Polyhedral Optimizations for a Data-Flow Graph Language

Alina Sbîrlea[1] Jun Shirako[1] Louis-Noël Pouchet[2] Vivek Sarkar[1]

[1] Rice University, USA
[2] Ohio State University, USA

Abstract. This paper proposes a novel optimization framework for the Data-Flow Graph Language (DFGL), a dependence-based notation for macro-dataflow model which can be used as an embedded domain-specific language. Our optimization framework follows a "dependence-first" approach in capturing the semantics of DFGL programs in polyhedral representations, as opposed to the standard polyhedral approach of deriving dependences from access functions and schedules. As a first step, our proposed framework performs two important legality checks on an input DFGL program — checking for potential violations of the single-assignment rule, and checking for potential deadlocks. After these legality checks are performed, the DFGL dependence information is used in lieu of standard polyhedral dependences to enable polyhedral transformations and code generation, which include automatic loop transformations, tiling, and code generation of parallel loops with coarse-grain (fork-join) and fine-grain (doacross) synchronizations. Our performance experiments with nine benchmarks on Intel Xeon and IBM Power7 multicore processors show that the DFGL versions optimized by our proposed framework can deliver up to 6.9× performance improvement relative to standard OpenMP versions of these benchmarks. To the best of our knowledge, this is the first system to encode explicit macro-dataflow parallelism in polyhedral representations so as to provide programmers with an easy-to-use DSL notation with legality checks, while taking full advantage of the optimization functionality in state-of-the-art polyhedral frameworks.

1 Introduction

Hardware design is evolving towards manycore processors that will be used in large clusters to achieve exascale computing, and at the rack level to achieve petascale computing [29], however, harnessing the full power of the architecture is a challenge that software must tackle to fully realize extreme-scale computing. This challenge is prompting the exploration of new approaches to programming and execution systems, and specifically, re-visiting of the dataflow model — but now at the software level.

In the early days of dataflow computing, it was believed that programming languages such as VAL [5], Sisal [27], and Id [7] were necessary to obtain the benefits of dataflow execution. However, there is now an increased realization that "macro-dataflow" execution models [30] can be supported on standard multi-core processors by using data-driven runtime systems [4,3,36]. There are

many benefits that follow from macro-dataflow approaches, including simplified programmability [12], increased asynchrony [15], support for heterogeneous parallelism [32], and scalable approaches to resilience [39]. As a result, a wide variety of programming systems are exploring the adoption of dataflow principles [21,28,31], and there is a growing need for compiler and runtime components to support macro-dataflow execution in these new programming systems.

At the other end of the spectrum, polyhedral and other compiler frameworks implicitly uncover dataflow relationships in sequential programs through dependence analysis and related techniques. Though this approach can result in good performance, it usually requires a sequential program as input, which often limits portability when compared to higher-level dataflow program specifications.

We argue that a combination of declarative dataflow programming and imperative programming can provide a practical approach both for migrating existing codes and for writing new codes for extreme-scale platforms. We propose the use of a Data-Flow Graph Language (DFGL) as an embedded domain-specific language (eDSL) for expressing the dataflow components in an application. The DFGL notation is based on the Data Flow Graph Representation (DFGR) introduced in [31]. It enables individual computations to be implemented as arbitrary sequential code that operates on a set of explicit inputs and outputs, and defers the packaging and coordination of inter-step parallelism to the compiler and the runtime system. We propose a novel optimization framework for DFGL which enables correctness analysis of the application as well as low level transformations using a polyhedral compiler. Our performance experiments with nine benchmarks on Intel Xeon and IBM Power7 multicore processors show that the DFGL versions optimized by our proposed framework can deliver up to 6.9× performance improvement relative to standard OpenMP versions of these benchmarks.

Section 2 provides the background for this work, Section 3 discusses the motivation for the DFGL approach, Section 4 gives an overview of the compiler flow for DFGL subprograms, Section 5 describes the key technical points in our approach, Section 6 presents our experimental results, Section 7 discusses related work and Section 8 contains our conclusions.

2 Background

This section briefly summarizes the underlying DFGL programming model and the polyhedral compilation framework, which together form the foundation for the approach introduced in this paper.

2.1 DFGL model

The Data-Flow Graph Language (DFGL) model is a dependence based notation for dataflow parallelism, which is based on the Concurrent Collections (CnC) model [21,12] and the Data Flow Graph Representation (DFGR) [31]. DFGL describes computations using two main components: *steps*, that represent sequential subcomputations; and *items*, that represent data read and written by steps. The user describes an application by writing a graph that captures the relation among the items and steps.

As in the CnC model, steps are grouped into *step collections*, and represent all dynamic invocations of the same computational kernel. A unique identifier (*tag*) identifies a dynamic instance of a step `S` in a collection, `(S: tag)`. A special `env` step handles communications with "outside", e.g., initialization and emitting final results. Items are grouped into *item collections* and model all data used as inputs and outputs to steps. Analogous to tags for steps, elements in item collection `A` are uniquely identified by a *key*: `[A: key]`. In general, keys are represented as functions of step tags, such as affine functions or pure functions evaluated at run time [31]. The relations among steps and items are described by the "`->`" and "`::`" operations. The operation `->` describes data-flow as follows: `[A: key] -> (S: tag)` denotes item(s) read by a step[1], `(S: tag) -> [A: key]` denotes item(s) written by a step, and `(S: tag1) -> (S: tag2)` denotes a step-to-step ordering constraint. The operation `::` describes step creation; i.e., `(S: tag1) :: (T: tag2)` denotes instance(s) of `T` created by an instance of `S`[2]. The detailed semantics are shown in past work [31].

DFGL guarantees determinism and data race freedom by enforcing a dynamic single assignment rule. This rule states that any item in any collection can only be written once during the whole execution of the program. The model can be implemented to rely on different underlying runtimes. The compiler also has a lot of freedom in packaging the parallelism through code transformations such as loop tiling and generation of fine-grained (doacross) parallelism.

2.2 Polyhedral compilation framework

The polyhedral model is a flexible representation for arbitrarily nested loops. Loop nests amenable to this algebraic representation are called *Static Control Parts* (SCoPs) and represented in the SCoP format, where each statement contains three elements, namely, iteration domain, access relations, and schedule. SCoPs require their loop bounds, branch conditions, and array subscripts to be affine functions of iterators and global parameters.

Iteration domain, $\mathcal{D}^S$: A statement S enclosed by m loops is represented by an m-dimensional polytope, referred to as an iteration domain of the statement [19]. Each element in the iteration domain of the statement is regarded as a statement instance $\boldsymbol{i} \in \mathcal{D}^S$.

Access relation, $\mathcal{A}^S(\boldsymbol{i})$: Each array reference in a statement is expressed through an access relation, which maps a statement instance $\boldsymbol{i}$ to one or more array elements to be read/written [40]. This mapping is expressed in the affine form of loop iterators and global parameters; a scalar variable is considered as a degenerate case of an array.

Schedule, $\Theta^S(\boldsymbol{i})$**:** The sequential execution order of a program is captured by the schedule, which maps instance $\boldsymbol{i}$ to a logical time-stamp. In general, a schedule is expressed as a multidimensional vector, and statement instances are executed according to the increasing lexicographic order of their time-stamps.

Dependence Polyhedra, $\mathcal{D}^{S \to T}$: The dependences between statements S and T are captured by dependence polyhedra — i.e., the subset of pairs $(\boldsymbol{i}, \boldsymbol{i'}) \in \mathcal{D}^S \times \mathcal{D}^T$ which are in dependence. We note n the dimensionality of $\mathcal{D}^{S \to T}$.

[1] Step I/O may comprise a list of items, and item keys may include range expressions.
[2] A typical case is `env` step to create set of step instances where tag is a range.

Given two statement instances $\boldsymbol{i}$ and $\boldsymbol{i'}$, $\boldsymbol{i'}$ is said to depend on $\boldsymbol{i}$ if 1) they access the same array location, 2) at least one of them is a write and 3) $\boldsymbol{i}$ has lexicographically smaller time-stamp than $\boldsymbol{i'}$, that is $\Theta^S(\boldsymbol{i}) \prec \Theta^T(\boldsymbol{i'})$.

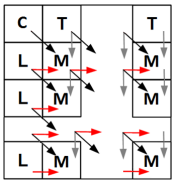


Fig. 1. Computation and dependence for Smith-Waterman.

```
[int A];
(corner:i,j) -> [A:i,j];
(top:i,j) -> [A:i,j];    (left:i,j) -> [A:i,j];
[A:i-1,j-1], [A:i-1,j], [A:i,j-1] -> (main_center:i,j) -> [A:i,j];
env::(corner:0,0);
env::(top:0,{1 .. NW});  env::(left:{1 .. NH},0);
env::(main_center:{1 .. NH},{1 .. NW});
[A:NH,NW] -> env;
```

Fig. 2. Input: DFGL for Smith-Waterman.

```
corner(0, 0);
for (c3 = 1; c3 <= NW; c3++) top(0, c3);
for (c1 = 1; c1 <= NH; c1++) left(c1, 0);
#pragma omp parallel for private(c3, c5, c7) ordered(2)
for (c1 = 0; c1 <= NH/32; c1++) {
  for (c3 = 0; c3 <= NW/32; c3++) {
#pragma omp ordered depend(sink: c1-1, c3) depend(sink: c1, c3-1)
    for (c5 = max(1, 32*c1); c5 <= min(NH, 32*c1+31); c5++)
      for (c7 = max(1, 32*c3); c7 <= min(NW, 32*c3+31); c7++)
        main_center(c5, c7);
#pragma omp ordered depend(source: c1, c3)
} }
```

Fig. 3. Output: optimized OpenMP for Smith-Waterman (using our system).

3 Motivating Example

The Smith-Waterman algorithm is used in evolutionary and molecular biology applications to find the optimal sequence alignment between two nucleotide or protein sequences, using dynamic programming to obtain the highest scoring solution. We show how this algorithm is encoded in our graph-based representation and then optimized by our polyhedral framework.

Figure 1 gives a visual representation of the Smith-Waterman algorithm, which contains 4 kind of steps: a single *corner* step (C) computing the top-left matrix corner and collections of steps computing the *top* row (T), *left* column (L) and the *main* body (M) of the matrix. The three-way arrows mark the flow of data between steps. As mentioned in Section 2.1, each instance of the same step collection is identified by a unique tag. Using a `(NH+1)×(NW+1)` integer matrix (which comprises item collection `A`), there are `NH` × `NW` main steps, each of which is identified by a tuple-tag `(i,j)`, with `1` ≤ `i` ≤ `NH` and `1` ≤ `j` ≤ `NW`.

The data dependences (represented by arrows in Figure 1) are modeled by using the tag `(i,j)` to identify a step instance and keys (affine functions of tag) to specify items; Note that all main steps read 3 items and write one item of collection `A`: `[A:i-1,j-1], [A:i-1,j], [A:i,j-1] -> (M:i,j) -> [A:i,j]`.

The DFGL specification for Smith-Waterman is shown in Figure 2. The first line of code declares an item collection, where each item is of type `int`. The next four lines of code specify, for each of the 4 steps, what items are read and written, as a function of the step instance's tag.

The final four lines specify what the environment needs to produce for the graph to start, and what it needs to emit after completion of the graph as output data. The environment starts all computation steps via `::` operation, (e.g., main steps of {`1 .. NH`} × {`1 .. NW`}). It also reads one item resulting from the computation (the bottom right corner, which contains the optimal sequence alignment cost).

Although the dependences in this DFGL program expose a wavefront parallelism (e.g., step instances `(M:1,10)`, `(M:2,9)`, ... `(M:10,1)` can run in parallel), the computation granularity of each instance is too small to be implemented as a concurrent task on current computing systems. Furthermore, there are several choices on how to implement this wavefront parallelism, e.g., as a regular forall loop parallelism via loop restructuring (skewing) or using a special runtime that supports software pipelining. Figure 3 shows the optimized code in OpenMP, as generated by our framework. Loop tiling is applied to the kernel so as to improve both data locality and computation granularity. To implement the pipeline parallelism, we rely on an OpenMP-based fine-grained synchronization library [34], which will be supported in OpenMP 4.1 standard [28]. These transformations brought significant improvements as reported in Section 6.

4 Converting DFGL to Polyhedral Representation

In this section, we first introduce the programming flow using DFGL as an embedded domain-specific language (eDSL) for expressing the dataflow components in an application. We also introduce the overview of our optimization framework, as well as the restrictions placed upon DFGL programs for compatibility with the polyhedral framework.

4.1 Embedded DFGL programming flow

As shown in Figure 4, we use pragma `dfgl` to specify a DFGL program embedded in a regular C program. Each item collection in the DFGL program requires a corresponding array that is declared and in scope at the `dfgl` pragma. Users can initialize items and obtain computation results outside the DFGL program

```
void foo() {
  //C region
  int A[NH+1][NW+1];
  ...
  #pragma dfgl
  {
    //DFGL region
    [int A];
    ...
  }
  print(A[NH][NW]);
}
```

Fig. 4. DFGL as an embedded DSL

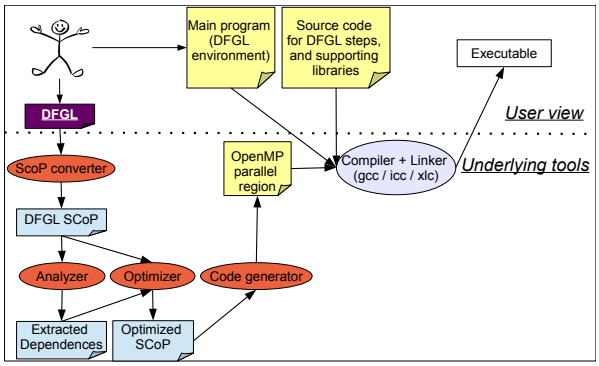


Fig. 5. Optimization and build flow for a DFGL parallel region.

via the corresponding arrays. To enable legality check in Section 5.2, users need to describe which items are to be initialized/emitted as a form of write/read on the environment, e.g., `env -> [A: key]` or `[A: key] -> env`. The flow for compiling a DFGL parallel region is shown in Figure 5. The user creates the DFGL description and provides the main program (DFGL environment) and codes for the compute steps. Then, they use our toolchain, which couples an extended translator [37] that we created for conversion to SCoP, and an extension to ROSE Compiler framework [2,33], to obtain an executable for running the application.

The first component of the toolchain is the *SCoP converter* that transforms the DFGL representation into a simplified SCoP format as described in Section 5.1. Next, we use the *Analyzer* to report errors in the input DFGL program and obtain the dependences. The dependences, along with the information from the DFGL SCoP, are then fed into the *Optimizer*. The final stage is the generation of the optimized OpenMP code, which is built together with the user-provided main program, kernels and libraries to obtain the executable.

4.2 DFGL restrictions for enabling polyhedral optimizations

To facilitate the conversion to a polyhedral representation, we focus on a restricted subset of DFGL that can be summarized as follows: (1) step tags are of the form $\boldsymbol{i} = (i_1, ..., i_k)$ with k the dimensionality of the associated step collection; (2) item keys are affine expressions of step tags; and (3) all steps are started by the environment such that the set of steps started can be described using only affine inequalities of the step tag. Note that a step-to-step dependence is converted into step-to-item and item-to-step dependences using a new item collection. Both rectangular regions (ranges [31]) and simple polyhedra shaped by affine inequalities of step tags are supported in DFGL. In practice, ranges and simple polyhedra are often enough to express the tag sets needed to model regular applications. They also come with the benefit of easy compilation to a loop-based language, which we will use to generate parallel OpenMP code.

The implementation we propose relies on generation of C code due to the benefits of high performance given by a low level language and the ease of pro-

gramming provided by DFGL, which abstracts applications using a high-level representation. This approach is also appropriate for using DFGL as an embedded DSL, since the OpenMP code that our toolchain generates can be integrated into larger code bases (in effect, an OpenMP parallel region is generated for each DFGL parallel region), while the user steps, which the generated code calls, can themselves be optimized routines or library calls (possibly with non-affine data accesses, since only the DFGL program is processed by the polyhedral framework, not the internal step code).

5 Polyhedral Optimizations for DFGL

In this section, we present the details of our analysis and optimizations for an input DFGL program, in the context of a polyhedral compilation framework.

5.1 Polyhedral representation of DFGL program

This section introduces our approach for creating a polyhedral representation of a DFGL program. Each step is viewed as a polyhedral statement, for which an iteration domain is constructed by analyzing the creation of step instances by the environment and access functions are constructed by analyzing the dataflow expressions.

SCoP for DFGL model As shown in Section 2.2, the original SCoP format consists of three components: iteration domain, access relation, and schedule. The restricted DFGL model defined in Section 4.2 allows to seamlessly create the iteration domain to be represented as a polyhedron bounded by affine inequalities, and the I/O relations of each step instance to be modeled as affine read/write access relations. Examples of DFGL code fragments and their SCoP representations are shown below.

`[A:i-1,j+1]->(S:i,j)->[B:i,j]` $\Leftrightarrow\ \mathcal{A}^{S_{R_1}}(i,j) = (A, i{-}1, j{+}1),\ \mathcal{A}^{S_{W_1}}(i,j) = (B,i,j)$

`env::(S:{1 .. N},{i .. M})` $\Leftrightarrow\ \mathcal{D}^S = \{(i,j) \in \mathbb{Z}^2 \mid\ 1 \leq i \leq \texttt{N}\ \wedge\ i \leq j < \texttt{M}\}$

Instead of specifying the sequential execution order (total order) among all step instances, the DFGL model enforces ordering constraints via dataflow: a step instance is ready to execute only once all of its input items (data elements) are available. Therefore, the SCoP format specialized for DFGL contains iteration domains and access functions, but no explicit schedule.

Dependence computations To compute polyhedral dependences between any two step instances, we need to determine their *Happens-Before* (HB) relation — i.e., which instance must happen before another [24]. By definition of the dynamic single assignment form, only flow dependences can exist and any read to a memory location must necessarily happen after the write to that location. So we can define the HB relation between instance $\boldsymbol{i}$ of step S and $\boldsymbol{i'}$ of step T as:

$$\mathcal{HB}^{S\to T}(\boldsymbol{i},\boldsymbol{i'})\ \equiv\ \mathcal{A}^{S_{W_l}}(\boldsymbol{i}) = \mathcal{A}^{T_{R_m}}(\boldsymbol{i'})\ \wedge\ (S \neq T \vee \boldsymbol{i} \neq \boldsymbol{i'})$$

This simply captures the ordering constraints of the DFGL model: step instance $\boldsymbol{i'}$ reading an item cannot start before step instance $\boldsymbol{i}$ writing that item completed, even if step instance $\boldsymbol{i'}$ of T appears lexically before instance $\boldsymbol{i}$ of step

S in the DFGL program. According to the definition in Section 2.2, dependence polyhedra between steps S and T are simply expressed as:

$$\mathcal{D}^{S \to T} \equiv \{(\boldsymbol{i}, \boldsymbol{i'}) \mid \boldsymbol{i} \in \mathcal{D}^S \wedge \boldsymbol{i'} \in \mathcal{D}^T \wedge \mathcal{HB}^{S \to T}(\boldsymbol{i}, \boldsymbol{i'})\}$$

which captures that $\boldsymbol{i}/\boldsymbol{i'}$ is an instance of step S/T, $\boldsymbol{i}$ writes the item read by $\boldsymbol{i'}$ (access equation), and $\boldsymbol{i}$ happens before $\boldsymbol{i'}$ (HB relation). Because of the dynamic single assignment rule, the DFGL model disallows Write-After-Write dependence and Write-After-Read dependences. The next section outlines how polyhedral analysis can be used to check of these error cases.

5.2 Legality analysis

This section introduces the compile-time analyses to verify the legality of a DFGL program. Enforcing the DFGL semantics, it detects the violation of the dynamic-single-assignment rule, plus three types of deadlock scenarios.

Violation of the single-assignment rule is equivalent to the existence of Write-After-Write dependences, and is represented by the following condition, which indicates that instances $\boldsymbol{i}$ and $\boldsymbol{i'}$ write an identical item (data element):

$$\exists \boldsymbol{i} \in \mathcal{D}^S, \ \exists \boldsymbol{i'} \in \mathcal{D}^T \ : \ \mathcal{A}^{S W_l}(\boldsymbol{i}) = \mathcal{A}^{T W_m}(\boldsymbol{i'}) \ \wedge \ (S \neq T \vee \boldsymbol{i} \neq \boldsymbol{i'} \wedge l \neq m)$$

Self deadlock cycle is the simplest case of deadlock. An instance $\boldsymbol{i}$ needs to read an item which is written by $\boldsymbol{i}$ itself, thereby resulting in indefinite blocking.

$$\exists \boldsymbol{i} \in \mathcal{D}^S \ : \ \mathcal{A}^{S W_l}(\boldsymbol{i}) = \mathcal{A}^{S R_m}(\boldsymbol{i})$$

General deadlock cycle is the second of deadlock scenarios, where the dependence chain among multiple step instances creates a cycle. Any instance on the cycle waits for its predecessor to complete and transitively depends on itself. As discussed in Section 5.3, transformations in the polyhedral model are equivalent to a multidimensional affine schedule such that, for each pair of instances in dependence, the producer is scheduled before the consumer. The existence of such legal schedule [18] guarantees the absence of general deadlock cycle, and optimizers are built to produce only legal schedules.

Deadlock due to absence of producer instance is the third deadlock scenario. Even without a cycle in the dependence chain, it can be possible that a step instance $\boldsymbol{i'}$ needs to read an item that any other step instance does not write. Detecting this scenario is represented by the following condition, which means there is no step instance $\boldsymbol{i}$ that writes an item to be read by $\boldsymbol{i'}$. Note that the items written/read by the environment `env` are also expressed as domains and access relations (Section 4.1)[3].

$$\exists \boldsymbol{i'} \in \mathcal{D}^T \ : \ \neg \left(\exists \boldsymbol{i} \in \mathcal{D}^S \ : \ \mathcal{A}^{S W_l}(\boldsymbol{i}) = \mathcal{A}^{T R_m}(\boldsymbol{i'})\right)$$

For instance, the following compile-time error message is shown if we remove the second line "`(corner:i,j) -> [A:i,j];`" in Figure 2:

```
Legality check: Deadlock due to no producer of (main_center:1,1)
```

[3] In future work, we may consider the possibility of not treating this case as an error condition by assuming that each data item that is not performed in the DFGL region has a initializing write that is instead performed by the environment.

5.3 Transformations

Given a set of dependence polyhedra $\{\mathcal{D}^{*\rightarrow *}\}$ that captures all program dependences, the constraints on valid schedules are:

$$\Theta^S(\boldsymbol{i}) \prec \Theta^T(\boldsymbol{i'}), \quad (\boldsymbol{i}, \boldsymbol{i'}) \in \mathcal{D}^{S\rightarrow T}, \quad \mathcal{D}^{S\rightarrow T} \in \{\mathcal{D}^{*\rightarrow *}\}$$

For any dependence source instance $\boldsymbol{i}$ of step S and target instance $\boldsymbol{i'}$ of step T, $\boldsymbol{i}$ is given a lexicographically smaller time-stamp than $\boldsymbol{i'}$. Because of the translation of the DFGL program into a complete polyhedral description, off-the-shelf polyhedral optimizers can be used to generate imperative code (i.e., C code) performing the same computation as described in the DFGL program. This optimization phase selects a valid schedule for each step based on performance heuristics — maximizing objective functions. There have been a variety of polyhedral optimizers in past work with different strategies and objective functions e.g., [11,33]. The schedule is then implemented to scan the iteration domains in the specified order, and a syntactic loop-based code structure is produced using polyhedral code generation [8].

We used the PolyAST [33] framework to perform loop optimizations, where the dependence information provided by the proposed approach is passed as input. PolyAST employs a hybrid approach of polyhedral and AST-based compilations; it detects reduction and doacross parallelism [17] in addition to regular doall parallelism. In the code generation stage, doacross parallelism can be efficiently expressed using the proposed doacross pragmas in OpenMP 4.1 [28,34]. These pragmas allow for fine-grained synchronization in multidimensional loop nests, using an efficient synchronization library [38].

6 Experimental Results

This section reports the performance results of the proposed DFGL optimization framework obtained on two platforms: (1) an IBM POWER7: node with four eight-core POWER7 chips running at 3.86GHz, and (2) an Intel Westmere: node with 12 processor cores per node (Intel Xeon X5660) running at 2.83 GHz. For benchmarks, we use Smith-Waterman, Cholesky Factorization, LULESH and six stencil kernels from PolyBench [25].

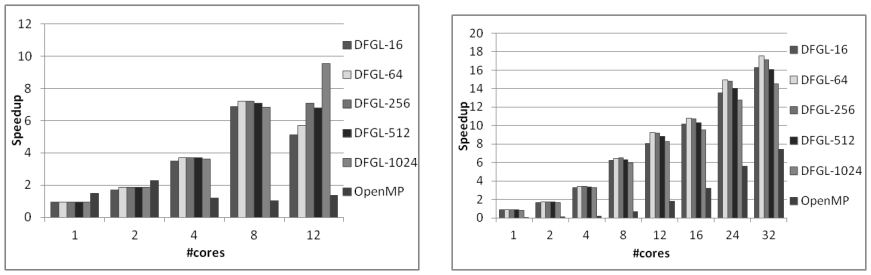
Smith-Waterman is used as our motivating example (Section 3). We run the alignment algorithm for 2 strings of size 100,000 each, with a tile size varying between 16 and 1024 in each dimension. As the baseline OpenMP implementation, we manually provided a wavefront doall version via loop skewing. Figure 6 shows the speedup results on our two test platforms, relative to the sequential implementation. We observe that the performance varies depending on the tile size chosen: for Westmere the best tile size is 1024, while for POWER7 the best tile size is 64. However our approach gives a big performance improvement compared with the skewed wavefront OpenMP implementation: up to 6.9× on Westmere and up to 2.3× on POWER7 for the maximum number of cores, due to cache locality enhancement via tiling and efficient doacross synchronizations.

To evaluate the efficiency of doacross (point-to-point synchronizations) and wavefront doall (barriers), we provided variants that removes all computations in the kerenel and only contains synchronizations. Table 1 shows the synchronization and overall execution times in second. When using 32 cores, the synchronization overheads for doacross with tile size = 64 and wavefront doall is

	OpenMP	DFGL-16	DFGL-64	DFGL-256	DFGL-512	DFGL-1024
Overall	9.863sec	4.508sec	4.188sec	4.283sec	4.571sec	5.047sec
Synch.	1.720sec	0.482sec	0.163sec	0.128sec	0.129sec	0.143sec

Table 1. Overall and synchronization time (Smith-Waterman onPower7 with 32 cores)

0.163[sec] and 1.72[sec], respectively. In addition to this synchronization efficiency, loop tiling by the optimization framework enhanced data locality; overall improvement over the OpenMP variant is 2.36× when using 32 cores and tile size = 64.



(a) Intel Westmere (b) IBM POWER7

Fig. 6. Smith-Waterman using 2 sequences of 100k elements each. Results are for DFGL optimized code with loop tiling using tile sizes between 16 and 1024, and OpenMP baseline with parallelism obtained via loop skewing.

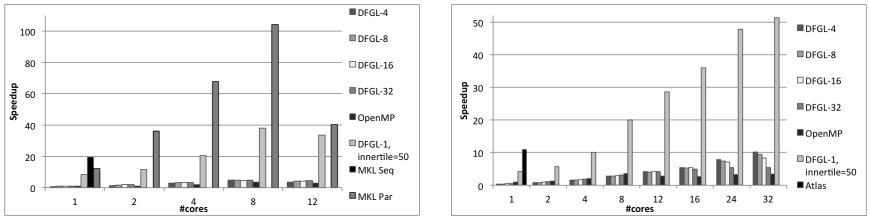
Cholesky Factorization is a linear algebra benchmark that decomposes a symmetric positive definite matrix into a lower triangular matrix and its transpose. The input matrix size is 2000 × 2000 and the generated code has 2D loop tiling with tile size varying between 4 and 32. In figure 7 that even though this approach does not yield a large speedup, it still gives improvement compared to the OpenMP implementation: 1.4× on Westmere and 3.0× on POWER7.

As reported in previous work [13], the combination of data tiling (layout transformation) and iteration tiling is a key technique for Cholesky Factorization while the current toolchain supports only iteration tiling. Alternatively, we manually implemented 50×50 iteration and data tiling within the user-provided steps and underlying data layout; the input DFGL is unchanged and our toolchain generated the same inter-step parallel code via doacross. This version brought significant improvements due to optimized cache locality, up to 15× on Westmere and up to 10.8× on POWER7 over standard OpenMP implementation. Furthermore, it gives on par performance with Parallel Intel MKL on 12 cores, on Westmere[4] and outperforms ATLAS on POWER7[5] on more than 4 cores.

[4] MKL is the best tuned library for Intel platforms. We compare against Sequential and Parallel MKL.

[5] On POWER7 we use ATLAS — the sequential library — as MKL cannot run on POWER7, and a parallel library was not available.

These results further motivate our work, since the application tuning can be accomplished both by the polyhedral transformations and the user by replacing the steps with optimized versions. For example, in the case of cholesky, it is possible to call optimized MKL/ATLAS kernels inside the user steps. In our results, these steps are regular sequential steps and all parallelism comes from the OpenMP code generated by the polyhedral tools. Further, since DFGL can be used as an embedded DSL, the OpenMP code being generated can be incorporated in larger applications and coupled with optimized user steps.



(a) Intel Westmere (b) IBM POWER7

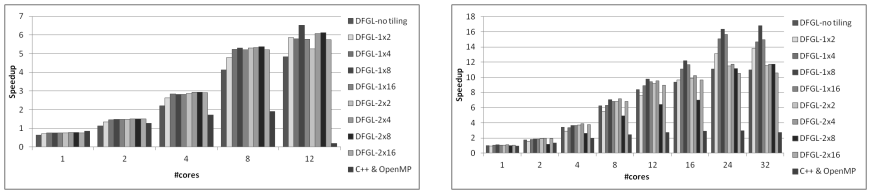
Fig. 7. Cholesky Factorization using 2000x2000matrix. Results are for loop tiling using tile sizes between 4 and 32, OpenMP parallelism, data tiling resulting of the inner steps and reference MKL/Atlas implementations.

LULESH is a benchmark needed for modeling hydrodynamics [1]. It approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. In this implementation each element is defined as a cube, while each node on the mesh is a point where mesh lines intersect and a corner to 4 neighboring cubes. The mesh is modeled as a 3D space with N^3 elements and $(N+1)^3$ nodes. The benchmark uses an iterative approach to converge to a stable state. We pre-tested the application and saw a convergence after 47 iterations; thus in our results we use a fixed number of 50 iterations for simplicity.

Figure 8 gives the results for a 100^3 space domain and our toolchain tiled both the time loop and the 3D loop nest corresponding to the space. We see that even with a time tile size of 2, this leaves only 25 parallel iterations at the outermost doacross loop, which for the POWER7 in particular leads to a smaller speedup. The best results are obtained with no time tiling and a space tile of 8^3, on both Westmere and POWER7. We also observe a significant increase in performance compared with the reference C++ implementation which uses OpenMP [22].

Finally, we summarize results for the stencil benchmarks from the Polybench suite [25]: *Jacobi-2D*, *Jacobi-1D*, *Seidel-2D*, *FDTD* (Finite Different Time Domain), *FDTD-APML* (FDTD using Anisotropic Perfectly Matched Layer) and *ADI* (Alternating Direction Implicit solver) in figures 9 when using the maximum number of cores on each platform. We created the baseline OpenMP implementations in a standard manner: parallelism added at the outer most loop for fully parallel loops and after skewing for loops with loop-carried dependences. We did not add manual polyhedral optimizations.

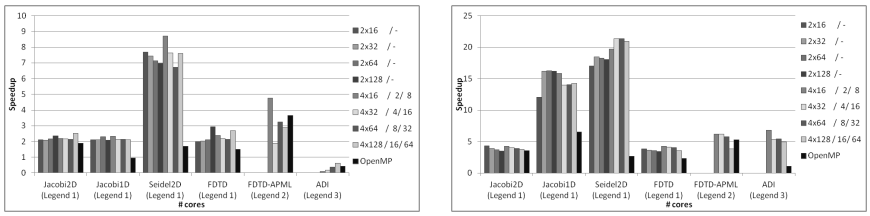
The results show that the best tile sizes vary between platforms: on the Westmere the best results are generally for the larger time tile (4) and the largest space tile size (128), while for the POWER7 the best results are for the



(a) Intel Westmere (b) IBM POWER7

Fig. 8. LULESH for 50 time iterations and a 100^3 space domain. Results are for time loop tiling with tiles 1,2 and space loop tiles 2,4,8,16, and reference C++ OpenMP implementation.

smaller time tile (2) and the smallest space tile (16). We also note that the results obtained using the DFGL toolchain outperform the OpenMP implementations for most cases, with up to 1.8× speedups.



(a) Intel Westmere, 12 cores (b) IBM POWER7, 32 cores

Fig. 9. Stencil benchmarks from the Polybench suite.Results compare DFGL tiling with standard OpenMP parallel versions.

7 Related Work

DFGL has its roots in Intel's Concurrent Collections (CnC) programming model [12,21], a macro-dataflow model which provides a separation of concerns between the high level problem specification and the low level implementation. The original CnC implementation did not offer a means for definiting dependences at a high level, and an extended CnC model proposed for mapping onto heterogeneous processors [32] became the foundation for DFGL.

Compared to past work related to CnC, DFGL pushes the use of a high-level data-flow model as an embedded DSL for enabling robust compiler optimizations using a state-of-the-art polyhedral compilation framework that is capable of generating code for the new OpenMP 4.1 doacross construct. In addition, to the best of our knowledge, this work is the first to use polyhedral analyses to detect potential deadlocks and violations of the dynamic single assignment rule in a dataflow graph program specification. Other data-flow models also use a parallel underlying runtime to achieve performance, either a threading library, such as pthreads used in TFlux [35], or a task library, such as TBB used in

Intel's CnC, or a parallel language such as Cilk used in Nabbit [6]. Legion [9] is another language which aims to increase programmability, however it requires an initial sequential specification of a program, similar to the input assumed by polyhedral compiler frameworks. DFGL eases programmability by separating the application description from its concrete implementation, and ensures that the optimized parallel code generated is not handled by the user. In addition, DFGL regions can be integrated in large scale applications as an embedded DSL, and can be coupled with optimized step code implementations or library calls.

Domain specific languages aim to give a high-level view of the applications and to ease programmability but are generally restricted to particular sets of problems, such as stencil computations [26] or graph processing problems [20]. In contrast, DFGL aims to combine the programmability benefits of DSLs with the optimizability of polyhedral regions, by using an approach that enables portable specifications of parallel kernels. Alpha [42] is a language which can be viewed as an eDSL for the polyhedral model. However the specification for Alpha is that of a full language, whereas DFGL can be composed with optimized step code defined in other languages, as long as these can be built together.

A number of papers addressed data-flow analysis of parallel programs using the polyhedral model, including extensions of array data-flow analysis to data-parallel and/or task-parallel programs [16,41]. These works concentrate on analysis whereas our main focus is on transformations of macro-dataflow programs. Kong et al. [23] applied polyhedral analysis and transformations for the OpenStream language, a representative dataflow task-parallel language with explicit intertask dependences and a lightweight runtime. PolyGlot [10] was the first end-to-end polyhedral optimization framework for pure dataflow model such as LabVIEW, which describes streaming parallelism via wires (edges) among source, sink, and computation nodes. On the other hand, our framework aims at optimizing macro-dataflow model, where asynchronous tasks are coordinated via input/output variables in data-driven manner.

8 Conclusions

In this paper, we proposed an optimization framework that uses as input the DFGL model, a dataflow graph representation that results in high performance generated by polyhedral tools while still allowing the programmer to write general (non-affine) code within computation steps. We outlined the language features of DFGL and presented our implementation of the model, which provides a tool that reads in the DFGL specification and generates the SCoP format for polyhedral transformations. We then described the technical details for computing dependences based on the access functions and domain, as described in the SCoP format, using the dynamic single assignment property of DFGL. Further we described compile-time analyses to verify the legality of DFGL programs by checking for potential dynamic single assignment violations and potential deadlocks. We have shown experimental results for our implementation of the DFGL model, which offers good scalability for complex graphs, and can outperform standard OpenMP alternatives by up to 6.9×. The current restrictions on DFGL are inherited from the polyhedral model itself and should be also addressed in future work [14]. This work focuses on the C language; future work could consider C++ notational variants.

Acknowledgments This work was supported in part by the National Science Foundation through awards 0926127 and 1321147.

References

1. Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Tech. Rep. LLNL-TR-490254
2. The PACE compiler project. http://pace.rice.edu
3. The Swarm Framework. http://swarmframework.org/
4. Building an open community runtime (ocr) framework for exascale systems (Nov 2012), supercomputing 2012 Birds-of-a-feather session
5. Ackerman, W., Dennis, J.: VAL - A Value Oriented Algorithmic Language. Tech. Rep. TR-218, MIT Laboratory for Computer Science (June 1979)
6. Agrawal, K., et al.: Executing task graphs using work-stealing. In: IPDPS ('10)
7. Arvind, Dertouzos, M., Nikhil, R., Papadopoulos, G.: Project Dataflow: A parallel computing system based on the Monsoon architecture and the Id programming language. Tech. rep., MIT Lab for Computer Science (March 1988), computation Structures Group Memo 285
8. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT. pp. 7–16 (2004)
9. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC (2012)
10. Bhaskaracharya, S.G., Bondhugula, U.: Polyglot: a polyhedral loop transformation framework for a graphical dataflow language. In: Compiler Construction. pp. 123–143. Springer (2013)
11. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI (2008)
12. Budimlić, Z., Burke, M., Cavé, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D., Sarkar, V., Schlimbach, F., Taşirlar, S.: Concurrent Collections. Scientific Programming 18, 203–217 (August 2010)
13. Chandramowlishwaran, A., Knobe, K., Vuduc, R.: Performance evaluation of concurrent collections on high-performance multicore computing systems. In: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on. pp. 1 –12 (April 2010)
14. Chatarasi, P., Shirako, J., Sarkar, V.: Polyhedral optimizations of explicitly parallel programs. In: Proc. of PACT'15 (2015)
15. Chatterjee, S., Tasrlar, S., Budimlic, Z., Cave, V., Chabbi, M., Grossman, M., Sarkar, V., Yan, Y.: Integrating asynchronous task parallelism with mpi. In: IPDPS (2013)
16. Collard, J.F., Griebl, M.: Array Dataflow Analysis for Explicitly Parallel Programs. In: Proceedings of the Second International Euro-Par Conference on Parallel Processing. Euro-Par '96 (1996)
17. Cytron, R.: Doacross: Beyond Vectorization for Multiprocessors. In: ICPP'86. pp. 836–844 (1986)
18. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part II: multidimensional time. Intl. J. of Parallel Programming 21(6), 389–420 (Dec 1992)
19. Feautrier, P., Lengauer, C.: The Polyhedron Model. Encyclopedia of Parallel Programming (2011)
20. Hong, S., Salihoglu, S., Widom, J., Olukotun, K.: Simplifying scalable graph processing with a domain-specific language. In: CGO (2014)
21. IntelCorporation: Intel (R) Concurrent Collections for C/C++, http://softwarecommunity.intel.com/articles/eng/3862.htm

22. Karlin, I., et al.: Lulesh programming model and performance ports overview. Tech. Rep. LLNL-TR-608824 (December 2012)
23. Kong, M., Pop, A., Pouchet, L.N., Govindarajan, R., Cohen, A., Sadayappan, P.: Compiler/runtime framework for dynamic dataflow parallelization of tiled programs. ACM Transactions on Architecture and Code Optimization (TACO) 11(4), 61 (2015)
24. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (Jul 1978), http://doi.acm.org/10.1145/359545.359563
25. Louis-Noel Pouchet: The Polyhedral Benchmark Suite, http://polybench.sourceforge.net
26. Lu, Q., Bondhugula, U., Henretty, T., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P., Chen, Y., Lin, H., fook Ngai, T.: Data layout transformation for enhancing data locality on NUCA chip multiprocessors. In: PACT (2009)
27. McGraw, J.: SISAL - Streams and Iteration in a Single-Assignment Language - Version 1.0. Lawrence Livermore National Laboratory (July 1983)
28. OpenMP Technical Report 3 on OpenMP 4.0 enhancements. http://openmp.org/TR3.pdf
29. Sarkar, V., Harrod, W., Snavely, A.E.: Software Challenges in Extreme Scale Systems (January 2010), special Issue on Advanced Computing: The Roadmap to Exascale
30. Sarkar, V., Hennessy, J.: Partitioning Parallel Programs for Macro-Dataflow. ACM Conference on Lisp and Functional Programming pp. 202–211 (August 1986)
31. Sbirlea, A., Pouchet, L.N., Sarkar, V.: DFGR: an Intermediate Graph Representation for Macro-Dataflow Programs. In: Fourth International workshop on Data-Flow Modelsfor extreme scale computing (DFM'14) (Aug 2014)
32. Sbîrlea, A., Zou, Y., Budimlić, Z., Cong, J., Sarkar, V.: Mapping a data-flow programming model onto heterogeneous platforms. In: LCTES (2012)
33. Shirako, J., Pouchet, L.N., Sarkar, V.: Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '14 (2014)
34. Shirako, J., Unnikrishnan, P., Chatterjee, S., Li, K., Sarkar, V.: Expressing DOACROSS Loop Dependencies in OpenMP. In: 9th International Workshop on OpenMP (IWOMP) (2011)
35. Stavrou, K., Nikolaides, M., Pavlou, D., Arandi, S., Evripidou, P., Trancoso, P.: TFlux: A portable platform for data-driven multithreading on commodity multicore systems. In: ICPP (2008)
36. The STE——AR Group: HPX, a C++ runtime system for parallel and distributed applications of any scale., http://stellar.cct.lsu.edu/tag/hpx
37. UCLA, Rice, OSU, UCSB: Center for Domain-Specific Computing (CDSC), http://cdsc.ucla.edu
38. Unnikrishnan, P., Shirako, J., Barton, K., Chatterjee, S., Silvera, R., Sarkar, V.: A practical approach to DOACROSS parallelization. In: Euro-Par (2012)
39. Vrvilo, N.: Asynchronous Checkpoint/Restart for the Concurrent Collections Model. MS Thesis, Rice University (2014), https://habanero.rice.edu/vrvilo-ms
40. Wonnacott, D.G.: Constraint-based Array Dependence Analysis. Ph.D. thesis, College Park, MD, USA (1995), uMI Order No. GAX96-22167
41. Yuki, T., Feautrier, P., Rajopadhye, S., Saraswat, V.: Array Dataflow Analysis for Polyhedral X10 Programs. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '07 (2013)
42. Yuki, T., Gupta, G., Kim, D., Pathan, T., Rajopadhye, S.: AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In: LCPC (2012)