AlphaZ: A System for Design Space Exploration in the Polyhedral Model*

Tomofumi Yuki[1], Gautam Gupta[2], DaeGon Kim[2], Tanveer Pathan[2], and Sanjay Rajopadhye[1]

[1] Colorado State University
[2] CORESPEQ Inc.

Abstract. The polyhedral model is now a well established and effective formalism for program optimization and parallelization. However, finding optimal transformations is a long-standing open problem. It is therefore important to develop tools that, rather than following predefined optimization criteria, allow practitioners to explore different choices through script-driven or user-guided transformations. More than practitioners, such flexibility is even more important for compiler researchers and auto-tuner developers. In addition, tools must also raise the level of abstraction by representing and manipulating reductions and scans explicitly. And third, the tools must also be able to explore transformation choices that consider memory (re)-allocation.

AlphaZ is a system that allows exploration of optimizing transformations in the polyhedral model that meets these goals. We illustrate its power through two examples of optimizations that existing parallelization tools cannot perform, but can be systematically applied using our system. One is time-tiling of a code from PolyBench that resembles the Alternating Direction Implicit (ADI) method, and the other is a transformation that brings down the algorithmic complexity of a kernel in UNAfold, a sequence alignment software, from $O(N^4)$ to $O(N^3)$.

1 Introduction

The recent emergence of many-core architectures has given a fillip to automatic parallelization, especially through "auto-tuning" and iterative compilation, of compute- and data-intensive kernels. The *polyhedral model* is a formalism for automatic parallelization of an important class of programs. This class includes *affine control loops* which are the important target for aggressive program optimizations and transformations. Many optimizations, including loop fusion, fission, tiling, and skewing, can be expressed as transformation of polyhedral specifications. Vasillache et al. [21, 28] make a strong case that a polyhedral representation of programs is especially needed to avoid the blowup of the intermediate program representation (IR) when many transformations are repeatedly applied, as is becoming increasingly common.

* This work was funded in part by the National Science Foundation, Award Number: 0917319

A number of polyhedral tools and components for generating efficient code are now available [2, 3, 7, 9–11, 17, 22]. Typically, they are source-to-source, and first extract a section of code amenable to polyhedral analysis, then perform a sequence of analyses and transformations, and finally generate output.

Many of these tools are designed to be fully automatic. Although this is a very powerful feature, and is the ultimate goal of the automatic parallelization community, it is still a long way away. Most existing tools give little control to the user, making it difficult to reflect application/domain specific knowledge and/or to keep up with the evolving architectures and optimization criteria. Some tools (e.g., CHiLL [3]) allow users to specify a set of transformations to apply, but the design space is not fully exposed.

In particular, few of these systems allow for explicit modification of the memory (data-structures) of the original program. Rather, most approaches assume that the allocation of values to memory is an inviolate constraint that parallelizers and program transformation systems must always respect. There is a body of work towards finding the "optimal" memory allocation [4, 23, 25, 26]. However, there is no single notion of optimality, and existing approaches focus on finding memory allocation given a schedule or finding a memory allocation that is legal for a class of schedules. Therefore, it is critical to elevate data remapping to first-class status in compilation/transformation frameworks.

To motivate this, consider a widely accepted concept, *reordering*, namely changing the temporal order of computations. It may be achieved through tiling, skewing, fusion, or a plethora of traditional compiler transformations. It may be used for parallelism, granularity adaptation, or locality enhancement. Regardless of the manner and motivation, it is a fundamental tool in the arsenal of the compiler writer as well as the performance tuner.

An analogous concept is "*data remapping*," namely changing the memory locations where (intermediate as well as final) results of computations are stored. Cases where data remapping is beneficial have been noted, e.g., in array privatization [16] and the manipulation of buffers and "stride arrays" when sophisticated transformations like time-skewing and loop tiling are applied [30]. However, most systems implement it in an ad hoc manner, as isolated instances of transformations, with little effort to combine and unify this aspect of the compilation process into loop parallelization/transformation frameworks.

In this paper, we present an open source polyhedral program transformation system, called `AlphaZ`, that provides a framework for prototyping analyses and transformations. We illustrate possible uses of our system through two examples that benefit from explicit representation of reductions and memory re-mapping.

2 Background

In this section we provide the necessary background of the polyhedral model, and summarize related work that use it for compiler optimization.

2.1 The Polyhedral Model

The strength of the polyhedral model as a framework for program analysis and transformation are its mathematical foundations for two aspects that should be (but are often not) viewed separately: program *representation/transformation* and *analysis*. Feautrier [5] showed that a class of loop nests called Affine Control Loops (or Static Control Parts) can be represented in the polyhedral model. This allows compilers to extract regions of the program that are amenable to analyses and transformations in the polyhedral model, and to optimize these regions. Such code sections are often found in kernels of scientific programs, such as dense linear algebra, stencil computations, or dynamic programming.

In the model, each instance of each statement in a program is represented as an *iteration point*, in a space called *iteration domain* of the statement. Each such point is hence, an *operation*. The iteration domain is described by a set of linear inequalities forming a convex polyhedron using the following notation, where z is iteration point, A is a constant matrix, and b is a constant vector.

$$D = \{z \mid Az + b \geq 0, \, z \in Z^n\}$$

Dependences are affine functions, expressed as[3] $(z \rightarrow z')$, where z' consists of affine expressions of z. *What* a program computes is completely specified by the set of operations and the (flow) dependences between them. As noted by Feautrier, program memory and data-structures need not figure in this representation.

2.2 Memory-Based Dependences

The results of array dataflow analysis are based on the values computed by instances of statements, and therefore do not need any notion of memory. Therefore, program transformation using dataflow analysis results usually requires re-considering memory allocation of the original program. Most existing tools have made the decision to preserve the original memory allocation, and include memory-based dependences as additional dependences to be satisfied.

2.3 Polyhedral Equational Model

The `AlphaZ` system adopts an *equational* view, where programs are described as mathematical equations using the `Alpha` language [15]. After array dataflow analysis of an imperative program, the polyhedral representation of the flow dependences can be directly translated to an `Alpha` program. Furthermore, `Alpha` has reductions as first-class expressions [12] providing a richer representation.

We believe that application programmers (i.e., non computer scientists), can benefit from being able to program with equations, where performance considerations like schedule or memory remain unspecified. This enables a separation

[3] In the literature of the polyhedral model, the word dependence is sometimes used to express flow of data, but here the arrow is from the consumer to the producer.

of what is to be computed, from the mechanical, implementation details of *how* (i.e., in which order, by which processor, thread and/or vector unit, and where the result is to be stored).

To illustrate this, consider a Jacobi-style stencil computation, that iteratively updates a 1-D data grid over time, using values from the previous time step. A typical `C` implementation would use two arrays to store the data grid, and update them alternately at each time step. This can be implemented using modulo operations, pointer swaps, or by explicitly copying values. Since the former two are difficult to describe as affine control loops, the Jacobi kernel in `PolyBench/C 3.2` [20] uses the latter method, and the code (`jacobi_1d_imper`) looks as follows:

```
for (t = 0; t < T; t++)
   for (i = 1; i < N-1; i++)
      A[i] = foo(B[i-1] + B[i] + B[i+1]);
   for (i = 1; i < N-1; i++)
      B[i] = A[i];
```

When written equationally, the same *computation* would be specified as:

$$A(t,i) = \begin{cases} t = 0: & B_{\texttt{init}}(i); \\ t > 0 \leq i < N-1: & \texttt{foo}(A(t-1,i-1), A(t-1,i), A(t-1,i+1)); \\ t > 0 = i: & A(t-1,i); \\ t > 0 \wedge i = N-1: & A(t-1,i); \end{cases}$$

where A is defined over $\{t, i | 0 \leq t < T \wedge 0 \leq i < N\}$, and B_{init} provides the initial values of the data grid. Note how the loop program is already influenced by the decision to use two arrays, an implementation decision, not germane to the computation.

2.4 Related Work

The polyhedral model has a long history, and there are many existing tools that utilize its power. Moreover, it is now used internally in the IBM XL compiler family. We now contrast `AlphaZ` with such tools. The focus of our framework is to provide an environment to try many different ways of transforming a program. Since many automatic parallelizers are far from perfect, manual control of transformations can sometimes guide automatic parallelizers as we show later.

PLuTo is a fully automatic polyhedral source-to-source program optimizer tool that takes C loop nests and generates tiled and parallelized code [2]. It uses the polyhedral model to explicitly model tiling and to extract coarse grained parallelism and locality. Since it is automatic, it follows a specific strategy in choosing transformations.

Graphite is an optimization framework for high-level optimizations that are being developed as part of `GCC` now integrated to its trunk [19]. Its emphasis is to extract polyhedral regions from programs that GCC encounters, significantly more complex task than what research tools address, and to perform loop optimizations that are known to be beneficial.

AlphaZ is not intend to be full fledged compiler. Instead, we focus on intermediate representations that production compilers may eventually be able to extract. Although codes produced from our system can be integrated into a larger application, we do not insist that the process has to be fully automatic, thus expanding the scope of transformations.

PIPS is a framework for source-to-source polyhedral optimization using interprocedural analysis [8]. Its modular design supports prototyping of new ideas by *developers*. However, the end-goal is an automatic parallelizer, and little control over choices of transformations are exposed to the user.

Polyhedral Compiler Collections (PoCC) is another framework for source-to-source transformations, designed to combine multiple tools that utilize the polyhedral model [22]. POCC also seeks to provide a framework for developing tools like Pluto, and other automatic parallelizers. However, their focus is oriented towards automatic optimization, and they do not explore memory (re)-allocation.

MMAlpha is another system with similar goals to AlphaZ [9]. It is also based on the Alpha language. The significant differences between the two are that MMAlpha emphasizes hardware synthesis. It does not treat reductions as first class , and does no tiling. MMAlpha does provide memory reuse in principle, but in its context, simple projections that directly follow processor allocations are all that it needs to explore.

RStream from Reservoir Labs performs automatic optimization of C programs [17]. It uses the polyhedral model to translate C programs into efficient code targeting multi-cores and accelerators. Vasillache et al. [27] recently gave an algorithm to perform a limited form of memory (re)-allocation (the new mapping must *extend* the one in the original program).

Omega Project has led to development of a collection of tools [10, 24] that cover a larger subset of the design space than most other tools. The Omega calculator partially handles *uninterpreted function symbols*, which no other tools support. Their code generator can also re-allocate memory [24]. However, reductions are not handled by Omega tools.

CHiLL is a high-level transformation and parallelization framework using the polyhedral model [3]. It also allows users to specify transformation sequences through scripts. However, it does not expose memory allocation.

POET is a script-driven transformation engine for source-to-source transformations [31]. One of its goals is to expose parameterized transformations via scripts. Although this is similar to AlphaZ, POET relies on external analysis to verify the transformations in advance.

Finally, we note that none of these tools do anything with reductions.

3 The AlphaZ System

In this section we present an overview of AlphaZ, focusing, due to space limitations, on only the specific features needed for the examples in later sections (see our technical report [32] for more details).

AlphaZ is designed to manipulate Alpha equations, either written directly or extracted from an affine control loop. It does this through a sequence of commands, written as a separate script. The program is manipulated through a sequence of transformations, as specified in the script. Typically, the final command in the script is a call to generate code (OpenMP parallel C, with support for parameterized tiling [7, 11]). The pen-ultimate set of commands specify, to the code generator, the (i) schedule, (ii) memory allocation, and (iii) additional (i.e., tiling related) mapping specifications.

The key design difference from many existing tools is that AlphaZ gives the user full control of the transformations to apply. Our ultimate goal is to develop techniques for automatic parallelizers, and the system can be used as an engine to try new strategies. This allows for trying out new program optimizations that may not be performed by existing tools with high degree of automation. The key benefits for this are:

- Users can *systematically* apply sequences of transformations without rewriting the program by hand.
- Compiler writers can *prototype* new transformations/code generators. New compiler optimizations may eventually be re-implemented for performance/robustness, but prototyping requires much less effort.

In the following, we use two examples to illustrate benefits of the ability to reconsider memory allocation, and to manipulate reductions. Section 4 illustrates the importance of memory re-mapping, with a benchmark from PolyBench/C 3.2 [20], and Section 5, presents an application of a very powerful transformation on reductions, called Simplifying Reductions. We show that the algorithmic complexity of an implementation of RNA secondary structure prediction alignment algorithm from UNAfold package [14] can be reduced from $O(N^4)$ to $O(N^3)$ through a systematic application of AlphaZ transformations.

4 Time-Tiling of ADI-like Computation

The Alternating Direction Implicit method is used to solve partial differential equations (PDEs). One of the stencil kernels in PolyBench/C 3.2 [20], adi/adi.c resembles ADI computation.[4]

ADI with 2D discretization solves two sets of tridiagonal matrices in each time step. The idea behind ADI method is to split the finite difference system of

[4] There is an error in the implementation, and time-tiling would not be legal for a correct implementation of ADI. The program in the benchmark nevertheless illustrates our point that existing tools are incapable of extract the best performance, largely because of lack of memory remapping.

equations of a 2D PDE into two sets: one for the x-direction and another for y. These are then solved separately, one after the other, hence the name *alternating direction implicit*.

Shown below is a code fragment from PolyBench, corresponding to the solution for one direction in ADI. When this code is given to PLuTo [2] for tiling and parallelization, PLuTo fails to find that all dimensions can be tiled, and instead, tiles the inner two loops individually. The key reason is as follows: the value written by `S0` is later used in `S3`, since computing `S3` at iteration `[t,i1,i2]` (written `S3[t,i1,i2]`) depends on the result of `S0[t,i1,i2]` and `S0[t,i1,i2-1]`. Since the dependence vector is in the negative orthant, this *value-based dependence* does not hinder tiling in any dimension.

```
  for (t = 0; t < tsteps; t++) {
     for (i1 = 0; i1 < n; i1++)
     for (i2 = 1; i2 < n; i2++) {
S0:     X[i1][i2] = X[i1][i2] - X[i1][i2-1] * A[i1][i2]
                        / B[i1][i2-1];
S1:     B[i1][i2] = B[i1][i2] - A[i1][i2] * A[i1][i2]
                        / B[i1][i2-1];
     }

S2 ... // 1D loop updating X[*,n-1] (details irrelevant here)

     for (i1 = 0; i1 < n; i1++)
     for (i2 = n-1; i2 >= 1; i2--)
S3:     X[i1][i2] = (X[i1][i2] - X[i1][i2-1]
               * A[i1][i2-1]) / B[i1][i2-1];

   ... //second pass for i1 direction

  }
```

However, the original C code reuses the array `X` to store the result of `S0` as well as `S3`. This creates a memory-based dependence $\mathtt{S3}[\mathtt{t},\mathtt{i1},\mathtt{i2}] \rightarrow \mathtt{S3}[\mathtt{t},\mathtt{i1},\mathtt{i2}+1]$ because `S3[t,i1,i2]` overwrites `X[i1,i2]` used by `S3[t,i1,i2+1]`. Hence, `S3` must iterate in a reverse order to reuse array `X` as in the original code, whereas allocating another copy of `X` allows all three dimensions to be tiled.

4.1 Additional Complications

The memory-based dependences are the critical reason why the PLuTo scheduler (actually, we use a variation implemented in Integer Set Library by Verdoolaege [29]) cannot find all three dimensions to be tilable in the above code. Moreover, two additional transformations are necessary to enable to scheduler to identify this. These transformations can be viewed as partially scheduling the polyhedral representation before invoking the scheduler. `AlphaZ` provides a command, called Change of Basis (CoB), to apply affine transforms to statements of polyhedral domains.[5]

One of them *embeds* S2 which nominally has a 2D domain (and the corresponding statement in the second pass) into 3D space, *aligning* it to be adjacent to a boundary of the domain of S1. The new domain of S2 becomes (note the last equality) $\{\mathtt{t}, \mathtt{i1}, \mathtt{i2} \mid \mathtt{0} \leq \mathtt{t} < \mathtt{tsteps} \wedge \mathtt{0} \leq \mathtt{i1} < \mathtt{N} \wedge \mathtt{i2} == \mathtt{n} - \mathtt{1}\}$.

The other complication is that because of the reverse traversal of the i2 loop of S3, dependences obtained by dataflow analysis [5] are affine, not uniform: $\mathtt{S3}[\mathtt{t}, \mathtt{i1}, \mathtt{i2}] \rightarrow \mathtt{S2}[\mathtt{t}, \mathtt{i1}, \mathtt{n} - \mathtt{i2} - \mathtt{1}]$. If a CoB $(\mathtt{t}, \mathtt{i1}, \mathtt{i2} \rightarrow \mathtt{t}, \mathtt{i1}, \mathtt{n} - \mathtt{i2} - \mathtt{1})$ is applied to the domain of S3 we get a uniform dependence. After these three transformations (removing memory-based dependences, and the two CoBs) the PLuTo scheduler discovers that all loops are fully permutable.

We are not sure of the precise reason why PLuTo scheduling is not able to identify all dimensions are tilable without these transformations. Parts of PLuTo scheduling is driven by heuristics, and our conjecture is that these cases are not well handled. We expect these difficulties can be resolved, and that it is not an inherent limitation of PLuTo. However, a fully automated tool, prevents a smart user from so guiding the scheduler. We believe that guiding automated analyses can significantly help refining automated components of tools.

4.2 Performance of Time Tiled Code

Since PLuTo cannot tile the outer time loop, or fuse many of the loops due to the issues described above, PLuTo parallelized code contains 4 different parallel loops within a time step. On the other hand, AlphaZ generated code with time-tiling consists of a single parallel loop, executing wave-fronts of tiles in parallel. Because of this we expect the new code to perform significantly better.

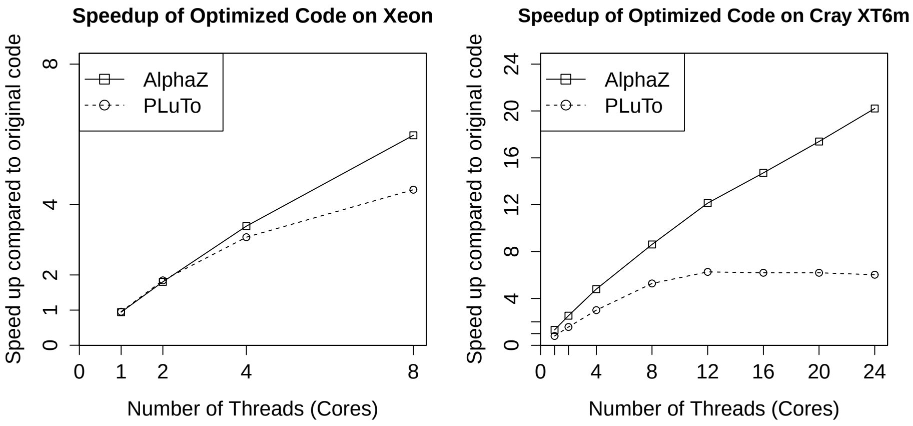
We measured the performance of the transformed code on a workstation, and also on a node in Cray XT6m. The workstation uses two 4 core Xeon5450 processors (8 cores total), 16GB of memory, and running 64-bit Linux. A node in the Cray XT6m has two 12 core Opteron processors, and 32GB of memory. We used `GCC/4.6.3` with `-O3 -fopenmp` options on the Xeon workstation, and `CrayCC/5.04` with `-O3` option on the Cray. PLuTo was used with options `--tile --parallel --noprevector`, since prevector targets ICC.

AlphaZ was supplied with the original C code along with a script file specifying pre-scheduling transformations described above, and then used the PLuTo scheduler to complete the scheduling. Memory allocation was specified in the script as well, and additional copies of X were allocated to avoid the memory-based dependences discussed above.

For all generated programs, only a limited set of tile sizes were tried (8, 16, 32, 64 in all dimensions), and we report the best performance out of these. The

[5] This is similar to the preprocessing of code generation from unions of polyhedra [1], where affine transforms are applied such that the desired schedule is followed by lexicographic scan of unions of polyhedra. Since the program representation in AlphaZ is equational, any bijective affine transformation is a legal CoB.

problem size was selected to have cubic iteration space that runs for roughly 60 seconds with the original benchmark on Xeon environment (tsteps = n = 1200).



(a) Xeon

(b) Cray XT6m

Fig. 1: Speedup of `adi.c` parallelized with PLuTo and `AlphaZ`, with respect to the execution time of the unmodified `adi.c` from `PolyBench/C 3.2`. Observe that coarser grained parallelism with time-tiling leads to significantly better scalability with higher core count on the Cray.

The results are summarized in Figure 1, confirming that the time-tiled version performs much better. On the Cray, we can observe diminishing returns of adding more cores with PLuTo parallelized codes, since only the inner two loops are parallelized. `AlphaZ` generated code does require more memory (this can actually, be further reduced), but at the same time, time-tiling exposes temporal reuse of the memory hierarchies.

5 Reducing Complexity of RNA folding

In this section, we outline steps to reduce the complexity of an application for RNA folding. Complete details, including the source `Alpha` program as well as the script, can be found in related Master's thesis and technical report [18, 33]. RNA secondary structure prediction, or RNA folding, is a widely used algorithm in bio-informatics. The original algorithm has $O(N^4)$ complexity, but an $O(N^3)$ algorithm has been proposed by Lyngso et. al [13]. However, no implementation of the $O(N^3)$ algorithm is publicly available.[6] This example illustrates one of the most powerful transformations in AlphaZ that is enabled through explicit representation of reductions. Specifically, we show how the equations that describe the algorithm can be systematically transformed to derive the $O(N^3)$ algorithm.

5.1 Reductions in AlphaZ

Reductions, associative and commutative operators applied to collections of values, are first class Alpha expressions [12]. It is well known that reductions are important patterns and have important performance implications. Moreover, reductions raise the level of abstraction over chains of dependences.

Alpha reductions are written as $\texttt{reduce}(op, f_p, E)$, where op is the reduction operator, f_p is a projection function, and E is the expression being reduced. The projection function f_p is an affine function that maps points in Z^n to Z^m, where m is usually smaller than n. When multiple points in Z^n is mapped to the same point in Z^m, the values of E at these points are combined using the reduction operator. For example, commonly used mathematical notations such as $X_i = \sum_{j=0}^{n} A_{i,j}$ is expressed as $X(i) = \texttt{reduce}(+, (i, j \to i), A(i, j))$. This is more general than mathematical notations, allowing us to easily express reductions with non-canonic projections, such as $(i, j \to i + j)$.

5.2 Simplifying Reductions

Simplifying Reductions [6] is the key transformation for reducing complexity of programs. We first explain the key idea behind this transformation with a simple (almost trivial) example. Consider an Alpha program computing a single variable, X_i, over a domain $\{i \mid 0 \leq i < N\}$ using the following equation

$$X[i] = \texttt{reduce}(+, \{j < i\} : A[j])$$

where each element is the sum of subsets of values $A_j, 0 \leq j < i < N$. Viewed naively, this would specify that each element of X is an (independent) reduction, and this would take $O(N^2)$ time to compute. Of course this is actually a prefix (scan) computation, and can be written as:

$$X_i = \begin{cases} i = 0 : A_i \\ i > 0 : A_i + X_{i-1} \end{cases}$$

Automatically detecting scans is the core of the reduction simplification algorithm [6]. The key idea is based on the observation that the expression inside the reduction (i.e., the reduction body) exhibits reuse: for the example above, at all

[6] Discussion with the original authors elicited the response that (i) the algorithm was "too complicated to implement" except in an early prototype, and (ii) limiting one of the parameters to 30 was "good enough" in practice.

points in a 2D space the value of the expression is $X[j]$ so there is reuse along the i direction. Reuse in the body of a reduction and its interaction with the domain boundaries leads to a scan. All the required transformations are implemented as `AlphaZ` commands. Some of the analyses performed are also implemented, but applying simplifying reductions to RNA folding requires additional human analysis, and thus human guided transformation.

5.3 RNA folding in UNAfold

UNAfold computes the RNA secondary structure through a dynamic programming algorithm, that uses a prediction model based on thermodynamics [14] and finds a structure with minimal free energy. For an RNA sequence of length N, the algorithm computes, for each subsequence from i to j, three tables (arrays) of free energy such that $1 \leq i \leq j \leq N$. The tables $Q(i,j)$, $Q'(i,j)$, and $QM(i,j)$ represent the free energy for three different substructures that may be formed. The following equation is the part of the original formulation corresponding to the dominant term that makes the algorithm $O(N^4)$.

$$Q'(i,j) = \min \begin{cases} \ldots \\ \min_{i<i'<j'<j} \begin{cases} E_{BI}(i,j,i',j') \\ Q'(i',j') \end{cases} \\ \ldots \end{cases} \tag{1}$$

Notice that the term uses four free variables i,j,i' and j', and since the constraints on these indices constitute the domain $\{i,j,i',j' | 1 \leq i < i' < j' < j \leq N\}$, it is easy to see the $O(N^4)$ complexity. The term corresponds to a substructure called *internal loops.*

5.4 Simplification

We focus on the dominating term in calculating the energy to illustrate the simplification. The term rewritten as a separate equation in `Alpha` is as follows

$$Q'(i,j) = \texttt{reduce}(\min, (i,j,i',j' \to i,j), E_{BI}(i,j,i',j') + Q'(i',j'));$$

where, the function E_{BI} is defined as follows:

$$E_{BI}(i,j,i',j') = Asym(i'-i-j+j') + S_P(i'-i+j-j'-2) + E_S(i,j) + E_S(i',j')$$

The body of the reduction does not exhibit any reuse, so we need to first inline the energy function E_{BI}. Doing this, and distributing out $E_S(i,j)$ gives the following:

$$Q'(i,j) = E_S(i,j) + \texttt{reduce}\left(\min, (i,j,i',j' \to i,j), \begin{cases} Asym(i'-i-j+j') & + \\ S_P(i'-i+j-j'-2) & + \\ E_S(i',j') + Q'(i',j') & \end{cases}\right) \tag{2}$$

The reduction still cannot be simplified, and the simplification algorithm [6] algorithm attempts to decompose the reduction from into two reductions (like expressing a double summation as a sum of a sum). The algorithm uses a dynamic programming algorithm to search through possible decompositions. One of the decompositions that lead to complexity reduction is the following:

$$Q'(i,j) = E_S(i,j) + \texttt{reduce}\left(\min, (i,j,d \to i,j), Q''(i,j,d)\right)$$

$$Q''(i,j,d) = \texttt{reduce}\left(\min, (i,j,i',j' \to i,j,j'-i'), \left\{\begin{array}{ll} Asym(i'-i-j+j') & + \\ S_P(i'-i+j-j'-2) & + \\ E_S(i',j') & + \\ Q'(i',j') & \end{array}\right.\right)$$

After the decomposition, the expression $S_P(i'-i+j-j')$ can be distributed out from the inner reduction. This can be found through analysis using null spaces of projection and access functions, which is also part of the simplifying reduction algorithm. In short, the analysis finds that null space of the access $S_p(i'-i+j-j'-2)$ contains the null space of the projection function $(i,j,i',j' \to i,j,j'-i')$. Thus, S_p term can be factored out from the reduction.

Then the remaining expressions evaluate to the same value for all points $[i',j',x]; x = j-i$. Taking advantage of this reuse and the property of the $\langle \min, + \rangle$ semi-ring allows the reduction to be simplified.

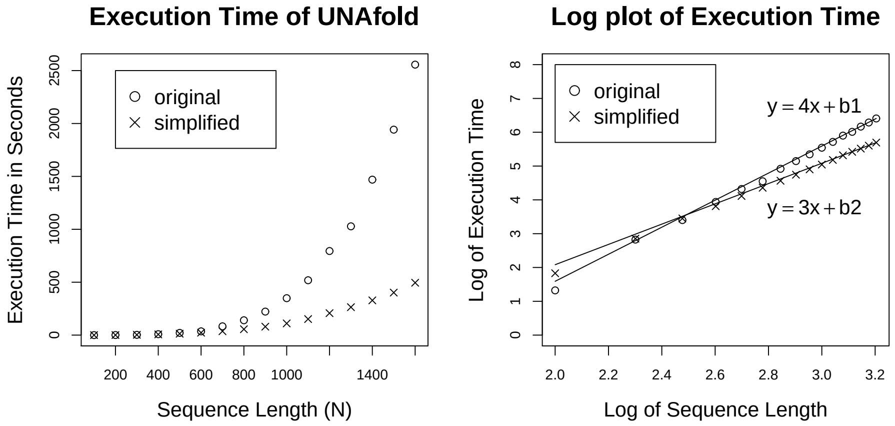
5.5 Need for Human Guidance

The above steps leading to reduction simplification can be mostly automated. In fact, once we have Equation 2, all the analyses required to apply the sequence of transformations are available. However, extracting Equation 2 from Equation 1 requires separating out boundary cases and other branches. In addition, E_{BI} must be inlined for the algorithm to detect reuse in the reduction.

Although our eventual goal would be fully automatic these steps, the current implementation of `AlphaZ` provides a powerful set of transformations that enable the user to systematically derive the lower complexity program. For RNA folding, the presence of reuse in the reduction was known [13], and such domain specific knowledge can be utilized by our system that gives the users flexible control over different transformations when needed. The specific semantics preserving transformations that we used are:

- `SimplifyingReduction` This is the key transformation that replaces a reduction with reuse by a scan.
- `Inline` (Inline EBI)
- `FactorOutFromReduction` Use distributivity to factor out terms from within reductions, where possible.
- `ReductionDecomposition` Decomposition of multidimensional reductions into a reduction of (sub) reductions.

In addition, some pre-processing transformations were also used.



(a) Raw Execution Time

(b) Log-Log Plot

Fig. 2: Execution Time of UNAfold after simplifying reduction compared with the original implementation. The two lines shown are with slopes 4 and 3.

5.6 Validation

We have applied the above transformation using `AlphaZ` to the UNAfold 3.8 [14]. The function `fillMatrices_1` in `hybrid-ss-min.c` was written in `Alpha`, and the simplifying transformation was applied. The sequential code generator in `AlphaZ` code was used to generate the simplified version of `fillMatrices_1` and replaced with the original function. Both the original and the simplified versions were compiled with `GCC/4.5.1`, with `-O3` option and the execution times were measured on a machine with Core2Duo 1.86GHz and 6GB of memory.

Figure 2 shows the measured performance (raw and log-log). It clearly shows the reduction in complexity, and, as expected, the speedups with transformed code becomes greater and greater as the sequence length grows.

6 Conclusions and Future Work

We have presented a system for exploring analyses and transformations in the polyhedral model. The two key features in our system are (i) the ability to re-consider memory allocations, and (ii) explicit representation of reductions.

Polyhedral representations of programs are expressed as systems of equations; which can either be extracted from loop nests, or programmed directly in an equational language. These polyhedral programs are manipulated using script driven transformations, to reflect human analyses or domain specific knowledge to help guide optimizing translations. Then executable code is generated by specifying schedule, memory allocation, and other implementation details.

AlphaZ has a number of transformations and code generators, and others are actively being developed. In addition to what previous tools have focused on, we believe that exploring memory allocations is very important. We expect it to become even more important as we target distributed memory machines.

While many tools focus on fully automated program transformations, a tool like AlphaZ that expose as much control to the user is helpful in developing and prototyping new ideas.

Although we have not presented details of our code generators, our code generators are highly modularized and extensible, enabling exploration of code generators as well. Our ongoing efforts are towards extending the code generators to other platforms such as CUDA, OpenCL, etc., and in implementing high level optimizations involving reductions, and many more.

References

1. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: Proceedings of the 13th IEEE International Conference on Parallel Architecture and Compilation Techniques. pp. 7–16. PACT '04, Washington, DC, USA (2004)
2. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 101–113. PLDI '08, ACM, New York, NY, USA (2008)
3. Chen, C., Chame, J., Hall, M.: Chill: A framework for composing high-level loop transformations. U. of Southern California, Tech. Rep pp. 08–897 (2008)
4. Darte, A., Schreiber, R., Villard, G.: Lattice-based memory allocation. IEEE Transactions on Computers 54(10), 1242–1257 (2005)
5. Feautrier, P.: Dataflow analysis of array and scalar references. International Journal of Parallel Programming 20(1), 23–53 (1991)
6. Gautam, G., Rajopadhye, S.: Simplifying reductions. In: POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 30–41. ACM, New York, NY, USA (2006)
7. Hartono, A., Baskaran, M.M., Bastoul, C., Cohen, A., Krishnamoorthy, S., Norris, B., Ramanujam, J., Sadayappan, P.: Parametric multi-level tiling of imperfectly nested loops. In: Proceedings of the 23rd international conference on Supercomputing. pp. 147–157. ACM, New York, NY, USA (2009)
8. Irigoin, F., Jouvelot, P., Triolet, R.: Semantical interprocedural parallelization: An overview of the pips project. In: Proceedings of the 5th international conference on Supercomputing. pp. 244–251. ACM (1991)
9. IRISA, C.: The MMAlpha environment
10. Kelly, W., Pugh, W., Rosser, E.: Code generation for multiple mappings. In: Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation. pp. 332–341. IEEE (1995)
11. Kim, D., Rajopadhye, S.: Efficient tiled loop generation: D-tiling. In: The 22nd International Workshop on Languages and Compilers for Parallel Computing (2009)
12. Le Verge, H.: Reduction operators in alpha. In: Etiemble, D., Syre, J.C. (eds.) Parallel Algorithms and Architectures, Europe. pp. 397–411. LNCS, Springer Verlag, Paris (June 1992), see also, Le Verge Thesis (in French)

13. Lyngs, R., Zuker, M., Pedersen, C., et al.: Fast evaluation of internal loops in rna secondary structure prediction. Bioinformatics 15(6), 440–445 (1999)
14. Markham, N., Zuker, M.: Software for nucleic acid folding and hybridization. Methods Mol. Biol 453, 3–31 (2008)
15. Mauras, C.: ALPHA: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones. Ph.D. thesis, L'Université de Rennes I, IRISA, Campus de Beaulieu, Rennes, France (December 1989)
16. Maydan, D., Amarasinghe, S., Lam, M.: Array-data flow analysis and its use in array privatization. In: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 2–15. ACM (1993)
17. Meister, B., Leung, A., Vasilache, N., Wohlford, D., Bastoul, C., Lethin, R.: Productivity via automatic code generation for PGAS platforms with the R-Stream compiler. In: Workshop on Asynchrony in the PGAS Programming Model (2009)
18. Pathan, T.: RNA Secondary Structure Prediction using AlphaZ. Master's thesis, Colorado State University, Computer Science Department (August 2010)
19. Pop, S., Cohen, A., Bastoul, C., Girbal, S., Silber, G., Vasilache, N.: Graphite: Loop optimizations based on the polyhedral model for gcc (2006)
20. Pouchet, L.N.: PolyBench, `www.cs.ucla.edu/~pouchet/software/polybench/`
21. Pouchet, L.N., Bastoul, C., Cohen, A., Vasilache, N.: Iterative optimization in the polyhedral model: Part I, one-dimensional time. In: IEEE/ACM Fifth International Symposium on Code Generation and Optimization (CGO'07). pp. 144–156. IEEE Computer Society press, San Jose, California (March 2007)
22. Pouchet, L.N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P.: Hybrid iterative and model-driven optimization in the polyhedral model. Tech. Rep. 6962, INRIA Research Report (June 2009)
23. Quilleré, F., Rajopadhye, S.: Optimizing memory usage in the polyhedral model. ACM Trans. Program. Lang. Syst. 22(5), 773–815 (2000)
24. Shen, T., Wonnacott, D.: Code generation for memory mappings. In: Proceedings of the 1998 Mid-Atlantic Student Workshop on Programming Languages and Systems
25. Strout, M., Carter, L., Ferrante, J., Simon, B.: Schedule-independent storage mapping for loops. ACM SIGOPS Operating Systems Review 32(5), 24–33 (1998)
26. Thies, W., Vivien, F., Sheldon, J., Amarasinghe, S.: A unified framework for schedule and storage optimization. ACM SIGPLAN Notices 36(5), 232–242 (2001)
27. Vasilache, N., Meister, B., Hartono, A., Baskaran, M., Wohlford, D., Lethin, R.: Trading off memory for parallelism quality. In: International Workshop on Polyhedral Compilation Techniques (IMPACT) (2012)
28. Vasilache, N.: Scalable Program Optimization Techniques In The Polyhedral Model. Ph.D. thesis, University of Paris-Sud 11 (2007)
29. Verdoolaege, S.: isl: An integer set library for the polyhedral model. Mathematical Software–ICMS 2010 pp. 299–302 (2010)
30. Wonnacott, D.: Achieving scalable locality with time skewing. International Journal of Parallel Programming 30(3), 181–221 (2002)
31. Yi, Q.: Poet: a scripting language for applying parameterized source-to-source program transformations. Software: Practice and Experience (2011)
32. Yuki, T., Basupalli, V., Gupta, G., Iooss, G., Kim, D., Pathan, T., Srinivasa, P., Zou, Y., Rajopadhye, S.: Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model. Tech. rep., CS-12-101, Colorado State University (2012)
33. Yuki, T., Gupta, G., Pathan, T., Rajopadhye, S.: Systematic implementation of fast-i-loop in UNAfold using AlphaZ. Tech. rep., CS-12-102, Colorado State University (2012)