Polyhedral Dataflow Programming: a Case Study

Romain Fontaine, Laure Gonnord, Lionel Morel

Polyhedral Dataflow Programming: a Case Study

Romain Fontaine
Univ Lyon, INSA Lyon, CITI
F-69621 Villeurbanne, France
Email: romain.fontaine@insa-lyon.fr

Laure Gonnord
Univ Lyon, Université Claude Bernard Lyon 1
CNRS, ENS de Lyon, Inria,
LIP, F-69342, LYON Cedex 07, France
Email: laure.gonnord@ens-lyon.fr

Lionel Morel
Univ Grenoble Alpes, CEA, List
F-38000 Grenoble, France
Email: lionel.morel@cea.fr

***Abstract*—Dataflow languages expose the application's potential parallelism naturally and have thus been studied and developped for the past thirty years as a solution for harnessing the increasing hardware parallelism. However, when generating code for parallel processors, current dataflow compilers only take into consideration the overall dataflow network of the application. This leaves out the potential parallelism that could be extracted from the internals of agents, typically when those include loop nests, for instance, but also potential application of intra-agent pipelining, or task spliting and rescheduling.**

In this work, we study the benefits of jointly using polyhedral compilation with dataflow languages. More precisely, we propose to expend the parallelization of dataflow programs by taking into account the parallelism exposed by loop nests describing the internal behavior of the program's agents. This approach is validated through the development of a prototype toolchain based on an extended version of the ΣC language. We demonstrate the benefit of this approach and the potentiality of further improvements on relevant case studies.

***Keywords*-parallelism; dataflow programming; compilation; runtime system; load-balancing.**

I. Introduction

Multi-core processors are everywhere, from high-end servers to user-oriented embedded systems like cellphones or task-specific accelerators. Applications that need to take advantage of the parallelism offered by these hardware are now numerous and range from latency-sensitive compression/decompression algorithms (eg video processing) to compute-intensive ones like machine-learning algorithms.

Programming these applications and taking advantage of the hardware parallelism is still a considerable challenge from the programmer's point-of-view. For some of these applications, dataflow programming is a premier choice because this programming style naturally fits the designer abstraction model. As a plus, dataflow is amenable to quite efficient parallelization because it naturally exposes task, data and pipeline parallelism.

In a dataflow programming language, programmers describe their application as a set of side-effect-free actors or *agents* that communicate solely through First-In-First-Out channels (FIFOs). Agents can be seen as independent processes, ie sequential programs, that interact through reading (resp. writing) data from (resp. to) their input (resp. output) FIFOs. There exists a large panel of dataflow languages, whose characteristics differ notably. One major point of variability is the scheduling of agents and their communications. There is indeed a continuum from the synchronous dataflow languages like Lustre [1] or Streamit [2], where the scheduling is fully static, to general communicating networks like KPNs [3] or RVC-Cal [4] where a dedicated runtime is responsible for scheduling tasks dynamically, when they *can* be executed.

So far, parallelization techniques for dataflow programs have focused on taking advantage of the decomposition in agents, potentially duplicating some agents to have several instances that work on different data items in parallel [5]. In the presence of big agents, the programmer is left with the splitting (or merging) of these agents by-hand if she wants to further parallelize her program, or at least give this opportunity to the runtime, which in general only sees agents as non-malleable entities. In the presence of arrays and loop-nests or, more generally, some kinds of regularity in the agent's code, we believe however that the programmer would benefit from automatic parallelization techniques such as those implemented within polyhedral compilation tools. We propose to use such sequential code parallelization techniques to parallelize the sequential code that describes an agents' behavior.

The work reported in this paper aims at demonstrating the practical advantage of combining the dataflow paradigm with the polyhedral optimization framework. We empirically demonstrate this by building a proof-of-concept tooling approach, using existing tools on existing languages. This paper's contributions are:

- A tentative approach combining dataflow programming with polyhedral compilation in order to enhance program parallelization by leveraging both inter-agent parallelism and intra-agent parallelism (ie regarding loop nests inside agents).
- An implementation of this approach using a state-of-the-art dataflow language as well as classical polyhedral tools.

Part of this work was carried out while Lionel Morel was with INSA Lyon, Université de Lyon.

- An evaluation of the approach on several example benchmarks.
- A discussion about the costs and benefits of the approach.

The rest of the paper is organized as follows. Section II introduces background on both dataflow programming languages (II-A), including the ΣC dataflow language, and the polyhedral model (II-B). Section III explains our proposition, namely conjointly parallelizing loop nests that are found *inside* dataflow agents. Section IV describes our the experimental setting, the toochain that supports the ΣC language and the Pluto tool we use for automatic loop parallelization. Section V describes our experiments. These comprise three applications: a toy matrix multiplication, an implementation of the Deriche algorithm for image edge detection and a simple instance of artificial neural network used for image classification. Finally, section VIII concludes and draws perspectives to this work.

II. Background

A. Data Flow Programming

First dataflow programming models were proposed in the seminal works of G. Kahn [3] and J. Dennis [6]. The main concept consists in decomposing the application one wants to build as a set of independent processes (we will call them agents in the rest of the paper) that communicate solely through First-In-First-Out (FIFO) channels.

Since the mid 70s, models and languages following the same philosophy have flourished. A main characteristic that distinguishes these models is the ability to determine statically (or not) how many data tokens are exchanged by agents on the FIFO channels. When these quantities can be determined at compile-time, these models are named static (Boolean, static, cyclo-static, etc). This gives nice properties to such programs, namely boundedness of memory usage and static schedulability. When data rates are not fixed statically, agents need to be scheduled at runtime. Such languages are qualified as dynamic dataflow.

ΣC: Our experiments are applied to programs written using the ΣC programming language [7]. ΣC implements the Cyclo-Static Dataflow (CSDF) model [8] where data rates of agents are known statically and can change periodically.

Throughout this paper, we use the example of the Deriche algorithm [9], an optimal edge-detection algorithm for discrete bi-dimensional images. Here we only describe the structure of the program. Written in ΣC, the implementation is composed of 6 agents that are connected as shown in Figure 1. Each of these agents applies a transformation to an image-size matrix. Links between agents represent FIFO channels, which are the unique mean that can be used to share data among agents.

An agent's behavior is defined using a DSL in which one describes:

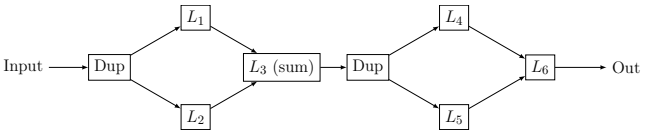


Figure 1. Representation of the dependency graph in Deriche's algorithm.

```
agent L1()
{
  interface
    {
      in<float> input;
      out<float> output;
      spec{ input[HEIGHT]; output[HEIGHT] };
    }

  void start (void) exchange (input in[HEIGHT], output o[
       HEIGHT])
  {
    float ym1 = 0, ym2 = 0, xm1 = 0;
    for(int i = 0; i<HEIGHT;i++){
      o[i] = a1 * in[i] + a2 * xm1 + b1 * ym1 + b2 * ym2;
      xm1 = in[i];
      ym2=ym1;
      ym1=o[i];
    }
  }
}
```

Figure 2. ΣC implementation of the `L1` agent.

- the interface of the agent (its input and output ports)
- a set (at least one) of functions that describe what happens when the agent is activated.

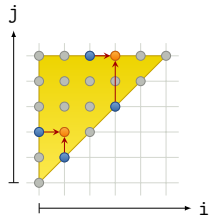
The code given in Figure 2 defines the agent `L1` to have one input (resp. output) flow on which the agent, everytime it is triggered, will read (resp. write) `HEIGHT` float values. The start function is triggered at each of `L1`'s activations and performs some computations to define the value to be written on its output flow. Here, the computation consists in a for loop iterating over all the elements read from (resp. written to) input and output ports.

The compilation process implemented in the ΣC toolchain is in charge of transforming these dataflow language concepts into runtime concepts. Agent activities are translated into sequential programs encapsulated into runtime threads. Depending on the target architecture, FIFO communication channels can be translated into efficient shared-memory implementations or into distributed communication mechanisms. The ΣC runtime is then in charge of allocating memory regions and scheduling the activities corresponding to agents, either relying on an underlying operating system or through dedicated scheduling policies.

B. Polyhedral model

The second building block of our approach is a compilation and optimisation framework for imperative kernels that perform intensive computations, namely, the polyhedral model. This framework [10] provides exact dependence analysis information where statement instances (i.e., statements executed at different loop iterations) and array elements are distinguished. The exact dependence information

obtained through this analysis and the use of linear programming techniques to explore the space of legal schedules [11] is what constitutes the base of the polyhedral model for loop transformations.




```
for (i=0; i<N; i++)
  for (j=i; j< ; j++)
S:  A[j] = f(A[i], A[j]);
```

Figure 3. An example of polyhedral representation. Loop nests that fit the polyhedral model can be viewed as mathematical (constraint-based) objects, which can also be visualized geometrically. Dependencies here are depicted as arrows from producers to consumers.

Figure 3 illustrates the polyhedral representation with an example. The statement S is executed approximately $\frac{N^2}{2}$ times during the execution of this loop. The triangular region expressed as a set of constraints, called the *domain* of S, represents this set of dynamic execution instances. Accesses to array A from each of these statement instances can be succinctly captured through affine functions of the loop iterators. The dependences are also expressed as a function between two statement instances. From this mathematical description, the framework is capable of extracting a set of valid *schedules* (*i.e.* new ways to execute loop statements), compute tiled, parallel, pipelined versions of the same code, depending on the optimization function (locality, maximum parallelization, ...), and then generate their code.

The "traditional" use of polyhedral techniques in optimizing compilers typically focuses on loop transformations of *polyhedral kernels*. PLuTo [12] is a now widely used push-button tool for automatically parallelizing polyhedral loop nests that tries to optimize locality in addition to parallelization. There is also significant work in data layout optimization for polyhedral programs where analyses are performed to minimize the program's memory requirement [13]. Polyhedral techniques for loop transformations are now adopted by many production level compilers, such as GCC, IBM XL, and LLVM.

Recently, polyhedral techniques are being applied to many different areas besides loop transformations. One natural application of automatic parallelization techniques is in verification of given parallelizations where the tools take parallelized programs as inputs, and use polyhedral analysis to guarantee the absence of parallel bugs [14], [15]. Moreover, a huge amount of research is done to extend the applicability of polyhedral optimizations [16], such as hybrid techniques that mix static compilation and dynamic tests [17]–[19].

The polyhedral model is becoming a standard to reason about regular programs and to effectively perform kernel optimizations inside SCOPS (static control parts of a program, where it is effectively applicable).

Our proposal, described next, is an integrated language approach to use both task and instruction parallelism within a unique setting.

III. General Approach

This section describes our arguments in favor of a new integrated approach that would mix in the same language dataflow idioms to express dataflow parallelism and " la polyhedral" model to express instruction parallelism.

The advantages of a dataflow approach are:

- Streaming algorithms [4], scientific workflows [20], and many other applications, are already thought as agents communicating data through FIFOs.
- From Kahn [3] proposition to the Tensorflow [21] language, a wide variety of (more or less semantically well-founded) languages share the common idea of letting the programmer express as much parallelism as he can, dataflow being one (or the main) of the paradigm proposed.
- In some well-defined variants like the Static/Synchronous DataFlow [22], the problem of statically scheduling agents with a maximum parallelism is shown to be decidable. It is the case of the ΣC language we use in our experiment.
- In some other variants, the user benefits from clever runtime supports.

But this is theory. All these approaches compile separately (and agnostically) the code inside the agents, thus they may miss some opportunities for static compilation as well as static or dynamic scheduling, like:

- the intrinsic parallelism of a given agent;
- the organization of data inside a given agent.

With this information made explicit, the dataflow compiler and scheduler would take the decision of splitting agents, pipelining or merging them, in order to exploit all the intrinsic parallelism of a given application. We can even imagine being able to express and schedule several applications running on the same parallel machine [23].

However, up to now, the developer was in charge of writing "well-optimized" agents. This approach is clearly error-prone, and lack flexibility. Moreover, the agents' code then begins to be overspecialized and not portable anymore. We propose to solve this issue by letting the programmer write the application she has in mind, and have the compilation chain harness the application's potential parallelism.

This integrated approach gives opportunities to the developer to express her knowledge of the application she designs. However the theoretical counterpart would not be trivial as it involves being able to express and compute hierarchical schedules so that the actual aggressive compilations done statically do not interfere with the task scheduling of agents, whether scheduled statically or at runtime. Indeed, there is a huge design space from explicitly expressing all parallel

```
agent L2()
{
  interface
    {
      in<float> input;
      out<float> output;
      spec{ input[HEIGHT]; output[HEIGHT] };     }

  void start (void) exchange (input in[HEIGHT], output o[
       HEIGHT])
  {
    float yp1 = 0, yp2 = 0, xp1 = 0, xp2 = 0;
    for(int i = HEIGHT-1; i>=0;i--){
      o[i] = (a3 * xp1 + a1 * xp2 + b1 * yp1 + b2 * yp2) ;
      xp2 = xp1;
      xp1 = in[i];
      yp2 = yp1;
      yp1 = o[i]; }
  } }

agent L3()
{
  interface
    {
      in<float> input[2];
      out<float> output;
      spec{ input[][HEIGHT]; output[HEIGHT] }; }

  void start (void) exchange (input[] in[HEIGHT], output o
       [HEIGHT])
  {
    for(int i = 0; i<HEIGHT;i++){
      o[i]=in[0][i]+in[1][i]; }
  } }
```

Figure 4. ΣC implementation of agents L2 and L3.

statements as tasks and schedule them independently (with a possibly unreasonable cost) to individual two-steps scheduling of tasks, then sub-tasks (which may be simpler to design but may lose inter-tasks optimization opportunities).

Our long term objective is to go towards such a formal framework to express, compile and run dataflow applications with intrinsic instruction or pipeline parallelism.

Motivating example: As a motivational example, let's look again at the Deriche image transformation application, shown in Figure 1.

This program is intended to deal with images in a pipelined fashion. When L1 and L2 have finished manipulating a first image, the combination phase of their result (agent L3) can start while L1 and L2 are fed with data corresponding to a new image. This program is naturally described in a dataflow manner. More importantly, the succession of phases L1, L2, etc. follows the mathematical description (see [9]).

One of the limitations of dataflow programming as it is done today is that the dataflow compiler sees each agent as a separate compilation unit. It is therefore unable to optimize code across agent's boundaries. As an example, consider the agent L1 of Figure 2. Figure 4 gives the code of agents L2 and L3.

Each of these agents contains a loop iterating on its input data to produce its output data, Intuitively, we would like the compiler to consider these loop nests as a potential source of optimization. It could then decide to 1) fuse actors L1, L2 and L3 as well as 2) apply loop-based parallelization.This

```
agent L123()
{
  interface
    {
      in<float> input;
      out<float> output;
      spec{ input[HEIGHT]; output[HEIGHT] };     }
  float inter[2][HEIGHT];

  void start (void) exchange (input in[HEIGHT], output o[
       HEIGHT])
  {
  // Some declarations and init have been cut here
    if (HEIGHT >= 1) {
      for (t2=0;t2<=HEIGHT-1;t2++) {
        inter[1][HEIGHT-t2-1] = (a3 * xp1 + a1 * xp2 + b1
             * yp1 + b2 * yp2) ;;
        yp2 = yp1;;
        yp1 = inter[1][HEIGHT-t2-1];;
        xp2 = xp1;;
        xp1 = in[HEIGHT-t2-1];;
        inter[0][t2] = a1 * in[t2] + a2 * xm1 + b1 * ym1 +
              b2 * ym2;;
        ym2 = ym1;;
        ym1 = inter[0][t2];;
        xm1 = in[t2];; }
      lbp=0 ; ubp=floord(HEIGHT-1,32);
#pragma omp parallel for private(lbv,ubv,t3,t4)
      for (t2=lbp;t2<=ubp;t2++) {
        lbv=32*t2;
        ubv=min(HEIGHT-1,32*t2+31);
#pragma ivdep
#pragma vector always
        for (t3=lbv;t3<=ubv;t3++) {
          o[t3]=inter[0][t3]+inter[1][t3]; }
      }
    }
  } }
```

Figure 5. ΣC Ideal implementation of an actor L123 resulting in the fusion of agents L1, L2, L3 as well as loop based parallel optimizations.

would produce an agent L123 such as the one of Figure 5 where for instance a parallel loop has been identified. Of course, this is one possible transformation and our ideal compiler would have many possibilities to choose from.

Proposed approach: This paper is a first proposition of a combination of the polyhedral model framework with a production dataflow language, namely ΣC. Through the implementation of three non trivial case studies, we explore the relationships between dataflow, pipeline and instruction parallelisms, how they interfere at compile time and at runtime.

This relationship is non trivial to predict, since the polyhedral model is able to capture a potentially infinite parallelism that neither the ΣC compiler, or runtime, is capable to reason on. The scheduling of a "parallel" agent has to make a compromise between the number of resulting "sub agents" and the intrinsic cost of having too many agents to orchestrate at runtime. Moreover, the benefit of the potential parallelism may be lost if there is too many FIFOs that increase memory pressure.

In essence, what we state in this paper is that the polyhedral model and the dataflow paradigm are going toward two different directions that will be reconciliated only if we express all their capabilities in a unique formal framework. This paper is a first experimental step to validate

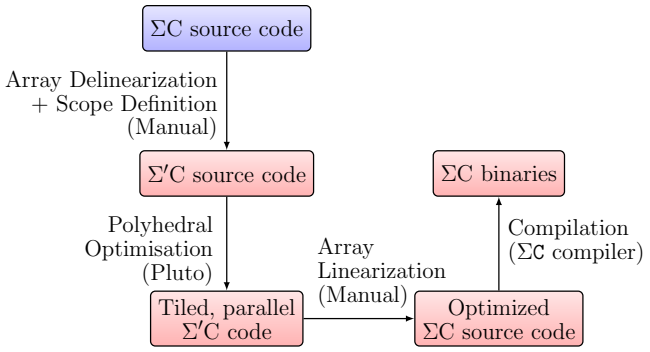


Figure 6. The complete toolchain.

the potentiality of the approach.

IV. Implementation

To experiment on the idea of combining polyhedral loop parallelization with agent-level parallelism, we have combined the ΣC language and the ΣC toolchain together with the Pluto optimizing tool.

The complete workflow is depicted in Figure 6. This workflow has been applied identically on all our running examples. First we have written the original ΣC program, defining agents, some with loop nests as required by the application's implementation choices. As the ΣC language only allows one-dimensional arrays, we then have delinearized this version (this could have been done automatically with a technique adapted from [24], for instance) and added Pluto annotations on loop nests for which we have identified a potential benefit.

The loop nests are then fed to Pluto individually. For each nest, Pluto returns a semantically-equivalent loop nest along with OpenMP annotations. The original loop nest is then replaced by the new one in the ΣC agent. The ΣC program is then compiled to C and then through GCC to an executable program. We compare the performance of this program with that of the one obtained by compiling the original ΣC program. The annotation needed by Pluto to operate on loop nests is straightforward, consisting, as shown in Figure 8 of simply adding `scop/endscop` pragmas around the loop nests that need to be parallelized.

V. Experiments

A. Use cases

We demonstrate our proposal on three non-trivial example programs.

The first one is the Deriche edge detection implementation already described in Section II-A. This application comprises several agents among which six include loop nests on which polyhedral compilation can be applied.

Our second program is an instance of an artificial neural network (ANN) program that can be used to classify images. ANNs are composed of layers that are pipelined . Each layer comprises a static number of perceptrons that produces a single dataflow output while reading a number of input flows coming from the preceding layer. The architecture of such an ANN is depicted in Figure 7.

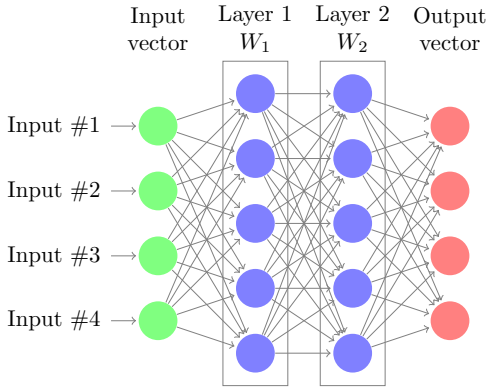


Figure 7. Graphical representation of layers

```
agent Layer(const int columns, const int rows,
    const double w[rows][columns],  const int activ_idx){
  interface
    {
      in<double> input;
      out<double> output;
      spec{input[rows];output[columns]};
    }
  // ...
#pragma scop
  for(int i = 0; i<columns;i++){
    for(int j = 0; j<rows;j++){
        out[i]+=in[j]*w[j][i];
      }
  }
#pragma endscop
  fns_activ[activ_idx](columns, out);
}
```

Figure 8. Example of a layer agent.

Figure 8 shows the ΣC implementation of a layer agent that gathers the activity of *column* perceptrons. Each perception j of a given layer computes a linear combination of its input $in[j]$ with its weight configuration $w[..][j]$. All the computations are gathered into a unique loop nest of depth 2. At the end, a non linear function (`fns_activ`) is applied to the output vector.

This network corresponds to the decision phase of the ANN only. The training of our ANN was performed with another program that exports the weights and the settings of the trained network. This is out of the scope of this paper, but suffice it to say that this training program has been implemented in Python, using the Keras library [25]. The specification of the network (weights and settings) is written by the python training program to a plain-text file. This file is then parsed, at compile time, by the root graph of the ΣC program. It specifies the number of layers, and for each layer, its size, its weight matrix, and the activation function to use. Hence, the whole structure (level of task parallelism, depth) of the ΣC program is configurable by the learning phase.

Finally, we implemented a block-based matrix multipli-

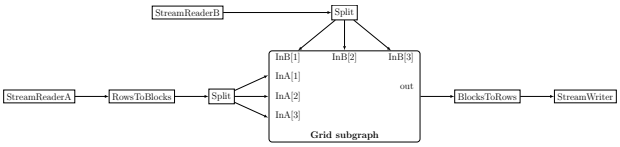


Figure 9. General structure of the matrix multiply ΣC program, exposing pipeline parallelism.

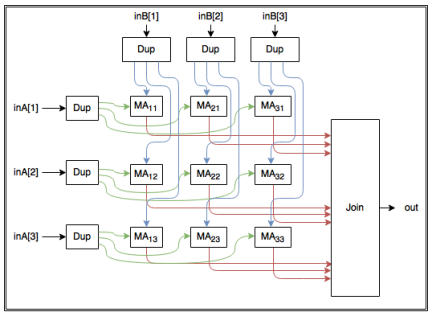


Figure 10. Structure of the Grid agent, exposing data parallelism.

cation whose general structure is depicted in Figure 9. The multiplication is based on a grid of multiply-accumulate agents, see Figure 10. At the highest level of the agent hierarchy, extra agents are needed to organize one of the matrix as blocks to be fed to these multiply-accumulate agents (agents `RowsToBlocks` and `BlocksToRows`). The whole structure of this dataflow program introduces significant potential parallelism. The level of parallelism (ie the number of agents) depends on 1) the size of the A, B and C matrices and 2) the number of blocks the matrices are split into to feed the Multiply-Accumulate agents.

B. Experimental setting

Our experimental results were obtained using two different configurations. The first one, tagged *desktop* in the rest of the paper, is a laptop with an Intel Core i7-7700U processor, providing 4 cores. The second platform, tagged *cluster*, is a Dell PowerEdge R430, with two Intel Xeon E5-2620 v4 CPUs, for a total of 16 cores and 32 threads. This machine was made available through the Grid5000 experiment platform. Both versions where running Fedora 24 (linux kernel version 4.5)

ΣC programs are compiled with a version of the ΣC compiler kindly provided to us by Kalray SA[1]. Several agent-to-thread mapping and agent scheduling strategies are available. In our experiment, we use the `thread` option which generates one POSIX thread for each agent in the application. From the original ΣC programs, the compiler generates C programs, which are then compiled using GCC 6.3.1. Loop nests are parallelized by Pluto (version 0.11.4[2]) which generates semantically equivalent forms annotated with OpenMP pragmas. GCC is then configured with the `-fopenmp` option and uses version 4.5.

[1]http://www.kalrayinc.com/

[2]http://pluto-compiler.sourceforge.net/

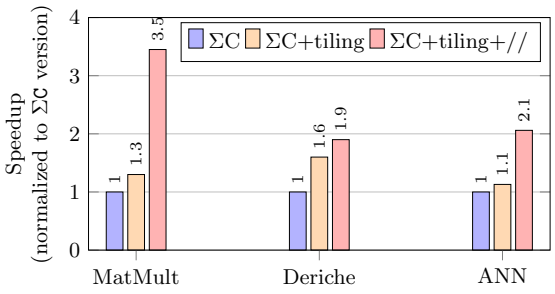


Figure 11. Speed-ups, on the cluster platform.

Concerning applications, we have made the following implementation and experimentation choices. To further study the capabilities offered by our approach, these choices should be extended to consider larger data and more agents as well, where applicable. In the case of the Matrix Multiplication, we have only manipulated matrices of size 1000*1000 elements. The multiply-accumulate grid consists of only 4 agents. This seemed a reasonable trade-off to limit the amount of data exchanged by agents on FIFO channels. The downside is that it limits the potential parallelism of the application. Concerning the Deriche case study, images used where of size 7786*3000 (ie 23 MPx). The number of agents is fixed by the global design choice and does not depend on data sizes, as it is the case for MatMult. Also, we have implemented a second version of Deriche, where agents `L1`, `L2` and `L3` have been manually replaced by agent `L123` of fig. 5. Finally, ANN was used to classify 1000 images of 784 pixels each. The network considered contains 3 dense layers of 784 neurons each and an output layer of 10 neurons. This topology has been chosen in order to have several *layer* agents pipelined as it allows to take advantage of pipeline parallelism.

VI. Results

Figure 11 depicts a comparison of performance results for our three programs running on the full Cluster environment (16 physical cores, 32 threads). Figure 12 shows a similar comparison for the Desktop environment (2 physical cores, 4 threads).

For each program, we use as baseline the timing performance of the *initial* ΣC program where agents have been implemented following the "dataflow informal semantics of the algorithm" (each agent implements a functionality). We then selected one or two agents as candidates for automatic optimization with tiling, and also compare with automatic parallelization of the same agent(s) with Pluto. Results are reported for execution on 4 cores on the desktop platform and 16 cores on the cluster platform.

The most notable result is that with little effort, essentially consisting in adding straightforward Pluto annotations to loop nests, the user can obtain non-negligible performance improvement on the execution time of her dataflow programs.

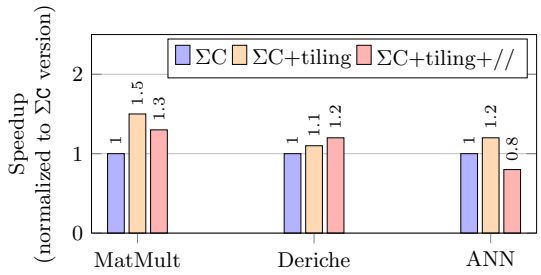


Figure 12. Speed-ups, on the desktop platform.

Performance are improved by tiling alone and can be further *slightly more* improved by using both tiling *and* OpenMP parallelization. Again, using these options is straightforward. We can then surely hope to obtain more interesting results by pursuing our goal of tightly combining polyhedral compilation with dataflow programming languages.

Two performance retreats can be observed, for MatMult and ANN, when using OpenMP on the desktop platform. This is explained by the fact that when using tiling all 4 cores of machines are already fully busy. So when trying to parallelize more by creating more threads at runtime, the OpenMP runtime tends to overload the system's scheduler: global performance suffers from the negative impact of too many thread context switches.

Finally, we show in figure 13 the speed-up (in %) that is gained by applying the transformations suggested in section III over the Deriche usecase. The fusion of agents `L1`, `L2` and `L3` presented in figure 4 leads to a gain in performance of at least 11% when using tiling on the code of the resulting agent. The gain is limited to a few percents when using both tiling and automatic parallelization. At the time of writing, this still needs to be further investigated in order to understand why both optimizations conflict in this manner. We also need to pursue experiments on cluster-like machines.

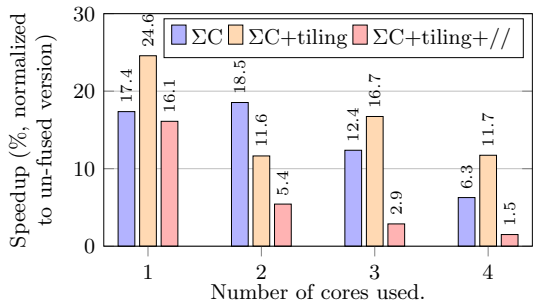


Figure 13. Speed-ups (in %) due to agent fusion.

VII. Related work

We already cited some of our inspirations in Section III. In this section we develop other related works.

In the polyhedral model community, the dataflow paradigm is more of an intermediate representation than an actual programming feature. The polyhedral process networks (PPN) [26], communicating regular processes (CRP) [27], and data-aware process networks (DPN) [28] are generated from a unique sequential program that is fully polyhedral, or at least, where SCOPs have been correctly identified. Further optimizations are made, like hierarchic polyhedral scheduling for CRPs, or efficient implementation of communicating buffers for DPNs [29]. The Compaan compiler [30] transforms applications in the field of signal and image processing (thus inherently dataflow applications [31], written in Matlab) to PPNs, from which they can automatically derive an hardware description for FPGA platforms. Although there is a significant amount of research for deriving efficient control for PPNs with particular constraints coming from the hardware [32], all this process remains polyhedral. Like the synchronous proposition of [33], we believe that these polyhedral works should be carefully integrated in our setting in order to gain benefit of both task and instruction parallelism; however we do not want to restrict the input language to a particular subclass of statically optimizable kernels, like in [34].

The approach described in [35] tries to conciliate what the authors call "macro dataflow" and the polyhedral model in order to benefit from the optimization facilities of the state-of-the-art polyhedral frameworks. This paper is the most related to ours, and is as far as we know the only other attempt at combining explicit agents and polyhedral kernels. Contrarily to our proposition to use an existing language (ΣC) and its ecosystem, they propose to use a more restrictive language (DFGL) to define the "macro" dataflow part as well as kernels, for which they propose a polyhedral compilation ("intra-step" optimization). The coordination of the "inter-step" parallelism is left to the underlying runtime system. The expressivity of the language actually compiled remains however in the classical polyhedral domain, since the scheduling problem is resolved by encoding the whole graph (macro and micro parts) as polyhedral dependencies that are solved (and scheduled) classically. We believe that a more general language approach should at least be able to have the same polyhedral expressivity, but be general enough to express non-regular behaviors inside and outside agents.

Finally, the authors of [36] propose an execution model for single program multiple data (SPMD) on GPUs, based on a polyhedral model based formulation. They propose a way to extract "thread parallelism" from actual (non fully polyhedral) applications. This approach is more a runtime approach than a language approach, and is specific to GPUs. In [37], the authors propose a polyhedral-based precompilation phase for their runtime system $DAGuE$, in order to expose data exchange information that is further used by the runtime. Our approach is more "friendly" to the compiler since the SigmaC language makes some of these

communications explicit. However both approaches share with us the idea of the integration of polyhedral techniques inside more pragmatic compilers and runtime that target full applications.

VIII. Conclusion

We have proposed an approach for parallelizing ΣC programs that takes advantage of the language's constructs to deal with task, pipeline and data parallelism *and* uses polyhedral compilation techniques to further parallelize loop nests inside the application's agents. The approach is validated with a set of non-trivial case studies. These case-studies show, that the use of polyhedral compilation to parallelize the internals of some agents increases programs' performance by a factor between 1.3 and 3.5, depending on the application and the parallelization technique used on loop nests. We also show in one example case that the combination of agent fusion and loop optimization can improve performance by 11 %.

These results have been obtained with a simple experimental approach only using off-the-shelve tools. These result encourage us to pursue research on combining expertise from dataflow programming languages and polyhedral compilation. Our long term objective is to go towards a formal framework to express, compile and run dataflow applications with intrinsic instruction or pipeline parallelism.

We plan to investigate the following directions:

- A language approach: propose new stream programing models where all kinds of parallelisms are expressed explicitly, and where all activities from code design to compilation and scheduling can be cleanly expressed.
- An experimental approach: explore various areas of applications from classical dataflow examples like radio signal and video processing to more recent applications in deep learning algorithmic. This will enable us to identify some potential (intra and extra) agents' optimization patterns that could be leveraged into new languages idioms.

Acknowledgment

We are grateful to people at Kalray for allowing us to experiment with their ΣC compiler. Some of the experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

This work was also partially funded by the French National Agency of Research in the CODAS Project (ANR-17-CE23-0004-01).

References

[1] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep 1991.

[2] W. Thies, "Language and compiler support for stream programs," Ph.D. dissertation, Massachusetts Institute of Technology, 2009.

[3] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*. North-Holland, 1974.

[4] I. Amer, C. Lucarz, G. Roquier, M. Mattavelli, M. Raulet, J.-F. Nezan, and O. Deforges, "Reconfigurable video coding on multicore," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 113–123, 2009. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5230810

[5] M. I. Gordon, "Compiler techniques for scalable performance of stream programs on multicore architectures," Ph.D. dissertation, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 2010.

[6] J. B. Dennis, "First version of a data flow procedure language," in *Symposium on Programming*, 1974, pp. 362–376.

[7] P. Aubry, P.-E. Beaucamps, F. Blanc, B. Bodin, S. Carpov, L. Cudennec, V. David, P. Dore, P. Dubrulle, B. D. d. Dinechin *et al.*, "Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor," *Procedia Computer Science*, vol. 18, pp. 1624–1633, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050913004730

[8] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *Signal Processing, IEEE Transactions on*, vol. 44, no. 2, pp. 397–408, 1996. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=485935

[9] R. Deriche, "Using canny's criteria to derive a recursively implemented optimal edge detector," *International Journal of Computer Vision*, vol. 1, no. 2, pp. 167–187, Jun 1987. [Online]. Available: https://doi.org/10.1007/BF00123164

[10] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[11] ——, "Some efficient solutions to the affine scheduling problem, II, multi-dimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, Dec. 1992.

[12] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, 2008, pp. 101–113.

[13] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1242–1257, 2005.

[14] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott, "ompVerify: Polyhedral analysis for the OpenMP programmer," in *Proceedings of the 7th International Workshop on OpenMP*, ser. IWOMP '11, Jun. 2011, pp. 37–53.

[15] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat, "Array dataflow analysis for polyhedral X10 programs," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, Feb. 2013, pp. 23–34.

[16] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, ser. CC'10/ETAPS'10, 2010, pp. 283–303.

[17] S. Rus, L. Rauchwerger, and J. Hoeflinger, "Hybrid analysis: Static and dynamic memory reference analysis," *International Journal of Parallel Programming*, vol. 31, no. 4, pp. 251–283, 2003.

[18] S. Rus, G. He, C. Alias, and L. Rauchwerger, "Region array SSA," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '06, 2006.

[19] A. Jimborean, P. Clauss, J.-F. Dollinger, V. Loechner, and M. Juan Manuel, "Dynamic and speculative polyhedral parallelization using compiler-generated skeletons," *International Journal of Parallel Programming*, vol. 42, no. 4, pp. 529–545, Aug. 2014.

[20] J. Travis and J. Kring, *LabVIEW for everyone: graphical programming made easy and fun*. Prentice-Hall, 2007.

[21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[22] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, Jan. 1987. [Online]. Available: http://dx.doi.org/10.1109/TC.1987.5009446

[23] L. Morel, M. Selva, K. Marquet, C. Saysset, and T. Risset, "CalMAR - a Multi-Application Dataflow Runtime (short paper)," in *Thirteenth ACM International Conference on Embedded Software 2017, EMSOFT'17*, Seoul, South Korea, Oct. 2017. [Online]. Available: https://hal.inria.fr/hal-01631691

[24] T. Grosser, S. Pop, R. J., and S. Sadayappan, "On recovering multi-dimensional arrays in Polly," in *IMPACT 2015 - 5th International Workshop on Polyhedral Compilation Techniques IMPACT 2015*, Amsterdam, The Netherlands, Jan. 2015, p. 9.

[25] F. Branchaud-Charron and F. R. Taehoon Lee, "Keras: The Python Deep Learning library." [Online]. Available: https://keras.io/

[26] S. Verdoolaege, *Polyhedral Process Networks*. New York, NY: Springer New York, 2013, pp. 1335–1375. [Online]. Available: https://doi.org/10.1007/978-1-4614-6859-2_41

[27] P. Feautrier, "Scalable and structured scheduling," *International Journal of Parallel Programming*, vol. 34, no. 5, pp. 459–487, Oct 2006. [Online]. Available: https://doi.org/10.1007/s10766-006-0011-4

[28] C. Alias and A. Plesco, "Data-aware Process Networks," Inria - Research Centre Grenoble – Rhône-Alpes, Research Report RR-8735, Jun. 2015. [Online]. Available: https://hal.inria.fr/hal-01158726

[29] C. Alias, "Improving Communication Patterns in Polyhedral Process Networks," INRIA Grenoble - Rhône-Alpes, Research Report RR-9131, Dec. 2017. [Online]. Available: https://hal.inria.fr/hal-01665155

[30] B. Kenhuis, E. Rijpkema, and E. F. Deprettere, "Compaan: deriving process networks from Matlab for embededd signal processing archictures," in *8th International Workshop on hardware/Software Codesign*, ser. CODES'2000, May 2000.

[31] J. T. Zhai, H. Nikolov, and T. Stefanov, "Modeling adaptive streaming applications with parameterized polyhedral process networks," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2011, pp. 116–121.

[32] S. Derrien, A. Turjan, C. Zissulescu, and E. F. Deprettere, "Deriving efficient control in Process Networks with Compaan/Laura," *International Journal of Embedded Systems*, 2008. [Online]. Available: https://doi.org/10.1504/IJES.2008.020298

[33] K. Didier, A. Cohen, A. Gauffriau, A. Graillat, and D. Potop-Butucaru, "Sheep in wolf's clothing: Implementation models for data-flow multi-threaded software," Inria Paris, Research Report RR-9057, Apr. 2017. [Online]. Available: https://hal.inria.fr/hal-01509314

[34] A. Cohen, A. Darte, and P. Feautrier, "Static Analysis of OpenStream Programs," CNRS ; Inria ; ENS Lyon, Research Report RR-8764, Jan. 2016, corresponding publication at IMPACT'16 (http://impact.gforge.inria.fr/impact2016). [Online]. Available: https://hal.inria.fr/hal-01184408

[35] A. Sbîrlea, J. Shirako, L.-N. Pouchet, and V. Sarkar, "Polyhedral optimizations for a data-flow graph language," in *Languages and Compilers for Parallel Computing*, X. Shen, F. Mueller, and J. Tuck, Eds. Cham: Springer International Publishing, 2016, pp. 57–72.

[36] A. Balevic and B. Kienhuis, "A data parallel view on polyhedral process networks," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '11. New York, NY, USA: ACM, 2011, pp. 38–47. [Online]. Available: http://doi.acm.org/10.1145/1988932.1988939

[37] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, and J. Dongarra, "From serial loops to parallel execution on distributed systems," in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 246–257.