Transforming Loop Chains via Macro Dataflow Graphs

Eddie C. Davis
Computer Science
Boise State University
Boise, Idaho, USA
eddiedavis@boisestate.edu

Michelle Mills Strout
Computer Science
University of Arizona
Tucson, Arizona, USA
mstrout@cs.arizona.edu

Catherine Olschanowsky
Computer Science
Boise State University
Boise, Idaho, USA
catherineolschan@boisestate.edu

Abstract

This paper describes an approach to performance optimization using *modified macro dataflow graphs*, which contain nodes representing the loops and data involved in the stencil computation. The targeted applications include existing scientific applications that contain a series of stencil computations that share data, i.e. loop chains. The performance of stencil applications can be improved by modifying the execution schedules. However, modern architectures are increasingly constrained by the memory subsystem bandwidth. To fully realize the benefits of the schedule changes for improved locality, temporary storage allocation must also be minimized.

We present a macro dataflow graph variant that includes dataset nodes, a cost model that quantifies the memory interactions required by a given graph, a set of transformations that can be performed on the graphs such as fusion and tiling, and an approach for generating code to implement the transformed graph. We include a performance comparison with Halide and PolyMage implementations of the benchmark. Our fastest variant outperforms the auto-tuned variants produced by both frameworks.

CCS Concepts • **Computing methodologies** → **Parallel computing methodologies**; *Parallel algorithms*; • **Software and its engineering** → **Compilers**; **Macro languages**;

Keywords stencil, dataflow, loop chain, storage optimizations

ACM Reference Format:
Eddie C. Davis, Michelle Mills Strout, and Catherine Olschanowsky. 2018. Transforming Loop Chains via Macro Dataflow Graphs. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3168832

CGO'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168832

1 Introduction

Partial Differential Equation (PDE) solvers are central to many scientific applications, including computational fluid dynamics (CFD), climate patterns [15], and medical imaging [11]. Frameworks and libraries ease the development of large applications [1–3, 8] and a large investment has been made in the development of complex scientific applications based on this infrastructure.

Performance improvements are possible for this type of application, but difficult to realize. Time-intensive portions of the application are often implemented as a series of parallel loops that perform stencil pattern operations and share data. Although there are many opportunities for parallelism, execution time is often dominated by the time required to move large quantities of data. This leads to poor parallel scaling, because increasing the number of threads increases competition for shared portions of the memory hierarchy, resulting in a bottleneck.

Approaches to improve the performance of these applications include rewriting them using embedded domain-specific languages (DSLs) and writing transformation scripts. Some DSL approaches that automate loop grouping/fusion, tiling, and storage reduction are PolyMage [17, 29] and Halide [22]. Scriptable loop transformation tools can be used to annotate or script loop transformations on stencil computations and include the CHiLL scripting framework [6], PET [32], and DFGL [27].

In our previous work, we leverage the loop chain abstraction, a series of loops that share data, to optimize such loop manually with a polyhedral code generator [19] and with pragmas [4]. However, achieving these improvements involves complex iteration space reordering and storage mapping optimizations; optimizations for which determining what optimizations to apply is difficult. This paper presents a representation loop chain schedules and data mappings, a methodology minimizing temporary storage requirements for the computation, and a cost model for comparing schedules and storage mappings.

Our approach provides a visual interface to aid the performance expert in guiding polyhedral code transformations

paired with storage mapping optimizations. In this work we explore the concept of a modified macro dataflow graph (M^2DFG). Based on macro dataflow graphs, M^2DFGs express dataflow at a high-level using sets of statements, include information about the data being passed between nodes, and use layout to express the execution schedule.

The contributions of this paper include:

- A procedure to generate M^2DFGs given annotated source code.
- A set of scheduling and data transformations for M^2DFGs.
- A systematic approach to minimizing temporary storage requirements within graph nodes after fusion.
- An approach to reducing storage allocations for the entire M^2DFG using liveness analysis.
- A high-level cost model useful for comparing graphs and execution schedules.
- A comparison of two overlapped tiling approaches.

The approach was evaluated by creating an implementation of the MiniFluxDiv CFD application benchmark [19]. The baseline implementation was annotated and a M^2DFG was used to develop alternate execution schedules. We demonstrate that there is a significant performance increase and that our implementation is competitive with rewrites of the benchmark in both Halide and PolyMage.

The remainder of this paper is organized as follows. Section 2 describes the loop chain abstraction and its relation to modified macro dataflow graphs. In Section 3, the applied macro dataflow graph model is explained, including graph components, layout, and operations. Code generation from dataflow graphs is covered in Section 4. Experimental results are provided in Section 5, followed by the related work in Section 6. Finally, conclusions are drawn in Section 7.

2 Background

We use the MiniFluxDiv benchmark as an application exemplar to demonstrate our approach. This benchmark was chosen because it captures some of the complexity of full-scale simulation-based applications. MiniFluxDiv has been annotated using Loop Chain pragmas. This work depends on annotations to provide information required to achieve a separation of concerns among statements, schedule, and storage mappings.

2.1 Motivating Example: MiniFluxDiv

MiniFluxDiv is an application benchmark modeled after finite difference applications such as those written with the Chombo framework [1]. The benchmark focuses on the shared-memory portion of a single time step in an iterative solve. The input is a 3D, immutable data structure padded with a layer of ghost cells (2 deep). The domain is broken into a set of independent subdomains called boxes. Boxes are decomposed into cells; 16^3 cells is a typical box size, but larger box sizes are desirable to reduce the space required for ghost cells. We explore box sizes of 16^3 cells and 128^3 cells in this work. Each cell represents a vector of five components, including density (ρ), energy (e), and the velocity in each direction (u, v, w).

The original implementation is a series of parallel loops. There are three loops for each dimension of the problem. The first loop performs a partial flux. This calculation results in face values, meaning that when the partial flux is calculated in the x-direction, a value is required for each border between cells. The second loop completes the flux calculation using data from the corresponding velocity components in each direction to produce partial fluxes. These steps are referred to as `Fx1` and `Fx2`. The fluxes in the y- and z- directions are referred to as `Fy1`, `Fy`, `Fz1`, and `Fz2`, respectively. The third loop calculates the differences between flux values and saves a cell-centered value at each point.

Each of the operations is applied to all five components. A naive implementation results in a series of 45 parallel loop nests. The performance baseline is hand-optimized to reduce the number of loops.

2.2 Loop Chains

A loop chain is a series of loop nests that perform operations on shared data [13]. The loop chain abstraction captures this pattern promotes decoupling the execution schedule from the primary expression of the algorithm. The abstraction can be implemented in a variety of ways: domain specific languages, libraries, or code annotations. A loop chain pragma language has been developed and a restricted version of it is used in this paper [4].

The first column in Figure 1 demonstrates how the pragmas are added. The outermost pragma, `omplc parallel(fuse)`, indicates the start of a loop chain and the schedule that should be applied, i.e., fuse. Each loop nest within the chain is labeled with a pragma, indicating its domain. The pragma `domain(0:X+1,0:Y,0:Z)`, for example, indicates that iterator x has domain 0 through X+1 (inclusive). Data read and write patterns are specified in the pragma following the *with* clause.

A loop chain compiler has been implemented by Bertolacci et. al [4] that uses the pragma specifications to apply a variety of transformations to the original application code, including shifts/skews, fusion, tiling, and wavefront. In the existing tool, the data access patterns help the compiler to ensure the legality of transformations, but is not used to optimize data accesses or temporary storage.

3 Macro Dataflow Graphs

A modified macro dataflow graph (M^2DFG) is a visual representation of a computation highlighting data dependences. Traditional dataflow graphs represent data dependences at a fine-grained level, typically per statement. M^2DFGs differ from traditional dataflow graphs in three primary ways:

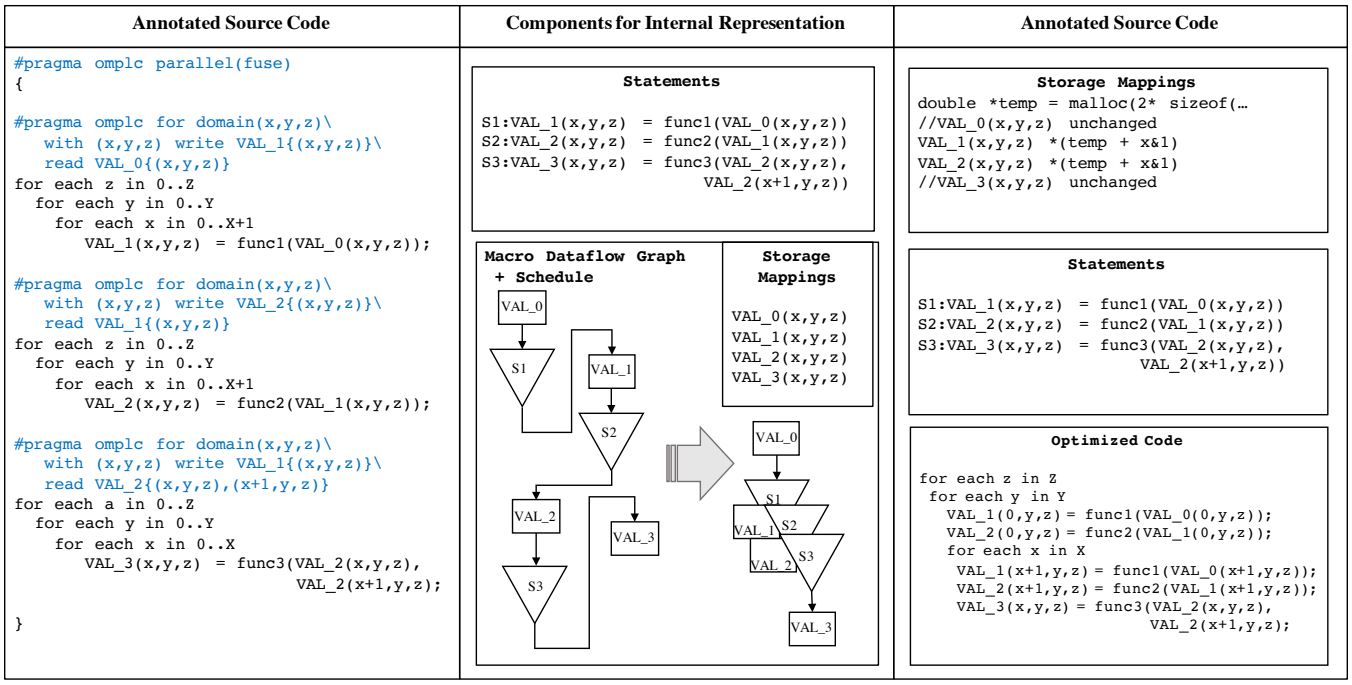


Figure 1. Overview of the three code generation phases using loop chain pragmas and the modified macro dataflow graph method and supporting cost model.

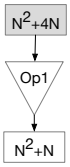


Figure 2. Summary of graph components, values, statements, and edges.

1. all iterations of a given loop nest are grouped into a single macro node,
2. data is explicitly represented as a node or nodes, and
3. the execution schedule is expressed.

This section describes the components of the modified M^{2}DFGs, the expression of execution schedule using graph layout, and a cost model to compare the potential performance of different graphs and layouts.

3.1 Graph Components

Each graph is represented by a tuple of three sets, $G = (V,S,E)$, where V is the set of *value* nodes, S is the set of *statement* nodes, and E is a set of directed edges that indicate the flow of data between nodes dictating a partial execution schedule. Each node v in V represents a value set in the program that will be mapped to memory. Each node s in S corresponds to a statement set in the code.

Value nodes are depicted as a rectangles. In Figure 2, for example, the node labeled $N^2 + 4N$ represents a set of values with that cardinality. There are two classes of value sets, persistent and temporary. Persistent value sets are accessed outside of the loop chain, and therefore, these value sets have a fixed storage mapping. Temporary values are allocated and accessed only within the loop chain. Persistent value nodes are shaded gray.

A statement node, depicted by an inverted triangle in the graphs, represents a statement set, one or more expressions applied to the value sets on incoming edges. The contents of statement sets are retrieved from loop bodies.

3.2 Execution Schedule

The edges of the graph indicate a partial execution schedule based on data dependences. The graph layout expresses the execution schedule. Graphs are executed from left to right, and top to bottom. Within the nodes the statements are executed over the domain in lexicographical ordering. An exception is made after fusion operations. In this case any shifting will be automatically applied to ensure legal execution.

The original `MiniFluxDiv` schedule over a 2D domain is represented by the M^{2}DFG provided in Figure 3. The graph is organized into four columns, one for each component in the 2D space. The persistent value nodes in the top row labeled ρ_0, u_0, etc. represent the initial input data for each box. Similarly, the persistent value nodes along the bottom,

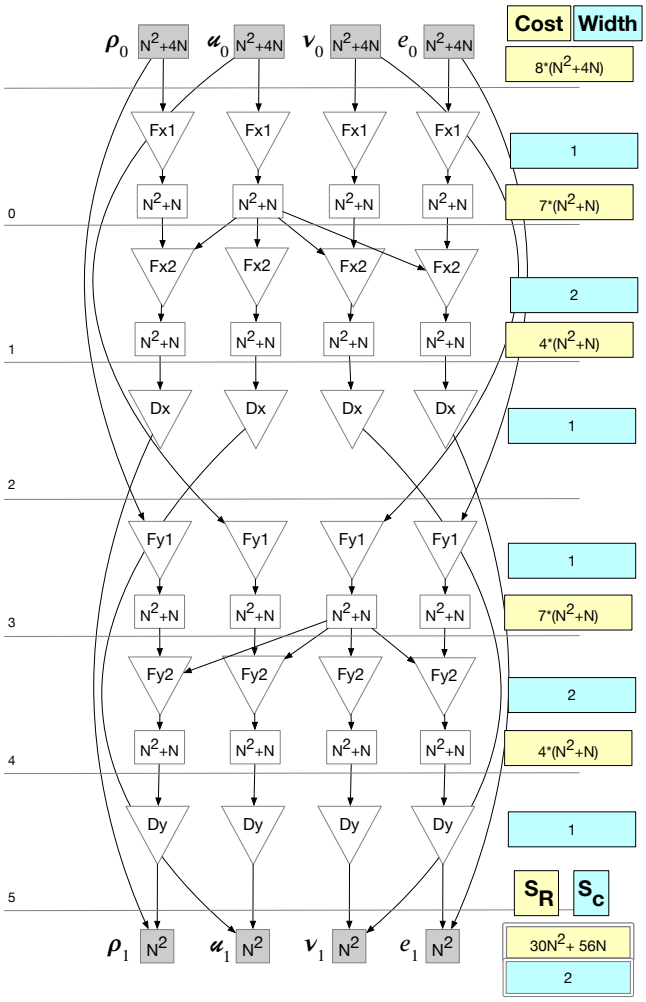


Figure 3. A graph representation of the series of loops implementation of the `MiniFluxdiv` benchmark. This schedule uses static single assignment for all values produced within the represented computation.

e.g., ρ_1, u_1, etc., contain the resulting output data. The input nodes are of size $N^2 + 4N$, and the output nodes N^2, the difference is due to ghost cells.

The first face-centered flux loop is represented by the statement nodes labeled `Fx1`. Note that the velocity component of the `Fx1` statement node, u, is read by the `Fx2` statement nodes for all components. The same is true for `Fy1`(v). This dependence pattern is common in computational fluid dynamics, and is necessary to obtain realistic performance results.

3.3 Cost Model

A cost model is derived using the value nodes in M^{2}DFGs. Two primary metrics are calculated: the total amount of data read (S_R), and the maximum number of streams being accessed simultaneously (S_c).

The total amount of data read for each value set is the number of outgoing edges multiplied by the size of the value set. The total for the entire graph is the sum of those values. For example, in Figure 3 the total amount of data read in each row is summed in the yellow boxes at the right. The total is on the yellow box at the bottom right labeled S_R.

The maximum number of streams being accessed simultaneously (S_c) determines whether or not the prefetching capabilities of the target architecture have been exceeded. This metric is calculated by taking the maximum incoming degree among all of the statement sets. This definition for the maximum number of streams being simultaneously accessed can be improved in a case that there are wide multi-dimensional stencils in the statement node. This type of pattern needs to be detected and included as additional edges if the prefetch distance for the target machine is exceeded.

The number of simultaneously read data streams, or width, is given in the blue boxes. The total number of streams read in this case is $S_c = 2$. The graph operations described in the following section are intended to reduce S_R, and keep S_c below a threshold to avoid exceeding the capabilities of the prefetcher.

4 Graph Operations

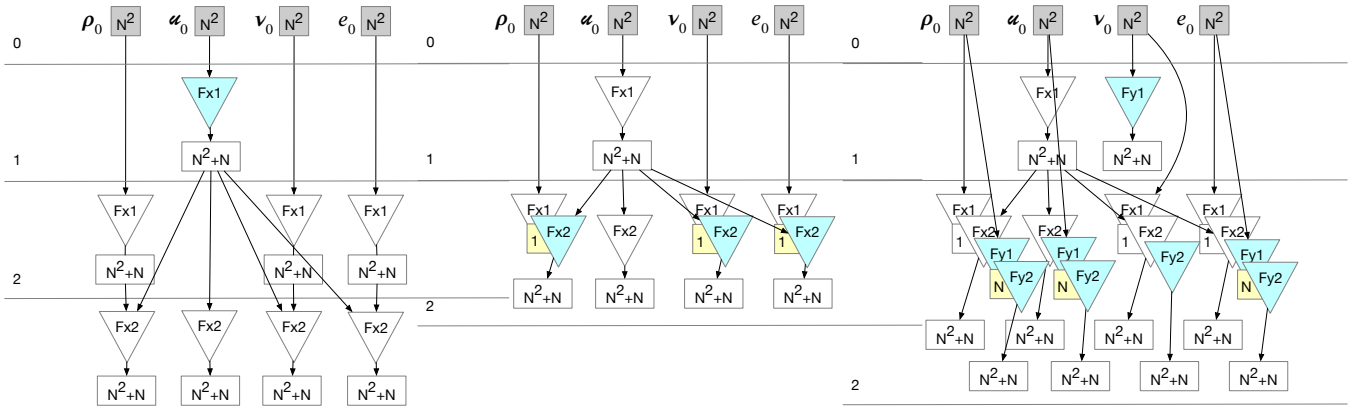
The graphs can be thought of as a visual representations of an ISCC [31] (built on the Integer Set Library [30]) script. Each operation described here is implemented as a relation in the ISCC script. The translation from graph to ISCC is in the process of being automated. Once the ISCC is written the resulting code is automatically generated.

There are three operations defined for the M^{2}DFGs, each corresponds to a transformation in the generated code. Figure 4 provides visualizations to describe the operations, as detailed in the following subsections. These include the `reschedule` operation, and two types of `fuse` operations, producer-consumer and read reduction.

Tiling transformations are considered separately from the reschedule and fuse operations. A tiling approach is defined and applied to the entire graph. Overlapped tiling as we implemented it is described in this section.

4.1 `reschedule` Operation

The `reschedule` operation moves a node from one row to another within the graph layout, effectively changing the execution schedule. For example, Figure 4(a) demonstrates relocating the velocity component (u) of the `Fx1` operation so that it will be executed before the other components. Rescheduling is provided as a convenience operation to enable subsequent optimizations, or to allow easier interpretation of the graph for code generation.



a) Reschedule: The cyan node has been rescheduled in preparation for rows 2 and 3 to be fused.

b) Producer-Consumer Fusion: The cyan nodes have been fused with the Fx1 nodes. This enables the yellow value sets' memory footprint to be reduced.

c) Read Reduction Fusion: The cyan nodes have been fused with the previously fused nodes. While the nodes are executed in a fused iteration space they each produce distinct value sets.

Figure 4. The set of operations that are defined for dataflow graphs.

4.2 fuse Operations

Fusing nodes in the graph directly corresponds to loop fusion. Producer-consumer fusion results in a single, more complex statement node. The benefit is a temporary data storage requirement reduction.

A read reduction fusion occurs when two statement nodes read data from the same value node. Each reader still produces its own value space, so there is no storage reduction. However, it provides an opportunity to reduce the number of times the same data are read.

The compiler transformations for the two fusion types are the same. However, the differences affect the cost model. The producer-consumer fusion of `Fx1` and `Fx2` is given in Figure 4(b). The subsequent read reduction fusion of the various operations is given in Figure 4(c). Fusion of statement nodes is indicated by the overlapping triangles.

4.3 Overlapped Tiling

The operations presented previously focus on the execution schedule among nodes of the graph. Global operations, like tiling, are applied to the graph as a whole. Tiling transformations divide a problem domain into smaller subdomains called tiles. In stencil-based applications, this leads to improved temporal locality and decreased data movement. This approach supports two types of overlapped tiling, and we provide a comparison.

In classical tiling, each iteration in the original space is executed by exactly one tile. This translates to each statement node in the graph being tiled separately. In overlapped tiling, an iteration can be executed in multiple tiles. This results in redundant computation overhead, but improved parallelism [20, 34].

Consider the two statement nodes, `Fx2` and `Dx`, as introduced in Figure 3. Each iteration of `Dx` reads two values produced by `Fx2`. This is illustrated in Figure 5(a).The arrows indicate dataflow. Classical tiling with a tile size of four results in three tiles, Figure 11(b). The dependences between tiles require a barrier to be placed after the `Fx2` statements finish execution and before `Dx` can begin.

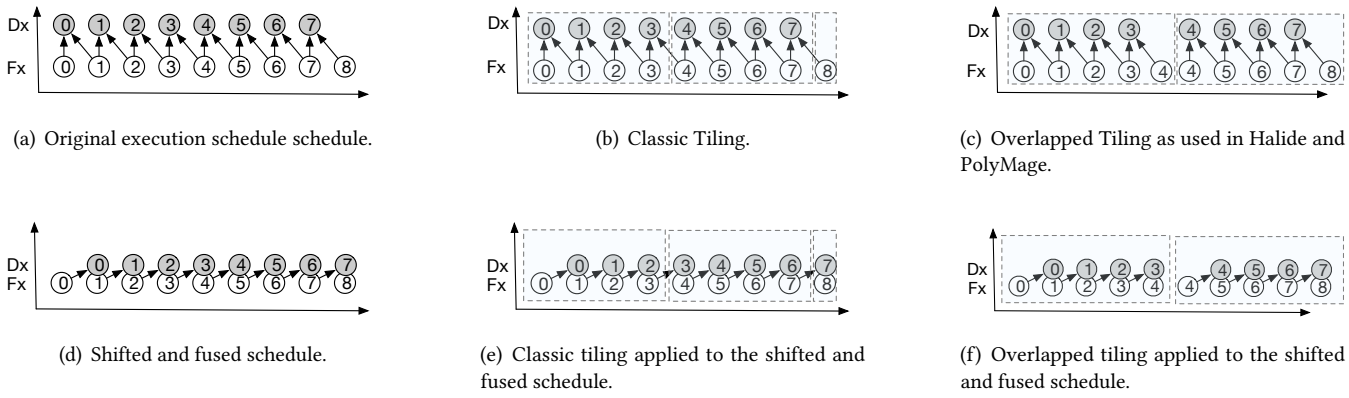
Overlapped tiling involves redundant computation within tiles to alleviate dependences. Figure 5(c) demonstrates overlapped tiling as it is applied in Halide [22], hierarchical overlapped tiling [34], and others. The tile size remains four, but is only applied to the final statement set (*Dx*). The previous statement sets in the execution schedule are expanded to satisfy dependences. In this case, the `Fx2` statement set is expanded by one in the positive direction for each tile, and the fourth iteration is executed by two tiles.

A second approach to overlapped tiling fuses producer/-consumer loop nests before tiling, Figure 11(d). In this example, the loop nest must be shifted for legal fusion. Classic tiling after fusion forces serial execution, Figure 11(f). To create overlapped tiles the domains of the previous statement sets also expanded. This approach is illustrated in Figure 5(f).

Each overlapped tiling approach has distinct advantages. The first preserves the parallelism available in the inner loop, and enables vectorization. The second reduces the temporary storage required per tile. In this case, the first approach requires space for as many iterations as are in the tile. In the second, only two scalars are required. The preferred approach depends on the application and the target hardware. According to the performance results demonstrated in Figure 11, sacrificing vectorization for reduced memory traffic is advantageous to this benchmark.

4.4 Mapping Data to Memory

Each value set representing temporary data expresses its space requirements in its label. A map is generated differently



(a) Original execution schedule schedule.

(b) Classic Tiling.

(c) Overlapped Tiling as used in Halide and PolyMage.

(d) Shifted and fused schedule.

(e) Classic tiling applied to the shifted and fused schedule.

(f) Overlapped tiling applied to the shifted and fused schedule.

Figure 5. Illustrations of the transformations that create the two overlapped tiling variants.

depending of if the value node is standalone, or if it has been pulled into a statement node through fusion. Standalone nodes use a one-to-one mapping between the iterator of the writing statement node and memory locations. Each of these maps are relative, meaning that the actual address to the space in memory is a parameter.

The map for a node subject to producer-consumer fusion is calculated from the data access patterns defined in the loop chain pragmas, along with the reuse distance in the transformed schedule. The distance is 1 in Figure 4(b), and only one value is read, therefore, the required space can reduced to a single scalar value. Fusing an operation with a stencil reading pattern will result in greater space requirements. For instance, fusing a `Dx` operation from Figure 3 produces a reuse distance of only 1, but two values need to be maintained. The data dependence for a stencil in the y-direction requires even more space to satisfy. Fusing a `Dy` node with a `Fy1` would require saving two values for each operation. The reuse distance is the domain length in the x-direction (N). In this case, the dependences can be satisfied with a buffer of size $2N$.

The address is provided by static liveness analysis applied to the graph as a whole. The liveness analysis proceeds by processing the graph in reverse execution order. A table is maintained with a list of spaces, the corresponding pointer ID, capacity, and a boolean indicating whether the location is active. During graph traversal a value node is assigned to an existing space that is of equal or greater capacity and marked as inactive. If no existing, inactive space can accommodate the node an existing smaller space is expanded. If no inactive spaces exist a new space is added to the table, the node is assigned to it, and the space is marked as active. When the node that writes to the value node is visited, the space is marked as inactive.

5 Experimental Results

This section details the experiments performed on the MiniFluxDiv benchmark and the larger AMR-Godunov application. M^{2}DFGs were used to guide a series of optimizations on MiniFluxDiv. Our performance measurements demonstrate scheduling optimizations are less effective without the corresponding reduction in temporary data, the overlapped tiling variant focusing on memory traffic reduction outperforms the vectorized version for this benchmark, and our performance is competitive with the performance achieve using Halide's and PolyMage's autotuning capabilities.

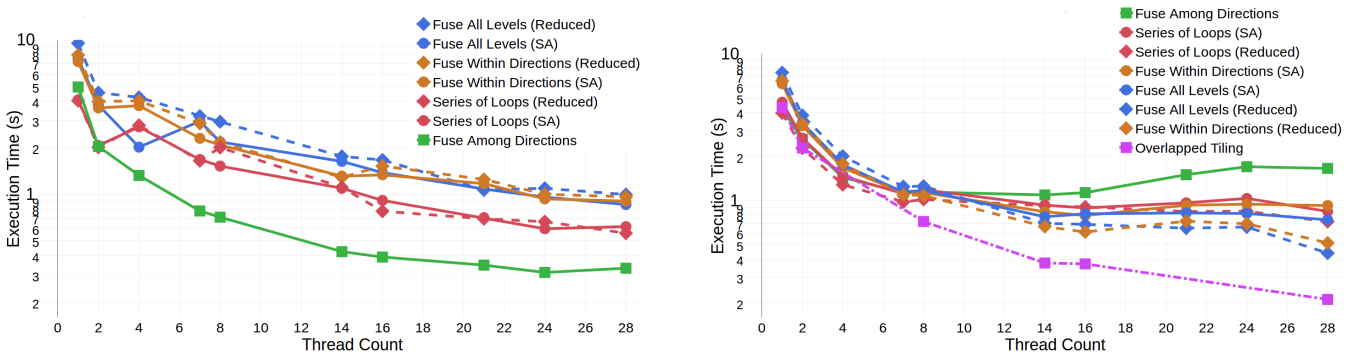
The performance of a larger example application, AMR-Godunov, was explored using M^{2}DFGs. The application was manually optimized and a performance improvement of 17% was observed.

5.1 Experimental Setup

The benchmark was optimized using several different schedules, each schedule was applied to a small box size of 16^3 and a large size of 128^3. The total number of cells per experimental run is 58,720,256 cell and the number of boxes is calculated accordingly (14,336 and 28, respectively). The scalability of each variant is explored by varying the thread count from 1 to 28, i.e., the number of cores on the target machine, with per thread parallelism over the boxes. Each experiment was run five times and the mean execution time is presented here.

All MiniFluxDiv experiments were conducted on the R2 cluster at Boise State University. Each node of R2 is a dual socket, Intel Xeon E5-2680 v4 CPU at 2.40 GHz clock frequency with 28 cores (14 per socket). The cores include a 32KB L1, 256KB L2, and 35840K L3 caches. The system contains 192GB of RAM split over 2 NUMA domains. GCC g++ version 6.1.0 was used to compile all the benchmarks, with optimization level `-O3` used by the compiler.

The experiments for AMR-Godunov were performed on Atlantis at Colorado State University. Atlantis is a 20-core



(a) Results with box size of 16 cells.

(b) Results with box size of 128 cells.

Figure 6. Performance of the `MiniFluxdiv` benchmark on 28-Core Intel Xeon E5-2680 CPU for both small boxes (16^3) and large (128^3). The y-axis is in log scale.

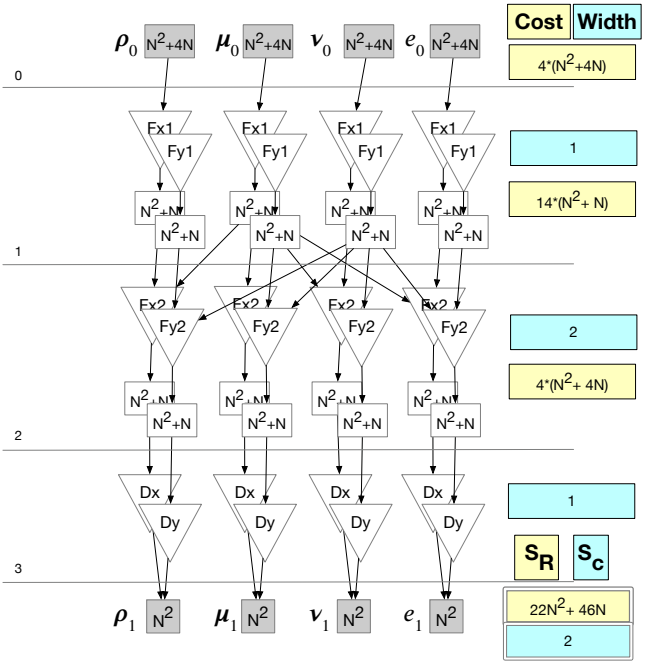


Figure 7. Graph for fuse among directions variant (green line in Figure 6).

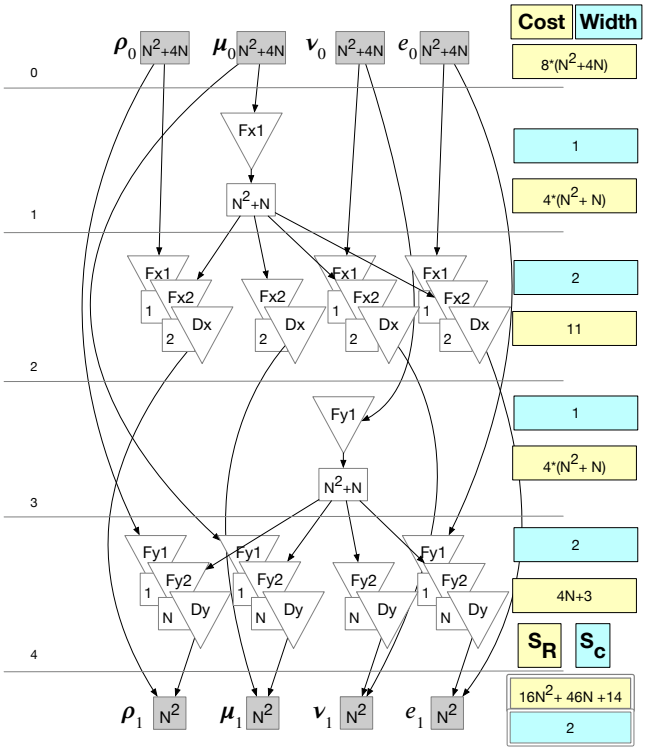


Figure 8. Graph for fuse within directions variant (orange lines in Figure 6).

machine composed of two 10-core Intel Ivy Bridge E5-2670v2 chips running at a clock rate of 2.50 GHz. The system is configured with 128 GB of DDR3 RAM in a quad-channel configuration with a clock rate of 1600 MHz, giving 51.2 GB/s of bandwidth per socket or an aggregate system bandwidth of 102.4 GB/s. Each core has a 32 KB of level 1 instruction cache, 32 KB of level 1 data cache, and 256 KB level 2 cache. All cores on a socket share 25 MB of level 3 cache.

5.2 Benchmark Variants

Experiments were conducted using five variants of a 3D implementation of the benchmark. Four of the variants did not use tiling: 1) series of loops, 2) fuse among directions, 3) fuse all levels, and 4) fuse within directions. Series of loops is the baseline variant. This is the original implementation and is used as the performance baseline. Variants 2-4 were created using M^{2}DFGs. An overlapped tiling variant was implemented using schedule 3 (fuse all levels) as the execution schedule within the tiles. Two versions of the first four variants were created. A single assignment (SA) version with

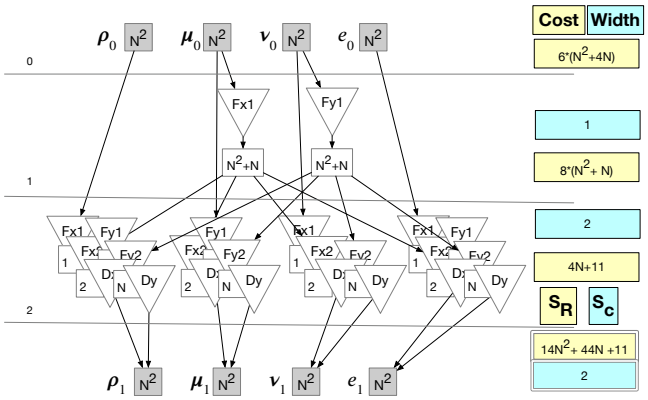


Figure 9. Graph for fuse all levels variant(blue lines in Figure 6).

no storage optimizations and, when possible, a version with storage optimizations (reduced).

The diagrams in this paper present a 2D version of Mini-FluxDiv. This was done to save space. In the diagrams there are four components and a series of 24 loop nests. All experimentation is done with the full 3D version. The 3D version has five components and a series of 36 loop nests.

Series of loops. The baseline implementation is series of loops with storage optimizations. This is the original implementation of the benchmark mirroring the implementation in the Chombo framework. Figure 3 displays the original schedule. This schedule performs well for small box sizes. Figure 6 shows this variant in red. The solid line is without temporary storage optimizations and the dashed is with. The parallelization is straight-forward and can be done within boxes or over boxes, using OPENMP parallel for pragmas on each loop nest or on the outer loop nest over boxes. On our target machine the parallelization over boxes performed better and is used in all results unless labeled otherwise.

Fuse among directions. This variant is shown in Figure 7. Read reduction fusion is performed on the Flux operations (Fx1,Fy1) and the fusion of the Diff (Dx,Dy) operations results in better locality for writing to the output buffers. Only the SA version was implemented, because there are no opportunities for storage reduction. Figure 6 displays this variant in green. This is the only schedule that improves on the baseline code for small boxes. However, the performance is poor for the large boxes.

Fuse all levels. This schedule is displayed in Figure 9. This schedule maximizes both producer-consumer and read reduction fusion. Both versions of this schedule perform well for large boxes, with the data reduced version being the most performant.

Fuse within Directions. The fuse within directions graph variant is given in Figure 8. This schedule maximizes the use of producer-consumer fusion. The Fx1 and Fy1 operations that are applied to velocity components cannot be included in the fusion. They are rescheduled before the fused row as to respect the data dependences. Fusing within directions is scalable, but does not outperform the series of loops for small boxes, or the fuse all rows schedule for large boxes.

Overlapped Tiling. Overlapped tiling was applied to the Fuse all levels schedule. The overlapped tiling variant performs the best for large box cae. This result is an improvement on our previous work. The improvement came from changing the intra-tile schedule. The use of M^2DFGs and code generation allowed for a larger set of intra-tile schedules to be attempted.

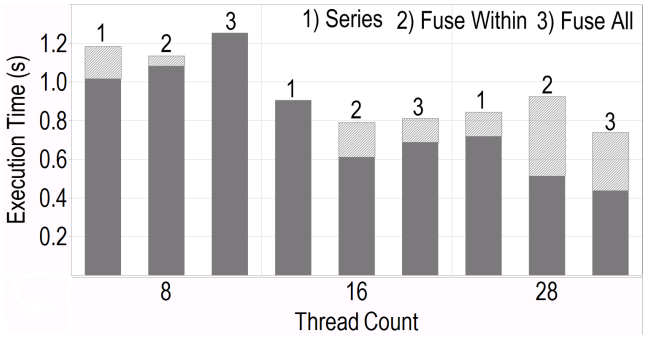


Figure 10. The execution time for schedules with storage mapping optimizations are significantly faster for most schedules. The original times are represented by the light gray bars, and the dark bars indicate the reduced times.

5.3 Temporary Storage Reductions

Figure 10 shows a subset of the schedules explored in this work. Each bar represents a variant with only scheduling changes, and the corresponding variant with both scheduling and data reduction optimizations. The benefits of the data reductions are most clearly seen for the large box sizes.

Figure 6 uses dashed and solid lines of the same color to display the impact of storage reduction. Each variant is show twice in the same color. The solid line represents the variant without temporary storage reductions and the dashed line with those reductions. The impact of the storage reductions is most clearly seen at high thread counts and with the large box sizes.

5.4 Overlapped Tiling Comparison

The data reduced variant of the fuse all levels schedule was selected to test the overlapped tiling implementation. This transformation produces the tiling as illustrated in Figure 5(f), or fusion within tiles. Tiling enables even further data reduction, as each thread only needs to allocate enough space for one tile. To compare with the overlapped tiling method given in Figure 5(c), the original series of loops schedule was tiled and then fused, referred to as fusion of tiles. The measurement results are compared with the baseline

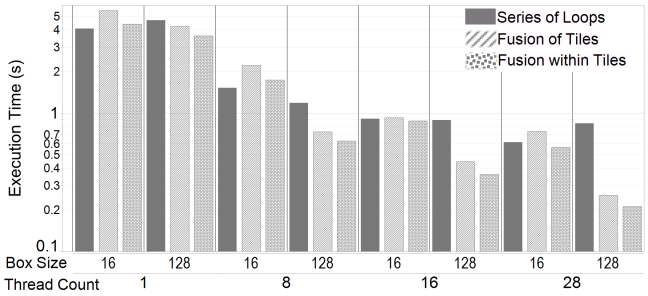


Figure 11. Overlapped tiling comparison of the two techniques applied to the `MiniFluxDiv` benchmark, including the original series of loops implementation as a reference. The x-axis is tiling method within box size and thread count. The y-axis is in log scale.

schedule and displayed in Figure 11. The fusion within tiles technique outperforms fusion over tiles for both small and large boxes on all four thread counts. It also outperforms the baseline schedule as the thread count increases.

5.5 Halide and PolyMage Comparisons

The `MiniFluxDiv` benchmark was implemented using the Halide [22] library and PolyMage [17]. The performance results show that the M^{2}DFG-guided schedules outperform the autotuned versions using Halide [16] and PolyMage (see Figure 12).

Results for the smaller box size (16^3) are omitted. The Halide and PolyMage implementations are limited to parallelization within the boxes. This limitation is not fundamental to the approach; it is implementation specific.

Our overlapped tiling variant outperforms both the Halide and PolyMage variants. The results for the large boxes are limited to within boxes. In this case, we applied both parallelization over and within boxes for a fair comparison, each of those variants outperform Halide and PolyMage.

The primary difference in the execution schedule is the fusion of iterations, complicating vectorization, but reducing temporary storage needs. Preserving straight-forward vectorization requires an increase in temporary storage use. Scaling out to 28 cores puts pressure on the memory subsystem and reducing that pressure takes precedence. This is not true for all applications, however, this is an important insight because MiniFluxDiv represents patterns commonly found in scientific applications.

5.6 Case Study: AMR-Godunov

AMR-Godunov [7] is an example AMR application published with the Chombo [1] software. It is an unsplit, second-order Godunov method. The application is written in a combination of C++ and Fortran. Each of the primary computational kernels is written in Fortran. M^{2}DFGs were used to organize a set of optimizations. The optimizations were applied by hand in the Fortran code. The final schedule reduced the size of the temporary space required by approximately 14KB. The overall execution time was reduced by 17%.

Figure 13 shows the M^{2}DFG for a subroutine that consumes approximately 80% of the execution time at each time step. Each time step involves communicating ghost cells and then processing each box independently. Optimizations were applied only within this subroutine. The problem domain is decomposed into independent subdomains called boxes. Each box contains a set of five component values for each 3D cell. In this example, the boxes are held at size 16^3.

The process for optimizing the code started at the bottom of the graph. Each of the *qlu* (quasi-linear-update) nodes were executed in pairs and were fused. This created a simple producer-consumer pair between the fused *qlu* nodes and the following Riemann solve nodes which were subsequently fused. Fusion was accomplished by creating a new fusion-specific Fortran kernel. Each fusion was coded seperatley as the stencil dependences required shifting that was slightly different for each case. Figure 14 shows the final graph.

6 Related Work

This work compliments other efforts to optimize legacy applications as well as work using domain specific libraries. It builds upon previous work done with dataflow and macro dataflow graphs. The overlapped tiling schedule minimizes temporary storage requirements at the expensive of vectorization and our results with MiniFluxDiv demonstrate that this is beneficial for some applications.

6.1 Legacy Application Optimization.

Application annotations and optimizations plans have been explored in previous work. These include Orio [18] and related projects that include more complex optimization scripts such as POET [33], CHiLL [10], and URUK [9]. This work is complementary to those efforts as it will help guide the decision process.

6.2 Domain Specific Libraries/Languages.

Several project have proposed libraries and languages to be used when implementing applications. The design encourages the separation of execution schedules and storage mapping from the primary expression of the algorithm. Our work targets existing applications.

Firedrake [24] identifies four different expert roles in the development of a scientific application and provides unique interfaces for each. This reduces the breadth of expert knowledge required and results in application code that is more maintainable and easily ported between compute resources. The scheduling work presented here is applicable to the parallel programming expert interface in Firedrake.

The projects most closely related to loop chaining in terms of scheduling across loops are Halide [22] and PolyMage [17].

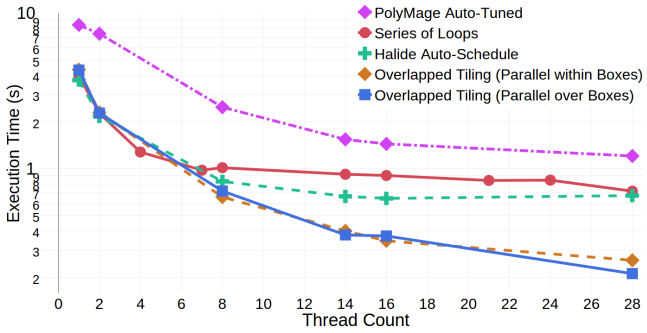


Figure 12. Series of loops, Overlapped tiling, Halide, and PolyMage with box size of 128 cells. The y-axis is in log scale.

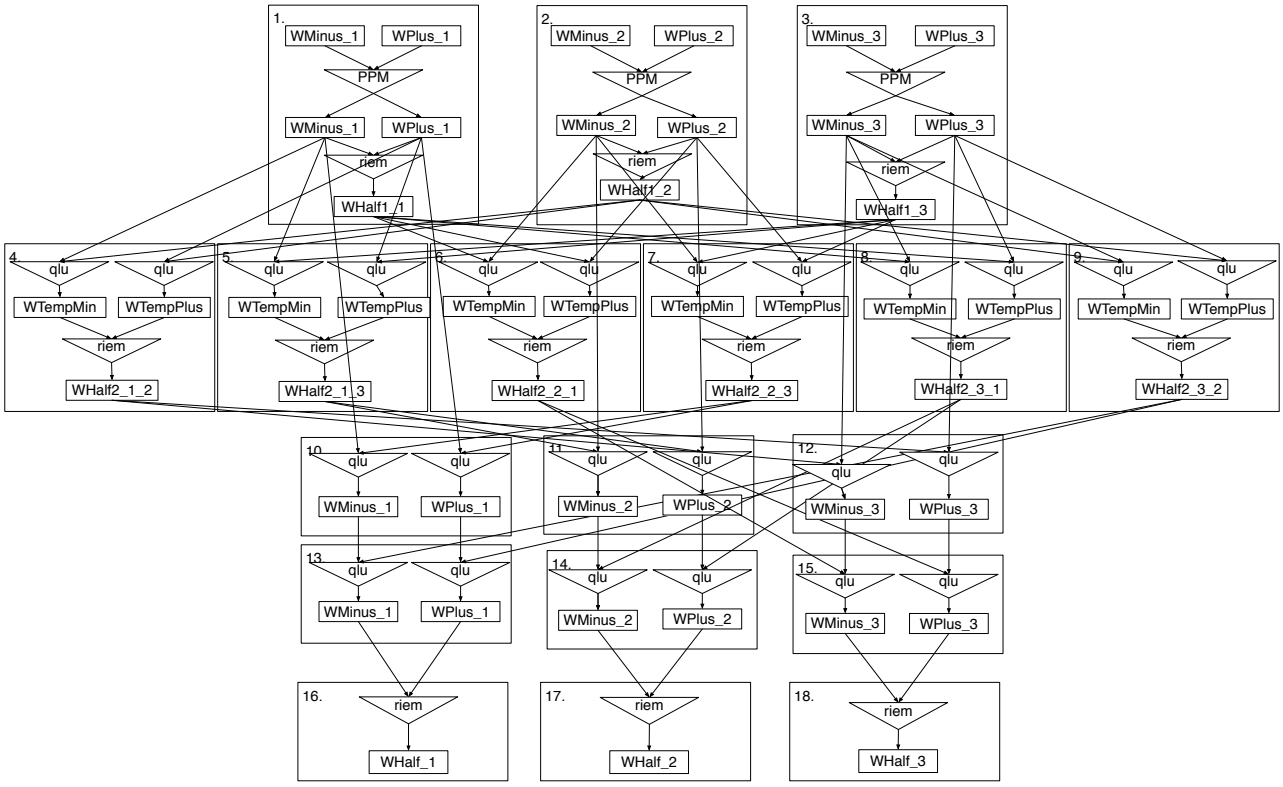


Figure 13. The M^{2}DFG for the original implementation of ComputeWHalf. ComputeWHalf is a subroutine that is part of a single timestep of AMR-Godunov.

Each of these projects exposes a functional programming interface to the scientific programmer and a separate scheduling interface for the performance expert. The overlapped tiling approach described in Figure 5 differs from the overlapped tiling variants introduced by Halide and Polymage. Additionally, we aim to accommodate legacy applications rather than requiring them to be be rewritten in a functional domain specific language.

6.3 Dataflow Representations

Original dataflow graphs were directed acyclic graphs with nodes at the iteration granularity. The idea of macro dataflow graphs was introduced to increase the granularity and group iterations into a single node [25]. Functional representations of the application are easily translated into macro dataflow graphs. This representation can be exploited to find parallelism and inform the associated code generation [23].

Prasanna et al. [28] take a hierarchical approach. Each macro node is scheduled for parallel execution on a node, unlike the previous work that assume serial execution. The entire graph is then partitioned and scheduled for distributed memory execution.

More recently, the Halide and PolyMage projects build DAGs internally based on functional representations. Both

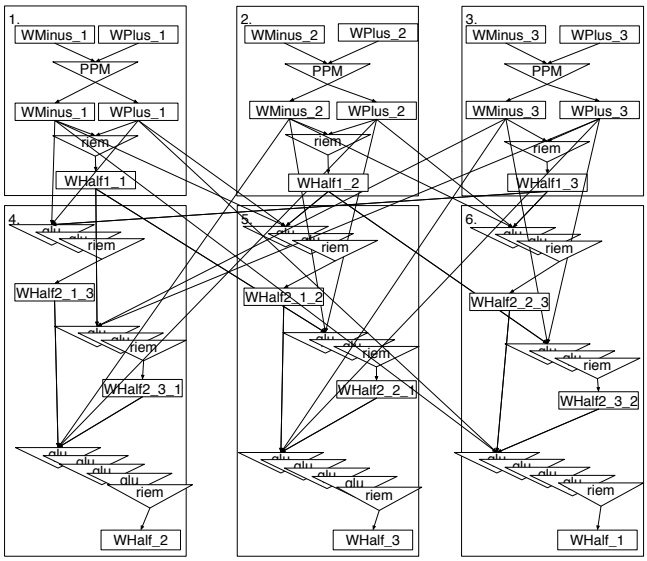


Figure 14. The M^{2}DFG for ComputeWHalf after optimization. The coding for the optimizations was performed by hand and was guided by manipulation of the M^{2}DFG.

projects target image processing pipelines. PolyMage utilizes polyhedral code generation, as do we, and Halide uses interval analysis. We have included a comparison of the performance results with both of these tools. The primary difference is that our approach targets legacy applications.

DFGR [26] allows developers to express programs at a high level with dataflow graphs as an intermediate representation. DFGR graphs are composed of *step* nodes that represent computation and *item* nodes that represent data. Steps and items partitioned into collections by unique *tag* identifiers.

The Data-Flow Graph Language (DFGL) [27] is an optimization framework based on DFGR that allows graph based dependences to be represented in the polyhedral model. Program graphs are compiled into Habanero-C, a variant of C that is built on the Intel Concurrent Collections (CnC) [5], that uses thread building blocks (TBB). As with Loop Chaining or PET [32], static control parts (SCoP) of programs are specified via pragmas (`dfgl`), an embedded domain specific language (eDSL). The DFGL code generator uses the ROSE compiler [14] to generate OpenMP 4 compatible code, including task level parallelism.

TIDeFlow [21] is an execution model specifying the precedence of computations without concern for scheduling or synchronization. It differs from other dataflow models by introducing the use of transitions and places from Petri nets to determine node weights.

Communication avoiding optimizations are closely related to this work in that they share similar goals. The most closely related to our work is the work performed by Demmel et, al. [12].

To the best of our knowledge, this fusion of producer-consumer dataflow graph nodes combined with temporary storage reduction using a reuse distance calculation is a novel contribution. Nodes within the graphs in previous work [23] can be coalesced, combining lightweight nodes into fewer heavyweight nodes, thereby reducing communication. The work presented here differentiates between coalescing and fusing nodes. The difference is that the iterations within the nodes are co-scheduled to reduce the temporary storage overhead.

Our approach is unique in that it includes fine-grained information about memory interactions, while the graph itself remains coarse-grained. Cost models have been used to compare the anticipated performance of macro dataflow graphs consistently since their inception. Given that the goal of many of these graphs is to identify parallelism opportunities, most of the cost models focus on execution cost of computation nodes [25], and the communication costs associated with the adjacent edges. These concepts are very much related, however, we are measuring interaction with memory rather than an interconnect.

7 Conclusion

In this paper, we present modified macro dataflow graphs (M^{2}DFGs) to visually represent series of stencil computations (i.e., loop chains) that often occur in scientific applications. Transformations on M^{2}DFGs correlate to schedule changes including overlapped tiling. An algorithm for determining what temporary storage reductions can be done is provided. A cost model to compare schedules based on memory traffic is presented, and results show it enables comparing relative performance between variants. Experimental results on a Computational Fluid Dynamics (CFD) benchmark and another application show that performance obtained by some of the schedule variants can outperform the state of the art embedded DSLs Halide and PolyMage.

Acknowledgments

The authors would like to thank Dr. Keshav Pingali and anonymous reviewers for their invaluable feedback. We would like to thank Dr. Uday R. Bondhugula for his technical assistance with Halide and PolyMage. We would like to acknowledge high-performance computing support of the R2 compute cluster, DOI: 10.18122/B2S41H, provided by Boise State University's Research Computing Department.

This material is based upon work supported by the National Science Foundation under Grant No. 1422725 and by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program under Award Number DE-SC-04030.

References

[1] M. Adams, P. Colella, D. T. Graves, J. N. Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen. 2014. *Chombo Software Package for AMR Applications - Design Document.* Technical Report LBNL-6616E. Lawrence Berkeley National Laboratory.

[2] Ann Almgren. 2017. AMReX. https://github.com/AMReX-Codes/AMReX-Codes.github.io. (July 2017).

[3] Satish Balay, Shrirang Abhyankar, M Adams, Peter Brune, Kris Buschelman, L Dalcin, W Gropp, Barry Smith, D Karpeyev, Dinesh Kaushik, et al. 2016. *Petsc users manual revision 3.7.* Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).

[4] I. J. Bertolacci, M. M. Strout, S. Guzik, J. Riley, and C. Olschanowsky. 2016. Identifying and Scheduling Loop Chains Using Directives. In *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*. IEEE Press, 3 Park Ave, New York, NY, USA, 57–67. https://doi.org/10.1109/WACCPD.2016.010

[5] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. 2010. Concurrent collections. *Scientific Programming* 18, 3-4 (2010), 203–217.

[6] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations.* Technical Report. Technical Report 08-897, U. of Southern California.

[7] P. Colella, D. T. Graves, T. J. Ligocki, D. F. Martin, and B. Van Straalen. 2008. *AMR Godunov Unsplit Algorithm and Implementation.* Technical Report. Lawrence Berkeley National Laboratory.

[8] W. Crutchfield and M. Welcome. 1993. Object-Oriented Implementation of Adaptive Mesh Refinement Algorithms. *Scientific Programming* 2 (1993), 145–156.

[9] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming* 34, 3 (2006), 261–317.

[10] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. 2010. Loop Transformation Recipes for Code Generation and Auto-Tuning. In *Languages and Compilers for Parallel Computing*, Vol. 5898. Springer Berlin Heidelberg, Springer Publishing, Salmon Tower Building, New York, NY, USA, 50–64.

[11] Ramgopal Kashyap and Pratima Gautam. 2016. Fast Level Set Method for Segmentation of Medical Images. In *Proceedings of the International Conference on Informatics and Analytics*. ACM, ACM, 2 Penn Plaza, Ste 701, New York, NY, USA, 20.

[12] Nick Knight. 2015. *Communication-Optimal Loop Nests.* Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-185.html

[13] Christopher D. Krieger, Michelle Mills Strout, Catherine Olschanowsky, Andrew Stone, Stephen Guzik, Xinfeng Gao, Carlo Bertolli, Paul H.J. Kelly, Gihan Mudalige, Brian Van Straalen, and Sam Williams. 2013. Loop Chaining: A Programming Abstraction For Balancing Locality and Parallelism. In *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*. IEEE, IEEE Press, 3 Park Ave, New York, NY, USA, 375–384.

[14] Chunhua Liao, Daniel J Quinlan, Thomas Panas, and Bronis R De Supinski. 2010. A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. *IWOMP* 6132 (2010), 15–28.

[15] Kyle T Mandli, Aron J Ahmadia, Marsha Berger, Donna Calhoun, David L George, Yiannis Hadjimichael, David I Ketcheson, Grady I Lemoine, and Randall J LeVeque. 2016. Clawpack: building an open source ecosystem for solving hyperbolic PDEs. *PeerJ Computer Science* 2 (2016), e68.

[16] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 83.

[17] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. Polymage: Automatic optimization for image processing pipelines. In *ACM SIGPLAN Notices*, Vol. 50. ACM, ACM, 2 Penn Plaza, Ste 701, New York, NY, USA, 429–443.

[18] Boyana Norris, Albert Hartono, and William Gropp. 2007. Annotations for Productivity and Performance Portability. In *Petascale Computing: Algorithms and Applications*. Chapman & Hall / CRC Press, Taylor and Francis Group, 3848 FAU Blvd, Boca Raton, FL, USA, 443–462. http://www.mcs.anl.gov/uploads/cels/papers/P1392.pdf Preprint ANL/MCS-P1392-0107.

[19] Catherine Olschanowsky, Michelle Mills Strout, Stephen Guzik, John Loffeld, and Jeffrey Hittinger. 2014. A Study on Balancing Parallelism, Data Locality, and Recomputation in Existing PDE Solvers. In *In The IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Press, IEEE Press, 3 Park Ave, New York, NY, USA, 793–804.

[20] Catherine Olschanowsky, Michelle Mills Strout, Stephen Guzik, John Loffeld, and Jeffrey Hittinger. 2014. A study on balancing parallelism, data locality, and recomputation in existing PDE solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, IEEE Press, 3 Park Ave, New York, NY, USA, 793–804.

[21] Daniel Orozco. 2011. Tideflow: A parallel execution model for high performance computing programs. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, IEEE Press, 3 Park Ave, New York, NY, USA, 211–211.

[22] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[23] S. Ramaswamy and P. Banerjee. 1993. Processor Allocation and Scheduling of Macro Dataflow Graphs on Distributed Memory Multicomputers by the PARADIGM Compiler. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, Vol. 2. IEEE Press, 3 Park Ave, New York, NY, USA, 134–138. https://doi.org/10.1109/ICPP.1993.153

[24] Florian Rathgeber, David A Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew TT McRae, Gheorghe-Teodor Bercea, Graham R Markall, and Paul HJ Kelly. 2016. Firedrake: automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software (TOMS)* 43, 3 (2016), 24.

[25] Vivek Sarkar and John Hennessy. 1986. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM, ACM Press, 2 Penn Plaza, Ste 701, New York, NY, USA, 202–211.

[26] Alina Sbirlea, Louis-Noel Pouchet, and Vivek Sarkar. 2014. Dfgr an intermediate graph representation for macro-dataflow programs. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2014 Fourth Workshop on*. IEEE, IEEE Press, 3 Park Ave, New York, NY, USA, 38–45.

[27] Alina Sbîrlea, Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. 2015. Polyhedral optimizations for a data-flow graph language. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, Springer Publishing, Salmon Tower Building, New York, NY, USA, 57–72.

[28] G. N. Srinivasa Prasanna, A. Agrawal, and B. R. Musicus. 1994. Hierarchical Compilation of Macro Dataflow Graphs for Multiprocessors with Local Memory. *IEEE Trans. Parallel Distrib. Syst.* 5, 7 (July 1994), 720–736. https://doi.org/10.1109/71.296318

[29] Vinay Vasista, Kumudha Narasimhan, Siddharth Bhat, and Uday Bondhugula. 2017. Optimizing Geometric Multigrid Method Computation Using a DSL Approach. In *In The IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*

(SC). ACM Press, 2 Penn Plaza, Ste 701, New York, NY, USA, 1–13.

[30] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. *Mathematical Software ICMS 2010* 6327 (2010), 299–302. http://link.springer.com/chapter/10.1007/978-3-642-15582-6

[31] Sven Verdoolaege. 2015. *barvinok: User Guide*. Compsys. http://compsys-tools.ens-lyon.fr/iscc/barvinok.pdf

[32] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*. ACM Press, 2 Penn Plaza, Ste 701, New York, NY, USA, 1–8.

[33] Qing Yi. 2012. POET: a scripting language for applying parameterized source-to-source program transformations. *Software: Practice and Experience* 42, 6 (2012), 675–706. http://dx.doi.org/10.1002/spe.1089

[34] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H Kuhn, Yang Ni, and David Padua. 2012. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, ACM Press, 2 Penn Plaza, Ste 701, New York, NY, USA, 207–218.