	Differentiation Time(ms)			Overall Time(ms)		
			baseline	opt	speedup	baseline	opt	speedup
devServer	BGDHyperOpt	1	5.44± 6%	0.20± 1%	27.44X	10.44± 5%	1.22± 2%	8.52X
		2	5.22± 8%	0.20± 2%	26.18X	10.12± 7%	1.21± 2%	8.37X
		3	5.37± 6%	0.20± 2%	26.82X	10.36± 6%	1.21± 2%	8.56X
	Branchist.	1	0.04± 3%	0.04± 7%	1.10X	0.09± 3%	0.04± 7%	2.51X
		2	0.19± 36%	0.15± 1%	1.22X	0.32± 34%	0.15± 1%	2.08X
		3	0.55± 2%	0.39± 4%	1.41X	0.69± 2%	0.39± 4%	1.79X
	CartPole	1	52.15± 4%	46.86± 1%	1.11X	55.06± 4%	49.38± 1%	1.12X
		2	15.69± 1%	14.51± 1%	1.08X	16.99± 1%	15.65± 1%	1.09X
		3	15.77± 2%	14.06± 2%	1.12X	17.07± 2%	16.24± 2%	1.05X
	HMC	1	2.89± 5%	0.67± 2%	4.29X	3.26± 5%	0.94± 3%	3.46X
		2	4.55± 1%	1.59± 4%	2.86X	5.17± 1%	2.11± 4%	2.45X
		3	5.58± 7%	2.19± 5%	2.54X	6.35± 7%	2.79± 5%	2.27X
	HookeanSpring	1	0.10± 4%	0.01± 2%	6.62X	0.16± 5%	0.01± 2%	11.02X
		2	0.11± 8%	0.03± 2%	4.09X	0.18± 8%	0.03± 2%	6.72X
		3	0.23± 12%	0.05± 13%	4.52X	0.38± 10%	0.05± 13%	7.53X
	QWOP	1	1.97± 6%	1.46± 4%	1.35X	3.89± 4%	2.76± 3%	1.41X
		2	25.71± 5%	17.81± 6%	1.44X	44.50± 3%	29.61± 4%	1.50X
		3	32.17± 5%	21.20± 6%	1.52X	56.86± 3%	36.37± 4%	1.56X
macBook	BGDHyperOpt	1	3.59± 4%	0.14± 2%	25.15X	7.23± 4%	0.86± 2%	8.41X
		2	3.59± 3%	0.15± 6%	23.43X	7.31± 3%	0.91± 4%	8.06X
		3	3.66± 4%	0.15± 3%	24.23X	7.41± 3%	0.91± 4%	8.18X
	Branchist.	1	0.04± 18%	0.04± 12%	1.03X	0.10± 14%	0.04± 12%	2.38X
		2	0.13± 4%	0.12± 2%	1.08X	0.23± 4%	0.12± 2%	1.91X
		3	0.44± 2%	0.31± 5%	1.42X	0.55± 2%	0.31± 5%	1.80X
	CartPole	1	29.46± 3%	26.42± 2%	1.12X	31.16± 3%	27.92± 2%	1.12X
		2	9.43± 2%	8.76± 0%	1.08X	11.89± 2%	11.03± 1%	1.08X
		3	9.06± 0%	8.35± 1%	1.09X	12.14± 1%	11.25± 0%	1.08X
	HMC	1	2.60± 1%	0.59± 2%	4.42X	2.90± 1%	0.82± 2%	3.56X
		2	3.88± 1%	1.24± 1%	3.13X	4.37± 1%	1.64± 1%	2.67X
		3	4.30± 1%	1.61± 1%	2.67X	4.85± 1%	2.06± 0%	2.35X
	HookeanSpring	1	0.08± 8%	0.01± 7%	5.72X	0.14± 8%	0.01± 7%	9.84X
		2	0.09± 11%	0.02± 2%	4.17X	0.15± 8%	0.02± 2%	7.13X
		3	0.10± 2%	0.04± 4%	2.46X	0.17± 2%	0.04± 4%	4.09X
	QWOP	1	1.57± 3%	1.16± 2%	1.35X	3.60± 4%	2.45± 2%	1.47X
		2	20.53± 2%	14.70± 2%	1.40X	40.43± 2%	26.46± 2%	1.53X
		3	24.69± 2%	16.81± 1%	1.47X	50.01± 2%	31.89± 1%	1.57X

(2) Coarsening is consistently beneficial; it saves the execution time across benchmarks, configurations, and machines. The main reasons for the time savings are three: (i) the savings of the operator overloading overhead of AD, which comes from object boxing and memory allocations; (ii) the simplifications of the computations thanks to the large-scoped symbolic differentiation and optimizations; (iii) the removal of unnecessary primal computations. We next elaborate these benefits through in-depth examinations of each of the benchmarks.

BGDHyperOpt. BGDHyperOpt is a meta-learning program and has been introduced in Example III in Section 4.4 and Figure 6. Meta-learning entails inspecting and optimizing a machine learning process, which has recently drawing lots of interest. This program tries to optimize learning rates through gradient descent. The three configurations correspond to three different sets of inputs.

The SOI in this program is the entire function that computes the learning error as shown in Figure 6(a). There are nine ϕ functions in the SOI. Despite the control complexities, coarsening is able to get the closed-form expression for the entire gradient computation and apply symbolic

differentiation on it, giving more than 23× differentiation speedups in all cases. The primal cannot be removed because the generated differentiation code must use the trip-counts of the `while` loop in the primal. As a result, the overall speedup is about 8×. Meanwhile, coarsening saves over 70% of memory allocations because many of the data allocations in the default AD are for holding intermediate data objects which are no longer needed after coarsening.

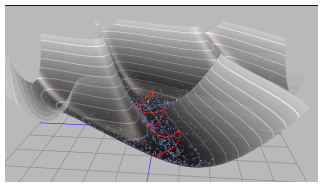
We did an ablation study to examine the benefits from the ϕ-calculus. In the study, we apply coarsening without ϕ-calculus; symbolic differentiation is hence applied to only the inner-most loop (which is written as a single Tensor statement) and the code after the `while` loop. The differentiation speedups drop from 23-27× to 8-9×:

	devServer			macBook		
Config	1	2	3	1	2	3
Speedup(X) w/o ϕ-calculus	8.46	7.80	8.08	9.24	9.34	9.39
Speedup(X) w/ ϕ-calculus	27.44	26.18	26.82	25.15	23.43	24.23

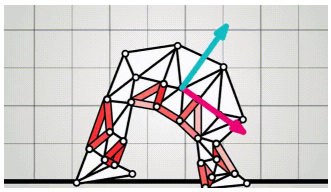
It can be seen that the larger scope of optimizations enabled by ϕ-calculus boosts the speedups by a factor of three. Its effects are multi-fold: (i) It allows a complete removal of the boxing overhead of the Tensor data structure from the differentiation process, whereas without ϕ-calculus, as only part of the differentiation is symbolically done, Tensors have to be used so that the operator overloading can still work, which is what the remaining part of the AD depends on. (ii) It exposes large-scoped loop-invariant calculations, for both the primal and the differentiation. Symbolic transformation and analysis of Lines 6-7 in Figure 6(a) can show that the loop involves the calculations of $\sum_i x[i] * y[i]$ and $\sum_i x[i] * x[i]$; when the analysis scope spans across the entire `while` loop via ϕ-calculus, the optimization can easily recognize that the two summations repeat in every iteration of the `while` loop and can be hoisted out of the `while` loop. Similar phenomena are in the differentiation. Neither the default optimizers in the Java Runtime underlying Kotlin or the coarsening without ϕ-calculus can recognize and take advantage of that. (iii) It saves the remaining AD overhead that the version by coarsening without ϕ-calculus has to suffer.

Brachist. This program calculates the Brachistochrone curve (i.e., curve of fastest descent), which is the one lying on the plane between a point A and a lower point B (called anchor points), where B is not directly below A, on which a bead slides frictionlessly under the influence of a uniform gravitational field to a given end point in the shortest time. In each iteration, the program computes the time taken by the bead to slide down the slope by summing the time it takes for each section of the current curve, and then gets the gradient of every section over the total time. The three configurations correspond to the number of sections that the target curve is regarded to be composed of. This is a relatively easy case, but it demonstrates an important scenario where coarsening can remove the entire primal computation. The entire primal code to compute the time taken to slide down the slope is identified as the SOI. With coarsening, the AD tool can symbolically differentiate the entire computation. Because the program only needs the gradients to update the curve in each iteration, it can now forego the computation of the total time as the gradients can be directly computed. Therefore the coarsening optimization removes the entire primal computation, making the program's overall speedups even more than the speedups on the gradients calculations. As Table 3 shows, the speedups on the differentiation part is modest (due to the simplicity and regularity of the code), but the end-to-end executions get more significant speedups (e.g., 1.8-2.38× versus 1.03-1.42× on macBook). The overall speedups are more pronounced on the smaller inputs because the primal computation weights more in those runs.

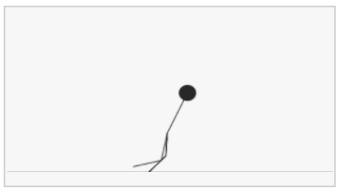
CartPole. CartPole is a deep reinforcement learning program as already introduced in Section 3. The three configurations corresponding to the number of exploration steps observed before learner updates the model parameters. It shows the least speedups among all the benchmarks, not because



(a) HMC sampling in a space ([hmc [n.d.]])



(b) HookeanSpring ([Rojas et al. 2019])



(c) QWOP

Fig. 9. Illustrations of several benchmarks.

coarsening is not effective, but because the small portion of the optimized code weighs in the overall program. Recall in Figure 2, the primal computation of CartPole contains two parts, the Neural Networks(NN) and the environment update. As CartPole uses a simple simulation environment, the environment update part weighs only about 15% of the primal time, with the rest dominated by the NN. As the NN has a standard structure and the default gradient calculation is through a manually written highly polished vendor library rather than the AD, the SOI is the second part, which updates the environment. There are five ϕ functions in the SOI. The speedups on the differentiation of the environment update part are actually significant:

	devServer			macBook		
Config	1	2	3	1	2	3
Speedup(X) on differentiating the environment update part	2.68	2.63	3.19	3.01	2.73	3.21

In cases where reinforcement learning is applied to more complex environments, the speedups on the end-to-end execution by coarsening are expected to be more substantial.

HMC. HMC stands for Hamiltonian Monte Carlo. It is one of the main algorithms in Probabilistic Programming or Statistics for finding out posterior distributions of random variables through a carefully designed Monte Carlo sampling process [nea 2011], as illustrated in Figure 9(a).

We first gives a conceptual view of HMC. The core sampling function in HMC takes in two functions as part of the arguments: function "U" and function "grad_U". The former does the primal computation and the latter computes the derivative of "U". HMC calls them many times on different values, more often on "grad_U" than on "U". In this benchmark, HMC is used for logistic regression. Logistic regression is a classic method for classification. HMC is used to estimate the posterior distribution of the parameters in the logistic regression model. Its "U" function is as follows:

$$U(\beta) = \beta^T X^T (y - 1_n) - 1_n^T [log(1 + e^{-X\beta})] - \frac{\beta^T \beta}{2\sigma_\beta^2}$$

where, β are the parameters in logistic regression, X and y are the input and response data (training data), σ_β^2 is a hyperparameter (1000). All are vectors except that X is a matrix. The three configurations correspond to three different training data sets.

A feature of HMC is that when it needs the gradient on some values, it often cares about the gradients but not the actual primal value. It does call the primal function at some places, but on values different from those needed for the calculation of the gradients.

But in the default implementation on AD, anytime gradient is needed, the primal is called because of the inherent requirement for AD to work as we have described in Section 2. Coarsening takes the entire primal function "U" as the SOI, and there are one two-level nested loop and another two single-level loops in the code with one of them containing an if-else statement. Coarsening is able to symbolically differentiate it and generate the entire "grad_U" function. It hence can save many primal computations. Overall, the speedups are 2.3-3.6×.

One special note on HMC is that the exponential term $e^{-X\beta}$ can easily result in value overflow; special treatments must be given to large exponents (e.g., using $-X\beta$ to approximate $log(1 + e^{-X\beta}$ if $-X\beta$ exceeds 80). The default AD-based tool uses a masking function to deal with that (similar to "where" in PyTorch). Without the masking scheme, an alternative would be to wrap each element in $-X\beta$ in a Tensor to discern their following uses; which creates huge runtime overhead, making the program run about 25× slower. Without the dependence on Tensor or operator overloading, coarsening is not subject to the problem; when it generates the code, it directly generates the appropriate code for the cases where the exponential value is large.

HookeanSpring. HookeanSpring is a physical simulation program. It simulates mass-spring systems as illustrated in Figure 9 [Rojas et al. 2019]. It demonstrates the transitions of physics-based states as energy minimization procedures. The program keeps optimizing the vertex positions of a spring system to find some configuration that minimizes the total elastic energy. Every spring has some preferred rest length and they naturally tend to recover their rest shapes over time. Each optimization step uses the gradients of the spring vertex locations regarding to the system energy. The three configurations correspond to three sizes of the spring system in terms of the number of spring vertices. Coarsening is able to take the entire energy calculation of the Spring system as the SOI and symbolically differentiate it. As a result, the primal computation which computes the system energy can be completely removed. The speedups are 4–11×. The program even runs faster than the original primal computation alone; the following table shows the times taken in one iteration of the simulation and the relative speedups (median values of repeated measurements are used):

	devServer			macBook		
Config	1	2	3	1	2	3
Original primal only(μs)	49.19	51.92	114.20	43.37	47.01	52.58
Exec. after coarsening(μs)	14.49	26.76	51.49	14.46	21.61	41.17
Speedups(×)	3.39	1.94	2.22	3.00	2.18	1.28

This result is significant because there has been a common perception that a program would take a lot more time to run if automatic differentiation is added into it. For example, a previous work considers 2.4-4× slowdown after adding automatic differentiation as already close to the optimal [Hog 2014]. This coarsening result shows that with coarsening, after adding automatic differentiation, a program can even run several times faster.

QWOP. QWOP is an avatar motion optimization program. It trains a virtual stick figure to run as far as possible by providing a schedule for how much each muscle should be extended, as illustrated in Figure 9(c). The three configurations correspond to three configurations of the mass of the body parts of the stick figure. The special aspect about this program is that its core part is a 225-line function with 13 loops and many if-else statements. After loop unrolling, the function becomes 1117-line long. Coarsening can successfully deal with the function, getting two SOIs, and achieving 1.17-1.51× overall speedups.

7.3 Potential on Other AD Tools

Coarsening is a general optimization for AD. To check its potential benefits to AD tools beyond DiffKt, we examined the performance of the benchmark `BGDHyperOpt` on three other AD tools: JAX [Bradbury et al. 2018] for Python, Zygote [Innes 2020] for Julia, and Adept [Hog 2014] for C++.

For each of the three AD tools, we have two versions of the benchmark BGDHyperOpt: (i) the baseline version which uses the default AD offered by the tool; (ii) the coarsened version optimized by coarsening. The latter was written based on the results from our symbolic engine. Table 4 reports the speedups of the coarsened versions over the baseline counterparts. Please note that the JAX baseline version already uses its JIT (the JIT gives 1.3-1.47X speedups over the default version that uses no JIT). The execution times were measured on the Macbook after warm-ups. The speedups are 66×-335×, even greater than on DiffKt, indicating the potential of coarsening as a general AD optimization technique.

Table 4. Speedups of the coarsened version over the baseline version on `BGDHhyperOpt`

AD Tool (Language)	Input Size	
	1000	2000
JAX (Python) [Bradbury et al. 2018]	87.4X	335.1X
Zygote (Julia) [Innes 2020]	150X	90.8X
Adept (C++) [Hog 2014]	66X	96.2X

8 DISCUSSIONS

The study has demonstrated the significant benefits of coarsening on the Kotlin AD tool on first-order backward AD, the most popular kind of AD. It is easy to see that the technique can help other types of AD implementations (e.g., forward or mixed directions and higher-order differentiation) as the outcome of coarsening can always be used as a shortcut on the AD chains.

Our exploration of coarsening is at the static compilation time. The technique is potentially applicable at runtime as well, which could be especially meaningful for languages (e.g., Python) that are difficult for static time analysis and transformations. In that case, runtime profiling could be useful, and extra care (e.g., hot paths based selective optimizations) may be necessary to minimize the time overhead of symbolic manipulations.

The current coarsening optimization applies to both regular and irregular loops as mentioned before. But there is code with unstructured control flows where it is even unclear what the loop is (e.g., code formed by go-to statements in certain languages). In those cases, the SOIs could be set to the sections within the branches that form the unstructured control flows.

With coarsening, the compilation time does not have noticeable changes except for the time taken by the symbolic engine in doing symbolic differentiation and other symbolic manipulations. As mentioned, as a proof of concept, the current implementation uses an extended Sympy for that. Written in Python, Sympy is not the most efficient symbolic engine. For the benchmarks in the experiment, it takes up to a minute to do symbolic differentiation.

Coarsening is based on the SSA form of a program. When a program has assignments to arrays, array SSA [Knobe and Sarkar 1998] would be necessary to discern the different ranges of data elements in an array when they are treated differently in the program. The representation and corresponding analysis are more complicated than on the basic SSA form. We found that for AD programs written in Tensor-based AD libraries, in most cases, array SSA is not necessary. It is because in those programs, if there are large arrays, operations on them are typically written as Tensor operations (e.g., $C = A + B$ for `Loop: c[i] = a[i]*b[i]`) without explicit references to

individual array elements; for such representations, the standard SSA still applies. Even if sometimes a part of the array elements are treated differently from others, Tensor operations still suffice via Tensor masking operations (e.g., the HMC case). In the cases where individual array elements are used and updated differently, those arrays are usually short and are used in small loops; loop unrolling and scalar conversion can easily turn the code into a form amenable for the standard SSA. Nonetheless, integration of array SSA with coarsening could still be useful especially for AD tools without Tensor-like abstractions.

9 RELATED WORK

There is a large body of work on efficient AD. Optimizations range from checkpointing [Dauvergne and Hascoet 2006] to edge/vertex eliminations on computation graphs [Dixon 1991], combination of forward and backward differentiation [Baydin et al. 2018], loop transformations [Shaikhha et al. 2019], and so on. A recent work [Sherman et al. 2021] proposes a differentiable programming language to deliver a semantics for higher-order functions, higher-order derivatives, and Lipschitz but non-differentiable functions. Some of the relevant studies have been mentioned in earlier sections, and more on AD for machine learning can be seen in recent surveys [Baydin et al. 2018; Margossian 2019; van Merriënboer et al. 2018]. Coarsening can be regarded as an optimization complementary to those existing AD optimizations: They can be used together, with coarsening offering shortcuts and the other optimizations improving the remaining AD operations.

There are many tools capable of doing symbolic differentiation. Examples include the Calculus module in Julia [cal [n.d.]], SageMath [sag [n.d.]], KotlinGrad [Considine et al. 2019], and Acumen which maps from analytical models to simulation codes via symbolic differentiation [Zhu et al. 2010]. None of them have addressed the complexities from control flow on symbolic differentiation, or the systematic integration of symbolic differentiation with AD. The existing symbolic engines can differentiate only expressions not programs. For cases with simple control flows where the problem of interest involves only several conditional cases, the user could enumerate those cases and use the existing symbolic engines to differentiate them each. That practice does not apply to code with loops or many branches. The ϕ-calculus in this work offers a solution to the complexity.

Several recent studies have challenged the common criticisms of "expression swell" of symbolic differentiation [Laue 2019; Wang et al. 2018]. Even though the arguments may differ in form, the main points are similar: If placeholders are used to store intermediate differentiation results for reuses, the problem can be largely alleviated. The design of our reuse-aware SOI identification in coarsening is based on a similar insight, but provides a systematic way to deal with the tradeoff between reuse and the granularity of symbolic differentiation.

Expression templates have been used in both forward and backward AD implementations to reduce runtime space and time overhead[Sag 2017; Aubert et al. 2001; Phipps and Pawlowski 2012]. In Adept, for instance, during the primal computations, the algorithm records backward operations onto a stack. Its use of expression templates in C++ helps avoid invocations of virtual function calls at runtime, and hence reduces the amount of objects needed to allocate to hold intermediate results. Differentiation happens however still at each individual operation. There is no symbolic differentiation or symbolic simplifications or optimizations in a large scope. Moreover, as with all other operator overloading based AD, these solutions also require primal computations to be executed before gradients can be computed. As a result, the highly optimized implementations are still 2.4-4X slower than the original algorithm (without gradients calculations) [Hog 2014]. Coarsening optimization, in comparison, harnesses large-scope optimization opportunities, and can sometimes forego the primal computations completely, yielding even a higher speed than the original algorithm has as Section 7 has shown.

Since it was first proposed in late 1980s [Cytron et al. 1989], SSA has been widely adopted in program representations in compilers. Gated SSA (GSA) proposes to explicitly specify the conditions in the ϕ functions, and introduces notations to distinguish loop entry, loop exit, and normal ϕ-functions [Ottenstein et al. 1990], which share some similarities with part of the ϕ-notations in this work. The loop notations in ϕ-calculus is inspired by the notations in Glore [Ding and Shen 2017], a work that detects large-scoped loop invariants. In 1990s and early 2000s, there were a number of papers on recognizing and substituting inductive variables in loops based on SSA through symbolic analysis [Tu and Padua 1995]. An example is the symbolic analysis based on Chains of Recurrences [van Engelen 2001; van Engelen et al. 2004]. The purpose is to convert the array subscripts in a loop into a form ready for parallelization-oriented dependence analysis. For symbolic differentiation, what is needed is not only symbolic treatment to inductive variables but derivations of the closed-form expressions for all the computations that are related with the active variables, hence the need for the ϕ-calculus. As a type of standard IR, SSA is also the IR leveraged in recent AD compilers, such as the Zygote for Julia [Innes 2018, 2020], which is a pure AD tool without systematic integration of symbolic differentiation.

Besides symbolic and algorithmic differentiation, there is another approach called numerical differentiation, which uses finite difference approximations. But because it is inaccurate and scales poorly for gradients, it is rarely used for machine learning where gradients with respect to millions of parameters are common.

10 CONCLUSION

This paper has presented *coarsening*, a novel optimization that expands the scope of symbolic differentiation and systematically integrates symbolic differentiation with AD. It builds on two key innovations: the *ϕ-calculus* and the *reuse-aware SOI identification*. The *ϕ-calculus* offers the first mechanism that allows symbolic differentiation to apply on code with complicated control flow, while the *reuse-aware SOI identification* provides an algorithm to deal with the tension between computation reuse and coarsening. Experiments on several AD tools and various settings demonstrate that coarsening is an effective optimization for AD. It can remove the overloading overhead in AD and at the same time harness the benefits of symbolic optimizations and differentiation, yielding several times to two orders of magnitude speedups.

REFERENCES

[n.d.]. Calculus package for Julia. Available at https://github.com/JuliaMath/Calculus.jl.
[n.d.]. HMC Explained. Available at https://arogozhnikov.github.io/2016/12/19/markov_chain_monte_carlo.html.
[n.d.]. SageMath. Available at https://www.sagemath.org/.
[n.d.]. *Sympy software*. https://www.sympy.org/en/index.html.
1988. Fast reverse-mode automatic differentiation using expression templates in C++. *Perspectives in Computing* 19 (1988). Source of expression swell.
2011. Handbook of Markov Chain Monte Carlo. (May 2011). https://doi.org/10.1201/b10905
2014. Fast reverse-mode automatic differentiation using expression templates in C++. *Trans. Math. Software* 40, 26 (2014). Issue 4. ADEPT AD tool in C++.
2017. High-Performance Derivative Computations using CoDiPack. *Trans. Math. Software* 45 (2017). Issue 4. CoDiPack.
A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison Wesley.
P. Aubert, N. Di Cesare, and O. Pironneau. 2001. Automatic differentiation in C++ using expression templates ´ and application to a flow control problem. *Comput. Vis. Sci.* 3 (2001), 197–208.
Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research* 18, 1 (2018).
James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. https://jax.readthedocs.io/.

Breandan Considine, Michalis Famelis, and Liam Paull. 2019. *Kotlin∇: A Shape-Safe eDSL for Differentiable Programming*. https://github.com/breandan/kotlingrad.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 25–35.

B. Dauvergne and L. Hascoet. 2006. The Data-Flow Equations of Checkpointing in Reverse Automatic Differentiation. *Lecture Notes in Computer Science* 3994 (2006).

Y. Ding and X. Shen. 2017. GLORE: Generalized Loop Redundancy Elimination upon LER-Notation. In *Proceedings of OOPSLA at The ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*.

L. C. Dixon. 1991. Use of automatic differentiation for calculating Hessians and Newton steps. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application* (1991), 114–125.

Michael Innes. 2018. Don't Unroll Adjoint: Differentiating SSA-Form Programs. *CoRR* abs/1810.07951 (2018). arXiv:1810.07951 http://arxiv.org/abs/1810.07951

Michael J Innes. 2020. Sense & Sensitivities: The Path to General-Purpose Algorithmic Differentiation. In *Proceedings of the 3rd MLSys Conference*. https://fluxml.ai/Zygote.jl/latest/.

Kathleen B Knobe and Vivek Sarkar. 1998. Array SSA form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.

Sören Laue. 2019. On the Equivalence of Forward Mode Automatic Differentiation and Symbolic Differentiation. *CoRR* abs/1904.02990 (2019). arXiv:1904.02990 http://arxiv.org/abs/1904.02990

Dougal Maclaurin. 2016. *Modeling, Inference and Optimization with Composable Differentiable Procedures*. Ph.D. Dissertation. Harvard University.

Charles C. Margossian. 2019. A review of automatic differentiation and its efficient implementation. *WIREs Data Mining and Knowledge Discovery* 9, 4 (Mar 2019). https://doi.org/10.1002/widm.1305

Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. 1990. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *ACM SIGPLAN 1990 conference on Programming language design and implementation*. 257–271.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *Proceedings of NIPS 2017 Workshop Autodiff*.

Eric Phipps and Roger Pawlowski. 2012. Efficient Expression Templates for Operator Overloading-BasedAutomatic Differentiation. In *Recent Advances in Algorithmic Differentiation*, Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 309–319.

Junior Rojas, Stelian Coros, and Ladislav Kavan. 2019. Deep reinforcement learning for 2D soft body locomotion. In *NeurIPS Workshop on Machine Learning for Creativity and Design 3.0*.

Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient Differentiable Programming in a Functional Array-Processing Language. *Proc. ACM Program. Lang.* 3, ICFP, Article 97 (July 2019), 30 pages. https://doi.org/10.1145/3341701

Benjamin Sherman, Jesse Michel, and Michael Carbin. 2021. Computable Semantics for Differentiable Programming with Higher-Order Functions and Datatypes. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.

Nazanin Tehrani, Nimar S. Arora, Yucen Lily Li, Kinjal Divesh Shah, David Noursi, Michael Tingley, Narjes Torabi, Sepehr Masouleh, Eric Lippert, and Erik Meijer. 2020. Bean Machine: A Declarative Probabilistic Programming Language For Efficient Programmable Inference. In *Proceedings of the 10th International Conference on Probabilistic Graphical Models, PMLR 138*.

Peng Tu and David Padua. 1995. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th International Conference on Supercomputing*. 414–423.

Robert A. van Engelen. 2001. A method for recognizing and substitutions of generalized inductive variables through Chains of recurrences (CRs). In *Proceedings of the International Conference on Compiler Constructions*.

Robert A. van Engelen, J. Birch, Y. Shou, B. Walsh, and Kyle A. Gallivan. 2004. A Unified Framework for Nonlinear Dependence Testing and Symbolic Analysis. In *Proceedings of the International Conference on Supercomputing*.

Bart van Merriënboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. 2018. Automatic differentiation in ML: Where we are and where we should be going. *CoRR* abs/1810.11530 (2018). arXiv:1810.11530 http://arxiv.org/abs/1810.11530

Fei Wang, Xilun Wu, Grégory M. Essertel, James M. Decker, and Tiark Rompf. 2018. Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator. *CoRR* abs/1803.10228 (2018). arXiv:1803.10228 http://arxiv.org/abs/1803.10228

Yun Zhu, Edwin Westbrook, Jun Inoue, Alexandre Chapoutot, Cherif Salama, Marisa Peralta, Travis Martin, Walid Taha, Robert Cartwright, Aaron Ames, and Raktim Bhattacharya. 2010. Mathematical equations as executable models of

mechanical systems. In *Proceedings of International Conference on Cyber-Physical Systems*.