Combining dataflow programming and polyhedral optimization, a case study

Romain Fontaine, Lionel Morel, Laure Gonnord

Combining dataflow programming and polyhedral optimization, a case study

Romain FONTAINE, Lionel MOREL, Laure GONNORD

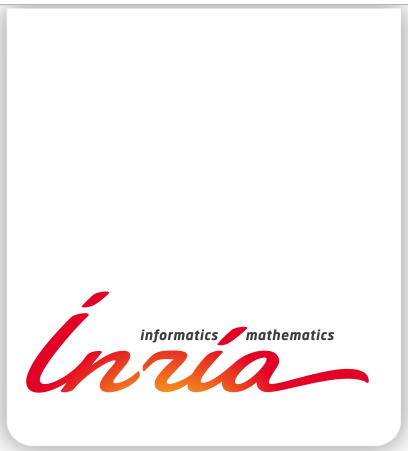
TECHNICAL REPORT

N° 0490

July 2017

Project-Teams SOCRATE and ROMA

ISSN 0249-0803 ISRN INRIA/RT--0490--FR+ENG



Combining dataflow programming and polyhedral optimization, a case study

Romain FONTAINE*, Lionel MOREL †, Laure GONNORD‡

Project-Teams SOCRATE and ROMA

Technical Report n° 0490 — July 2017 — 40 pages

Abstract:
Nowadays, parallel computers have become ubiquitous and current processors contain several execution cores, anywhere from a couple to hundreds. This multi-core tendency is due to constraints preventing the increase of clock frequencies, such as heat generation and power consumption. A variety of low-level tools exist to program these chips efficiently, but they are considered hard to program, to maintain, and to debug, because they may exhibit non-deterministic behaviors. This project focuses on adding an abstraction level in order to have as much performance as possible while not dealing with low-level mechanisms. The approach is based on data flow programming, which allows the programmer to specify only the operations to perform and their dependencies, without actually scheduling them. This project combines this paradigm with the Polyhedral Model, which allows automatic parallelization and optimization of loop nests, in order to make the programming easier by delegating work to the compilers and static analyzers.

Key-words: Parallel Computing, Data Flow Paradigm, Static Data flow, SigmaC, Polyhedral Model, Compiler Optimization, OpenMP, Massively Parallel Processor Array.

* INSA de Lyon
† etc
‡ University of Lyon, LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), F-69000 Lyon, France

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Une étude de cas: optimisations polyédriques pour la programmation flot de données

Résumé : Dans ce rapport nous étudions la combinaison du paradigme flot de données statique avec les optimisations de code “polédriques”, via quelques études de cas.

Mots-clés : Programmation Parallèle, Paradigme Flot de Données, Dataflow Statique, SigmaC, Modèle Polyédrique, Optimisation de Code, OpenMP, MPPA.

1 Introduction

1.1 History

Moore's law states that the number of transistors that can be on a chip doubles every 18 month. Until 2005, processor designers have raised the clock speed in order to improve performance by increasing the number of operations completed in a time unit. This raise caused an increase in power consumption and a greater heat generation.

Since 2005, clock speed could not be increased efficiently anymore, therefore manufacturers started to increase the number of execution cores on a single chip. In 2017, most CPUs are multicore, with typically 2 to 16 cores. Some companies such as Kalray even design Massively Parallel Processor Arrays, which are chips intended for scientific computations, with hundreds or even thousands of cores.

Most programming languages have a sequential approach and were originally designed to be executed sequentially, in only one process, on a single core of execution. Their performance was improved when the clock frequencies were increasing, but since 2005, computer programmers cannot rely on this anymore to increase their programs performance.

For high performance computing, all of the potential of multicore architectures needs to be used. Usually, writing programs that can benefit from the computing power of several processors is not trivial, and there are many different ways to achieve it.

1.2 Terminology

The following notions are important when thinking about parallelizing sequential programs:

Definition 1 (Speedup). *The speedup is the ratio between the execution time of the sequential program and the one of the parallelized program. The higher the speedup, the better. If the speedup is 2, the parallel program runs twice as fast as the sequential one.*

$$S = \frac{TSeq}{TPar} \tag{1}$$

Where:

- *S is the speedup*
- *TSeq is the execution time for the sequential version of the program*
- *TPar is the execution time for the parallel version of the program.*

Note: A super-linear speedup (i.e. a speedup greater than the number of processors used) can occur. It can happen with certain types of algorithms, where the order of execution is changed (e.g. when backtracking) or because of cache effects; sometimes, the processor's cache is used more efficiently in the parallel algorithm, resulting in an additional acceleration.

Definition 2 (Amdahl's Law). *Amdahl's Law states that the maximum theoretical speedup achievable when parallelizing a program is limited by the inherently sequential part of the program.*

$$Smax = \frac{N}{(Pseq * N) + 1 - Pseq} \tag{2}$$

Where:

- *Smax is the theoretical maximum speedup that can be achieved.*

- *Pseq is the proportion of the program that is sequential (i.e. that cannot be parallelized).*
- *N is the number of processors used.*

Definition 3 (Throughput)**.** *The throughput is the number of operations completed in a fixed amount of time.*

Definition 4 (Latency)**.** *The latency is the time taken to complete one operation.*

1.3 Types of parallelism

The three following types of parallelism can be used to improve performance, when parallelizing algorithms:

- *Task* parallelism: Running different tasks (i.e. sequences of operations) on the same data. The amount of parallelism depends on the number of independent tasks to be performed. Figure 1 shows an example of two independent tasks (tasks 2 and 3), which can be performed at the same time.
- *Data* parallelism: Running the **same task** on **different pieces of data**. The amount of parallelism depends on the size of the data.
- *Pipeline* parallelism: Once there is enough data in the pipeline, **several stages of computation can be active at the same time**. Pipeline parallelism can be seen as an assembly line: each worker's state depends only on its predecessor(s), and it is either working or waiting for work to arrive. Pipelining does not reduce the total time of production of a single input, but it allows to increase the throughput of the production line. In the example of Figure 2, when operating on a stream of data, the following can happen (assuming that each task takes one unit of time to be completed):
  - t=1
    - Task 1 is performed on D1
  - t=2
    - Task 1 is performed on D2
    - Task 2 is performed on D1
  - t=3
    - Task 1 is performed on D3
    - Task 2 is performed on D2
    - Task 3 is performed on D1

  Where D1 is the first piece of data of the stream, D2 the second one, and D3 the third one.

  It is important to note that the application must be composed of a sequence of steps in order to benefit from this kind of parallelism. The grain of the pipeline greatly depends on the nature of the algorithm. An image processing algorithm may only need to process the image line by line, which allows a relatively fine-grain pipeline parallelism. On the other hand, when multiplying a vector and a matrix, all of the input data must be available in order to compute the result. This grain is significantly coarser than in the previous case, and the application will need to work on a stream of inputs in order to benefit from it.

  The amount of parallelism depends on the size of the pipeline (i.e. on the number of steps in the sequence).

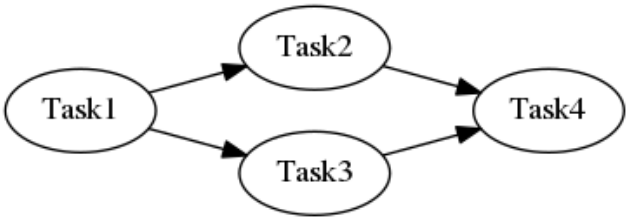


Figure 1: Task parallelism example (Tasks 2 and 3)

Figure 2: Pipeline parallelism example

1.4 Hardware

1.4.1 CPU

Central Processing Units are suited for general purpose programming and are optimized for sequential execution. They often provide several cores, have few constraints, and a large part of the chip's silicon is focused on branch prediction, out-of-order execution, deep pipelines, and large cache systems to reduce the impact of memory accesses on performance.

1.4.2 GPU

Graphical Processing Units can be used to do general purpose computing as well, but they are optimized for highly parallel execution, with thousands of execution cores. They require writing specific code and a significant amount of work is needed to adapt a program to be executed on a GPU. They also impose hard constraints, such as synchronization at the level of the instruction. This constraint makes it more powerful but less expressive, strongly limiting the types of algorithms that can efficiently be parallelized on those chips.

1.4.3 MPPA

Massively Parallel Processor Arrays, also known as *MPPAs*, are integrated circuits that contain hundreds or thousands of processing units and RAM memories. Their highly parallel structure aims to bring more computing power with a lower energy consumption. MPPAs are often used in low-latency applications, especially for stream handling, such as video and image processing, cryptography, and signal processing.

They can usually be programed with standard languages such as C and C++, in order to reuse source code. However, it is still a considerable investment because work is required to parallelize algorithms at such a scale. It is worth mentioning that not all algorithms are suited for an execution on this hardware, because they need to be massively parallelizable.

The company Kalray has designed a MPPA that holds 256 cores, and Adapteva produces a version with 1024 processing units. One of the main selling point of manufacturers is to provide an abundance of execution cores that are more flexible and easier to program than GPUs.

1.5 Parallelization techniques

There is a variety of techniques used to parallelize algorithms, and the following presents the most common ones. Some of them require the programmer to explicitly create parallelism (Threads, Message Passing, OpenMP), some require a change of paradigm (Data Flow), and some others rely on static analysis of programs (Polyhedral model), without forcing the programmer to explicitly handle parallelism or to significantly alter the source code.

Several of the following techniques may be used at the same time, when, for example, programming for execution on a cluster. *Message Passing* can be used for communication between the nodes of the cluster, and *Threads* can be used to parallelize the computations of the nodes.

1.5.1 Programming models

1. Threads

   A thread contains a sequential program that can be executed and scheduled independently by the operating system. A process can create threads, which are able to communicate through shared memory, and can be used to achieve parallelism.

   When parallelizing non-trivial algorithms, communication is often needed, and it can be achieved with shared variables. In order to prevent concurrent accesses from breaking consistency in the program's memory, synchronization mechanisms are needed. These mechanisms are not trivial to use; when not used properly, the programs can suffer from race conditions (i.e. have a non-deterministic behavior). The programs usually have to be altered significantly to get performance improvements, and by doing so, their ease of maintenance decreases.

   OpenMP is a tool that adds a layer of abstraction above the threads, and allows the programmer to parallelize programs with only a few compiler directives. Both data and task parallelism can be achieved, through for loops and the definition of independent tasks. This abstraction allows parallelization and performance improvements without significantly modifying the source code, which is positive for development costs and maintenance.

2. Message Passing

   Message passing is a standard for communication between different processes, which can be used both in shared memory and distributed memory contexts. This paradigm is widely used to develop large scale parallel applications, even though its programming costs are still relatively high, because the source code must be transformed radically to be parallelized.

3. Data Flow Programming

   The Data Flow programming paradigm allows the programmer to specify the flow of data of programs instead of the flow of instructions, as it is usually done in most imperative programming languages.

   This programming model is therefore inherently parallel and allows the computation of several tasks at the same time, without having to use low-level synchronization mechanisms.

   The variety of programs that can be expressed in this paradigm is not as important as in imperative programming with threads, but the constraints allow the compiler to do more of the work, thus making programs more portable and easier to maintain.

1.5.2 Automatic parallelization

The Polyhedral Model is an approach for automatic program parallelization, which allows to parallelize programs without any effort required from the programmer.

It is a compiler optimization based on integer linear programming and parametric linear algebra. This mathematical approach provides an abstraction to represent **nested for-loops** and their **dependencies**. This representation can be used to change the order of execution in order to parallelize and improve data locality, without altering the output of the program.

Some compilers, such as *LLVM* and *GCC* are using this approach for optimization and automatic parallelization. Some external tools based on the Polyhedral Model are also available. *Pluto* is one of these optimizers, and works at the source code level, by taking lightly annotated sequential C code and outputting parallelized and optimized C code. The annotations only specify the parts of the program (i.e. the loop nests) to be processed.

1.6 Aim of the Project

In this project, two seemingly promising parallelization techniques are combined. On the one hand, automatic **for loop optimization and parallelization** through the polyhedral model. On the other hand, **data flow programming paradigm**, with the SigmaC programming language.

Since SigmaC is based on the C programming language, source-to-source polyhedral optimizers such as Pluto can be used to further optimize and parallelize SigmaC source code.

At first, the sections of code to optimize will be chosen manually, and the performance will be evaluated. The performance improvements will be evaluated, on Kalray's MPPA, if possible. If this proves to be beneficial, such a tool could be integrated to the SigmaC toolchain in order to benefit from both the parallelism from the Data Flow paradigm, but also from data parallelism found in loop nests.

2 Background

2.1 Data Flow Programming

2.1.1 Motivations

The data flow paradigm has been designed to focus on data and to make its flow explicit. It consists of a directed graph of agents, which perform computations on streams of data. The dependencies are therefore explicit, and an agent is executed when all of its dependencies are available. Each agent has an internal state and is independent from the global state of the application.

This paradigm allows to avoid the need of explicit synchronization, which is error-prone and hard to debug. It also provides better encapsulation of the state of the program's components and provides a higher abstraction level, compared to threads or message passing which can be tedious to implement and maintain.

Data Flow Programming allows the programmer to focus on the computations to do and their logical order instead of the scheduling of the parallel execution itself.

2.1.2 Principles

The data flow paradigm allows the programmer to create **agents**, which are autonomous entities. They are linked to other agents through queues, these queues being their only means of

communication. The agents depend only on their **internal state**, and their inputs from other agents. They are independent of the global state of the application. They can be considered as *black boxes*, only connected to other agents by a set of well-defined inputs and outputs. Since memory sharing is restricted to queues, a better encapsulation level is achieved, and it is easier to reason about and maintain the programs.

The programmer creates a directed graph, where the nodes are the agents and the edges the one-way queues between them. Since communication and dependencies are explicit, the programmer does not need to handle synchronization manually. An agent starts its operations as soon as its dependencies are satisfied. This model is therefore inherently parallel.

The program is expressed at a higher level of abstraction and it allows the programmer to make a generic program that a compiler will map as efficiently as possible onto any architecture. Consequently, the development and maintenance costs are hopefully lower than those for a specific implementation using lower-level mechanisms such as threads, for example.

2.1.3 Types of data flow paradigms

Several variations of this paradigm are available. **Static Data Flow** is more restrictive, but allows more compile-time analysis and optimizations, and offers more guarantees. **Dynamic Data Flow**, on the other hand, relaxes some constraints, which increases the expressiveness of the languages, but prevents some compiler optimizations and allows issues to appear, such as deadlocks and buffer overflows. **Cyclo-Static Data Flow** lies in the middle by combining features from both sides.

1. Static Data Flow

   In static data flow, there is an important constraint: at compile-time, the number of inputs and outputs, consumed and produced by an agent must be known and have to be constant.

   This allows the compiler to infer useful information for scheduling, such as the size of the queues in order to avoid deadlocks. This also ensures that a given program will be able to run with a bounded memory usage.

2. Cyclo-Static Data Flow

   As in Static Data Flow, the number of tokens produced and consumed by an agent must be known at compile-time. But the agent can specify a sequence of behaviors and repeat it periodically. This allows more complex behaviors than Static Data flow while still providing some constraints to the compiler.

   The SigmaC programming language belongs to this paradigm [**?**].

3. Dynamic Data Flow

   In this paradigm, the constraint of the fixed number of tokens produced and consumed is relaxed. This allows to express more programs, such as compression algorithms that can produce or consume arbitrary numbers of tokens. On the other hand, it allows to implement programs that may contain deadlocks or that may overflow the communication queues.

2.1.4 SigmaC

1. Introduction

SigmaC is a Cyclo-Static Data Flow programming language developed by a French company called Kalray, based in Grenoble, France. It is built upon the C programming language, in order to enable partial code reuse between C code and SigmaC [**?**].

Kalray provides a SigmaC compiler, which is able to compile programs for several architectures, including its Massively Parallel Processor Array (MPPA). It allows the user to provide information about the target architecture such as the number of processors, the memory hierarchy, and the topology, in order for the compiler to map and schedule the agents on the execution cores as efficiently as possible.

2. Simple example

   An agent is a processing unit that reads data from a queue, processes them, and writes back data to another queue. In the source code, it is defined with the keyword *agent*. It can accept compile-time parameters, and define variables as well as functions (in the same way as objects in an object-oriented language).

   A sample agent is defined with the code listed in listing 5. This agent has two parameters (width and height), defining the size of the matrix it handles. These parameters must be known at compile-time.

Listing 1: Declaring an agent

```
1 agent transpose(int width, int height)
```

It then specifies an *interface*, which defines the communication behavior of the agent. First, a list of input and output *ports* is declared, respectively with the keyword *in* and *out*. Each of these ports have a data type, specified with a template-like notation, and a user-defined name. It is important to note that an agent might have 0 or several input ports, and 0 or several output ports.

Listing 2: Declaring the agent's ports.

```
1 interface
  {
3   in<float> input;
    out<float> output;
5   [...]
  }
```

Then, the *spec* block outlines the behavior of the agent, on each *iteration*. For each channel, the number of tokens, either consumed or produced is specified. Here, the port called *input* will deal with *width*height* tokens (/float/s, in this case). The port called output will have the same behavior.

Listing 3: Declaring the agent's specification.

```
  interface
2 {
    [...]
4   spec{input[width*height];output[width*height]};
  }
```

The function *start()* defined inside the agent is its entry point. The C function notation is extended, with an optional *exchange* keyword, followed by more parameters. The types of these parameters must be the names of the ports previously defined. The name of the parameter is defined by the user, and the array size must match the one defined in *spec*, inside the *interface* block.

Listing 4: Declaring the start function.

```
1 void start() exchange(input in[width*height], output out[width*height
    ])
```

The *exchange* keyword makes a function able to use ports of the agent, for consuming data, producing data, or both. The ports that a given function can use are specified with a parameter-like notation, after the *exchange* keyword.

N.B.: The scheduling pass during compilation will assume that the specification matches the real behavior of the agent. Therefore, it is the programmer's responsibility to make sure that it does.

Listing 5: Complete example of a parameterized agent with one input channel and one output channel.

```
1
agent transpose(int width, int height)
3 {
  interface
5   {
      in<float> input;
7     out<float> output;
      spec{input[width*height];output[width*height]};
9   }
  void start() exchange(input in[width*height], output out[width*
      height]){
11   for(int i = 0; i<height;i++){
      for(int j = 0; j<width;j++){
13      out[j*height+i]=in[i*width+j];
      }
15   }
  }
17 }
```

The entry point of a SigmaC application is the *subgraph* named *root*, which is defined in listing 6. Subgraphs are used to encapsulate a composition of agents, to facilitate code-reuse by only exposing a well-defined interface, composed of input and output ports. The *interface* of the *root subgraph* is empty, and its *map* section instantiates agents with the keyword *new* (similarly to Object Oriented Programming), and *connects* their ports.

In this example, three agents are instantiated: two agents for I/O, reading from and writing to memory, and one for transposing the data.

Each input (respectively output) port must be connected to one and only one output (resp. input) port in order for the program to compile. The calls to the function *connect* are the ones creating the links in the data flow graph.

N.B.: When connecting ports, the compiler only checks for coherency in the size of the type, not for actual type-compatibility. Therefore, an output port of type *int* can be connected to an input port of type *float* without warning, if sizeof(int)==sizeof(float).

Listing 6: Simple example of a subgraph instianciating and connecting the agents.

```
2 subgraph root()
  {
```

```
4       interface{}

6       map
        {
8       const int height = 10;
        const int width = 20;
10      agent sr = new StreamReader(0x001, height*width, height*width);
        agent t = new transpose(width,height);
12      agent sw = new StreamWriter(0xfff, height*width, height*width);
        connect(sr.output, t.input);
14      connect(t.output, sw.input);
        }
16  }
```

This example does not benefit from all of SigmaC's abilities because the specification of the agent is trivial; it has a static behavior, always consuming and producing the same amount of data.

3. Advanced example

   Listing 8 represents an agent with a more advanced behavior, benefiting from the Cyclo-Static Data Flow abilities of SigmaC. Its purpose is to multiply two vectors, value by value. Combined with other agents organized in a bidimensional grid, this agent communicates with its neighbors by receiving data, operating on it, and forwarding it, to implement a matrix-matrix multiplication. Once an agent has repeated this step enough times, it can output the result, which is a single value of the resulting matrix. This is not how it would be done in reality, but it is a good example for demonstrating SigmaC's syntax.

   Here, the *spec* section defines the behavior of the agent as a *circular state machine*. This state machine is defined by a sequence of list of ports, inside curly brackets and separated by semicolons. A list of ports corresponds to a state, called a *transition* in SigmaC. The number of tokens produced or consumed on a port can be defined between square brackets, otherwise the default value is one. A given transition can be repeated several times by specifying the number of repetitions before it, between parentheses.

   Listing 7: *Spec* section defining a circular state machine.

```
1  spec{(SIDE){iNorth; iWest; oSouth; oEast};{res}};
```

   In this example, the first transition will be repeated *SIDE* times (*(SIDE){iNorth; iWest; oSouth; oEast}*), by consuming one token on *iNorth* and *iWest*, and producing one token on *oSouth* and *oEast*. The other transition will produce one output token on the port *res* (*{res}*), and will go back to the beginning. One *iteration* will have been completed.

   This behavior must be coherent with the one executed in the start function. Each exchange function called must match this specification in order for the SigmaC compiler to schedule the program as well as possible.

   In this case, the specification is correct, because the start function calls the function *compute() SIDE* times, and then the function *result()*. This is correct because the exchange parameters of compute matches the first transition, and the one of result matches the second one.

   Listing 8: Example of an agent with an advanced behavior. It has a cyclic specification and two exchange functions.

```
1  agent Multiplicator()
```

```
{
  interface
    {
      in<int> iNorth;
      in<int> iWest;
      out<int> oSouth;
      out<int> oEast;
      out<int> res;
      spec{(SIDE){iNorth; iWest; oSouth; oEast};{res}};
    }
  int tmp_result;

  void compute() exchange(iNorth n, iWest w, oSouth s, oEast e){
    s = n;
    e = w;
    tmp_result+=n*w;
  }

  void result() exchange(res r){
    r = tmp_result;
  }

  void start(){
    tmp_result = 0;
    for(int i = 0; i<SIDE;i++){
      compute();
    }
    result();
  }
}
```

In this case, a SigmaC feature called *Implicit Copy* could have been used: The exchange function *compute()* copies the values of the input ports to the output ports, which can be automated by SigmaC using the following *!* syntax:

```
...
spec{(SIDE){iNorth!oSouth;iWest!oEast};{res}};
...
void compute() exchange(iNorth!oSouth n, iWest!oEast w){
  tmp_result+=n*w;
}
...
```

These values can then be modified and can reduce the syntactic complexity of the function to implement. However, according to the SigmaC Language Reference [**?**], this feature is implemented as a copy, and not as pointer sharing yet. It is therefore syntactic sugar and does not provide a performance increase.

4. Generic Agents

   SigmaC provides the following generic agents for handling data flow:

   - Split: Splits a stream of data in packets of a fixed size, in a round-robin way.
   - Join: Merges several streams of data into one, in a round-robin way, for a fixed packet size.

- Dup: Duplicates a data stream, particularly useful for task parallelism (two different tasks performed on the same data).
- Sink: Discards the content of a data stream. This agent is useful because every port must be connected.
- StreamReader: Reads a data stream of a given size from a memory address.
- StreamWriter: Writes a data stream of a given size to a memory address.

2.2 Polyhedral Model

2.2.1 Introduction

As seen in section 1.5.2, the Polyhedral Model is a method for automatically **optimizing** and **parallelizing** sequential programs through **static analysis** [?].

Its scope is limited to specific algorithmic patterns, which are nested for-loops. Inside these loops, the array accesses and the loop bounds must be affine functions of the loop indices, and they may contain conditional statements.

The polyhedral model represents the program as a parametric polyhedron, which is an abstraction that allows a reordering of the loop iterations. This reordering is done to enable as much parallelism as possible, while optimizing the locality of memory accesses and conserving the **data-dependencies** in order to guarantee the same output.

The process of optimizing the locality of memory accesses is called tiling, and it is achieved by dividing for loops into tiles that can fit in the processors cache, as shown in figure 3.

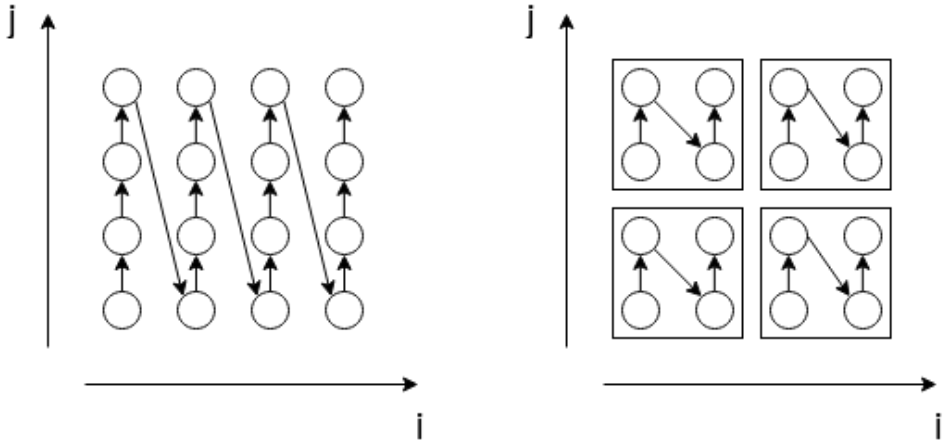


Figure 3: Represents the iteration space (circles) of two nested for-loops (indexes i and j). On the left, the original order of execution is represented. Assuming that there are no dependencies between iterations, they can be executed in the order shown on the left, which is tiled (squares). Memory locality is then improved.

This process can also be illustrated with the simple example of matrix transposition, listing 9 representing the initial code and listing 10 representing the tiled code.

Listing 9: Initial code for matrix transposition

```
1 int n = 100;
  int i, j, a[n][n], t[n][n];
```

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    t[i][j] = a[j][i];
  }
}
```

Listing 10: Tiled code with a tile size of 2*2.

```
int n = 100;
int i, j, x, y, a[n][n], t[n][n];

for (i = 0; i < n; i += 2) {
  for (j = 0; j < n; j += 2) {
    for (x = i; x < min(i + 2, n); x++) {
      for (y = j; y < min(j + 2, n); y++) {
        t[x][y] = a[y][x];
      }
    }
  }
}
```

2.2.2 Pluto

Pluto [1] [?] is a loop nest optimizer designed for the C programming language. It is based on the polyhedral framework and can optimize loop nests through tiling and parallelization with OpenMP directives.

When tiling, the size of a tile can be specified to Pluto in order to target any specific memory hierarchy. Pluto also allows performance gains by parallelizing for loops with OpenMP, when the dependencies in the source code allow it.

We chose Pluto because it is a source-to-source tool, taking C source code as input and outputting C. This is convenient in order to feed the optimized C code back to the SigmaC compiler, as explained more in depth in Section 3.2.

For example, the following code performs a matrix multiplication, assuming that the array C, containing the result, is initialized to 0. The area to be optimized by Pluto is delimited by the directives *#pragma scop* and *#pragma endscop*. The array C is assumed to be initialized for clarity: Pluto could have handled this initialization inside the first loop, but it would have produced a longer code, harder to understand.

```
#pragma scop
for(int i = 0; i<SIZE;i++){
  for(int j = 0; j<SIZE;j++){
    for(int k = 0; k<SIZE;k++){
      C[i][j]+=A[i][k]*B[k][j];
    }
  }
 }
#pragma endscop
```

The following is the previous code, transformed by Pluto, for parallelization and tiling (comments and declarations omitted):

[1] `http://pluto-compiler.sourceforge.net/`

```
lbp=0;
ubp=floord(SIZE-1,32);
#pragma omp parallel for private(lbv,ubv,t2,t3,t4,t5,t6)
for (t1=lbp;t1<=ubp;t1++) {
  for (t2=0;t2<=floord(SIZE-1,32);t2++) {
    for (t3=0;t3<=floord(SIZE-1,32);t3++) {
      for (t4=32*t1;t4<=min(SIZE-1,32*t1+31);t4++) {
        for (t5=32*t3;t5<=min(SIZE-1,32*t3+31);t5++) {
          lbv=32*t2;
          ubv=min(SIZE-1,32*t2+31);
#pragma ivdep
#pragma vector always
          for (t6=lbv;t6<=ubv;t6++) {
            C[t4][t6]+=A[t4][t5]*B[t5][t6];;
          }
        }
      }
    }
  }
}
```

The constant 32 appears in the generated code because the tile size used was 32*32*32. Pluto allows to use customized tile sizes, which should be chosen according to the size of the cache and the amount of parallelism needed.

3 Experiments

3.1 Introduction

It is important to note that the SigmaC compiler does not exploit data parallelism inside an agent's source code. This seems to be a missed opportunity because there exist tools such as Pluto, based on the polyhedral model, that allow automatic parallelization and optimization of sequential source code.

The SigmaC compiler allows to create pipeline parallelism as well as task parallelism automatically. SigmaC could also bring data parallelism and optimize nested for-loops in the code of individual agents, by integrating a tool like Pluto in the toolchain.

The purpose of these experiments is therefore to combine automatic polyhedral optimization (with Pluto) with a data flow compilation (with the SigmaC programming language), with the goal of, hopefully, increasing even more the level of parallelism achieved.

Three sample programs have been implemented in order to combine Pluto and SigmaC. The first one is the classical matrix-matrix multiplication. The second one is the Deriche Edge Detector, and the last one is an Artificial Neural Network.

Table 1 presents a summary of the different types of parallelism found in the programs implemented.

Table 1: Summary of the types of parallelism used in the experiments

Algorithm \ Type of parallelism	Data	Task	Pipeline
Matrix - Matrix Multiplication	X		X
Deriche	X	X	X
Artificial Neural Network	X		X

3.2 Pluto and SigmaC

As seen in section 2.2.2, Pluto only processes the source code located between the directives *#pragma scop* and *#pragma endscop*. Therefore, providing that the section contains only C code and no SigmaC keywords, SigmaC files can be optimized by Pluto. This is often the case because SigmaC keywords are found in agent and function definitions, which are distinct from the loops nests to be optimized.

However, SigmaC agents tend to use linearized array because their communication is achieved by queues (FIFOs). Pluto cannot optimize loop nests accessing to linearized array because it cannot infer the size of their dimensions. This is a well-known problem in compiler optimization because, usually, the compiler's intermediate representations are based on linearized arrays, such as Polly, with LLVM [**?**] .

The following sections present two solutions that can enable Pluto optimization of SigmaC loop nests over linearized arrays.

3.2.1 Temporary transformation

This transformation is manual and consists in modifying temporarily the source code in order to 'delinearize' the array accesses. The source code can then be optimized by Pluto, and Pluto's output is linearized back by the developer. Figure 4 illustrates this process.

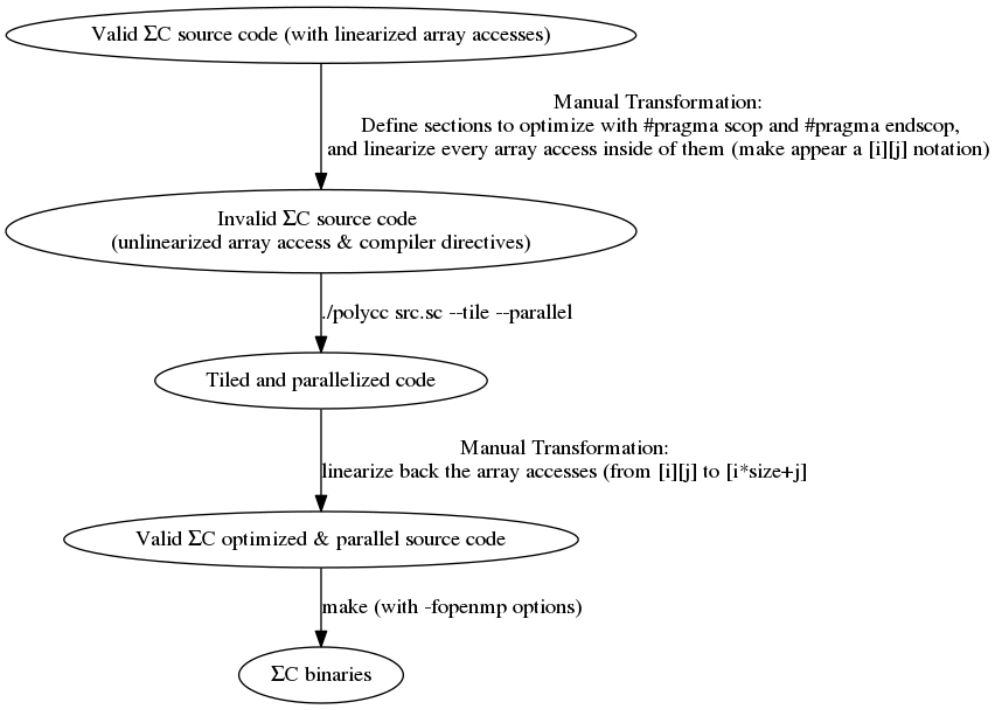


Figure 4: Process of optimizing loop nests in SigmaC code, through temporary modifications of the source code.

This transformation cannot be automated easily in the compilation toolchain and can become tedious to maintain manually.

Example:

- Initial code:

```
agent MyAgent()
{
  interface
    {
      in<unsigned char> input;
      out<unsigned char> output;
      spec{ input[WIDTH*HEIGHT]; output[WIDTH*HEIGHT] };
    }

  void
    start (void) exchange (input in[WIDTH*HEIGHT], output out[WIDTH*HEIGHT
        ])
  {
    for(int i = 0; i < HEIGHT; i++){
      for(int j = 0; j < WIDTH; j++){
        out[i*WIDTH+j] = myFunction(in[i*WIDTH+j]);
      }
    }
  }
}
```

- Delinearized code. It can be optimized by Pluto, but not compiled by the SigmaC Compiler.

```
for(int i = 0; i < HEIGHT; i++){
   for(int j = 0; j < WIDTH; j++){
      out[i][j] = myFunction(in[i][j]);
   }
}
```

- Optimized by Pluto (comments omitted, *polycc file.sc –tile –parallel*):

```
/* Includes and declarations omitted for space */
  if ((HEIGHT >= 1) && (WIDTH >= 1)) {
    lbp=0;
    ubp=floord(HEIGHT-1,32);
  #pragma omp parallel for private(lbv,ubv,t2,t3,t4)
    for (t1=lbp;t1<=ubp;t1++) {
      for (t2=0;t2<=floord(WIDTH-1,32);t2++) {
        for (t3=32*t1;t3<=min(HEIGHT-1,32*t1+31);t3++) {
          lbv=32*t2;
          ubv=min(WIDTH-1,32*t2+31);
  #pragma ivdep
  #pragma vector always
          for (t4=lbv;t4<=ubv;t4++) {
            out[t3][t4] = myFunction(in[t3][t4]);;
          }
        }
      }
    }
  }
```

- Pluto code, linearized manually. This code compiles with the SigmaC Compiler:

```
// [...]
if ((HEIGHT >= 1) && (WIDTH >= 1)) {
  lbp=0;
  ubp=floord(HEIGHT-1,32);
#pragma omp parallel for private(lbv,ubv,t2,t3,t4)
  for (t1=lbp;t1<=ubp;t1++) {
    for (t2=0;t2<=floord(WIDTH-1,32);t2++) {
      for (t3=32*t1;t3<=min(HEIGHT-1,32*t1+31);t3++) {
        lbv=32*t2;
        ubv=min(WIDTH-1,32*t2+31);
#pragma ivdep
#pragma vector always
        for (t4=lbv;t4<=ubv;t4++) {
          out[t3*WIDTH+t4] = myFunction(in[t3*WIDTH+t4]);;
        }
      }
    }
  }
}
```

3.2.2 Using a port array

This method permanently modifies the source code in order to be able to use the double bracket syntax for array accesses. In order to achieve this, the input must be split, and the output must be joined. The performance might suffer as a Split agent must be created and several links must be maintained as well. In this case, each port of the array receives a row of a matrix.

Note: Because this is a 'trick' from the SigmaC data flow paradigm, this method can only add one dimension to the array used. Therefore, it generally does not work for linearized arrays with a dimension greater than or equal to 3.

Example:

- Initial code:

```
agent MyAgent()
{
  interface
    {
      in<unsigned char> input;
      out<unsigned char> output;

      spec{ input[WIDTH*HEIGHT]; output[WIDTH*HEIGHT] };
    }

  void
    start (void) exchange (input in[WIDTH*HEIGHT], output out[WIDTH*HEIGHT
        ])
  {
    for(int i = 0; i < HEIGHT; i++){
      for(int j = 0; j < WIDTH; j++){
        out[i*WIDTH+j] = myFunction(in[i*WIDTH+j]);
```

```
17        }
      }
19    }
}
21
subgraph root()
23 {
  interface
25    {
        spec{};
27    }

29  map
      {
31      agent sr = new StreamReader<unsigned char> (ADDRIN, HEIGHT * WIDTH,
            WIDTH);
        agent sw = new StreamWriter<unsigned char> (ADDROUT, HEIGHT * WIDTH,
            WIDTH);
33      agent ma = new MyAgent();
        connect (sr.output, ma.input);
35      connect (ma.output, sw.input);
      }
37 }
```

- Updated to use a port array: the source code is usable by both Pluto and the SigmaC compiler.

```
1  agent MyAgent()
{
3   interface
      {
5       in<unsigned char> input[HEIGHT];
        out<unsigned char> output[HEIGHT];
7
        spec{ input[][WIDTH]; output[][WIDTH] };
9     }

11  void
      start (void) exchange (input[] in[WIDTH], output[] out[WIDTH])
13  {
      for(int i = 0; i < HEIGHT; i++){
15      for(int j = 0; j < WIDTH; j++){
          out[i][j] = myFunction(in[i][j]);
17      }
      }
19  }
}
21
subgraph root()
23 {
  interface
25    {
        spec{};
```

```
27      }

29    map
        {
31        agent sr = new StreamReader<unsigned char> (ADDRIN, HEIGHT * WIDTH,
              WIDTH);
          agent sw = new StreamWriter<unsigned char> (ADDROUT, HEIGHT * WIDTH,
              WIDTH);
33        agent s = new Split<unsigned char>(HEIGHT, WIDTH);
          agent ma = new MyAgent();
35        agent j = new Join<unsigned char>(HEIGHT, WIDTH);
          connect (sr.output, s.input);
37        for(int i = 0; i<HEIGHT;i++){
            connect(s.output[i], ma.input[i]);
39          connect(ma.output[i], j.input[i]);
          }
41        connect (j.output, sw.input);
        }
43  }
```

3.3 Experiment environments

This section describes the contexts of execution of the experiments. Two radically different environments have been used in order to evaluate the approach at different scales.

In both cases, the programs were run through the SigmaC runtime (using threads and the *fast* mode), which may cause a performance overhead.

3.3.1 Desktop

The first experiment machine is a laptop with an Intel Core i5-5200U processor (2.2 GHz), providing 2 cores and 4 threads.

3.3.2 Cluster

The second experiment machine is from the Grid5000 platform [2]. It is a Dell PowerEdge R430, with two Intel Xeon E5-2620 v4 CPUs (2.10GHz), for a total of 16 cores and 32 threads.

3.4 Matrix-Matrix Multiplication

3.4.1 Introduction

The following describes the implementation of the multiplication operation, for two matrices, in SigmaC. It contains a loop nest that has been optimized and parallelized by Pluto.

This program computes the dot product C←A·B, where A, B and C are square matrices, of size N*N. This operation can be described in the following way:

```
1   int N;
    int A[N][N], B[N][N], C[N][N];
3   for(int i = 0; i<N;i++){
      for(int j = 0; j<N;j++){
5       C[i][j]=0;
```

[2] https://www.grid5000.fr/

```
        for(int k = 0; k<N;k++){
7           C[i][j]+=A[i][k]*B[k][j];
        }
9     }
  }
```

3.4.2 Approach

The agents are arranged in a logical 2D grid. Each agent computes a part of the result matrix (a *block*). In order to do so, it must have a row of blocks from the matrix A and a row of blocks from the matrix B. This can be illustrated by the figures 5 and 6.

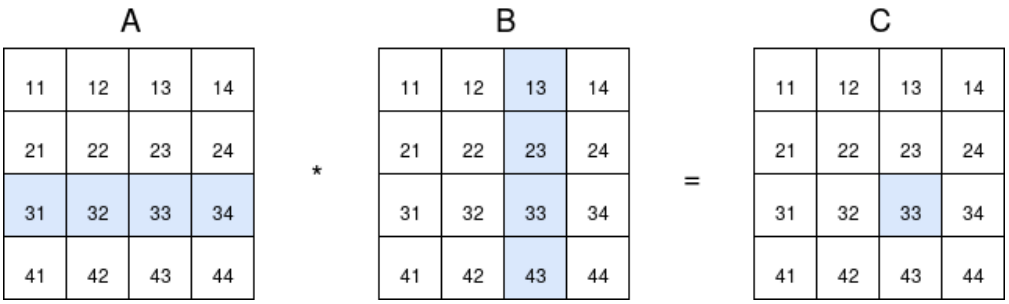


Figure 5: In order to compute the value of the cell C33, the third row of matrix A and the third column of matrix B are needed.

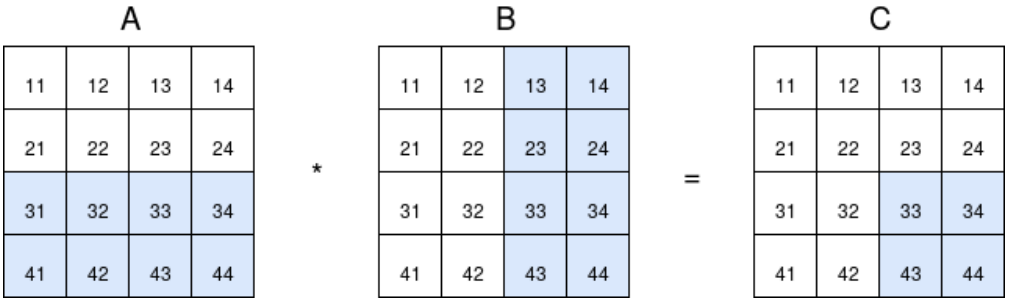


Figure 6: In order to compute the values of the block containing the cells C33, C34, C43, and C44, the third and fourth rows of A are needed as well as the third and fourth columns of B.

Let:

- N be the size of the matrix (N*N elements in total)
- P be the number of multiplying agents

Assumptions:

- Operating on square matrices of 32 bits signed integers.
- $P = n^2, n \in \mathbb{N}$ (i.e. the agents can be organized in a 2D grid).
- N is divisible by $\sqrt{P}$ (i.e. each block has the same size, which is an integer).

3.4.3 Overview

1. Create a square matrix of agents of size $\sqrt{P} * \sqrt{P}$
2. Partition A and B into P blocks each
3. Each agent receives a row-wise stream of blocks from matrix A, and a column-wise stream of blocks from matrix B.
4. Each agent computes a block of the product with a for-loop nest of depth 3, accumulating a partial result.
5. Each agent repeats $\sqrt{P}$ times the steps 3 and 4, accumulating the results in a temporary matrix.
6. The results of every agent are joined, and put back in a square matrix format instead of a stream of blocks.

3.4.4 Parallelism

In this implementation, parallelism is achieved at three different levels. From the finer granularity to the coarser:

- Data parallelism: For-loop parallelization inside an agent's local matrix multiplication.
- Data parallelism: The resulting matrix is divided into independent blocks that are computed concurrently.
- Pipeline parallelism: The program can process a stream of matrices to be multiplied together (e.g. A1*B1, A2*B2, ... AN*BN).

3.4.5 Topology

The topology of the data flow graph, when using 9 agents, or a 3*3 agent grid, is shown in figures 7 and 8.

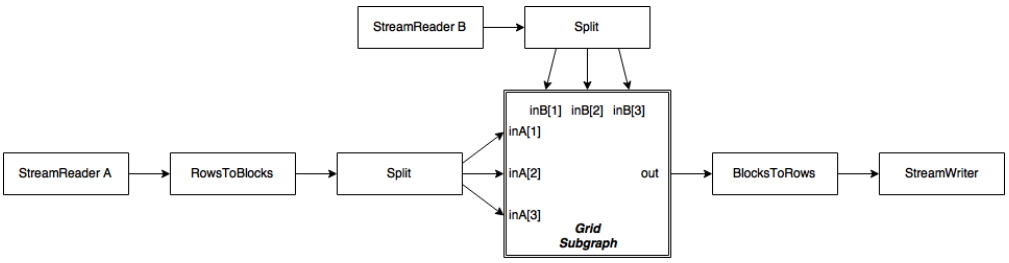


Figure 7: Agents, subgraphs, and data flow in this implementation

1. Agents
   - MultiplyAccumulate: Handles the local multiplication of a block of the resulting matrix.
   - MatrixToHorizontalBlocks: Splits the matrix A into blocks (in a horizontal way).

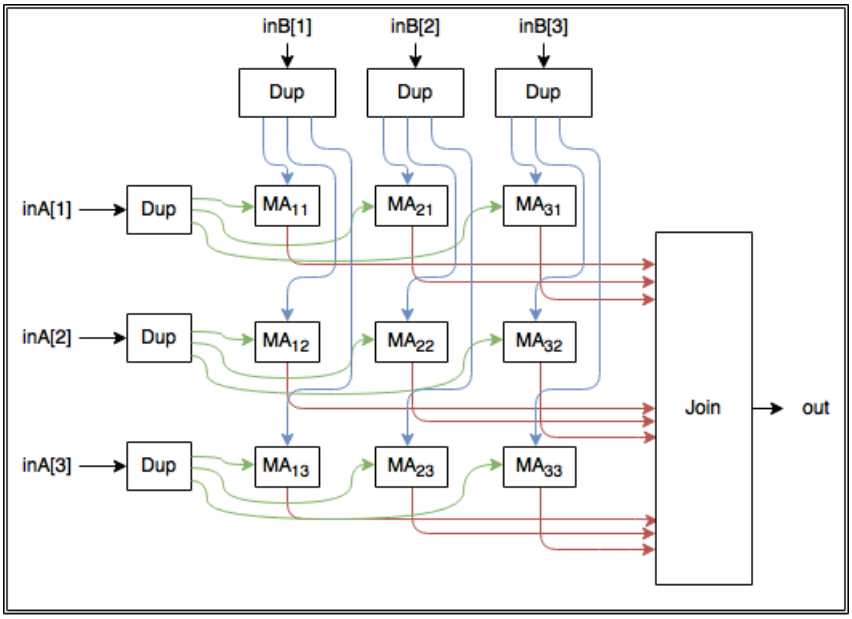


Figure 8: Agents and data flow inside the *Grid* subgraph.

- HorizontalBlocksToMatrix: Joins blocks to recreate matrix C from blocks.

2. Subgraphs
   - Grid:
     - Contains a 2D-array of *MultiplyAccumulate* agents.
     - Inputs: Two arrays of streams of blocks: One array for each matrix, one stream for each row/column of agents.
     - Outputs: A stream of blocks representing the result matrix.

3.4.6 Choices

Several approaches have been studied for this implementation. The following presents these approaches.

1. Systolic array

   This approach assigns one agent to compute each element of the output matrix. This is not an efficient implementation, and it scales poorly (With a matrix size of 30*30, 900 agents are created).

   However, this approach is simple and interesting because of the communication patterns: agents are arranged in a logical bidimensional grid and they receive information from their neighbors, work on it, and then forward it.

   It is a good starting point for implementing a similar approach, with each agent of the grid working on a larger part of the matrix.

2. Systolic array - block-wise

This approach is similar to the previous one, each agent of the grid communicating with its neighbors in order to obtain blocks of the matrices. However, every multiplying agent processes a submatrix instead of processing only one value.

It has been studied because pipeline parallelism appears clearly with this approach. However, it does not exploit completely data parallelism, as each sumbatrix of the result can be computed concurrently because of the lack of data dependency.

3. Block-wise

   Another strategy fully exploits data parallelism by computing each block of the result concurrently. This increases the memory complexity, because at a given time, each agent stores a row and a column of blocks ($O(P * N * N)$), instead of just storing two blocks ($O(N * N)$).

   This strategy has been chosen because it is simpler to implement and it is the one that benefits the most from data parallelism. It is going to be explained in detail in the following sections.

3.4.7 Detailed implementation

1. Input

   - Read the matrices A and B with *StreamReaders*, stored in a row-major order, and coded as 4 bytes signed integers.

2. Dividing the work into blocks

   - Split the matrices A and B in streams of *blocks* (i.e. square sub-matrices).

   In the following example, the matrix A of size (4,4) can be split in four blocks (matrix B) in the following way:

Table 2: Initial Matrix A

A11	A12	A13	A14
A21	A22	A23	A24
A31	A32	A33	A34
A41	A42	A43	A44

Table 3: Grid of blocks

B11	B12
B21	B22

Table 4: Elements contained in the block B12

A13	A14
A23	A24

In this case, the matrix A (from the product C←A·B) needs to be converted to a stream of blocks (row-wise). It needs to output the following sequence of blocks:

B11, B12, B21, B22.

The agent *MatrixToHorizontalBlocks* does this operation. The result is then *Split*, in order to get one stream of block-columns for each block-row.

The matrix B is to be converted to a stream of blocks as well, but column-wise: The expected order is the following:

B11, B21, B12, B22.

This operation is achieved with a simple *Split* agent.

The resulting streams are linked to *Dup* agents that duplicate them in order for each agent of the multiplication array to get the data it needs.

3. Multiplication array

   The multiplication grid is an array of *MultiplyAccumulate* agents.

   Each agent behaves in the following way:

   - Initialize the accumulator (matrix with the size of a block - i.e. $N * N/P$) to a zero matrix.
   - For each pair of blocks received (i.e. $\sqrt{P}$ iterations):
     - Consume two inputs (A block from matrix A and one from matrix B)
     - The partial product is computed and stored by the agent.

```
int size = N*N/(P*P);
for(int i = 0; i<size;i++){
  for(int j = 0; j<size;j++){
    for(int k = 0; k<size;k++){
      C[i][j]+=A[i][k]*B[k][j];
    }
  }
}
```

   - When the result is complete, the agent outputs its part of the result.

   N.B.: This behavior is achieved by exploiting the Cyclo Static abilities of SigmaC. On each iteration, the agent does not necessarily have the same behavior. They cycle through their specification, with two distinct phases:

   - Taking input and accumulating a partial result (repeated $\sqrt{P}$ times)
   - Producing the output.

4. Result-gathering phase

   The result output of each *MultiplyAccumulate* agent is linked to a *Join*, which takes P inputs of the size of the block. Since *Join* consumes its inputs in a round-robin way (i.e. in a cyclic-sequential order, from the first port to the last), the order of the blocks will be maintained.

   This stream of blocks must be converted back to a matrix, which is exactly the inverse operation of the one done when converting matrix A to blocks previously. Done by the agent *HorizontalBlocksToMatrix* and written back to memory with a *StreamWriter*.

3.4.8 Side tools

1. Data generation

   Using Python3 and the Numpy package:

   - Initialize a seeded random number generator, to get repeatable results.
   - Generate two random square matrices, A and B.
   - Multiply them to get the expected output matrix E.
   - Write each of these matrices to binary files, each element coded on 4 bytes signed integers. The file does not contain other data than the matrix's values.
   - Create a parameter file for the SigmaC toolchain: It specifies the addresses to write to and read from as well as the size of the matrices. This file is a makefile containing several definitions of variables, which is included in the main makefile. The addresses have been parameterized in order to allow an arbitrary increase of matrix sizes without overlapping memory sections.

2. Output checking

   Using the command diff in order to compare the result file and the expected result:

   ```
   diff C E
   ```

3.4.9 Results

This implementation, benefits from a block-wise partitioning on the side of the data flow, and inside these blocks, from tiling and parallelizing for loops.

This experiment consists of multiplying two matrices of size 1000*1000, by a grid of 4 agents (2*2).

Table 5 shows a slight performance increase when tiling on the cluster, and better improvements when parallelizing as well.

Concerning the Desktop environment, table 6 demonstrates a notable performance increase when tiling, and a drop when parallelizing. This is due to the fact that 4 agents were doing dense computations, and no more execution cores were available. The extra context-switching caused a significant overhead.

Table 5: Experiment run in the *Cluster* environment.

Implementation	Pure Data Flow	Tiling	Tiling and Parallelizing
Runtime (s.)	3.8	2.9	1.1
Speedup	-	1.3	3.45

Table 6: Experiment run in the *Desktop* environment.

Implementation	Pure Data Flow	Tiling	Tiling and Parallelizing
Runtime (s.)	8.1	5.5	6.2
Speedup	-	1.5	1.3

3.4.10 Further improvements

- Because this operation is common, the code could be encapsulated in a subgraph for reuse in another project.
- In order to benefit more from pipeline parallelism, a stream of matrices to be multiplied together could be taken as an input.
- Make the program more generic by accepting non-square matrices and matrices with floating-point values.

3.5 Deriche edge detector

This is an optimal edge-detection algorithm for discrete bi-dimensional images. It has been chosen because it is well suited for parallelization since in contains both independent tasks and loop nests.

It is composed of 6 major steps. The result of each step is respectively called L1, L2, ..., L6.

1. Horizontal pass, left to right computations on the image.
2. Horizontal pass, right to left computations on the image.
3. Computing the sum of L1 and L2.
4. Vertical pass, top to bottom computations on L3.
5. Vertical pass, bottom up computations on L3.
6. Adding the two previous results together, and applying a threshold.

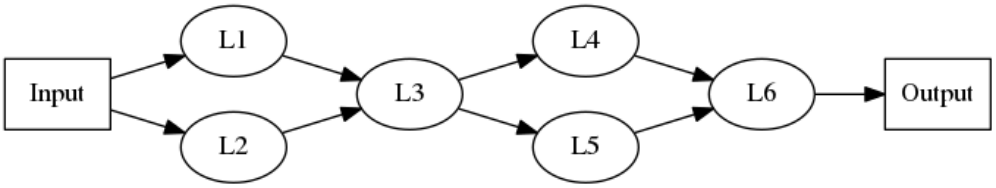


Figure 9: Representation of the dependency graph in Deriche's algorithm.

This graph makes this algorithm quite easy to implement in Data flow, because each step can be written as an agent, and the links are explicit.

This algorithm can be implemented efficiently by making agents working on a stream of rows, computing only one row at a time. However, this eliminates the loop nests that can be parallelized and optimized by Pluto.

Because the algorithm both needs to read the image vertically and horizontally, it needs to be transposed. The subject of matrix transposition in SigmaC is treated in section 3.5.1.

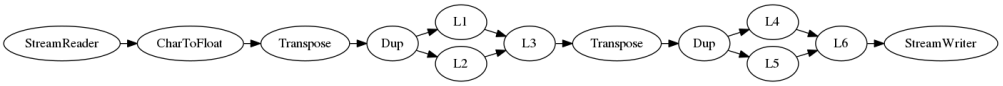


Figure 10: Representation of the implemented data flow graph

3.5.1 Transposing a matrix in SigmaC

Matrix transposition is an often-needed operation, and it can be achieved in several ways in SigmaC.

It is a 'blocking' operation, because the whole matrix is needed to perform it. In Deriche's algorithm, it represents a bottleneck for pipeline parallelism since the rest of the algorithm operates on a stream of rows of the array.

1. With *Split* and *Join* agents

   In SigmaC, a matrix transposition can be done with two agents, as shown in listing 11.

   Listing 11: Subgraph of a matrix transposition using two agents *Split* and *Join*.

```
subgraph transpose(int width, int height)
{
  interface
    {
      in<float> input;
      out<float> output;
      spec{input[width*height]; output[width*height]};
    }
  map
    {
      agent split = new Split<float>(width, 1);
      agent join = new Join<float>(width, height);
      connect(input, split.input);

      for(int i = 0; i<width; i++)
        {
          connect(split.output[i], join.input[i]);
        }
      connect(join.output, output);
    }
}
```

   This method splits the image on every pixel of each row, and joins them in columns.

   In order to optimize performance with this approach, the *Split* agent can be replaced by a *FastSplit* agent. It has the same behavior but works in a single *transition*, instead of *width* transitions, in this case [**?**].

   This method is relatively slow when working with large matrices because the number of links created is linearly proportional to the width of the matrix to transpose.

2. With loop nests

   Listing 12 shows an iterative approach, as it would be done in most imperative programming languages:

Listing 12: Agent achieving a matrix transposition through a loop nest.

```
1
  agent transpose(int width, int height)
3 {
    interface
5     {
        in<float> input;
7       out<float> output;
        spec{input[width*height];output[width*height]};
9     }
    void start() exchange(input in[width*height], output out[width*
        height]){
11    for(int i = 0; i<height;i++){
        for(int j = 0; j<width;j++){
13        out[j*height+i]=in[i*width+j];
        }
15    }
  }
17 }
```

It has the advantage of being completed with only one agent, one input link and one output link. Moreover, it makes nested for loops appear, which can be optimized and parallelized by Pluto.

3.5.2 Results

In this implementation, several approaches have been used for matrix transposition, and their performance is evaluated here, on an image of size $7786 * 3000$ (23MP).

It is important to note that, when using simple Split and Join agents, the experiment lasted more than 5 minutes. These results are therefore omitted here.

The tendency of the results are similar in both contexts. Tables 7 and 8 show a notable performance increase when tiling, and small improvements when parallelizing with Pluto as well. These results also show that transposition achieved with FastSplit and Join does not scale well on big matrices, compared to the loop nests.

Table 7: Experiment run in the *Cluster* environment.

Implementation	Loop nest	Tiling	Tiling and Parallelizing	FastSplit and Join
Runtime (s.)	2.8	1.7	1.5	30.5
Speedup	-	1.6	1.9	0.1

Table 8: Experiment run in the *Desktop* environment.

Implementation	Loop nest	Tiling	Tiling and Parallelizing	FastSplit and Join
Runtime (s.)	2.8	2.5	2.4	34.1
Speedup	-	1.1	1.2	0.1

3.5.3 Further Improvements

1. Merging agents

An optimization could be done in order to limit memory needs and remove an agent: One of them is currently dedicated to casting the unsigned chars to doubles. It could be merged with the two agents using these values, but there is a transposition agent between them, operating on doubles.

Templates are deprecated in SigmaC [?], therefore it is not easy to achieve that without duplicating code.

Performance would be increased because one agent doing little work is removed, and the size of the data is reduced in one part of the data flow graph, because the type *char* is usually stored on fewer bytes than *double*.

2. Parameters

   The implementation could be parameterized instead of using constant coefficients for edge detection.

3. Pipelining

   The implementation could take a stream of images as input in order to benefit more from pipeline parallelism.

4. Coarser Grain

   Each agent could operate on one whole image at a time. The transposition phase would not be needed, and every agent could be optimized by Pluto for data parallelism.

3.6 Artificial Neural Network

3.6.1 Introduction

Artificial neural networks are computer programs inspired by nature, and especially by brains. They are composed of layers of simple computing units, called perceptrons, that are connected together. Each of these units takes several inputs, and produces an output according to their input and their configuration.

During the training process, this configuration evolves in order to classify data, approximate a function, or recognize a pattern. The training phase will not be treated here as it is not suited for data flow programming. The program implemented here can simulate such a neural network once it is given the parameters of an already-trained network.

In this section, Artificial Neural Networks are introduced in a bottom-up fashion. The most basic units, the perceptrons, will be presented first, then they will be grouped in layers, which, once grouped, will finally form neural networks.

1. The Perceptron

   The perceptron is the basic processing unit in a neural network, which can be thought of as a neuron. It has a fixed number of inputs N, and a weight for each of these inputs. It also has an activation function, and a single output value. The input(s), the weight(s) and the output are real numbers.

   Its output value is the activation function applied to the weighted sum of the input. It can also be seen as the product of two vectors (the input and the weights), followed by a function application on the result.

   The activation function is used to add non-linear capabilities to such a system, and a variety of functions can be used, such as the Hyperbolic Tangent and the sign function, for example.

The graphical representation of a perceptron is shown in figure 11. Its behavior can be modeled with the following code:

```
double Perceptron(int N, double inputs[N], double weights[N], double
    (*activationFunction)(double))
{
  double output = 0;
  for(int i = 0; i<N;i++){
    output+=inputs[i]*weights[i];
  }
  return activationFunction(output);
}
```

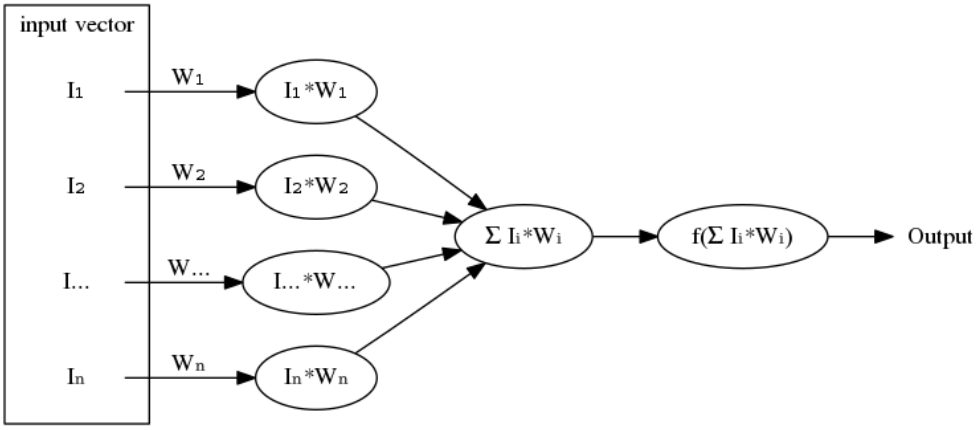


Figure 11: Graphical representation of a perceptron.

However, this simple model is limited to linearly separable problems, and very close to a linear regression. In order to increase the complexity of the functions that can be approximated, perceptrons can be organized in layers.

2. Neural Networks

   A neural network can be created by organizing perceptrons in layers, and by linking the output of every perceptron of the layer N-1 to every perceptron in the layer N. Such a layer is said to be fully connected.

   The first layer (input layer) must contain as many perceptrons as it takes input values. The last layer (output layer) must contain as many perceptrons as it produces output values.

   The computation of a layer is equivalent to a vector-matrix multiplication between the input vector and the weight matrix.

   - I is the input vector, a line vector of size N.
   - W is the weight matrix, of size N x M (N rows, M columns).
   - O is the output vector, a line vector of size M.

   The output of a layer can be computed as follows:

(a) O'=I*W, the star symbol representing the dot product, described below.

(b) Each output element should then be transformed by the activation function. O=activation(O')

This algorithm could be written as shown in Listing 13. Figures 12 and 13 show the flow of data between entities in this algorithm.

Listing 13: Implementation of a simple neural network with three fully-connected layers.

```
void layer(int N, int M, double In[N], double W[N][M], double Out[M],
    double (*activation)(double[],int)){
  for(int i = 0; i<M;i++){
    Out[i]=0;
    for(int j = 0; j<N;j++){
      Out[i]+=In[j]*W[j][i];
    }
  }
  activation(Out, N);
}
void activation(double Out[N], int N){
  // Updates O
}

void myNetwork(double In[N], double Out[N], int N)
{

  double W1[N][10] = {...};
  double W2[10][20] = {...};
  double W3[20][N] = {...};

  double o1[10];
  double o2[20];

  layer(N, 10, In, W1, o1, &activation);
  layer(10, 20, o1, W2, o2, &activation);
  layer(20, N, o2, W3, Out, &activation);
}
```

3. Learning

   The learning process will not be treated here as this experiment only focuses on the feed-forward part of the algorithm. It uses pre-computed weights, and is not meant to be trained.

3.6.2 Implementation

1. Parallelism

   In this case, two kinds of parallelism can be benefited from:

   - Pipeline parallelism: When executing a batch evaluation (i.e. with a stream of input vectors), each layer can be executed in a pipelined way: If there are three layers, three input vectors can be handled at the same time: The first input in the last layer, the second input in the middle layer, and the last input in the first layer. The more layers the network contains, the more pipeline parallelism will have potential. At any

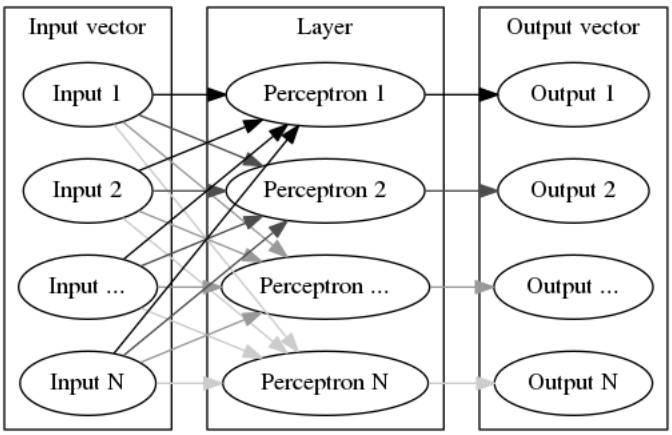


Figure 12: Graphical representation of a single layer.

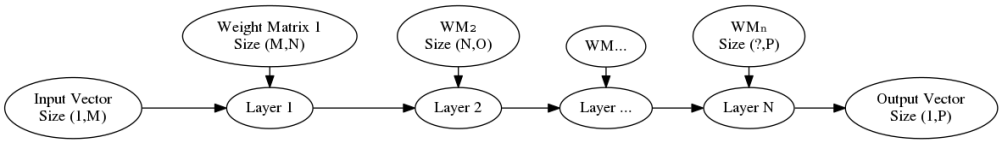


Figure 13: Graphic representation of a network.

moment, the program can compute as many inputs as its number of layers. This does not decrease the latency but it increases the throughput of the system.

- Data parallelism: through for loops parallelism during the vector-matrix multiplication.

Note: Task parallelism cannot be used here because the layer N+1 needs the whole output vector of the layer N to start its computations.

2. Building the network

   The network has to be trained by another program that can export the weights and the settings of the trained network. In this experiment, a Python program has been implemented based on the library Keras [3].

   The specification of the network is written to a plain-text file, which is parsed, at compile time, by the root graph of the SigmaC program. It specifies the number of layers, and for each layer, its size, its weight matrix, and the activation function to use. This workflow is described in figure 14.

   One agent is instantiated for each layer, and it is given the parameters provided in the file. The agents (*layers*) are then linked together as they appear in the text file.

3. Format of the file describing the network

   On the first line of the file, an integer specifies the number of layers composing the network.

[3] `https://keras.io/`

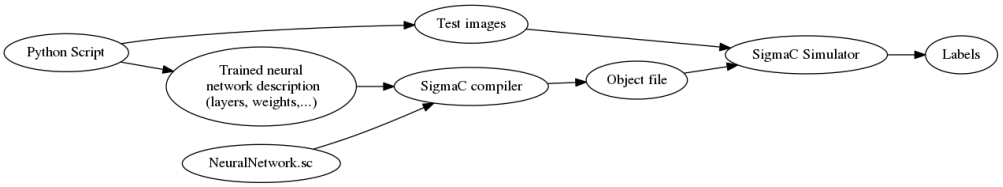


Figure 14: Graphic representation of the workflow of this experiment.

Then, for each layer: There is a new line, containing two integers, I and O, and a string. I and O represent, respectively, the number of inputs and outputs of the current layer. I is either the number of neurons in the previous layer, or the number of inputs of the network. O is the number of neurons in the layer. The string is the name of the activation function. The SigmaC program will not compile if the activation function is unknown.

Then follow I lines each containing O floating-point numbers (separated by spaces), which represent each value of the weight matrix.

A sample description file is shown in listing 14.

Listing 14: Sample file describing a network containing two layers with respectively 3 and 2 neurons (for the sake of brevity in the weight matrices).

```
2
4 3 tanh
0.1 0.2 0.3
0.4 0.5 0.6
0.7 0.8 0.9
1 1.1 1.2
3 2 softmax
0.1 0.2
0.3 0.4
0.5 0.6
```

It is important to note that the number of inputs of the layer N+1 must match the number of outputs of the layer N.

4. Runtime

   Each agent performs its local vector/matrix multiplication, tiled and parallelized by Pluto (with OpenMP), executes the activation function on the result, and outputs its result (either to the next layer or to the StreamWriter).

3.6.3 Results

The program is parallelized both by Pluto for loop nests and by the SigmaC compiler for data flow. It takes a stream of input vectors as input, and outputs a stream of output vectors. It has been successfully able to classify images from the MNIST dataset about Handwritten Digits Recognition.

The classification's accuracy is not important here, because it mostly depends on the training phase, which is independent from this implementation.

This experiment consists of classifying 1000 images of 784 pixels by a neural network containing 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, therefore benefiting more from pipeline parallelism and having more computation to perform.

Table 9 shows a slight performance increase when tiling on the cluster, and better improvements when parallelizing with Pluto as well.

On the Desktop configuration, table 10 shows a small performance increase when tiling, but it drops when parallelizing with OpenMP. This is due to the fact that the 3 layers are already keeping most of the cores busy, and managing more threads and context-switching causes too much overhead.

Table 9: Experiment run in the *Cluster* environment.

Implementation	Pure Data Flow	Tiling	Tiling and Parallelizing
Runtime (s.)	3.5	3.1	1.7
Speedup	-	1.13	2.06

Table 10: Experiment run in the *Desktop* environment.

Implementation	Pure Data Flow	Tiling	Tiling and Parallelizing
Runtime (s.)	5.8	5.0	7.3
Speedup	-	1.2	0.8

In order to push the experiment further on the cluster, a finer OpenMP configuration has been done at run-time, by changing the value of the environment variable *OMP_NUM_THREADS*. This allows a finer control over the number of threads spawned for each loop nest. Table 11 presents the results. The performance steadily increases when increasing the number of threads. With values greater than 8, the performance decreases because the number of threads exceeds the number of execution cores (4 agents doing dense computations, using 8 threads each). It is worth mentioning that the performance of the parallelized version with only one thread per loop (Table 11) is worse than the one of the version that is only tiled (Table 9), because of OpenMP's overhead.

Table 11: Experiment run in the *Cluster* environment, when using different values for the variable *OMP_NUM_THREADS*.

Number of threads	1	2	4	8
Runtime (s.)	3.8	2.1	1.2	1.1
Speedup	-	1.8	3.2	3.5

3.6.4 Further improvements

- More activation functions could be implemented in order to be able to model more networks.
- Adding support for bias neurons, in order to improve the accuracy of the network and the variety of supported networks.
- Adding support for convolutional layers, in order to support more networks.
- Encoding weights in binary instead of ASCII characters in the file describing the network, for lower space requirements.
- Using lower-precision floating point numbers, for performance [**?**].

3.7 Results

The results of the previous experiments are quite positive. Depending on the nature of the loop nests, the only process of tiling provides significant performance improvements, without any cost apart from the static optimization. Parallelizing the loop nests is not always beneficial, especially on processors containing relatively few cores, which may already be busy running SigmaC agents. The number of threads spawned must be adjusted according to the machine. Further experimentation should also be done in order to tune more finely tile sizes, according to the amount of memory accessed and the processor's cache hierarchy.

In some cases, when parallelizing with OpenMP, the SigmaC runtime ended with a *Segmentation Fault*, while still providing the correct output. The creation of threads inside agents must not have been expected by Kalray, and the OpenMP threads may be accessing memory that has already been cleaned up at the end of the simulation.

It should also be kept in mind that running inside the SigmaC Simulator causes an overhead. The performance could probably be improved by compiling SigmaC to native machine code. In order to be able to do so, the source code must not contain any *StreamReader* or *StreamWriter* agents. Therefore, user-defined agents should be created using POSIX interfaces, in order to, for example, read and write from files. As the documentation encourages to use *StreamReader* and *StreamWriter* agents and this information was obtained late, this modification is left as a further experiment.

4 Integrating Polyhedral optimizations to SigmaC

This section presents a reflection about integrating a polyhedral optimizer to the SigmaC toolchain.

In order to integrate a source-to-source polyhedral optimizer such as Pluto, the SigmaC programming language should be slightly modified. It would have to allow the conversion of function parameters that are linearized arrays to N-dimensional arrays, by specifying the dimensions' sizes in the parameter signature.

This would allow programs to be syntactically correct for both Pluto and SigmaC. For example, when doing a matrix transposition, the notations shown in listings 15 and 16 should be able to be used interchangeably, the second one allowing Pluto optimization. It is important to note that listing 15 cannot be processed by Pluto and has an error-prone syntax. Listing 15 provides a syntax that is both easier to understand and compatible with Pluto.

Listing 15: Current way of transposing a matrix in SigmaC

```
void function () exchange(input in[300*600], output out[300*600])
{
  for(int i = 0; i<300;i++)
    for(int j = 0; j<600;j++)
      out[j*300+i]=in[i*600+j];
}
```

Listing 16: Transposing a matrix using a new syntax for array accesses in SigmaC

```
void function () exchange(input in[300][600], output out[600][300])
{
  for(int i = 0; i<300;i++)
    for(int j = 0; j<600;j++)
    out[j][i]=in[i][j];
}
```

Otherwise, another polyhedral optimizer could be used, working at the level of the *Intermediate Representation*, such as Polly[4] with LLVM, which works on linearized arrays. This would probably require SigmaC to use an LLVM back end, and maybe a compiler such as Clang[5].

With Pluto, the programmer has to specify sections of code to process. Only loop nests containing a significant amount of computation and poor locality are worth optimizing. It is important to remember that tiling brings some overhead, since it increases the number of loops, which increases the size of the code (using less efficiently the instruction cache) and increases the number of branches to execute (wrong branch prediction causes pipeline stalls).

To take this off the programmer's shoulders, heuristics could be used in order to determine if it is worth tiling a given piece of code. The granularity of this element could be at the function scale, at the loop nest scale, or for each statement inside a for loop.

The tile sizes should also be determined, because they need to vary according to the size and number of memory accesses inside a given loop nest. The granularity of this should be determined as well. However, the tile size also depends on the target's memory hierarchy, and it is certainly a poor idea to add such a low-level dependence to the source code.

Parallelizing should only be done on loop nests containing a significant amount of computation. The number of threads created on each loop nest should be reasonable according to the scheduling (i.e. the number of agents running at the same time) in order to avoid creating more threads than the number of execution cores, which would cause an overhead with context switches, as seen in section 3.

This could be achieved through annotations from the programmer, specifying which proportion of the available threads should be used on a given loop nest. The SigmaC compiler would then allocate the available threads (according to the target's specification), if any, to further parallelize loop nests.

5 Conclusion

Experiments run with the data flow programming language combined with polyhedral parallelization and optimization have proven to be interesting.

Data Flow provides a higher level of abstraction than *Threads* or *Message Passing*, which is interesting because it allows the programmer to focus on the algorithm and the data dependencies rather than the explicit parallelization and scheduling of the program. The SigmaC compiler takes platform independent code and parallelizes it for any given architecture. When using threads, the programmer has to do this work: the code complexity is increased, and it is often only specific to one architecture.

Bringing polyhedral optimization with Pluto allows to further parallelize these programs with few efforts, by automatically optimizing data locality in for loops and by parallelizing them when possible. Memory latency is often a bottleneck, and tiling helps to reduce this issue. Tiling provides a significant performance increase when combining SigmaC and Pluto, and parallelism increases performance even more, if enough execution cores are available.

The combination of these two tools allows an easier translation from C to SigmaC, since for loops can be kept without increasing execution time.

However, Cyclo-Static Data Flow and Static Data Flow have constraints that make them less expressive than imperative paradigms, therefore they are not suited for all algorithms. Yet, they are adapted to many scientific computing problems and they are more flexible and easier to program than GPU's.

[4] `https://polly.llvm.org/`
[5] `https://clang.llvm.org/`

While it is relatively easy to program in the SigmaC language, the compilation and the execution of SigmaC programs can be complex. The SigmaC toolchain may not be mature enough and compilation is far from being as easy as it is when compiling C with GCC, for example. It is important to note that the experiments have been run inside the SigmaC runtime, therefore the results might not be completely accurate. It also implicates that running experiments requires the SigmaC toolchain, which is quite heavy to install.

It is also often not trivial to integrate polyhedral optimization with Pluto into SigmaC programs, mostly because of syntactic issues. It would be simpler for the programmer if such a tool was integrated to the SigmaC compiler, as Polly is in LLVM. Also, in these experiments, tiling was performed with a constant tile size, but it heavily depends on the target's memory hierarchy. Our approach, which consists in manually modifying the source code, forces to update the source code when changing the target. Integrating such a tool in the compiler would allow to automate this process and make the source code even less platform-dependent.

A further experiment to this work would be to run SigmaC applications on Kalray's MPPA, in order to test how well it scales on Many-Core architectures.

Contents

1 Introduction **3**
1.1 History . . . 3
1.2 Terminology . . . 3
1.3 Types of parallelism . . . 4
1.4 Hardware . . . 5
1.4.1 CPU . . . 5
1.4.2 GPU . . . 5
1.4.3 MPPA . . . 5
1.5 Parallelization techniques . . . 6
1.5.1 Programming models . . . 6
1.5.2 Automatic parallelization . . . 7
1.6 Aim of the Project . . . 7

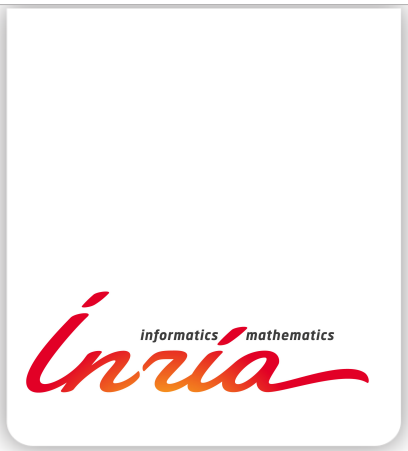
2 Background **7**
2.1 Data Flow Programming . . . 7
2.1.1 Motivations . . . 7
2.1.2 Principles . . . 7
2.1.3 Types of data flow paradigms . . . 8
2.1.4 SigmaC . . . 8
2.2 Polyhedral Model . . . 13
2.2.1 Introduction . . . 13
2.2.2 Pluto . . . 14

3 Experiments **15**
3.1 Introduction . . . 15
3.2 Pluto and SigmaC . . . 16
3.2.1 Temporary transformation . . . 16
3.2.2 Using a port array . . . 18
3.3 Experiment environments . . . 20
3.3.1 Desktop . . . 20
3.3.2 Cluster . . . 20
3.4 Matrix-Matrix Multiplication . . . 20
3.4.1 Introduction . . . 20
3.4.2 Approach . . . 21
3.4.3 Overview . . . 22
3.4.4 Parallelism . . . 22
3.4.5 Topology . . . 22
3.4.6 Choices . . . 23
3.4.7 Detailed implementation . . . 24
3.4.8 Side tools . . . 26
3.4.9 Results . . . 26
3.4.10 Further improvements . . . 27
3.5 Deriche edge detector . . . 27
3.5.1 Transposing a matrix in SigmaC . . . 28
3.5.2 Results . . . 29
3.5.3 Further Improvements . . . 29
3.6 Artificial Neural Network . . . 30

3.6.1 Introduction 30
3.6.2 Implementation 32
3.6.3 Results 34
3.6.4 Further improvements 35
3.7 Results 36

4 Integrating Polyhedral optimizations to SigmaC **36**

5 Conclusion **37**



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803