Automatic Complexity Reduction with the Polyhedral Equational Model

Extended Abstract

Tomofumi Yuki, Sanjay Rajopadhye and Gautam Gupta

1 Introduction

The Bird-Meertens formalism [1, 2] is one of many well known techniques for systematic program derivation. The primary focus of such formalisms is to provide a framework for validating the transformations used. They usually are not intended to help decide what transformations to apply. The framework may allow concise validation/proof of each step in the transformation, but human intuition is still involved in the transformation process.

On the other hand, optimizing compilers focus on *automatically* transforming programs to improve performance, while preserving the original semantics. Algorithmic improvement in complexity is rarely achieved in the compiler. One rare case where it is achieved in the compiler was proposed by Gupta and Rajopadhye [3]. They developed a technique that analyzes program expressions that use reductions as high level operators. The analysis finds values that are reused across multiple instances of a reduction, and exploits the reuse to obtain an equivalent program with lower asymptotic complexity.

The polyhedral model, is a mathematical framework for analysis and transformation of compute- and data-intensive kernels in programs where the iteration space is modeled as unions of polyhedra. We use an equational language based on the polyhedral model. The original motivation for this language was to let the scientists specify the computation as equations, without worrying about the performance, and then let the compiler optimize and generate executable code. This closely resembles one of the motivations of program derivation: start simple and successively transform the program for efficiency. However, the precise representation of the shape and size of the computation as polyhedral objects enables powerful and automated analyses, including complexity reduction.

We illustrate how equational reasoning in the polyhedral model can also be used for automatic program derivation. Although the model is applicable to only a limited class of programs, it provides automatic derivation and provide optimality guarantees. Specifically, we extend the simplification algorithm proposed by Gupta and Rajopadhye to deal with nested reductions. The extended algorithm simplifies nested reduction with reuse across different operators using the distributive property of a semi-ring.

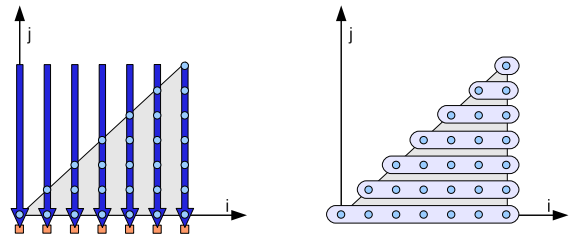
2 Simplifying Reductions

The key idea behind simplifying reductions is to detect hidden reuse among different instances of reductions. An example of such reuse can be seen in prefix sum computation. Prefix sum can be naively specified as $X[p] = \sum_{i=0}^{p} \mathtt{A}[i]$ where $\{p \mid 0 \leq p < N\}$, with $O(N^2)$ complexity, since there are N different instances of summations, each for a different prefix.

Of course, we all know that this computation is not really quadratic, but let us see how a compiler that can do polyhedral (a.k.a. geometric) analysis would discover this information automatically. It is clear that $x[p]$ accumulates the same set of values as $x[p-1]$ except for the value $\mathtt{A}[p]$. The simplifying reductions presented in [3] is a method for systematic detection and exploitation of such reuse for a single reduction.

Figure 1 visualizes the iteration space of prefix sum for N=8. The body of the reduction have a triangular domain $\{i, j | 0 \leq j \leq i < N\}$, and there are 7 independent reductions along the vertical axis. Because $\mathtt{A}[j]$ is accessed within a 2D domain, it can be observed that all points along the horizontal access that have the same j but different i all share the same value.

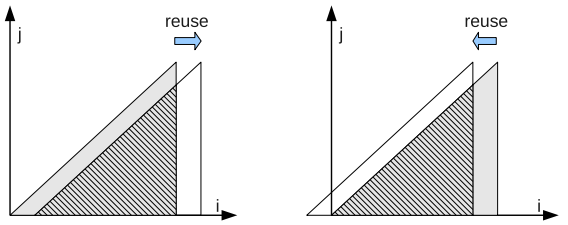
Simplifying Reduction is a program transformation that takes as input, constant vector r_E in the reuse space, and rewrites the equation so that an instance of reduction at z reuses the result of another instance at $z - r_E$. Unless the values used at different instances of reductions are identical, reusing the result of another instance by itself is not enough. The "residual" computation required can be computed from the polyhedral representation of the iteration



(a) Iteration Space and Reductions

(b) Reuse of A

Figure 1: Geometric view of the iteration space and reductions involved in prefix sum computation for N=8. The iteration space has a triangular domain where all integer points represent a computation. The reduction is along the vertical axis so that all points with the same i contribute to the same answer. Because A is indexed only with j, all points with the same j share the same value.



(a) When reuse vector $(1, 0)$ is used

(b) When reuse vector $(-1, 0)$ is used

Figure 2: Visualization of the reuse and simplification. $\mathcal{D}_{\mathtt{E}'}$ is the domain translated by the reuse vector. The intersection of the two domains (striped and filled) is the value being reused. In Figure (a), the diagonal strip of filled domain that does not have the stripe, $\mathcal{D}_{add} = \mathcal{D}_E - \mathcal{D}_{\mathtt{E}'}$ is the domain that needs to be computed in addition to the reuse. In Figure (b), the diagonal strip of unfilled domain, $\mathcal{D}_{sub} = \mathcal{D}_{\mathtt{E}'} - \mathcal{D}_E$ is the domain of values that needs to be undone from the reused value.

space. Figure 2 illustrates the reuse space and how the required computation in addition to the reuse is computed. Domains of residual computations are derived from the original domain $\mathcal{D}_E$ (filled domain) and its translation by the reuse vector $\mathcal{D}_{\mathtt{E}'}$ (unfilled domain). Domain with diagonal stripes is the intersection $\mathcal{D}_{int} = \mathcal{D}_E \bigcap \mathcal{D}_{\mathtt{E}'}$. $\mathcal{D}_{int}$ is where the result of two reductions r_E apart overlap and can be reused. Thus, the diagonal strip of filled domain that does not have the stripe, $\mathcal{D}_{add} = \mathcal{D}_E - \mathcal{D}_{\mathtt{E}'}$ is the domain that needs to be computed in addition to the reuse.

Depending on the shape of the domain and the direction of reuse being exploited, it is possible that a column in the translated domain has more points than the original one. In this case, some computation must be "undone" in order to exploit the reuse. In such cases, the reduction operator must have a corresponding inverse operator in order to undo parts of the computation. For example, if the vector[-1,0] was used instead in the above example, $P(x)$ is computed from $P(x+1)$ by *subtracting* $A[x+1]$. Such a domain, called *subtract domain*, can be computed as well, and it must be empty if the operator does not have an inverse.

This transformation can be used to systematically reduce the complexity of RNA secondary structure prediction algorithm [4].

3 Maximum Segment Sum

The Maximum Segment Sum (MSS) problem and its linear time algorithm became well known after its presentation as a Programming Pearl [5]. The problem is to find a consecutive sub-array, of a one-dimensional array that has the largest sum. The original problem was on two-dimensional arrays in the context of pattern recognition [5]. The problem is often used to show that a program derivation technique can successfully prove the equivalence of the cubic time algorithm and linear time algorithm. An extension to the Simplifying Reductions to simplify nested reductions with different reduction operators, in one transformation, enables the compiler to *automatically* deduce linear-time algorithm for MSS.

References

[1] Bird, R.: A calculus of functions for program derivation. (1990) 287–307

[2] Gibbons, J.: An introduction to the Bird-Meertens formalism. In: New Zealand Formal Program Development Colloquium Seminar, Hamilton. (1994)

[3] G., G., S., R.: Simplifying reductions. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '06 (2006) 30–41

[4] Yuki, T., Gupta, G., Pathan, T., Rajopadhye, S.: Systematic implementation of fast-i-loop in UNAfold using AlphaZ. Technical report, CS-12-102, Colorado State University (2012)

[5] Bentley, J.: Programming pearls. (1986)