DxTer: An Extensible Tool for Optimal Dataflow Program Generation

Bryan Marker Martin Schatz Devin Matthews
Isil Dillig Robert van de Geijn Don Batory

March 25, 2015

Abstract

DxTer (pronounced "dexter") is a tool for generating high-performance program implementations of an input dataflow graph. Given a specification $\mathtt{S}$ of a program in some domain $\mathcal{D}$, DxTer uses its *knowledge base* for $\mathcal{D}$ to explore a space of implementations of $\mathtt{S}$ and outputs an optimal implementation with respect to the cost model in $\mathcal{D}$. While DxTer search spaces can be massive (e.g., 10^{16}), this paper describes new techniques that allow DxTer to search such spaces and generate optimal code in seconds. In a case study, we apply DxTer to the domain of tensor contractions and show that DxTer generates programs that are competitive with (and sometimes superior to) a state-of-the-art tensor contraction tool.

1 Introduction

Scientific computing applications (SCAs) constitute an important class of software for modeling and simulating physical phenomena. Examples include computations in quantum chemistry, astrophysics, fluid mechanics, stress analysis, neutron transport, and the quantum scattering of charged particles. A characteristic of SCAs is that they are compute bound and run on distributed-memory supercomputers, for which one pays per minute of runtime on each core. Thus, it is very important to reduce SCA runtimes.

Unfortunately, the state-of-the-art in developing SCAs leaves much to be desired. In particular, while optimizing compilers have become very good at performing low-level optimizations such as strength reduction and loop invariant code motion, they offer no help in exploiting higher-level, domain-specific equivalences that can offer orders of magnitude in performance speed-ups. Consider the matrix multiplication expression $\mathtt{A} \cdot \mathtt{A}^{-1}$. While this expression can clearly be replaced by the identity matrix, optimizing compilers do not take advantage of such equivalences. Therefore, scientists who implement SCAs must manually apply these domain-specific optimizations until they achieve acceptable performance. Since this process is laborious, error-prone, and expensive,

code generation has become a popular alternative for alleviating the burden of developing high-performance SCAs [2, 4, 8, 29, 41, 43, 44].

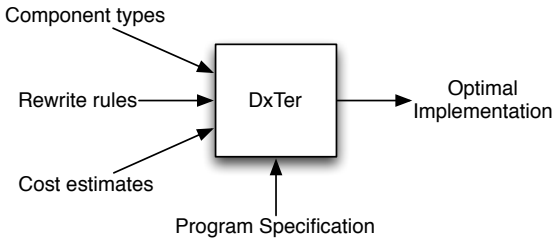

Figure 1: DxTer's input and output.

This paper presents the program generation engine underlying *DxTer*, a tool for generating high-performance, domain-specific programs such as SCAs. As shown in Figure 1, DxTer allows experts to rigorously encode their domain-specific expertise in the form of a *knowledge base* comprised of *component types*, *rewrite rules*, and a *cost model*. Component types define domain-specific computations, and rewrite rules define equivalences between dataflow graphs of these components. In addition, a cost model estimates the runtime of components as a function of their input.

Once an expert creates a knowledge base $\mathcal{B}_\mathcal{D}$ for a domain $\mathcal{D}$, non-experts can use DxTer to generate high-performance programs in domain $\mathcal{D}$. Specifically, a user only needs to provide a high-level specification `S` of the desired program in the form of a *dataflow graph*, a *directed acyclic multigraph (DAG)* whose nodes are components and whose edges represent input-output relations [34]. Given a specification `S`, DxTer explores the space of possible implementations of `S` and outputs an optimal implementation of `S` with respect to its knowledge base $\mathcal{B}_\mathcal{D}$. Thus, $\mathcal{B}_\mathcal{D}$ is reusable across user specifications. Further, since DxTer's search algorithm is domain-agnostic, DxTer can be used to generate programs in different domains by providing an appropriate knowledge base.

The key challenge underlying DxTer is to efficiently search the space of all possible implementations for a given specification. In practice, since the space of all implementations of a program is huge (e.g., 10^{16}), a naive code generation algorithm that explicitly enumerates all implementation options simply does not scale. To avoid this, DxTer implicitly represents the space of possible implementations using *partitioned dataflow graphs (PDGs)* and identifies independent subgraphs to decompose the search. Our novel PDG-based algorithm dramatically reduces the search space without sacrificing optimality and allows DxTer to generate high-performance code in a matter of seconds.

To demonstrate the usefulness and practicality of DxTer, we use it to generate high-performance SCAs that are based on *tensor contractions*. Tensor contractions, which are generalizations of matrix multiplication, have numerous applications in scientific computing, including in quantum chemistry [26], high-energy physics [28], and fluid mechanics [24]. We show that programs generated by DxTer are competitive with (and sometimes superior to) a state-of-the-art

tool for tensor contractions.

Our paper makes the following contributions:

- We give a high-level overview of DxTer and show how to encode domain-specific knowledge;
- We present DxTer's *domain-agnostic* search algorithm, including optimizations that enable it to explore massive search spaces with 10^{16} implementations in seconds; and
- We apply DxTer to the domain of tensor contractions and present strong experimental evidence that DxTer is capable of generating code that is competitive with – and sometimes superior to – domain-specific tools.

2 Overview of DxTer

At a high-level, there are two ways in which one can view DxTer. From the viewpoint of a compiler writer, DxTer is an extensible framework for implementing optimizing compilers for *domain-specific languages (DSLs)* or *application programming interfaces (APIs)*, targeting dataflow domains [34]. From the viewpoint of a programmer, DxTer is a generator of optimal dataflow programs from high-level specifications. This section gives an overview of DxTer from both perspectives and introduces key concepts that are used throughout the paper.

2.1 DxTer from a Compiler Writer's Perspective

From the perspective of a domain expert (e.g., compiler writer for a DSL), instantiating DxTer for a specific domain $\mathcal{D}$ requires specifying a knowledge base $\mathcal{B}_\mathcal{D} = (\mathcal{K}, \mathcal{R}, \mathcal{C})$ containing the following elements:

- **Components $\mathcal{K}$:** A *component* represents a computation. In DxTer, there are two kinds of components: *interfaces*, whose implementations are to be generated by DxTer, and *primitives*, the building blocks from which implementations are constructed. The expert needs to specify (i) the types of interface and primitive components of $\mathcal{D}$, and (ii) how primitives are implemented using existing code. For instance, a generic mathematical operation such as *matrix multiplication* is an example of an interface. On the other hand, a specific implementation of matrix multiplication, such as a call to the `Gemm` routine in the BLAS [7] library, exemplifies a primitive.

- **Rewrite rules $\mathcal{R}$:** *Rewrite rules* in DxTer specify semantic equivalences between graphs of components. Some rewrites are *refinements*, which indicate how a particular interface is implemented using primitives or lower-level interfaces. Other rewrites, called *optimizations*, specify equivalences between different graphs.[1]

- **Cost model $\mathcal{C}$:** An expert provides a *cost model* for each primitive. A cost model for a primitive $\mathcal{P}$ is a function of the parameters of $\mathcal{P}$ and yields an estimate of its execution cost. DxTer uses these estimates for rank-ordering different implementations. While cost models can estimate different kinds computational resources (e.g., memory), this paper focuses on cost models that only predict runtime.

Fundamental to DxTer is the use of dataflow graphs. Each vertex $\mathtt{v} \in \mathtt{V}$ of a dataflow graph $\mathtt{G} = (\mathtt{V}, \mathtt{E})$ represents a stateless component $\mathcal{K}$ in DxTer's knowledge base, and each arc $\mathtt{e} \in \mathtt{E}$ represents a data dependence (i.e., an input-output relation). For example, an edge from component `A` to component `B` indicates that the output of `A` is an input of `B`.

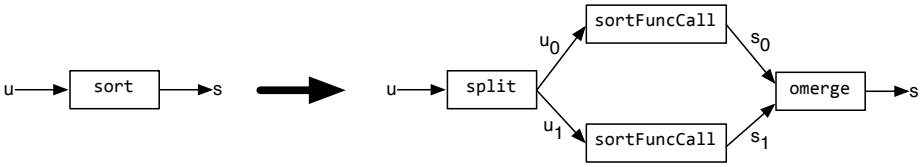

Figure 2: A graph rewrite rule where the interface to the left of the thick arrow is replaced by the graph on the right.

DxTer uses dataflow graphs to represent both specifications and implementations of programs. Hence, rewrite rules $\mathcal{R}$ in DxTer's knowledge base are specified as transformations from one dataflow graph, called the *left-hand side (LHS)* of the rewrite, to another one, called *the right-hand-side (RHS)*. Figure 2 shows a rewrite rule (specifically, a refinement) describing a map-reduce implementation for a `sort` component.

2.2 DxTer from a Programmer's Perspective

From the perspective of a non-expert user, it is straightforward to use DxTer to generate programs in $\mathcal{D}$, assuming an expert has already created a knowledge base $\mathcal{B}_\mathcal{D}$ for $\mathcal{D}$. The user only needs to provide a *specification*, which is a dataflow graph where typically all nodes are interfaces. In contrast, an *implementation* (or *implementing graph*) in DxTer is a dataflow graph where all nodes are primitives.

[1]The term *optimization* does *not* necessarily indicate that applying a given rewrite rule is *guaranteed* to improve performance but, rather, that it *may* result in a speed-up when combined with other optimizations or when used on some hardware architectures.

Internally, DxTer maps a specification to a large number of implementing graphs, all of which implement the specification and are correct-by-construction [3, 11, 27, 35] with respect to DxTer's knowledge base. DxTer then uses expert-supplied cost functions to rank implementations. In particular, given cost function $\mathcal{C}$, we define the cost of an implementing graph $\mathtt{G} = (\mathtt{V}, \mathtt{E})$ as:

$$\mathcal{C}(\mathtt{G}) = \sum_{\mathtt{v} \in \mathtt{V}} \mathcal{C}(\mathtt{v}).$$

DxTer ranks implementing graphs and translates the lowest-cost implementation to code.

3 Search

We present DxTer's basic search algorithm and then describe optimality-preserving improvements to make it practical.

3.1 Basic Search

```
1:  procedure EXPAND(S, R)
2:      S.HasExpanded ← false
3:      Ω ← {S}
4:      while ∃G. G ∈ Ω ∧ ¬G.HasExpanded do
5:          for all G ∈ Ω ∧ ¬G.HasExpanded do
6:              for all τ ∈ R do
7:                  if CanApply(τ, G) then
8:                      Φ ← Apply(τ, G)
9:                      Ω ← Ω ∪ Φ
10:                 end if
11:             end for
12:             G.HasExpanded ← true
13:         end for
14:     end while
15:     return Ω
16: end procedure
```

```
1:  procedure BASICSEARCH(S, R, C)
2:      return MinCostImpl(EXPAND(S, R), C)
3:  end procedure
```

Figure 3: BASICSEARCH algorithm

DxTer's basic search algorithm is shown in Figure 3. It uses a procedure called EXPAND that *expands* specification S by iteratively applying rewrite rules $\mathcal{R}$. That is, EXPAND generates a set Ω of *partial implementations*, which are dataflow graphs containing a combination of primitives and interfaces.

Each iteration of EXPAND finds a dataflow graph $\mathtt{G} \in \Omega$ to which it has not already applied rewrite rules. Then, for each rewrite $\tau \in \mathcal{R}$, it tests whether τ applies to graph G using the CANAPPLY function. Here, CanApply$(\tau, \mathtt{G})$ returns

true if and only if the left hand side of τ is a subgraph of G. If τ applies to G, it calls the APPLY procedure to obtain a new set of partial implementations Φ. In particular, APPLY applies rewrite τ to G by replacing subgraphs of G that match the LHS of τ with τ's RHS. Note that G may contain n subgraphs that match τ's LHS; hence, the resulting set Φ will have n graphs, one for each distinct application of τ.[2]

Next, we add the new partial implementations Φ to set Ω and repeat this process to a fixed point until all dataflow graphs in Ω have been expanded. Note that applying rewrites in different orders can produce the same graph; only unique graphs of Φ are added to Ω. As a result, EXPAND(S, $\mathcal{R}$) yields the reflexive transitive closure of S with respect to rewrite rules $\mathcal{R}$.

Since BASICSEARCH explores the space of all possible implementations, it is guaranteed to contain the optimal implementation of S with respect to DxTer's knowledge base. However, since the search space is at least exponential in the number of rewrite rules, BASICSEARCH is not always practical. In what follows, we describe meta-optimizations that significantly reduce the time and space to conduct a search without omitting the optimal implementation.[3]

3.2 Meta-Optimizations Based on Graph Partitioning

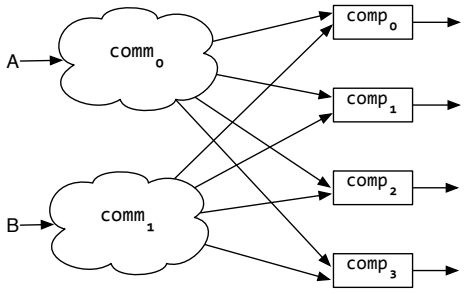

Figure 4: A common graph pattern that leads to inefficiencies in the basic search.

To understand the intuition underlying our meta-optimizations, consider the graph in Figure 4 where clouds represent subgraphs, and comm and comp stand for "communication" and "computation" respectively. Dataflow graphs such as the one from Figure 4 arise commonly in distributed-memory applications, where data can be redistributed among processes in many ways and the combination of redistributions must be implemented and optimized in concert.

Assuming $\texttt{comm}_0$ and $\texttt{comm}_1$ in Figure 4 have n and m implementing graphs respectively, BASICSEARCH generates at least $\texttt{n} \times \texttt{m}$ expansions of the dataflow graph in Figure 4. As there are *many* ways to implement each interface; n and

[2]Each graph in Φ will eventually have τ applied to it, producing all $2^{\texttt{n}}$ possible applications of τ on G.

[3]We do not discuss complexity or termination results from the search algorithm because they are dependent on the knowledge base input to DxTer.

m can be quite large. Hence, exploring the Cartesian product of the implementations of $\texttt{comm}_0$ and $\texttt{comm}_1$ is often infeasible in practice.

Fortunately, there are many situations in which we can explore the implementations of different subgraphs independently, thereby generating n+m rather than n×m implementations. For example, in Figure 4, assuming A ≠ B (and that the rewrite rules in the knowledge base satisfy a certain technical criterion), it is safe to explore $\texttt{comm}_0$ and $\texttt{comm}_1$ independently without losing our optimality guarantee. Similarly, if no rewrite rule applies to a *combination* of communication and computation components, implementations of any $\texttt{comm}_i$ and $\texttt{comp}_j$ can again be explored separately. We refer to the independent exploration of two different parts of the same graph as *partitioning*.

3.2.1 Partitioned Dataflow Graphs

We henceforth represent programs as *partitioned dataflow graphs (PDG)*. A PDG $(\mathcal{P}, \texttt{E})$ is a dataflow graph that contains vertices, denoted $\mathcal{P}$, that are *partitioned sets (PSets)* representing possible implementations of some specification. One can think of a PSet as a collection of dataflow graphs all of which implement the same functionality. Edges E in the PDG represent dataflow relations between PSets.

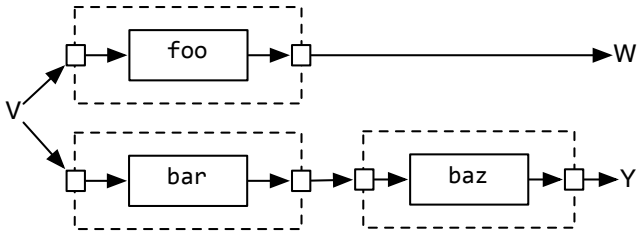

Figure 5: An example PDG.

Figure 5 shows an example PDG where dotted rectangles represent PSets. Inside a dotted rectangle is the specification graph of that PSet. Here, interfaces foo, bar, and baz belong to three different PSets. The "boxes" on the boundary of PSets are referred to as *tunnels* and allow a PSet to act like a single node by connecting the components outside the PSet to those within. Another important point about PDGs is that they can be hierarchical. In other words, just like procedures in code can call other procedures, a PSet in a PDG can contain other PDGs nested inside it.

3.2.2 Sufficient Conditions for Partitioning

Given a PDG, we are interested in the conditions under which two PSets inside this PDG can be explored independently. For this purpose, we first define *weakly connected dataflow graphs* and then *local dataflow dependence*:

Definition 3.1. (Weakly connected dataflow graph) A dataflow graph G is *weakly connected* if there is a path between each pair of nodes when every directed edge in G is replaced by an undirected edge.

In other words, a dataflow graph G is weakly connected if the undirected graph induced by G is connected.

Definition 3.2. (Local dataflow dependence) We say there is a *local dataflow dependence (LDD)* between two PSets $\texttt{p}_0$ and $\texttt{p}_1$ of graph G if (i) $\texttt{p}_0$ and $\texttt{p}_1$ share the same input or (ii) G contains an edge between $\texttt{p}_0$ and $\texttt{p}_1$.

Figure 5 shows both forms of LDDs between PSets. Specifically, foo and bar have a LDD as they share the same input V, and bar and baz have a LDD because the output of bar is an input of baz. On the other hand, there is no LDD between the interfaces $\texttt{comm}_0$ and $\texttt{comm}_1$ in Figure 4 when $\texttt{A} \neq \texttt{B}$.

> ***Key observation #1:*** *Given a PDG* $(\mathcal{P}, \texttt{E})$ *with no local dataflow dependence between its nodes, we can explore each PSet independently as long as the LHS of all rewrite rules are weakly connected.*

Specifically, since no rewrite rule can apply to a combination of PSets in such a PDG, it is safe to separately search for optimal implementations of each PSet without losing global optimality. As we will see shortly, our optimized search algorithm exploits this observation to significantly reduce the search space.

It turns out that we can further improve the above observation: Even if there is a LDD between some pair of nodes $\texttt{p}_\texttt{i}$ and $\texttt{p}_\texttt{j}$ in the PDG, we may still be able to explore them independently if $\texttt{p}_\texttt{i}$ and $\texttt{p}_\texttt{j}$ have a *type-based independence* property. To explain this property, we first define the notion of *derived types*:

Definition 3.3. (Derived types) Let S be a specification in a domain $\mathcal{D}$ with corresponding rewrite rules $\mathcal{R}$. The *derived types* of S, written DerivedTypes(S), is the set:

$$\left\{ \texttt{T} \;\middle|\; \begin{array}{c} \texttt{G} \in \textsc{Expand}(\texttt{S}, \mathcal{R}) \wedge \texttt{n} \in \text{Nodes}(\texttt{G}) \\ \wedge\ \texttt{T} = \text{Type}(\texttt{n}) \end{array} \right\}$$

In other words, DerivedTypes(S) is the set of all component types that arise when we apply the reflexive transitive closure of the rewrite rules $\mathcal{R}$ to S.

Definition 3.4. (Type-based independence) Let p, p′ be two nodes in a PDG implementing specifications $\texttt{S}_0$, $\texttt{S}_1$ such that $\text{DerivedTypes}(\texttt{S}_0) = \theta_0$ and $\text{DerivedTypes}(\texttt{S}_1) = \theta_1$. We say that p and p′ satisfy the *type-based independence* property if no rewrite rule contains a pair of components of type $\texttt{t}_\texttt{i} \in \theta_0$ and $\texttt{t}_\texttt{j} \in \theta_1$ in its LHS.

To gain intuition about this property, recall $\texttt{comm}_0$ and $\texttt{comp}_1$ from Figure 4. Assuming there is no rewrite rule that contains both a communication and a computation component on its LHS, then $\texttt{comm}_0$ and $\texttt{comp}_1$ have the type-based independence property.

Key observation #2: *Given a PDG* $(\mathcal{P}, \mathtt{E})$ *where all nodes have the type-based independence property, we can explore each PSet separately and still guarantee optimality.*

We suspect type-based independence to be common in many domains. For instance, in our tensor knowledge base (see Section 4), no rewrite rule applies to both *communication* components (those that move data) and *computation* components; hence, implementations of such components can be searched independently.

We now put these two observations together to state a key lemma about separable PSets:

Definition 3.5. (Separable PSets) A pair of nodes $\mathtt{p}_0$ and $\mathtt{p}_1$ in a PDG are called *separable* if either (i) the LHS's of all rewrites are weakly connected and there is no LDD between $\mathtt{p}_0$ and $\mathtt{p}_1$ or (ii) $\mathtt{p}_0$ and $\mathtt{p}_1$ have type-based independence.

Lemma 3.1. Let $\mathtt{S}$ be a PDG with nodes $\mathtt{p}_0 ... \mathtt{p}_n$, implementing specifications $\mathtt{S}_0 ... \mathtt{S}_n$, respectively, and suppose $\mathtt{G}_i = \textsc{BasicSearch}(\mathtt{S}_i, \mathcal{R}, \mathcal{C})$. Further, suppose that the LHS of each rewrite rule in $\mathcal{R}$ is weakly connected. In this case, $\mathtt{G} = \textsc{BasicSearch}(\mathtt{S}, \mathcal{R}, \mathcal{C})$ contains $\mathtt{G}_0 ... \mathtt{G}_n$ as subgraphs if every pair of nodes $\mathtt{p}_i$ and $\mathtt{p}_j$ are separable.

Proof. The proof of this lemma is by contradiction: Assume $\mathtt{G}$ does not contain $\mathtt{G}_i$ and instead contains $\mathtt{G}'_i$. Let $\theta_i = \mathtt{DerivedTypes}(\mathtt{S}_i)$. The lowest-cost implementation of $\mathtt{S}_i$ is not $\mathtt{G}_i$, but $\mathtt{G}'_i$. If $\mathtt{G}'_i$ was found in $\textsc{Expand}(\mathtt{S}_i, \mathcal{R})$, it would have been returned from $\textsc{BasicSearch}(\mathtt{S}_i, \mathcal{R}, \mathcal{C})$. Therefore, some rewrite τ lowers the cost of some implementation of $\mathtt{S}_i$. The LHS of τ must partially apply to a subgraph of an implementation of $\mathtt{S}_i$ and a subgraph of an implementation of $\mathtt{S}_j$ in another PSet; otherwise, it would have been applied in $\textsc{Expand}(\mathtt{S}_i, \mathcal{R})$.

By the lemma, there are two cases: 1) the LHS's of all transformations are weakly connected and there is no LDD between $\mathtt{p}_i$ and $\mathtt{p}_j$ OR 2) there is an LDD, but there exists no $\mathtt{t}_i \in \theta_i$ and $\mathtt{t}_j \in \theta_j$ that are found together on the LHS of some rewrite τ.

1) As the LHS's of all transformations are weakly connected, no such τ can apply to an implementation in $\mathtt{S}_i$ and $\mathtt{S}_j$ that are not connected by a LDD (a contradiction).

2) As no τ contains component types $\mathtt{t}_i$ and $\mathtt{t}_j$ on its LHS, no transformation applies partially to subgraphs of implementations of $\mathtt{S}_i$ and $\mathtt{S}_j$ (a contradiction). □

Lemma 3.1 immediately gives us a way to optimize BasicSearch using the separability of search. In particular, given a PDG where all PSets are separable, we do not need to explore all possible combinations of implementations of those PSets explicitly. Instead, we can generate optimal implementations for each of them by separately calling our algorithm with each component as the initial specification and still guarantee optimality of the combination.

3.3 Search Algorithm with Partitioned Sets

We now present our PDG-based improved search algorithm that exploits separable PSets (recall Lemma 3.1). The ImprovedSearch algorithm shown in Figure 6 takes a PDG $\mathtt{S}$ where each interface belongs to its own PSet and then calls the ExpandWithPSets procedure.

At a high level, the ExpandWithPSets algorithm independently explores implementations of different PSets but merges two PSets $\mathtt{p}_0$ and $\mathtt{p}_1$ if they are determined not to be separable. Conceptually, when we *merge* two PSets $\mathtt{p}_0$ and $\mathtt{p}_1$, we replace them with a single PSet $\mathtt{p}_{01}$ that represents their combined functionality. When there are no more PSets to merge (i.e., all remaining PSets are separable), Lemma 3.1 implies that we can find the optimal implementation of specification $\mathtt{S}$ by composing the lowest-cost implementation of each PSet within $\mathtt{S}$.

```
 1: procedure ExpandWithPSets(S, R)
 2:     if ¬ IsPDG(S) then
 3:         return Expand(S, R)
 4:     end if
 5:     keepGoing ← true
 6:     while keepGoing do
 7:         keepGoing ← false
 8:         for all p ∈ S do
 9:             for all G ∈ p.impls ∧ ¬G.HasExpanded do
10:                 G.HasExpanded ← true
11:                 Ω ← ExpandWithPSets(G, R)
12:                 if Ω ≠ {G} then
13:                     keepGoing ← true
14:                     p.impls ← p.impls ∪ Ω
15:                 end if
16:             end for
17:         end for
18:         for all p0, p1 ∈ S ∧ p0 ≠ p1 do
19:             if MustMerge(p0, p1, R) then
20:                 Merge(S, p0, p1)
21:                 keepGoing ← true
22:             end if
23:         end for
24:     end while
25:     return {S}
26: end procedure
 1: procedure ImprovedSearch(S, R, C)
 2:     return MinCostImpl(ExpandWithPSets(S, R), C)
 3: end procedure
```

Figure 6: Improved search algorithm using PDGs

Let us now consider the algorithm from Figure 6 in more detail. ExpandWithPSets is a recursive procedure that expands dataflow graph $\mathtt{S}$ with respect to rewrite rules $\mathcal{R}$. The input graph $\mathtt{S}$ of ExpandWithPSets can either be a PDG or a regular dataflow graph without partitions. If $\mathtt{S}$ is not a PDG, we simply call the basic Expand algorithm defined in Figure 3; hence, lines 2–3 of Figure 6 correspond to the base case of the overall algorithm.

On the other hand, if S does correspond to a PDG, the algorithm performs a fixed point computation that expands and merges PSets within S as necessary (lines 6–24). Inside the while loop, we first expand all PSets of S that have not yet been processed (lines 8–17). In particular, for each PSet p of S, the algorithm iterates through the dataflow graphs p.impls that implement p and that have not yet been transformed (i.e., their HasExpanded flag is false). We then recursively call ExpandWithPSets on each $G \in p.impls$ to generate all possible implementations of G. Hence, when the for loop in lines 9–16 terminates, p.impls contains all implementations of partition p.

Next, in lines 18–23, the algorithm tests whether we need to merge any PSets within S.[4] Specifically, for every pair (p_0, p_1) of PSets within S, we use the auxiliary MustMerge function to determine whether or not p_0 and p_1 are separable according to Definition 3.5, In particular, MustMerge returns true if p_0 and p_1 have a local dataflow dependence within S or $\mathcal{R}$ contains a rewrite rule whose LHS matches types from both p_0.impls and p_1.impls.

If p_0 and p_1 are not separable (i.e., MustMerge returns true), we call the Merge procedure at line 20 to merge these two partitions. The procedure Merge removes p_0 and p_1 from S and creates a new PSet p_{new} in the revised S called S' such that:

- p_{new}.impls is obtained by taking the Cartesian product of p_0.impls and p_1.impls. In particular, let G be the subgraph of S that contains only p_0 and p_1 and the edges between them, and let $G(p, p')$ denote G with nodes p_0, p_1 replaced with p, p'. Then, for every $p \in p_0.impls$ and $p' \in p_1.impls$, we have $G(p, p') \in p_{new}.impls$, and $G(p, p')$.HasExpanded is false.
- For every edge (s, p_0) and (p_0, t) in S where $s, t \neq p_1$, we have $(s, p_{new}) \in S'$ and $(p_{new}, t) \in S'$, and
- For every edge (s, p_1) and (p_1, t) in S where $s, t \neq p_0$, we have $(s, p_{new}) \in S'$ and $(p_{new}, t) \in S'$.

Here, observe that the new PSet p_{new} may need to be merged with others in S'; hence, this merging operation is applied until a fixed point is reached.

To illustrate this merging process, consider the PDG in Figure 7(a) and suppose that we need to merge foo and bar. Figure 7(b) shows the result of calling Merge, which now introduces a LDD between the new partition and baz. Hence, we perform another merge operation, which results in the PDG shown in Figure 7(c).

[4] The reader may wonder why we merge PSets *after* generating their implementations. This is because determining whether two PSets are separable requires knowing their individual implementations (see Definitions 3.3, 3.4).

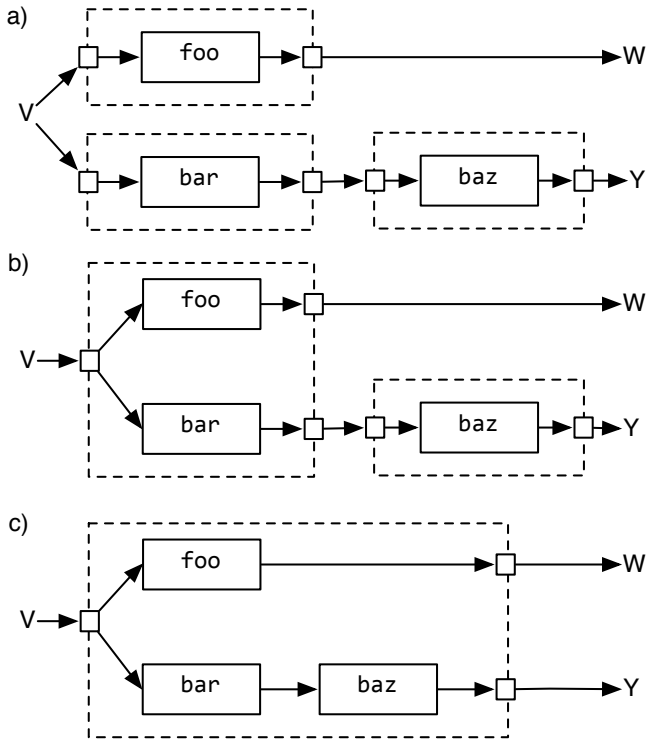

Figure 7: Two iterations of merging PSets.

Theorem 3.2. (Soundness) *The implementing graph returned by* ImprovedSearch *is a correct implementation of* `S` *with respect to DxTer's knowledge base.*

Proof. All implementations are obtained by applying rewrite rules $\mathcal{R}$ to specification `S`. Hence, if the rewrite rules $\mathcal{R}$ in the knowledge base are correct, then so is every implementing graph produced by DxTer. □

Theorem 3.3. (Optimality) *The implementing graph returned by* ImprovedSearch *is an optimal implementation of* `S` *with respect to DxTer's knowledge base.*

Proof. The fixed point computation in ExpandWithPSets terminates when `keepGoing` is set to false, which implies that all PSets are pairwise separable. Hence, by Lemma 3.1, the optimal implementation of `S` is composed of the optimal implementations of each PSet. □

4 A Target Domain: Tensor Contractions

We now give some background on tensors and explain how we used DxTer to generate code for SCAs that are composed of tensor contractions.

4.1 Background on Tensor Contractions

A *tensor* is a generalization of a matrix to n-*modes* (called dimensions in some texts), where n is the *order* of the tensor [32]. Vectors and matrices are tensors of order 1 and 2, respectively. For a tensor A of order n, the notation $\mathtt{A}_{\mathtt{i}_0\mathtt{i}_1\ldots\mathtt{i}_{(n-1)}}$ specifies a scalar element of the tensor by indexing into each of the n modes. If A is a matrix (i.e., a tensor of order 2), $\mathtt{A}_{\mathtt{ij}}$ is the element at row i and column j. In addition to indexing a scalar from a tensor, we can also select a subtensor in the same way we can index a subarray from an array. If $\mathtt{S}_\mathtt{j}$ denotes a set of indices and A is a tensor, then $\mathtt{A}[\mathtt{S}_0, \mathtt{S}_1, \ldots, \mathtt{S}_{(n-1)}]$ consists of the set of scalars $\mathtt{A}_{\mathtt{i}_0\mathtt{i}_1\ldots\mathtt{i}_{(n-1)}}$ such that $\mathtt{i}_\mathtt{j} \in \mathtt{S}_\mathtt{j}$.

A *tensor contraction* computes:

$$\mathtt{C}_{\pi(\mathtt{i}_0\ldots\mathtt{i}_{(m-1)}\mathtt{j}_0\ldots\mathtt{j}_{(n-1)})} = \sum_{\mathtt{k}_0\ldots\mathtt{k}_{(p-1)}} \mathtt{A}_{\pi(\mathtt{i}_0\ldots\mathtt{i}_{(m-1)}\mathtt{k}_0\ldots\mathtt{k}_{(p-1)})}\mathtt{B}_{\pi(\mathtt{k}_0\ldots\mathtt{k}_{(p-1)}\mathtt{j}_0\ldots\mathtt{j}_{(n-1)})}$$

where A, B, and C are tensors of order $\mathtt{m}+\mathtt{p}$, $\mathtt{p}+\mathtt{n}$, and $\mathtt{m}+\mathtt{n}$, respectively, and π is a permutation (shuffling of data). The familiar matrix multiplication is a special case for order-2 tensors (i.e., $\mathtt{m} = \mathtt{n} = \mathtt{p} = 1$): $\mathtt{C}_{\mathtt{ij}} = \sum_\mathtt{k} \mathtt{A}_{\mathtt{ik}}\mathtt{B}_{\mathtt{kj}}$. Similar to matrix multiplication, we might define a shorthand for tensor contractions $\mathtt{C} = \mathtt{A} \times \mathtt{B}$, although unlike matrix multiplication this shorthand alone does not fully specify the type of contraction. The *dot (scalar) product* is a tensor contraction, for which $\mathtt{m} = \mathtt{n} = 0$ in the above notation. In this case, all modes of the input tensors are *folded* (summed) over.

In general, input tensors of order r and s can have a t-fold contraction, where $0 \leq \mathtt{t} \leq \mathtt{r}, \mathtt{s}$. The output tensor then has $(\mathtt{r} + \mathtt{s} - 2\mathtt{t})$ modes. While for matrix multiplication the "folded" modes are generally the rows of A and the columns of B, any modes may be folded in a general tensor contraction. For example, two 3-mode tensors can be folded into a matrix: $\mathtt{C}_{\mathtt{ij}} = \sum_{\mathtt{kl}} \mathtt{A}_{\mathtt{kil}}\mathtt{B}_{\mathtt{klj}}$. Additionally, in the case of $\mathtt{t} = 0$, the contraction is an *outer product* including as a special case the usual matrix outer product $\mathtt{C}_{\mathtt{ij}} = \mathtt{A}_\mathtt{i}\mathtt{B}_\mathtt{j}$. In summary, both the number and positions of the modes to be folded are required to fully specify a tensor contraction.

As mentioned earlier, tensor contractions have many applications in scientific computing. Scientists produce equations that are k-fold contractions of order-n tensors for different values of $\mathtt{k} \leq \mathtt{n}$. From a programming perspective, the number of possible implementations of such equations is huge because one can express each contraction in many different ways due to algebraic laws of tensors. Since some implementations result in much better performance than others, scientists manually experiment with different implementation options and compare them against each other. Consequently, scientists can spend significant amounts of time manually optimizing distributed-memory implementations of tensor contractions – a tedious and error-prone process.

4.2 DxTer Knowledge Base for Tensor Contractions

We now describe how to instantiate DxTer to generate parallel, high-performance tensor contraction implementations on distributed-memory hardware using *Redistribution Operations and Tensor Expression (ROTE)* formalisms [32]. We used ROTE as it conveniently captured the tensor abstractions that DxTer needed.

Portions of each tensor are stored on each of the $\mathtt{p}$ processes. We use the ROTE notation to express the way data is distributed [32]. In particular, each process has its own predefined index sets (e.g., $\mathtt{D}_0$, $\mathtt{D}_1$, $\mathtt{D}_{01}$, and $\mathtt{D}_{123}$) that specify where tensor data is stored and how it is distributed. For example, $\mathtt{A}[\mathtt{D}_{123}, \mathtt{D}_0]$ indexes a subtensor of $\mathtt{A}$. As each process's interpretation of $\mathtt{D}_{123}$ and $\mathtt{D}_0$ is different, the specific portion of $\mathtt{A}$ that is indexed can be different for each process. We omit specific details of what these mean past saying each $\mathtt{D}_\mathtt{X}$ is a set of indices and, in general, $\mathtt{D}_\mathtt{X} \neq \mathtt{D}_\mathtt{Y}$ when $\mathtt{X} \neq \mathtt{Y}$. The set of all indices is denoted by the symbol $*$.

Components. Three types of components appear in the DxTer knowledge base for tensors. They arise from ROTE notation and basic mathematical operations on tensors:

1. **Contraction**. A *contraction* component is an interface whose implementation we want DxTer to generate. Contraction operations vary with instance-specific details such as tensor order and modes over which the contraction folds. This component subtype contracts two tensors and adds it to another: $\mathtt{C} = \mathtt{A} \times \mathtt{B} + \mathtt{C}$.

2. **Communication**. A *communication* component redistributes data between processes. As with contractions, communication components have different flavors based on the starting and ending distribution of data. Communication components that map directly to an MPI call are primitives implemented by a ROTE API call. The remaining are interfaces whose implementations DxTer must generate. Even a redistribution primitive can be implemented in alternate ways, which DxTer explores via optimizations.

3. **Local Contraction**. A *local contraction* component is a primitive that directly implements a specific kind of contraction by calling a ROTE API function. It is called "local" because it is executed by each process on locally stored data.

Rewrite Rules. There are 47 refinements and 15 optimizations in DxTer's current knowledge base for tensors [5]. Below, we show a few typical examples. Other rewrites explore implementation options for communication components in order to improve performance. For the complete set of rewrite rules, see [5].

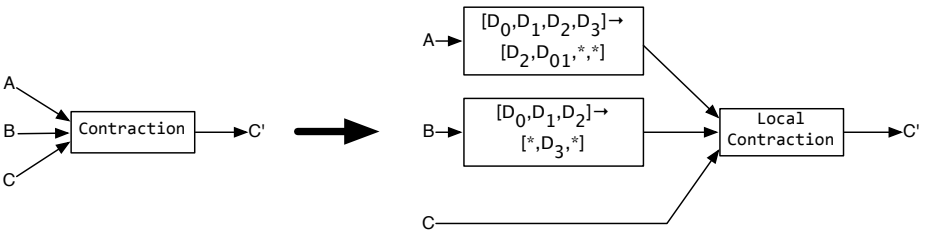

Figure 8: A particular instance of a sample refinement for a `Contraction` component.

Figure 8 shows a refinement of a `Contraction` interface that redistributes input tensors to parallelize computation. Components labeled $[\mathtt{X}] \rightarrow [\mathtt{Y}]$ correspond to data redistribution operations where data in distribution $[\mathtt{X}]$ are communicated into distribution $[\mathtt{Y}]$.[5]

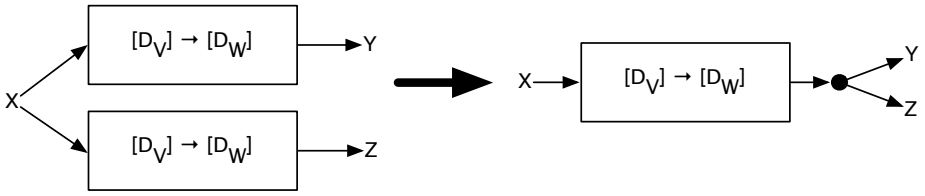

Figure 9: Sample optimization to remove redundant communication. $\mathtt{D_V}$ and $\mathtt{D_W}$ represent any distributions.

Figure 9 shows an optimization to replace a duplicate redistribution from $[\mathtt{D_V}]$ to $[\mathtt{D_W}]$ with a single instance, resulting in improved performance.

Cost Model. The final piece that completes DxTer's tensor knowledge base is the cost model.[6] In this domain, the key metric to optimize is execution runtime. For each primitive, we specify a function that estimates its runtime in terms of the size of its input parameters. For example, a `LocalContraction` primitive performs $2 \times \ell^{(\mathtt{m}+\mathtt{n}+\mathtt{p})}$ *floating point operations (FLOPS)*, where ℓ is the length of all modes of the locally stored portions of tensors in a contraction using the notation of Section 4.1 (in general, mode lengths are not the same, but we show the simplified case here). We model the cost of this primitive as $\gamma \times 2 \times \ell^{(\mathtt{m}+\mathtt{n}+\mathtt{p})}$ where γ is a constant representing the number of CPU cycles taken to perform a FLOP. We similarly specify cost estimates for communication primitives based on commonly accepted models [6, 25]. Note that we do not require the cost

[5]The rule in Figure 8 has distribution parameters ($[\mathtt{D_0}, \mathtt{D_1}, \mathtt{D_2}, \mathtt{D_3}]$) that must be supplied before the rule can be applied. These are parameters that are calculated using a non-trivial algorithm, so a single rewrite can represent rules for many distributions.

[6]DxTer can also generate a space of implementations and use empirical timing to determine the best performing code. For tensors, this is not feasible due to long runtimes, so it is standard to use cost models to judge performance.

model to predict true runtime since cost estimates are only used to rank one implementation option over another. While these estimates are rough, they are nonetheless sufficient for finding tensor contraction implementations competitive with hand-written code (see Section 5).

5 Experimental Results

5.1 Experimental Setup: CCSD Background

CCSD (coupled cluster single double) is a commonly used method in quantum computational chemistry [21, 33] that strikes a balance between communication cost and accuracy. It is a numerical, iterative method utilizing a set of equations to give an accurate reproduction of experimental results on electron correlation for molecules. Figure 10 lists the CCSD equations for closed-shell molecules as a set of ten terms, which are recomputed in each iteration. These terms have both computational and physical significance.

In our experiment, we use DxTer to generate an optimal implementation for each of the ten terms given in Figure 10. The result of each term is referenced in one or more later terms. To generate full CCSD code, we compose implementations of each individual term.

5.2 CCSD Code Performance

To evaluate the quality of code produced by DxTer, we compare the performance of code generated by DxTer against that of the *Cyclops Tensor Framework (CTF)* [40], which is a state-of-the-art distributed library for tensor contractions (see Section 6.4.1). We compare performance both for individual terms listed in Figure 10 as well as the full CCSD.

We test performance on a BlueGene/Q architecture where each node contains 16 shared-memory cores of IBM's 64-bit Power A2 architecture running at 1600 MHz. Each node has 16 GB of memory. We show results from 256 of these nodes, for a total of 4,096 cores. We run one MPI task on each core, and each of those runs with four shared-memory threads. Each node can attain a peak performance of 204.8 GFLOPS (10^9 floating-point operations per second) for a total of over 52.4 TFLOPS (10^{12} floating-point operations per second) combined peak performance.

For any particular execution of CCSD [21, 33], tensor modes are either of length n_v or n_o. These values depend on the molecule being studied; hence, we refer to them as *problem sizes*. We set $n_v = 10 \times n_o$, which is in the typical range of production CCSD calculations. In our evaluation, we show results for a range of n_o problem sizes.

$$
\begin{aligned}
\mathtt{W}^{\mathtt{bm}}_{\mathtt{je}} &= (2\mathtt{w}^{\mathtt{bm}}_{\mathtt{je}} - \mathtt{x}^{\mathtt{bm}}_{\mathtt{ej}}) + \sum_{\mathtt{f}} (2\mathtt{r}^{\mathtt{bm}}_{\mathtt{fe}} - \mathtt{r}^{\mathtt{bm}}_{\mathtt{ef}})\mathtt{t}^{\mathtt{f}}_{\mathtt{j}} - \sum_{\mathtt{n}} (2\mathtt{u}^{\mathtt{nm}}_{\mathtt{je}} - \mathtt{u}^{\mathtt{mn}}_{\mathtt{je}})\mathtt{t}^{\mathtt{b}}_{\mathtt{n}} \\
&\quad + \sum_{\mathtt{fn}} (2\mathtt{v}^{\mathtt{fe}}_{\mathtt{nm}} - \mathtt{v}^{\mathtt{fe}}_{\mathtt{mn}})(\mathtt{T}^{\mathtt{bf}}_{\mathtt{jn}} + \frac{1}{2}\mathtt{T}^{\mathtt{bf}}_{\mathtt{nj}} - \tau^{\mathtt{bf}}_{\mathtt{nj}}) \\
\mathtt{X}^{\mathtt{bm}}_{\mathtt{ej}} &= \mathtt{x}^{\mathtt{bm}}_{\mathtt{ej}} + \sum_{\mathtt{f}} \mathtt{r}^{\mathtt{bm}}_{\mathtt{ef}}\mathtt{t}^{\mathtt{f}}_{\mathtt{j}} - \sum_{\mathtt{n}} \mathtt{u}^{\mathtt{mn}}_{\mathtt{je}}\mathtt{t}^{\mathtt{b}}_{\mathtt{n}} - \sum_{\mathtt{fn}} \mathtt{v}^{\mathtt{fe}}_{\mathtt{mn}}(\tau^{\mathtt{bf}}_{\mathtt{nj}} - \frac{1}{2}\mathtt{T}^{\mathtt{bf}}_{\mathtt{nj}}) \\
\mathtt{U}^{\mathtt{mn}}_{\mathtt{ie}} &= \mathtt{u}^{\mathtt{mn}}_{\mathtt{ie}} + \sum_{\mathtt{f}} \mathtt{v}^{\mathtt{fe}}_{\mathtt{mn}}\mathtt{t}^{\mathtt{f}}_{\mathtt{i}} \\
\mathtt{Q}^{\mathtt{mn}}_{\mathtt{ij}} &= \mathtt{q}^{\mathtt{mn}}_{\mathtt{ij}} + (1 + \mathcal{P}^{\mathtt{mi}}_{\mathtt{nj}}) \sum_{\mathtt{e}} \mathtt{u}^{\mathtt{mn}}_{\mathtt{ie}}\mathtt{t}^{\mathtt{e}}_{\mathtt{j}} + \sum_{\mathtt{ef}} \mathtt{v}^{\mathtt{ef}}_{\mathtt{mn}}\tau^{\mathtt{ef}}_{\mathtt{ij}} \\
\mathtt{P}^{\mathtt{ji}}_{\mathtt{mb}} &= \mathtt{u}^{\mathtt{ji}}_{\mathtt{mb}} + \sum_{\mathtt{ef}} \mathtt{r}^{\mathtt{bm}}_{\mathtt{ef}}\tau^{\mathtt{ef}}_{\mathtt{ij}} + \sum_{\mathtt{e}} \mathtt{w}^{\mathtt{bm}}_{\mathtt{ie}}\mathtt{t}^{\mathtt{e}}_{\mathtt{j}} + \sum_{\mathtt{e}} \mathtt{x}^{\mathtt{bm}}_{\mathtt{ej}}\mathtt{t}^{\mathtt{e}}_{\mathtt{i}} \\
\mathtt{H}^{\mathtt{m}}_{\mathtt{e}} &= \sum_{\mathtt{fn}} (2\mathtt{v}^{\mathtt{ef}}_{\mathtt{mn}} - \mathtt{v}^{\mathtt{ef}}_{\mathtt{nm}})\mathtt{t}^{\mathtt{f}}_{\mathtt{n}} \\
\mathtt{F}^{\mathtt{a}}_{\mathtt{e}} &= -\sum_{\mathtt{m}} \mathtt{H}^{\mathtt{m}}_{\mathtt{e}}\mathtt{t}^{\mathtt{a}}_{\mathtt{m}} + \sum_{\mathtt{fm}} (2\mathtt{r}^{\mathtt{am}}_{\mathtt{ef}} - \mathtt{r}^{\mathtt{am}}_{\mathtt{fe}})\mathtt{t}^{\mathtt{f}}_{\mathtt{m}} - \sum_{\mathtt{fmn}} (2\mathtt{v}^{\mathtt{ef}}_{\mathtt{mn}} - \mathtt{v}^{\mathtt{ef}}_{\mathtt{nm}})\mathtt{T}^{\mathtt{af}}_{\mathtt{mn}} \\
\mathtt{G}^{\mathtt{m}}_{\mathtt{i}} &= \sum_{\mathtt{e}} \mathtt{H}^{\mathtt{m}}_{\mathtt{e}}\mathtt{t}^{\mathtt{e}}_{\mathtt{i}} + \sum_{\mathtt{en}} (2\mathtt{u}^{\mathtt{mn}}_{\mathtt{ie}} - \mathtt{u}^{\mathtt{nm}}_{\mathtt{ie}})\mathtt{t}^{\mathtt{e}}_{\mathtt{n}} + \sum_{\mathtt{efn}} (2\mathtt{v}^{\mathtt{ef}}_{\mathtt{mn}} - \mathtt{v}^{\mathtt{ef}}_{\mathtt{nm}})\mathtt{T}^{\mathtt{ef}}_{\mathtt{in}} \\
\mathtt{z}^{\mathtt{a}}_{\mathtt{i}} &= -\sum_{\mathtt{m}} \mathtt{G}^{\mathtt{m}}_{\mathtt{i}}\mathtt{t}^{\mathtt{a}}_{\mathtt{m}} - \sum_{\mathtt{emn}} (2\mathtt{U}^{\mathtt{mn}}_{\mathtt{ie}} - \mathtt{U}^{\mathtt{nm}}_{\mathtt{ie}})\mathtt{T}^{\mathtt{ae}}_{\mathtt{mn}} + \sum_{\mathtt{em}} (2\mathtt{w}^{\mathtt{am}}_{\mathtt{ie}} - \mathtt{x}^{\mathtt{am}}_{\mathtt{ei}})\mathtt{t}^{\mathtt{e}}_{\mathtt{m}} \\
&\quad + \sum_{\mathtt{em}} (2\mathtt{T}^{\mathtt{ae}}_{\mathtt{im}} - \mathtt{T}^{\mathtt{ae}}_{\mathtt{mi}})\mathtt{H}^{\mathtt{m}}_{\mathtt{e}} + \sum_{\mathtt{efm}} (2\mathtt{r}^{\mathtt{am}}_{\mathtt{ef}} - \mathtt{r}^{\mathtt{am}}_{\mathtt{fe}})\tau^{\mathtt{ef}}_{\mathtt{im}} \\
\mathtt{Z}^{\mathtt{ab}}_{\mathtt{ij}} &= \mathtt{v}^{\mathtt{ab}}_{\mathtt{ij}} + \sum_{\mathtt{mn}} \mathtt{Q}^{\mathtt{mn}}_{\mathtt{ij}}\tau^{\mathtt{ab}}_{\mathtt{mn}} + \sum_{\mathtt{ef}} \mathtt{y}^{\mathtt{ab}}_{\mathtt{ef}}\tau^{\mathtt{ef}}_{\mathtt{ij}} + (1 + \mathcal{P}^{\mathtt{ai}}_{\mathtt{bj}}) \Bigg\{ \sum_{\mathtt{e}} \mathtt{r}^{\mathtt{ej}}_{\mathtt{ab}}\mathtt{t}^{\mathtt{e}}_{\mathtt{i}} \\
&\quad - \sum_{\mathtt{m}} \mathtt{P}^{\mathtt{ij}}_{\mathtt{mb}}\mathtt{t}^{\mathtt{a}}_{\mathtt{m}} + \sum_{\mathtt{e}} \mathtt{F}^{\mathtt{a}}_{\mathtt{e}}\mathtt{T}^{\mathtt{eb}}_{\mathtt{ij}} - \sum_{\mathtt{m}} \mathtt{G}^{\mathtt{m}}_{\mathtt{i}}\mathtt{T}^{\mathtt{ab}}_{\mathtt{mj}} + \frac{1}{2}\sum_{\mathtt{em}} \mathtt{W}^{\mathtt{bm}}_{\mathtt{je}}(2\mathtt{T}^{\mathtt{ae}}_{\mathtt{im}} - \mathtt{T}^{\mathtt{ae}}_{\mathtt{mi}}) \\
&\quad - (\frac{1}{2} + \mathcal{P}^{\mathtt{i}}_{\mathtt{j}}) \sum_{\mathtt{em}} \mathtt{X}^{\mathtt{bm}}_{\mathtt{ej}}\mathtt{T}^{\mathtt{ae}}_{\mathtt{mi}} \Bigg\}
\end{aligned}
$$

Figure 10: Terms used inside the main loop of CCSD [21, 33]. Tensor modes are labeled with "names" in the context of each computation to specify the modes of tensors that match up. For matrices, we call the modes "rows" and "columns." For tensors, we use single-letter names that are specified in subscripts and superscripts. For example $\mathtt{r}^{\mathtt{bm}}_{\mathtt{fe}}$ labels tensor $\mathtt{r}$'s four modes $\mathtt{b}, \mathtt{m}, \mathtt{f}$, and $\mathtt{e}$, but $\mathtt{r}^{\mathtt{fn}}_{\mathtt{be}}$ labels those same modes $\mathtt{f}, \mathtt{n}, \mathtt{b}$, and $\mathtt{e}$, respectively. $\sum$ denotes contractions. The symbol $\mathcal{P}$ represents a permutation of data where $\mathcal{P}^{\mathtt{r}}_{\mathtt{p}}\mathtt{X}^{\mathtt{rs}}_{\mathtt{pq}} = \mathtt{X}^{\mathtt{ps}}_{\mathtt{rq}}$. Tensor variables $\mathtt{t}$, $\mathtt{T}$, and τ hold the data CCSD is calculating and are updated in each iteration. Other tensor variables (labeled with lower case letters) are constants that hold data related to the molecule being studied. Thus, tensors $\mathtt{r}, \mathtt{t}, \mathtt{T}, \tau, \mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{q}$, and $\mathtt{y}$ are inputs. $\mathtt{z}$ and $\mathtt{Z}$ are outputs.

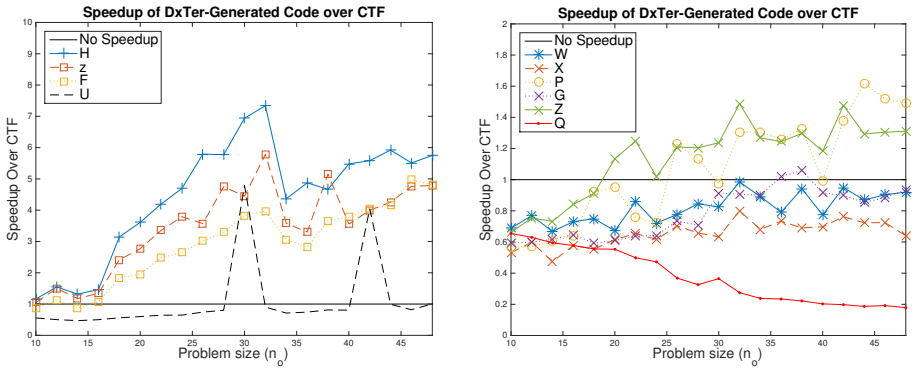

Figure 11: Speedup of DxTer-generated code over CTF on 4,096 for the ten terms, separated into terms with (left) big differences and (right) similar performance.

We generated DxTer code for a $4096 = 8 \times 8 \times 8 \times 8$ process grid, the configuration in which the code is run. For different process grids, DxTer would need to be re-run to generate correct and specially-optimized code.[7] We configure CTF as recommended by its lead developer in personal communication [39].

5.2.1 Individual Terms

We now compare the performance of DxTer-generated and CTF code on the 10 terms listed in Figure 10 across a range of problem sizes (10 to 48 in increments of 2). Figure 11 shows the speedup of DxTer-generated code over CTF; note that the graphs on the left and right sides of this figure use different ranges in their vertical axes. *Since DxTer-generated code works for larger problem sizes than CTF is able to handle, we only show results here up to CTF's largest size.* In Section 5.2.2, we show DxTer's performance on full CCSD for even larger problem sizes.

The left side of Figure 11 shows DxTer-generated code performs much better than CTF on some terms, particularly terms `H`, `z`, and `F`. The peaks in DxTer speedup over CTF performance for term `U` are caused by dips in CTF performance that were reproduced when we re-ran this experiment, but we do not have an explanation for this.

In contrast, the graph on the right side of Figure 11 shows the performance of CTF and DxTer-generated code are roughly comparable for most of the terms except for term `Q`, on which DxTer performs worse compared to CTF. For `Q`, CTF's default tensor distribution happens to match up with the distribution required for `Q` for one of the tensors, so it (luckily) avoids an expensive redistribution for that tensor. For other terms that use that tensor, redistributions

[7] We generate code for problem size $n_o = 50$ and run it for all problem sizes shown. DxTer could regenerate code for the various problem sizes, which might improve the performance we show.

are required because the default distribution does not line up.

Each term from Figure 10 accounts for different portions of the total CCSD computation (e.g., `Q` is less than 1% of runtime). For example, for a problem size of 50, `Z` requires a huge number (`6.72e14`) of floating-point operations while `Q` requires fewer (`5.55e12`). Thus, DxTer's poorer performance on some terms is overcome by better performance in others, especially `Z`, as shown by the full CCSD results next.

5.2.2 Full CCSD

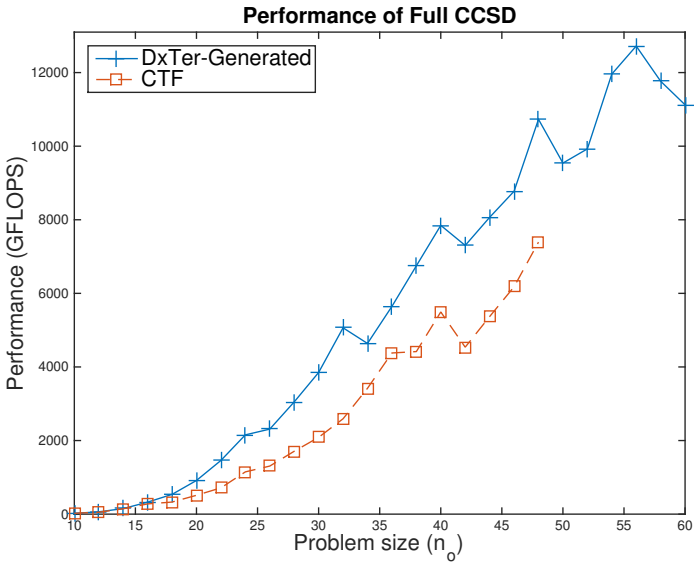

Figure 12: Performance of a single iteration of CCSD on 4,096 cores with one-quarter of peak performance at the top.

Figure 12 compares a single iteration of CCSD execution for CTF and DxTer-generated code across a range of problem sizes. Here, the cutoff for the vertical axis represents one-quarter of the theoretical peak performance on 4,096 cores. As Figure 12 shows, DxTer-generated code performs noticeably better than CTF. Further, DxTer-generated code can handle larger problem sizes. Since chemists typically use this code to run the largest problems on the smallest number of processes in the fastest time possible, we believe DxTer-generated code is particularly advantageous for the target users of this domain.

5.3 Code Generation Time

We now explore how long DxTer takes to generate code for each of the terms from Figure 10. Figure 13 shows the results for generating the ten CCSD terms on problem size $\mathtt{n_o} = 50$ and an $8 \times 8 \times 8 \times 8$ process grid. Here, the column

Term	Search Space Size	Lines of Code	With PSets	Without PSets (10 Minute Limit)
W	3.69e07	577	0.827s	Out of memory
X	5.39e05	414	0.331s	Time limit
U	6.6e01	199	0.013s	0.057s
Q	4.90e03	333	0.053s	0.277s
P	1.31e09	489	0.547s	Out of memory
H	8.82e02	209	0.050s	1.063s
F	2.59e03	382	0.035s	0.196s
G	5.10e03	409	0.059s	0.074s
z	1.33e07	657	0.069s	3.939s
Z	1.65e16	867	2.120s	Out of memory

Figure 13: Results from generating CCSD terms for problem size of 50 and 500.

labeled "Search Space Size" shows the size of the search space (which we can compute knowing the population sizes of separable PSets).

The second column labeled *"Lines of Code" (LOC)* shows the number of lines of code generated for the optimal implementation. While LOC is not necessarily a perfect proxy for code complexity, this statistic conveys that the generated code is not trivial.

The third column labeled "With PSets" shows the time DxTer needs to generate code using a single core of an Intel i7-930 processor (2.8 GHz). Finally, the last column, labeled "Without PSets," indicates whether DxTer can generate code within 10 minutes (where 10 was arbitrarily chosen) using the BasicSearch algorithm from Section 3.1 (i.e., no PSets or PDGs). If it can, we provide the time to generate. If it cannot, we specify if it runs out of memory or reaches the 10 minute cutoff.

As Figure 13 shows, DxTer's improved search algorithm from Section 3.3 is able to generate the optimal implementation of each term in less than a few seconds, and, in most cases, under one second. In contrast, for 40% of terms BasicSearch is either unable to finish code generation in ten minutes or exhausts 24GB of memory before the ten minute cutoff. Hence, these results indicate that exploiting the separability of search is essential to DxTer's ability to explore huge search spaces quickly.

6 Related Work

There are several projects related to DxTer both in the scientific computing as well as in the programming languages and software engineering communities. Here, we only address work that is closely related to DxTer.

6.1 DxT and Model Driven Engineering

DxTer is based on the *Design by Transformation (DxT)* paradigm for software development [1, 25, 31]. Users express program transformations as graph

rewrite rules, which are then used to explore the space of different implementation options. The DxT paradigm has its roots in *rule-based query optimization (RBQO)* and *model-driven engineering (MDE)*. In RBQO [23], a tool rewrites an input relational algebra expression using algebraic identities to produce a space of equivalent expressions and employs rough cost models to determine which is best. DxTer generalizes RBQO to new domains using graph rewrites.

In MDE, interface-only graphs called *Platform Independent Models (PIMs)* are transformed into architecture-specific (primitive-only) *Platform Specific Models (PSMs)* [10, 18]. Optimizations like those we use are not prominent in MDE, but are essential for DxTer to improve performance.

6.2 Dataflow Programming

DxTer uses *dataflow programming* to represent both program specifications and implementations. Dataflow programs are directed graphs that emphasize the movement of data between different components [14, 17, 42]. While there are numerous different ways to express dataflow computation, DxTer's visualization of dataflow programs follows LabVIEW, which is a general-purpose dataflow programming language and environment [19].

6.3 Program Synthesis

DxTer bears similarities to work on *program synthesis* in that it automatically generates implementations from high-level specifications. Recent work on program synthesis falls into three different categories: Approaches based on *program sketching* fill 'holes' in a program sketch with missing code fragments [37, 38]. Other synthesis systems generate programs from *input-output examples* specified by the user [12, 15, 20]. Finally, *component-based program synthesizers* generate programs using library components [13, 16]. DxTer differs from all of these synthesis techniques in the following ways: First, DxTer uses dataflow graphs as its specification rather than program sketches or input-output examples. Second, due to the presence of the rewrite rules in DxTer's knowledge base, generating a correct (but non-optimal) program is trivial. The real challenge lies in finding an optimal implementation of the specification (with respect to the user's cost model). Third, unlike most recent work on program synthesis [13, 16, 37, 38], DxTer does not use SAT or SMT solvers, but instead performs explicit search using partitioned dataflow graphs.

6.4 Code Generation for Tensor Contractions

In this paper, we applied DxTer to the domain of tensor contractions, for which there are other distributed-memory code generation tools, such as CTF [40], TCE [2], and RRR [30]. Unlike these projects which are tensor-specific, DxTer's code generation algorithm is domain-agnostic and can be used in other dataflow domains [1]. In addition to this key difference, we highlight some ad-

ditional differences below.

6.4.1 CTF

Cyclops Tensor Framework (CTF) is a new and efficient distributed-memory library for tensor contraction computing [40]. Unlike DxTer, CTF performs redistribution of the input tensors via the all-to-all collective [36], which works for any kind of redistribution but can be inefficient in some cases. In contrast, DxTer can combine different collectives to achieve higher performance. That is, DxTer searches for an efficient redistribution of tensors but CTF does not. Further, DxTer can optimize communication across a sequence of tensor contractions while CTF treats each contraction independently.

As mentioned earlier, DxTer uses the ROTE library for generating code. Unlike ROTE which currently only handles dense tensors, CTF can also deal with sparse tensors and their contractions.[8]

6.4.2 TCE

The *Tensor Contraction Engine (TCE)* [2] provides a domain-specific language and compiler for tensor contractions. Similar to DxTer, TCE performs search to generate high-performance code. However, TCE employs different phases of code generation, where each phase uses different sets of rewrite rules. In each phase, TCE selects the most promising representation and uses it in the next phase. While this strategy reduces the search space, it does not guarantee global optimality. Another important difference is that DxTer explores optimizations to reduce communication overhead, while TCE does not. We believe this can be a major performance advantage for DxTer over TCE.

We were not able to directly compare DxTer against TCE in our experimental evaluation for several reasons: First, TCE does not fully support the BlueGene/Q architecture on which we tested DxTer and CTF, it only provides basic correctness guarantees but not high performance. Second, the ROTE library underlying DxTer has rudimentary support for the Cray architecture on which TCE has been extensively evaluated. In particular, ROTE currently does not support a special form of communication that enables better performance on Cray machines.[9]

6.4.3 RRR

The *Reduction, Recursive broadcast, and Rotate (RRR)* [30] framework supports dense, distributed-memory tensor contractions (with symmetry). Similar to DxTer, RRR performs search to find good ways of parallelizing computation

[8] The tensor contractions used in our experimental evaluation do not have symmetry and only use dense tensors.

[9] While we do not have a direct comparison between DxTer and TCE, developers of CTF report that their system outperforms TCE [40] and in our preliminary tests on the Cray architecture DxTer-generated code performed roughly the same as CTF.

and optimizing data redistribution. However, a key difference is that, unlike DxTer, RRR does not support combinations of different tensor contractions. In our evaluation, we do not compare against RRR because the terms from Figure 10 involve combinations of tensors.[10]

6.5 Code Generation for Other SCAs

There are several domain-specific, scientific computing software generation projects [4, 8, 22, 29, 41, 43, 44] that apply rewrite rules to produce and then search a space of implementations. Built-to-Order BLAS [4], LGen [41], ATLAS [44], and AUGMEN [43] target the domain of dense linear algebra, SPIRAL [29] targets digital signal processing (as well as some types of dense linear algebra code), and FEniCS targets differential equations using finite element methods [22].

A key distinction between DxTer and these projects is that DxTer's code generation algorithm is domain-agnostic, while the afore-mentioned projects are domain-specific (e.g., [9]). Further, many of these projects target domains for which no cost model is sufficiently accurate because runtimes are very small and influenced by intricate cache behavior. Therefore, unlike DxTer, these projects use empirical evaluation rather than cost models to judge performance. Lastly, these projects provide no optimality guarantee as they do not search the entire space of implementations and instead use machine-learning-inspired techniques to search subspaces [4, 29, 41, 44].

7 Conclusions

We presented an extensible tool called DxTer for generating high-performance programs in dataflow domains. Given a knowledge base $\mathcal{B}_\mathcal{D}$ for domain $\mathcal{D}$ and a high-level specification `S` of a program, DxTer generates an optimal implementation of `S` with respect to knowledge base $\mathcal{B}_\mathcal{D}$. DxTer employs a new search algorithm based on partitioned dataflow graphs and is able to explore huge search spaces in seconds.

In this paper, we applied DxTer to the domain of tensor contractions, which have numerous applications in science and engineering. We evaluated the performance of DxTer-generated code on tensor contraction problems that arise in quantum chemistry and showed that DxTer-generated code can outperform CTF, a state-of-the-art system specifically designed for tensor contractions.

There are many possibilities for future work. One is to extend our tensor rule base to admit symmetric tensors. Another is to apply DxTer to other domains of interest in scientific computing. We expect to consider knowledge bases where the domain expert can specify the cost of multiple competing resources, such as runtime *and* memory, and then to extend our search algorithm to generate Pareto-optimal solutions for multi-objective optimization problems.

[10] We compared DxTer-generated code against RRR code for three of the five contractions reported in [30], and DxTer-generated code was either roughly the same or better than RRR code.

Acknowledgements

We gratefully acknowledge support for this work by NSF grants CCF-1212683, CCF-1320112, and ACI-1148125. Marker and Schatz held fellowships from Sandia National Laboratories. Marker also held a fellowship from the NSF (grant DGE-1110007).

We thank Tyler Smith for his help in tuning the runtime parameters for generated code.

This research used resources of the Argonne Leadership Computing Facility at Argonne National Lab, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References

[1] D. Batory, R. Goncalves, B. Marker, and J. Siegmund. Dark knowledge and graph grammars in automated software design. In *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 1–18. Springer International Publishing, 2013.

[2] G. Baumgartner et al. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. In *Proc. of the IEEE*, 2005.

[3] I. Baxter. Practical issues in building knowledge-based code synthesis systems. *6th Annual Reuse Workshop (WISR'93)*, 1993.

[4] G. Belter et al. Automating the generation of composed linear algebra kernels. In *SuperComputing*, 2009.

[5] Bryan Marker. DxTer Source. `https://github.com/DxTer-project/dxter`, 2015.

[6] E. Chan et al. Collective communication: theory, practice, and experience: Research articles. *Concurrency and Computation: Practice & Experience*, 19(13):1749–1783, September 2007. doi: `10.1002/cpe.v19:13`.

[7] J. J. Dongarra et al. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[8] D. Fabregat-Traver and P. Bientinesi. A domain-specific compiler for linear algebra operations. In *High Performance Computing for Computational Science – VECPAR 2010*, volume 7851 of *Lecture Notes in Computer Science*, pages 346–361, 2013.

[9] F. Franchetti. Private Correspondence, 2014.

[10] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., 2003.

[11] C. Green et al. Report on a knowledge-based software assistant. *Kestrel Institute Technical Report KES.U.83.2*, 1983.

[12] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[13] S. Gulwani et al. Synthesis of loop-free programs. In *ACM SIGPLAN Notices*, volume 46, pages 62–73. ACM, 2011.

[14] N. Halbwachs et al. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[15] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–328. ACM, 2011.

[16] S. Jha et al. Oracle-guided component-based program synthesis. In *ICSE*, volume 1, pages 215–224, 2010.

[17] W. M. Johnston et al. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[18] A. Kleppe et al. *MDA Explained: The Model-Driven Architecture*. Addison-Wesley, Boston, MA, 2003.

[19] LabView. NI LabVIEW. `http://www.ni.com/labview`, 2015.

[20] V. Le and S. Gulwani. Flashextract: A framework for data extraction by examples. In *PLDI*, page 55. ACM, 2014.

[21] T. J. Lee and J. E. Rice. An efficient closed-shell singles and doubles coupled-cluster method. *Chemical physics letters*, 150(6):406–415, 1988.

[22] A. Logg et al., editors. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer, 2012.

[23] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *ACM SIGMOD*, 1988.

[24] O. Malaspinas et al. Simulation of generalized newtonian fluids with the lattice boltzmann method. *International Journal of Modern Physics C*, 18 (12):1939–1949, 2007.

[25] B. Marker et al. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In *VECPAR*, 2012.

[26] D. Marx and J. Hutter. Modern methods and algorithms of quantum chemistry. *Grotendorst, J., Ed*, pages 301–449, 2000.

[27] M. Moriconi et al. Correct Architectural Refinement. *IEEE TSE*, 21:356–372, 1995.

[28] R. Orús and G. Vidal. Simulation of two-dimensional quantum systems on an infinite lattice revisited: Corner transfer matrix for tensor contraction. *Physical Review B*, 80(9):094403, 2009.

[29] M. Püschel and et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 2005.

[30] S. Rajbhandari et al. A communication-optimal framework for contracting distributed tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 375–386. IEEE Press, 2014.

[31] T. Riché et al. Pushouts in Software Architecture Design. In *GPCE*, 2012.

[32] M. Schatz. Anatomy of parallel compution with tensors. Technical Report TR-13-21, The University of Texas at Austin, Department of Computer Sciences, 2013.

[33] G. E. Scuseria et al. An efficient reformulation of the closed-shell coupled cluster single and double excitation (ccsd) equations. *The Journal of Chemical Physics*, 89(12):7382–7387, 1988.

[34] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[35] D. R. Smith and E. A. Parra. Transformational approach to transportation scheduling. *8th Knowledge-Based Software Engineering Conference*, 1993.

[36] M. Snir et al. *MPI: The Complete Reference*. The MIT Press, 1996.

[37] A. Solar-Lezama et al. Programming by sketching for bit-streaming programs. *PLDI*, pages 281–294, 2005.

[38] A. Solar-Lezama et al. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[39] E. Solomonik. Personal communication, 2015.

[40] E. Solomonik et al. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, Dec 2014.

[41] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 23:23–23:32. ACM, 2014. ISBN 978-1-4503-2670-4.

[42] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Langugage1.* Academic Press, 1985.

[43] Q. Wang et al. Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 25. ACM, 2013.

[44] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.