Beyond Polyhedral Analysis of OpenStream Programs

Work in progress

Nuno Miguel Nobre
The University of Manchester
Manchester, United Kingdom
nunomiguel.nobre@manchester.ac.uk

Andi Drebes
INRIA
Paris, France
andi.drebes@inria.fr

Graham Riley
The University of Manchester
Manchester, United Kingdom
graham.riley@manchester.ac.uk

Antoniu Pop
The University of Manchester
Manchester, United Kingdom
antoniu.pop@manchester.ac.uk

Abstract

Polyhedral techniques are, when applicable, an effective instrument for automatic parallelization and data locality optimization of sequential programs. This paper motivates their adoption in OpenStream, a task-parallel streaming language following the dataflow model of execution. We show that (1) it is possible to exploit the parallelism that naturally arises from dataflow task graphs with loop tiling transformations provided by the polyhedral model and (2) that a combination of dataflow task-parallelism and polyhedral optimizations performs significantly better than polyhedral parallelization techniques applied to sequential programs and dataflow task-parallelism without polyhedral optimization techniques. Our technique obtains parallel speedups superior to 1.2× when compared to state-of-the-art polyhedral-tiled OpenMP, and 1.6× when compared to manually tiled OpenStream implementations, for a simple Gauss-Seidel kernel.

However, stream indexing is often polynomial in the general case, severely limiting the set of OpenStream programs amenable to polyhedral tools and hindering the automation of our technique. We further investigate how the approach of Feautrier may offer a path not only to automatically convert fine-grained task-level concurrency to coarser-grained tasks, but also for scheduling under resource constraints.

1 Introduction

Programmers from all across the scientific spectrum tend to use high-level languages for productivity and portability. Quite naturally, their concern lies with the program semantics, i.e., with the results they can extract from the program, and not with the target architecture they will be executing it on. Multicore processors, deep pipelines, vector units and multiple levels of memory hierarchy make today's and future hardware architectures complex, variable in their capabilities and, therefore, difficult to target individually. As a result, the problem of lowering programmer-friendly and architecture-agnostic high-level language abstractions into well-orchestrated machine code for a specific target architecture is usually left to compilers, optimized libraries, and runtime systems. They are expected to enable the development of applications that fully exploit the parallel processing power of different machines and to take advantage of an increasing number of processing units.

Task-parallel programming models are an increasingly popular approach to address the issues above [3, 6–10, 19, 27, 28, 30, 36]. In particular, those based on streaming dataflow principles, e.g. StreamIt [36], ΣC [19], OpenStream [30], have strong assets to follow this architecture scaling trend. Computations are encapsulated in tasks that communicate through explicit data channels, or streams. A task-parallel dataflow program can be represented by a directed graph whose nodes are tasks and whose edges are defined by the producer-consumer task dependencies resulting from stream accesses. Given that the execution of tasks is triggered by the availability of their data operands, synchronization of parallel activities is implicit and latency can be naturally hidden. In addition, as the only sequencing between tasks stems from the data dependencies between them, a dataflow graph exposes numerous opportunities for parallelism [20, 26]. The functional determinism inherited from Kahn Process Networks [24] is enforced at the language level and both data and pipeline parallelism are naturally supported as particular instances of task parallelism [18]. However, exploiting the advantages of such models requires considerable effort and hinders productivity as the programmer has to manually implement tasks and precisely specify the data dependencies between them. Besides making the process more error-prone, this also increases the pressure on the programmer to make the right choices regarding task granularity. On the one hand finer-grained task graphs provide higher concurrency and better opportunities for load balancing, but on the other hand, coarser-grained task graphs are usually characterized by smaller inter-task communication and scheduling overhead.

One approach to automatically choose the optimal granularity is to start with fine-grained tasks and build successively bigger tasks in incremental fusion steps. A comprehensive review of graph algorithms, e.g., clustering or grain packing,

that follow this task coalescing procedure, is given in [34]. Here we simply note that the nodes of a task graph represent task executions or instances as opposed to task statements. While this allows for precise dependence specification between task instances as opposed to approximate dependencies between task statements, this also means the size of the data structure used to store such graphs is strongly problem-dependent. In present-day numerical applications it is not uncommon to include loops with millions of iterations and even in the best of these scenarios, memory requirements and code generation can easily become unmanageable [31].

In this paper, we motivate the applicability of existing polyhedral techniques to task-parallel dataflow programs in order to automate the granularity tuning process and generate suitable code for a given target architecture, relieving the user from such burden. These techniques use a compact abstract representation of programs, polyhedra in $\mathbb{Z}^n$, which is then subject to integer linear programming methods for analysis and high-level loop transformations [17]. As an alternative to graph algorithms, polyhedral transformations overcome the above limitations at the expense of limiting the range of both the available transformations and the amenable programs as detailed in Section 2.2.1, while still preserving a precise dependence description between task instances.

Section 2 briefly presents the OpenStream language and isolates its polyhedral fragment. Section 3 exemplifies, using a Gauss-Seidel kernel, how, for OpenStream programs within that fragment, it is possible to leverage existing polyhedral tools to build coarser-grained tasks with enhanced data locality. We describe how an intermediate and semantically equivalent C program is built and tiled with known polyhedral techniques and briefly outline the technical solutions that allowed those optimizations to be applied to the original OpenStream program. Section 4 discusses the expressiveness limitations of the polyhedral model, prompting a shift to a polynomial description of OpenStream programs. We recall important deadlock results from [11] and their implication on changing the granularity of task instances and, additionally, on bounding streams, i.e., on the problem of scheduling within a limited memory footprint. After describing how Feautrier's work [15] might provide some solutions and directions for future work in Section 5, we briefly discuss the most closely related work in Section 6 and conclude in Section 7.

2 A Brief Description of OpenStream

OpenStream is a task-parallel streaming dataflow language implemented as an extension to OpenMP supporting the specification of fine-grained task, data and pipeline parallelism. For a more detailed presentation the reader may refer to [30]. The fully fledged computational model underlying the operational semantics of the language is detailed in [29] and a simplified, partial-model is also defined in [12]. We briefly recall the three main concepts of OpenStream: streams, dataflow tasks and the control program.

2.1 Streams and Tasks

A stream is a one-dimensional array of indefinite size, whose elements are of the same type. Each element of a stream is written using Dynamic Single Assignment (DSA), i.e., each stream element is written at most once. Conceptually, a stream s has a read pointer J_s and a write pointer I_s that define which elements of the stream are affected by subsequent read and write accesses. Streams themselves can also be grouped in arrays of arbitrary size and dimension.

A task instance t of a task τ has a work function, i.e., an arbitrary sequence of instructions acting on local variables and on a finite set of streams. Access to each of the referenced streams is provided through windows. A window on a stream s is characterized by the access type it provides (read or write), its horizon $h_{t,s}$, and the burst $b_{t,s}$. The horizon is a positive integer specifying the size of the window. The burst is a non-negative integer specifying the amount by which the stream's read or write pointer is shifted after task creation. The burst of a write access window is equal to its horizon, guaranteeing single assignment, i.e., dataflow-only dependencies. The burst of a read access may be less than or equal to its horizon. A burst of zero corresponds to the peek operation. Figure 1 depicts how these concepts are used in OpenStream to express the flow of data between producer and consumer tasks.

Stream accesses through windows determine the producer-consumer relationships: a task is a producer for another task if its output window overlaps with the other task's input window. Such relationships are captured in the task graph, such as the one in Figure 2 for the example of the stream accesses above.

2.2 The Control Program

All task creations take place in the *control program*, an ordinary function containing task instantiation statements. Each such statement also includes the respective list of read and write accesses to streams, each specified as a reference to a stream, a burst and a horizon. The control program thus defines the order of task creation, in turn defining the producer-consumer relationships between tasks and thus the partial execution order of the task-parallel dataflow program.

2.2.1 The Polyhedral Fragment of OpenStream

The analysis is confined to the subset of OpenStream in which the control program fits the polyhedral model, as first defined in [11]. Communication between tasks and the control program using shared variables and OpenMP-inherited mechanisms like `firstprivate` or `copyin` is not allowed. We only consider programs that are deterministic by construction, i.e., programs for which the task creation order in the control program and thus the interleaving of data

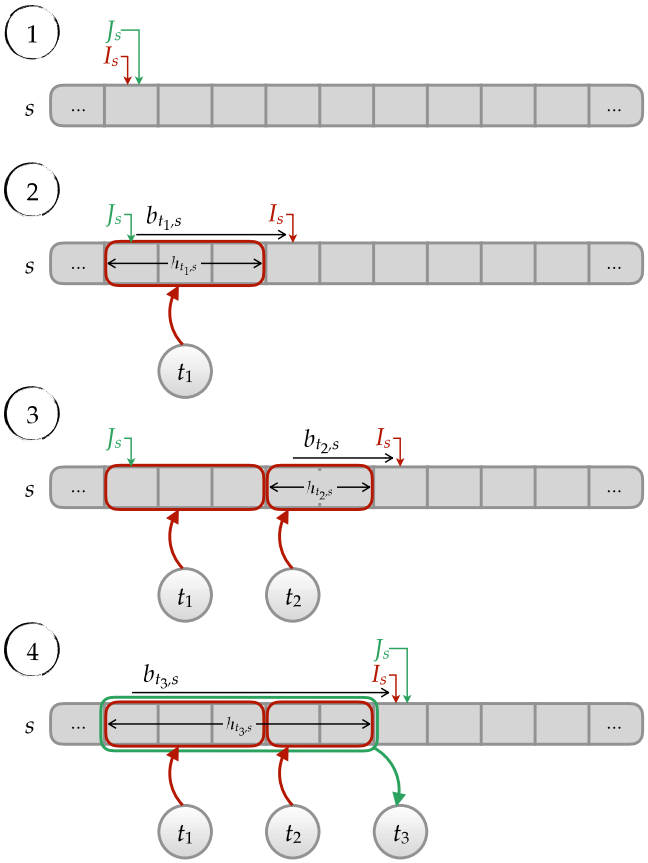

Figure 1. Illustration of stream accesses and the evolution of the read and write pointers for a stream s for $b_{t,s} = h_{t,s}$ for all task instances t. (1) shows the initial state of the stream before any access. In (2) resp. (3), producers t_1 resp. t_2 are created and the write pointer I_s updated. Lastly, in (4), consumer t_3 is created and the read pointer J_s advanced.

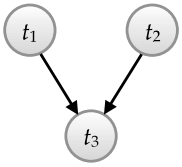

Figure 2. Task graph for the example of Figure 1. Task t_3 consumes data produced by both t_1 and t_2 and, thus, must be executed after both of them. t_1 and t_2, however, can be executed concurrently.

in streams is determined statically. A sufficient (but not necessary) condition is to require a sequential control program [29]. That is, the code of the tasks themselves can be arbitrary, but nested task creation is not allowed (tasks cannot instantiate other tasks). Control flow stems, exclusively, from the textual sequence of statements, affine conditional expressions and arbitrary nested counted loops with affine

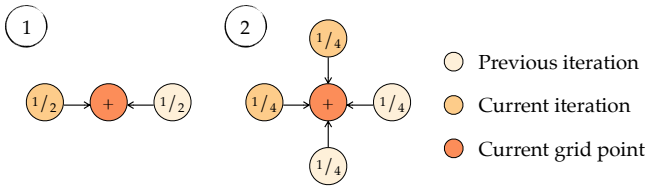

Figure 3. Finite-difference stencil for (1) the one-dimensional, two-point Gauss-Seidel kernel and (2) the two-dimensional, four-point kernel.

bounds in the surrounding loop iterators and integer parameters. Without loss of generality, and as suggested in [16], the only executable statements are task creation statements. Window bursts may be numerical or parametric constants, or polynomial expressions.

3 Gauss-Seidel Kernel

The solution for Laplace's equation $\nabla^2\phi = 0$ can be numerically approximated using finite-difference methods. These methods use a finite difference grid composed of points that provide an approximate solution to the original partial differential equation. The Gauss-Seidel method successively improves an initial estimate of the solution, such that, for the one-dimensional case, at iteration i and at grid point j of a grid with unit-spacing, ϕ_j can be approximated by $\bar{\phi}_j^{(i)}$:

$$\bar{\phi}_j^{(i)} = \frac{\bar{\phi}_{j-1}^{(i)} + \bar{\phi}_{j+1}^{(i-1)}}{2}.$$

Similarly, for a two-dimensional grid with unit-spacing, at a grid point (j,k), the value $\phi_{j,k}$ can be approximated by

$$\bar{\phi}_{j,k}^{(i)} = \frac{\bar{\phi}_{j-1,k}^{(i)} + \bar{\phi}_{j,k-1}^{(i)} + \bar{\phi}_{j+1,k}^{(i-1)} + \bar{\phi}_{j,k+1}^{(i-1)}}{4}.$$

Figure 3 depicts the stencils that correspond to both these equations.

3.1 Fine-Grained Implementations

The following code excerpt is a sequential C implementation of the one-dimensional, two-point stencil given in Figure 3 for a grid of size N and I iterations:

```
int I, N;

for (i = 0; i < I; ++i)
  for (j = 1; j < N - 1; ++j)
    phi[j] = (phi[j - 1] + phi[j + 1]) / 2;
```

The most straightforward way of building a task-parallel OpenStream-equivalent code is by encapsulating the inner loop body in a task. There are thus as many tasks as iterations of the innermost loop and the work function of each task is thus composed of a single statement. As we will show in Section 3.3, a direct implementation of this solution is

inefficient due to the high overhead for task creation, but serves as a starting point for transformations generating a more efficient, tiled version of the program.

The code for the naive implementation below is specified using the compact OpenStream pseudo-code introduced in [11, 16]. This notation focuses on the dependencies between tasks and, although it generally omits the actual work function, we have included it here for completeness. Dependencies are specified as follows: *read once from s* reads one element from a stream *s*, *write once into s* writes one element to a stream *s*, and *peek once from s* reads one element from *s* with a burst of 0, i.e., without advancing the read index of the stream. In the following snippet, $S(j)$ is one of the N streams in stream array S.

```
parameter I, N;

stream_array S[N];

for (i = 0; i < I; ++i)
  for (j = 1; j < N - 1; ++j)
    task {
      read once from S[j];      // phi[j] (discarded)
      peek once from S[j - 1];  // phi[j - 1]
      peek once from S[j + 1];  // phi[j + 1]
      write once into S[j];     // phi[j]

      // work function:
      // phi[j] = (phi[j - 1] + phi[j + 1]) / 2;
    }
```

The iteration domain and the flow dependence distance vectors of both implementations are shown in Figure 4. Notice how the iteration domain of the task is a polyhedron, fitting within the restrictions of Section 2.2.1.

3.2 Tiled Implementations

Directly applying polyhedral transformations to OpenStream program is out of reach of state-of-the-art polyhedral compilers. We thus resort to creating an intermediate and semantically equivalent C program to which we apply polyhedral tiling. To convert OpenStream code into a sequential C form tractable by existing polyhedral tools, we must preserve the dependence information between different tasks. Note that we cannot use the initial implementation, since in addition to flow dependencies, the sequential C code also contains anti-dependencies with the exact same distance vectors as the flow dependencies and an output dependence from to $(i-1, j)$ to (i, j).

Borrowing on the idea of I-structures introduced in [1], we build a new sequential C code where streams are encoded as columns in a two-dimensional array. This sequence of versions preserves the single assignment property of streams

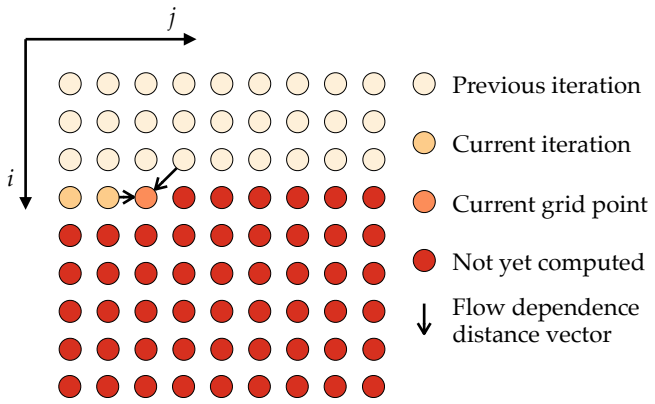

Figure 4. Iteration domain for a one-dimensional Gauss-Seidel kernel. The solution estimate on point (i, j) is updated using one point from the current iteration, $(i, j-1)$, and one other from the previous iteration, $(i-1, j+1)$. The dependence distance vectors depict the flow of data and are just the difference of the iteration vectors of the iterations in dependence. They are shown here for a single computation/task instance as they are the same for all others.

and thus eliminates name dependencies. Although in this particular case, the flow dependencies encode the same information as the name dependencies (notice how the output dependence can be transitively obtained from the flow dependencies), we chose to keep this step in the procedure for generality.

To apply loop tiling to the iteration domain, we use the polyhedral-based Pluto compiler [4, 5]. The resulting tiling is illustrated in Figure 5. Finally, we manually build an OpenStream program that implements the generated tiling. Since Pluto can only generate tiled code where the tile sizes are statically fixed at compile-time, we also manually generalized the code to use parametric tiling [22, 35]. Note that as task bodies are executed sequentially, some concurrency is lost in favor of a smaller number of tasks and better data locality.

For comparison, we additionally consider a completely manually built and parametrically and spatially tiled OpenStream version of the kernel as depicted in Figure 6. This differs from the previous tiling by only coarsening tasks along the spatial grid dimension and not across Gauss-Seidel iterations. Besides increasing the number of dependencies for each tile, this tiling-pattern also exploits data locality less efficiently.

3.3 Results

The experiments were carried out on GNU/Linux Mint 18.3 with kernel 4.15.0-42-generic using OpenStream's commit `e5158f5e`[1]. All OpenStream implementations were compiled with the OpenStream compiler, based on gcc 5.2.0, and the

[1] [Online] Available at http://www.openstream.info

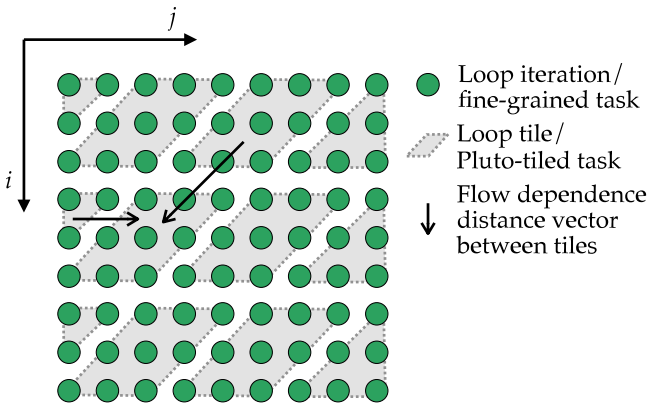

Figure 5. Tiling generated by Pluto for the one-dimensional Gauss-Seidel kernel. Each tile is traversed sequentially in row-major order. Notice how the edges of the parallelogram-shaped tiles lie along the dependence distance vectors. For illustration purposes, 3×3 tiles were chosen but the optimum size is, in general, architecture-dependent.

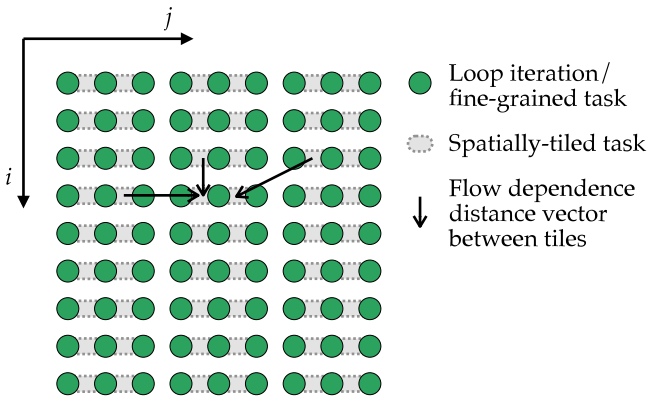

Figure 6. Spatially tiled implementation of the one-dimensional Gauss-Seidel kernel. Each tile now depends on three other tiles as opposed to two.

`-Ofast` optimization flag. All the remaining implementations were compiled using an unmodified gcc 5.2.0. The host is equipped with an Intel Core i7-6700 processor with four physical cores and eight logical threads, and 16 GiB of main memory.

Figure 7 shows the execution times of each implementation of the one-dimensional Gauss-Seidel kernel, normalized to the OpenStream Pluto-tiled version. The labels used in the figure are as follows.

Sequential implementations running on a single thread (1T):

[SeqC] The sequential C code from the beginning of Subsection 3.1 and its single-assignment counterpart (SA) described in Subsection 3.2.

Parallel implementations running on two, four and eight threads (2T/4T/8T):

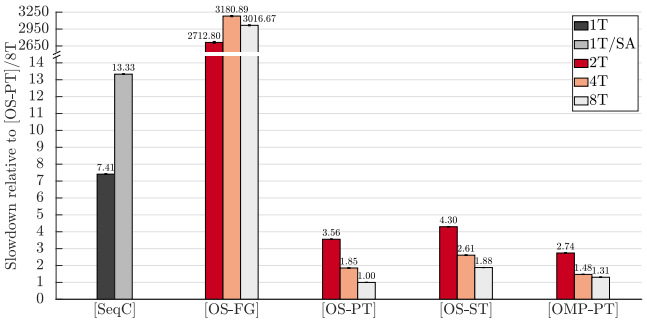

Figure 7. Execution times, normalized to [OS-PT] on 8T, for the fastest implementations of the one-dimensional Gauss-Seidel kernel with a spatial grid of $N = 32769$ points and $I = 32768$ iterations. See Table 1 for the selected tile sizes. The bars' height and the error bars represent the mean and 95% confidence intervals, respectively, over samples of ten runs.

Table 1. Optimal tile sizes for the tiled implementations of the one-dimensional Gauss-Seidel kernel with a spatial grid of $N = 32769$ points and $I = 32768$ iterations

	[OS-PT]	[OS-ST]	[OMP-PT]
2T	$2^6 \times 2^6$	2^{13}	$2^2 \times 2^2$
4T	$2^7 \times 2^7$	2^{12}	$2^2 \times 2^2$
8T	$2^8 \times 2^8$	2^{11}	$2^3 \times 2^3$

[OS-FG] The naive and fine-grained OpenStream implementation from the end of Subsection 3.1.

[OS-PT] The Pluto-tiled OpenStream code, see Figure 5.

[OS-ST] The spatially tiled OpenStream implementation, see Figure 6.

[OMP-PT] For completion, we also consider the OpenMP program corresponding to the tiling pattern of Figure 5, from which [OS-PT] was derived.

For the tiled implementations, we leveraged parametric tiling to repeatedly execute the kernel on the target machine for different tile sizes. This allowed us to optimize the tile sizes, similarly to how ATLAS [37] optimizes BLAS kernels. We exhaustively searched the space of tile configurations where the length along each of the tile dimensions is a power of two and equal across all such dimensions. Table 1 shows the final tile sizes retained from this exploration.

The single assignment form C code [SeqC]/SA is 1.8× slower than [SeqC] due to the additional memory requirements of version splitting, resulting in decreased data locality. The naive and fine-grained OpenStream implementation [OS-FG] exposes large amounts of parallelism. However, a quick analysis with Aftermath [13, 14], a tool with support for visualization and analysis of OpenStream trace files, shows that the increased parallelism is over-compensated by the high overhead incurred by task creation and dependence

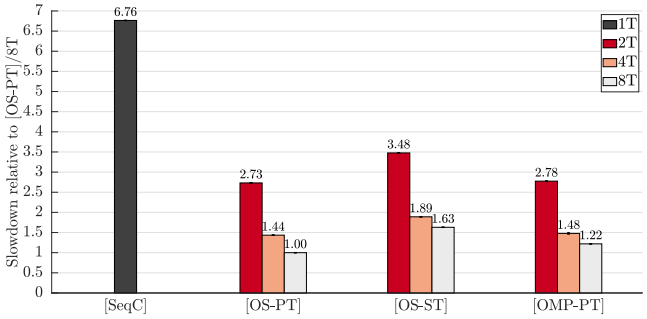

Figure 8. Execution times, normalized to [OS-PT] on 8T, for the equivalent implementations of the two-dimensional, four-point Gauss-Seidel stencil presented in Figure 3 with a spatial grid of $N \times N = 2049 \times 2049$ points and $I = 2048$ iterations. See Table 2 for the selected tile sizes. The bars' height and the error bars represent the mean and 95% confidence intervals, respectively, over samples of ten runs.

resolution in the runtime compared to the short execution time of the task body, composed of a single statement. This results in a 2700× slowdown compared to the Pluto-tiled version. The Pluto-tiled OpenStream code [OS-PT] generates a coarser-grained graph. In comparison with [OS-FG], there are fewer and longer-running tasks. This significantly limits the number of calls to the OpenStream runtime and thus lowers the runtime overhead. In addition, the combined tiling in the spatial and iteration dimensions in [OS-PT] improves data locality significantly compared to the spatial tiling in [OS-ST].

Lastly, [OS-PT] scales better than [OMP-PT] for a larger number of threads. While the point-to-point synchronization of [OS-PT] improves core utilization in comparison with the barrier synchronization of [OMP-PT], it does not fully explain the improved scalability. The effects on the utilization of the memory hierarchy and simultaneous multithreading are currently under investigation.

We now extend the analysis to the two-dimensional, four-point stencil of Figure 3. Since the process to obtain equivalent OpenStream and OpenMP implementations for the two-dimensional case is very similar to the one-dimensional case, a description of the procedure is omitted. As far as the tiling is concerned, the parallelogram-shaped tiles of Figure 5 are replaced by parallelepipeds and the one-dimensional tiles of Figure 6 by proper rectangles. The results are presented in Figure 8 for the tile sizes in Table 2. The major difference in comparison to the one-dimensional case is the performance advantage of [OS-PT] relative to [OMP-PT] for all thread configurations. This is due to the bigger, two-dimensional wavefronts which benefit the point-to-point synchronization in the dataflow implementation.

In conclusion, polyhedral-tiled OpenStream [OS-PT] is at least 1.2× faster than OpenMP [OMP-PT] and 1.6× faster than OpenStream with simple tiling techniques [OS-ST] for both Gauss-Seidel stencils, showing that a combination of dataflow task-parallelism with polyhedral optimizations performs significantly better than either approach alone.

Table 2. Optimal tile sizes for the tiled implementations of the two-dimensional Gauss-Seidel kernel with a spatial grid of $N \times N = 2049 \times 2049$ points and $I = 2048$ iterations.

	[OS-PT]	[OS-ST]	[OMP-PT]
2T	$2^5 \times 2^5 \times 2^5$	$2^8 \times 2^8$	$2^2 \times 2^2 \times 2^2$
4T	$2^5 \times 2^5 \times 2^5$	$2^8 \times 2^8$	$2^2 \times 2^2 \times 2^2$
8T	$2^5 \times 2^5 \times 2^5$	$2^7 \times 2^7$	$2^3 \times 2^3 \times 2^3$

4 Polynomial Stream Indexing

The following listing presents an implementation for the Pluto-tiled one-dimensional Gauss-Seidel kernel from Figure 5, Section 3.2, in which a task τ is instantiated for each parallelogram-shaped, B-element-wide block and for I/B iterations.

```
parameter I, N, B;
parameter IB = I / B, NB = N / B;

stream_array S[NB];

for (i = 0; i < IB; ++i)
  for (j = 1; j < NB - 1; ++j)
    task {                              // task τ
      read B times from S[j - 1];       // b_τ(i,j),S(j−1) = B
      read B times from S[j];           // b_τ(i,j),S(j) = B
      peek B times from S[j + 1];       // b_τ(i,j),S(j+1) = 0
      write B times into S[j - 1];
      write B times into S[j];
    }
```

As the task's iteration domain is a polyhedron, the code is within the polyhedral fragment of the OpenStream language, as defined in Section 2.2.1.

The index $J_s(t)$ of the first element of the stream s in the read window of an instance t of the task τ, where s is a stream of the array S, can be determined by calculating the sum of all bursts of all previously created task instances that read from s. For $s = S(j)$ and $t = \tau(i, j)$, the index $J_{S(j)}\big(\tau(i, j)\big)$ is thus:

$$
\begin{aligned}
J_{S(j)}\big(\tau(i,j)\big) &= \sum_{i'=0}^{i-1} b_{\tau(i',j),S(j)} + b_{\tau(i',j+1),S(j)} \\
&= \sum_{i'=0}^{i-1} b_{\tau(i',j),S(j)} + b_{\tau(i',j),S(j-1)} \\
&= \sum_{i'=0}^{i-1} 2B \\
&= 2B \cdot i\,.
\end{aligned}
$$

The stream index is thus polynomial with respect to the loop iterators and integer parameters of the control program. This is also true for the stream indexing in the two-dimensional Gauss-Seidel kernel.

Generally speaking, these indexes may be polynomial for OpenStream programs whose control program fits in the polyhedral fragment [11]. Intuitively, the one-dimensional nature of streams will inevitably lead to familiar array linearization functions whenever a program needs to map a multi-dimensional sequence of values into a stream, which is usually the case when the loop nest enclosing the task has more than one dimension.

5 Future Work

Polynomials of arbitrary degree as stream indexing functions have the undesirable consequence of rendering deadlock detection of polyhedral OpenStream programs undecidable. In [11], it is shown that deadlock detection requires resolving polynomial equations in the integers, thereby inheriting the insolvability proof of Hilbert's 10^{th} problem. The authors established this by using a procedure devised in [38] to determine that race detection in X10 is also undecidable.

Any valid loop transformation on the control program will preserve the semantics of the control program, i.e., the dependencies between task instances imposed by the original control program. This means that, although the task creation order might change, the task graph is unaffected and program execution is unchanged. On the other hand, over-constraining the runtime system by introducing new dependencies can have a greater impact on the task graph and program execution. However, any transformation that adds new dependencies is required to be safe, i.e., to retain the semantics of the original task-parallel program and, even though the necessary dependencies are guaranteed to be preserved, the same cannot be assumed about the absence of deadlocks.

Since loop transformations on the control program are ineffective to improve performance, different solutions are needed. In this section, we focus on two techniques:

- Bounding streams in order to limit requirements for memory resources
- Coarsening graphs with fine-grained tasks in order to limit the overhead of task management and improve data locality

Both techniques can potentially introduce deadlocks and thus require careful implementation. We provide an outline for procedures ensuring the validity of the applied transformations, based on the theoretical foundations provided in [11] and [15].

In [11], the authors show that the existence of a schedule for a polyhedral OpenStream program guarantees the absence of a deadlock and vice-versa. An algorithm for the construction of schedules for a subset of OpenStream programs with polynomial stream indexing, i.e., when dependence relations are represented as polynomial constraints, is given in [15], which extends Farkas' lemma to polynomials based on the mathematical work of Handelman [21] and Schweighofer [33].

In the remainder of this section, we exploit this property and algorithm to provide methods that certify the validity of the transformations described above.

5.1 Bounding Streams

Although the problem of bounding streams to limit memory requirements does not necessarily fall inline with the granularity problem identified above, it helps exemplifying how to add dependencies to the task graph. Indeed, as hinted in Subsection 5.2, this is not as obvious in the case of coarsening the task graph, where the range of possible transformations to the graph is huge. In addition, the ability to *statically* control the memory footprint of a program, is an interesting goal of its own.

Let l_s be the "live" size of a stream s, i.e., the maximum number of elements in a stream that have been produced but not yet consumed. While introducing such a limit helps limiting the memory footprint, it may not only drastically reduce the set of valid schedules for a given OpenStream program, but also introduce deadlocks.

For example, for a live size of $l_s = 5$ for s in Figure 1, the stream can simultaneously hold the data produced by both t_1 and t_2 before t_3 consumes anything. Any schedule for the task graph of Figure 2 thus remains valid. However, if $l_s = 3$, task t_2 cannot write to s before t_3 consumes, but t_3 depends on data produced by t_2. The newly introduced "back-pressure" dependence creates a cycle, which is shown in the task graph in Figure 9, and introduces a deadlock. In general, this can be modeled by extending the instance-wise dependence graph with new dependencies from task instance t' to t whenever there exist i and j, $J_s(t') \leq i \leq J_s(t')+h_{t',s}-1$ and $I_s(t) \leq j \leq I_s(t) + b_{t,s} - 1$, such that $i \leq j - l_s$, i.e., whenever $J_s(t') \leq I_s(t) + b_{t,s} - 1 - l_s$.

One way of proving the absence of deadlocks for a given stream size l_s consists in adding the necessary dependencies for back pressure to the task graph and to use Feautrier's

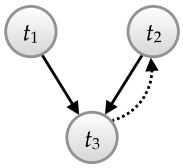

Figure 9. Task graph for the example of Figure 1 when $l_s = 3$. The new dependence from t_3 to t_2 creates a deadlock.

algorithm to determine the existence of a schedule. If the algorithm is able to determine a schedule, the absence of deadlocks is proven. The opposite, however, is not true, since there are some programs for which the algorithm is unable to find a schedule even though one might exist.

However, the subset of programs and stream sizes for which the existence of a schedule and thus the absence of a deadlock can be proven is still relevant for use in practice. We plan to investigate the properties of the programs which comprise such subset and their relation with the determined stream sizes.

5.2 Coarsening Task Graphs

We now focus on the problem of converting large amounts of fine-grained tasks into coarser-grained tasks, as solved manually in Section 3 for the Gauss-Seidel kernel. A strategy coarsening a task graph by merging sets of task instances into larger instances must take into account the direct and indirect (by transitivity) producer-consumer relationships of the merged tasks, since arbitrarily merging tasks might introduce deadlocks.

Figure 10 illustrates this. The left hand side of the figure (1) shows an extended version of the task graph of Figure 2, in which a task t_0 has been added that acts as a producer for both t_1 and t_2. If t_0 and t_3 were merged into a new task instance $t_0 + t_3$, as shown on the right hand side of the figure (2), the resulting program would contain a deadlock, due to the cyclic dependencies between t_0+t_3, t_1 and t_2, respectively.

Feautrier's algorithm can again provide a base to prove the validity of a coarsening transformation by constructing a schedule for the modified OpenStream program. Again, the algorithm failing to find a schedule does not prove the existence of a deadlock, but if the algorithm succeeds, the coarsening operation has produced a deadlock-free program.

With such a large search space, a good task coarsening strategy has to be devised. Loop strip-mining, fragmenting large loops of fine-grained tasks into segments or strips corresponding to coarser-grained tasks, seems like a promising candidate to improve performance. Together with a congruent increase in the bursts/horizons of the new tasks, these transformations should preserve the original dependencies, their validity relying solely on deadlock absence. It remains to be determined what class of programs are amenable to

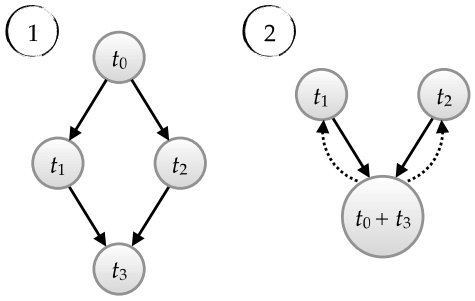

Figure 10. Task graph of a simple program before (1) and after (2) t_0 and t_3 are coalesced into t_0+t_3 creating a deadlock.

loop strip-mining and also to what extent (valid) loop transformations on the control program can, by reordering task instantiations and, thus, stream accesses, facilitate task coalescing. In fact, if otherwise irregular tile-accesses are reordered in such a way that those tile-accesses are contiguous in the stream, task coalescing becomes the loop strip-mining problem identified above, where we just need to change bursts/horizons.

6 Related Work

The urge to extend polyhedral techniques beyond imperative languages into the class of dataflow languages has been around since at least the beginning of the decade, leading to the development of polyhedral optimizers for domain specific languages. PolyGLoT [2] is a polyhedral automatic transformation framework for LabVIEW, a graphical dataflow programming language for laboratory data acquisition. Dataflow program parts are converted into a polyhedral representation amenable to state-of-the-art polyhedral tools whose resulting optimized C code is synthesized back into LabVIEW dataflow code. Likewise, R-Stream·TF [32] is a tool for polyhedral optimization of neural network computations expressed as TensorFlow dataflow graphs. Similarly to the procedure in Section 3, TensorFlow subgraphs are converted into sequential C code, optimized using the R-Stream compiler, wrapped in a custom TensorFlow operator and, lastly, replaced back into the original computation graph. Our work aims at extending these ideas to a more expressive dataflow programming model.

Furthermore, it has been previously shown [23, 25] that dataflow implementations of common linear algebra and stencil kernels often outperform the barrier-synchronized code generated by polyhedral compilers. In this sense, Section 3 complements these studies by providing additional evidence in this direction.

Lastly, in addition to Feautrier's work [15, 16], Zou and Rajopadhye [39] developed a code generator that implements polynomial schedules for parametric tile sizes. Although it is not yet clear how this work can be applicable to task-parallel

dataflow frameworks, the ideas described there might provide an alternative approach to the preliminary tiling strategy described in Subsection 5.2.

7 Conclusion

We have shown that the combination of static transformations using the polyhedral model and the dynamic nature of task-parallel dataflow OpenStream programs significantly increases performance compared to polyhedral optimizations or dataflow task-parallelism alone. We showed that this can be achieved by translating a parallel OpenStream implementation to an equivalent sequential program, which is then optimized using Pluto, a state-of-the-art polyhedral source-to-source compiler and translated back to OpenStream.

For a more general approach to coarsen task granularity of arbitrary task graphs and limiting the memory footprint by bounding streams, we have outlined a process for the verification of the correctness of such transformations based on Feautrier's approach [15]. We identified strip-mining as a promising candidate technique to deal with the large search space for granularity reduction.

References

[1] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (Oct. 1989), 598–632. https://doi.org/10.1145/69558.69562

[2] Somashekaracharya G. Bhaskaracharya and Uday Bondhugula. 2013. PolyGLoT: A Polyhedral Loop Transformation Framework for a Graphical Dataflow Language. In *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 123–143. https://doi.org/10.1007/978-3-642-37051-9_7

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel and Distrib. Comput.* 37, 1 (1996), 55 – 69. https://doi.org/10.1006/jpdc.1996.0107

[4] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *International Conference on Compiler Construction (ETAPS CC)*.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not.* 43, 6 (June 2008), 101–113. https://doi.org/10.1145/1379022.1375595

[6] François Broquedis, Thierry Gautier, and Vincent Danjean. 2012. LIBKOMP, an Efficient OpenMP Runtime System for Both Fork-join and Data Flow Paradigms. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World (IWOMP'12)*. Springer-Verlag, Berlin, Heidelberg, 102–115. https://doi.org/10.1007/978-3-642-30961-8_8

[7] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşirlar. 2010. Concurrent Collections. *Sci. Program.* 18, 3-4 (Aug. 2010), 203–217. https://doi.org/10.1155/2010/521797

[8] D. Callahan, B. L. Chamberlain, and H. P. Zima. 2004. The cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* 52–60. https://doi.org/10.1109/HIPS.2004.1299190

[9] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. ACM, New York, NY, USA, 51–61. https://doi.org/10.1145/2093157.2093165

[10] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.* 40, 10 (Oct. 2005), 519–538. https://doi.org/10.1145/1103845.1094852

[11] Albert Cohen, Alain Darte, and Paul Feautrier. 2016. Static Analysis of OpenStream Programs. In *6th International Workshop on Polyhedral Compilation Techniques (IMPACT'16), held with HIPEAC'16 (Proceedings of the IMPACT series)*. Michelle Strout and Tomofumi Yuki, Prague, Czech Republic. https://hal.inria.fr/hal-01251845

[12] Andi Drebes. 2015. *Dynamic optimization of data-flow task-parallel applications for large-scale NUMA systems.* Theses. Université Pierre et Marie Curie - Paris VI. https://tel.archives-ouvertes.fr/tel-01258451

[13] A. Drebes, A. Pop, K. Heydemann, and A. Cohen. 2016. Interactive visualization of cross-layer performance anomalies in dynamic task-parallel applications and systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 274–283. https://doi.org/10.1109/ISPASS.2016.7482102

[14] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. 2014. Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. In *Seventh Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2014)*. Vienna, Austria. https://hal.archives-ouvertes.fr/hal-01136508

[15] Paul Feautrier. 2015. The Power of Polynomials. In *5th International Workshop on Polyhedral Compilation Techniques (IMPACT'15)*, Alexandra Jimborean and Alain Darte (Eds.). Amsterdam, Netherlands. https://hal.inria.fr/hal-01094787

[16] Paul Feautrier and Albert Cohen. 2018. On Polynomial Code Generation. In *8th International Workshop on Polyhedral Compilation Techniques (IMPACT'18)*. Manchester, United Kingdom.

[17] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer US, Boston, MA, 1581–1592. https://doi.org/10.1007/978-0-387-09766-4_502

[18] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. *SIGPLAN Not.* 41, 11 (Oct. 2006), 151–162. https://doi.org/10.1145/1168918.1168877

[19] Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. 2011. ΣC: A Programming Model and Language for Embedded Manycores. In *Algorithms and Architectures for Parallel Processing*, Yang Xiang, Alfredo Cuzzocrea, Michael Hobbs, and Wanlei Zhou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 385–394.

[20] J. R Gurd, C. C Kirkham, and I. Watson. 1985. The Manchester Prototype Dataflow Computer. *Commun. ACM* 28, 1 (Jan. 1985), 34–52. https://doi.org/10.1145/2465.2468

[21] David Handelman. 1988. Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific J. Math.* 132, 1 (1988), 35–62. https://projecteuclid.org:443/euclid.pjm/1102689794

[22] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. 2009. Parametric Multi-level Tiling of Imperfectly Nested Loops. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 147–157. https:

//doi.org/10.1145/1542275.1542301

[23] Tian Jin, Nirmal Prajapati, Waruna Ranasinghe, Guillaume Iooss, Yun Zou, Sanjay Rajopadhye, and David G. Wonnacott. 2016. Hybrid Static/Dynamic Schedules for Tiled Polyhedral Programs. *CoRR* abs/1610.07236 (2016). arXiv:1610.07236 http://arxiv.org/abs/1610.07236

[24] G. Kahn. 1974. The semantics of a simple language for parallel programming. In *Information processing*, J. L. Rosenfeld (Ed.). North Holland, Amsterdam, Stockholm, Sweden, 471–475.

[25] Martin Kong, Antoniu Pop, Louis-Noël Pouchet, R. Govindarajan, Albert Cohen, and P. Sadayappan. 2015. Compiler/Runtime Framework for Dynamic Dataflow Parallelization of Tiled Programs. *ACM Trans. Archit. Code Optim.* 11, 4, Article 61 (Jan. 2015), 30 pages. https://doi.org/10.1145/2687652

[26] Ben Lee and A. R. Hurson. 1994. Dataflow Architectures and Multithreading. *Computer* 27, 8 (Aug. 1994), 27–39. https://doi.org/10.1109/2.303620

[27] OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface Version 4.0. *White Paper* (2013).

[28] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. 2009. Hierarchical Task-Based Programming With StarSs. *Int. J. High Perform. Comput. Appl.* 23, 3 (Aug. 2009), 284–299. https://doi.org/10.1177/1094342009106195

[29] Antoniu Pop and Albert Cohen. 2012. *Control-Driven Data Flow.* Research Report RR-8015. INRIA. 48 pages. https://hal.inria.fr/hal-00717906

[30] Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs. *ACM Trans. Archit. Code Optim.* 9, 4, Article 53 (Jan. 2013), 25 pages. https://doi.org/10.1145/2400682.2400712

[31] Louis-Noël Pouchet. 2013. Program transformations and optimizations in the polyhedral framework. 1st Polyhedral Spring School, St-Germain au Mont d'or, France.

[32] Benoît Pradelle, Benoît Meister, M. Baskaran, Jonathan Springer, and Richard Lethin. 2017. Polyhedral Optimization of TensorFlow Computation Graphs.

[33] Markus Schweighofer. 2002. An algorithmic approach to Schmüdgen's Positivstellensatz. *Journal of Pure and Applied Algebra* 166, 3 (2002), 307 – 319. https://doi.org/10.1016/S0022-4049(01)00041-X

[34] Oliver Sinnen. 2007. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing).* Wiley-Interscience.

[35] Sanket Tavarageri, Albert Hartono, Muthu Manikandan Baskaran, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2010. Parametric Tiling of Affine Loop Nests.

[36] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02).* Springer-Verlag, London, UK, UK, 179–196. http://dl.acm.org/citation.cfm?id=647478.727935

[37] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC '98).* IEEE Computer Society, Washington, DC, USA, 1–27. http://dl.acm.org/citation.cfm?id=509058.509096

[38] Tomofumi Yuki, Paul Feautrier, Sanjay V. Rajopadhye, and Vijay Saraswat. 2013. Checking Race Freedom of Clocked X10 Programs. *CoRR* abs/1311.4305 (2013). arXiv:1311.4305 http://arxiv.org/abs/1311.4305

[39] Y. Zou and S. Rajopadhye. 2018. A Code Generator for Energy-Efficient Wavefront Parallelization of Uniform Dependence Computations. *IEEE Transactions on Parallel and Distributed Systems* 29, 9 (Sept 2018), 1923–1936. https://doi.org/10.1109/TPDS.2017.2709748