POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di laurea in Ingegneria Informatica

Code Transformation in High Level Synthesis for iterative stencils

Relatore: Prof. Fabrizio Ferrandi
Correlatore: Ing. Marco Lattuada

Tesi di laurea di
Emanuele Vitali
Matr. 798867

Anno Accademico 2014/2015

Contents

List of Tables

List of Figures

Abstract

Scientific applications may be characterized by a high degree of parallelism which make them suitable to be implemented on programmable devices such as Field Programmable Gate Arrays. To automatize the creation of these hardware implementations, High Level Synthesis has been introduced. It consists of a set of methodologies aimed at creating a hardware implementation starting from a high level source code description. Among these methodologies the techniques for the data dependency analysis have a particular relevance. One of them is the polyhedral analysis which allows to identify some possibilities of parallelization inside loops that are due to the presence of the same operations applied to a large amount of data. To achieve this parallelism it is necessary to replicate the operations and to allow parallel accesses to the data. The contribution of this thesis is a methodology for the High Level Synthesis that is able to replicate the parallelizable section of the code, identified by the polyhedral analysis. This goal is reached with a data reorganization that satisfies data dependences, minimizes memory accesses and parallelizes the remaining accesses.

Sommario

Le applicazioni utilizzate nell'ambito del calcolo scientifico possono essere caratterizzate da un elevato grado di parallelismo che ben si adatta alla struttura di dispositivi programmabili come Field Programmable Gate Array. Per poter automatizzare la generazione dell'implementazione hardware è stata introdotta la Sintesi ad Alto Livello, una serie di metodologie che permettono la generazione automatica di descrizioni hardware partendo da codice di alto livello. All'interno di queste tecniche rivestono un ruolo fondamentale le tecniche di analisi delle dipendenze dati. Tra queste l'analisi poliedrale permette di individuare delle possibilità di parallelizzazione all'interno dei cicli dovute all'applicazione delle stesse operazioni su una gran quantità di dati. Per poter massimizzare questo parallelismo è necessario che le operazioni vengano replicate e i dati possano essere acceduti in modo parallelo. Questa tesi vuole contribuire allo stato dell'arte proponendo una metodologia per replicare durante la sintesi ad alto livello la parte di codice parallelizzabile riconosciuta mediante analisi poliedrale. Questo obiettivo è ottenuto mediante una riorganizzazione dei dati in modo che siano rispettate le dipendenze di dato, sia minimizzato il numero di accessi ai dati e siano ottimizzati gli accessi paralleli.

Riassunto in Italiano

Nel campo dell'Informatica la ricerca di maggior potenza computazionale è sempre stata una spinta per la ricerca di nuove tecnologie e nuovi algoritmi. Fino agli anni 2000 è stato possibile solo grazie al progresso tecnologico, seguendo la legge di Moore, aumentare il numero di transistor presenti su un chip. Si è però giunti al punto in cui la presenza di troppi transistor su un chip ha reso problemi come il consumo energetico e il raffreddamento veramente significativi, e questo ha portato a ricercare soluzioni diverse, come per esempio i processori multi-core. Questo genere di processori infatti permette di avere più potere computazionale, anche se suddiviso in appunto più core che possono processare parallelamente le istruzioni. Ciò limita i problemi di surriscaldamento dal momento che i transistor non sono più tutti insieme in un singolo core, ma suddivisi, ma introduce nuove problematiche come la sincronizzazione dei core e la consistenza dei dati. Questo tipo di approccio, portato a un livello estremo , ha portato alla creazione di sistemi come gli acceleratori hardware, che sono in grado di utilizzare la maggior parte della logica che si trova sul chip per eseguire le computazioni invece che per controllarle, eliminando l'overhead portato appunto dal controllo. In questo campo si trovano le FPGA (Field Programmable Gate Array) che sono circuiti integrati programmabili a livello hardware. Ma vi è una barriera all'utilizzo di questo tipo di soluzione, essendo questo tipo di programmazione estremamente complesso. Uno strumento che si propone di abbassare questa barriera è la Sintesi ad Alto Livello (HLS, da High Level Synthesis). Questa metodologia permette la sintesi automatica di implementazioni per FPGA senza avere la conoscenza del linguaggio di programmazione delle schede stesse. Questa metodologia infatti si occupa, partendo da codice di alto livello come per esempio C, di generare

il codice contenente la descrizione del circuito da sintetizzare sulla scheda. Questa descrizione proveniente dalla HLS è normalmente meno efficiente di codice Verilog scritto da un progettista, ma comunque permette di abbassare notevolmente il tempo di progetto di una soluzione per FPGA.

Le applicazioni utilizzate nell'ambito del calcolo scientifico possono essere caratterizzate da un elevato grado di parallelismo che ben si adatta alla struttura di dispositivi programmabili come le FPGA.

Per questo tipo di applicazioni rivestono un ruolo fondamentale le tecniche di analisi delle dipendenze dati. Tra queste l'analisi poliedrale permette di individuare delle possibilità di parallelizzazione all'interno dei cicli dovute all'applicazione delle stesse operazioni su una gran quantità di dati.

Per poter massimizzare questo parallelismo è necessario che le operazioni vengano replicate e i dati possano essere acceduti in modo parallelo. Questa tesi vuole contribuire allo stato dell'arte proponendo una metodologia per replicare durante il processo di HLS la parte di codice parallelizzabile riconosciuta mediante analisi poliedrale. Questo obiettivo è ottenuto mediante una riorganizzazione dei dati in modo che siano rispettate le dipendenze di dato, sia minimizzato il numero di accessi ai dati e siano ottimizzati gli accessi paralleli.

Questa tesi è organizzata nel modo seguente:

- Nel secondo capitolo (State of the Art) si descrive il modello poliedrale, per cosa è stato usato fino ad ora, focalizzandosi nel campo delle FPGA e della mappatura delle memorie.

- Nel terzo capitolo (Problem Statement) la soluzione proposta è spiegata nel dettaglio, partendo da come il modello poliedrale è usato, come i dati sono divisi e come la parte centrale del ciclo viene riscritta.

- Nel quarto capitolo (Implementation) viene descritta brevemente l'implementazione realizzata , sottolineando i vincoli introdotti dagli strumenti usati che invece non sarebbero presenti nella metodologia proposta.

- Nel quinto capitolo (Experimental Evaluation) il setup sperimentale e i benchmark usati sono descritti. I risultati delle sintesi vengono riportati e analizzati.
- Nel sesto capitolo (Conclusions and Future Work) i risultati sono brevemente esposti e alcuni modi per estendere la metodologia proposta vengono illustrati.

Chapter 1

Introduction

In the field of computer science the search for computational power is what drives the development of new technologies. Until the 2000s thanks to technological progress it has been possible to increase the number of transistor in a single chip following Moore's law. This has been possible until the presence of too many transistors has caused the power consumption and the heat dissipation to become a really important issue. A solution to this issue has been the switch to multi-core architectures. This kind of architectures, thank to having more cores able to perform computation, can lead to more computational power avoiding overheating issues. This sort of approach, on an extreme level, lead to the creation of optimized logic able to perform less operations but in a optimized way. They can do this with the elimination of part of the control logic that is situated in the processor and using the logic only to perform the computations. The problem of this kind of logic is that the greater is the efficiency requested the greater is the difficulty in writing code. The FPGA (Field Programmable Gate Array) are one of these programmable logic. This integrated circuits are able to obtain a great efficiency, since they are programmable at an hardware level. The issue with this approach is that programming hardware is difficult, and requires an high level of expertise. In this context an important enabling technology for the adoption of hardware accelerator technologies is the High Level Synthesis (HLS). Its purpose is to give the performance and energy efficiency of hardware designs with a lower

barrier to entry in design expertise, and shorter design time. HLS tools perform automatic synthesis of the circuit to be implemented on the FPGA starting from a high level language specification like c code. This allows the programmer to create circuits for FPGAs without even knowing the synthesis language.
The great difference between normal computing and hardware accelerators is that while in normal computer there is a limited amount of processors able to do all the computation, in programmable logic there is the possibility to create more computing units, that are able to do only the operations they were created to do. This kind of logic is no more general purpose (like processors), but specialized.

This work aims to seek and improve parallelization in a subset of possible situations, that are scientific calculation. This kind of calculation (compute-intensive) often spend most of time in nested loops doing the same operations on many different data ([KAI11],[BHRS08]). This sort of computation can be modeled with the Polyhedral Model. This instrument is important because it can reveal some parallelism that may be hidden because of how the code is written. The polyhedral techniques are used to analyze the source code and, if some possible parallelism is revealed, to exploit it.

The contribution of this thesis is an algorithm to create more cores able to perform the calculation of the section of the code that can be parallelized, making them work on different data. This requires a reorganization of those data to avoid violation of data dependences and allow the instructions to be executed at the same time. This work is organized as follows:

- Chapter 2 (State of the art) will explain the polyhedral model and how has been used until now in programming, with particular focus on FPGA and memory mapping.

- Chapter 3 (Problem Statement) will explain in detail the proposed solution, starting from how the polyhedral model is used, explaining how the data are divided into different arrays and how the core of the parallel section is modified.

- Chapter 4 (Implementation) will describe how the suggested methodology has been implemented, highlighting the constraints that have been introduced.
- Chapter 5 (Experimental evaluation) will describe the setup and the benchmarks used to test the algorithm, and analyzes the results obtained by the proposed solution.
- Chapter 6 (Conclusion and Future Work) will quickly summarize the results and will suggest how the methodology can be extended.

Chapter 2

State of the Art

In this chapter the Polyhedral Model will be explained, and the reason behind the use of this theory is explained. Since the focus of this work is not creating a new tool, but use the information given by the polyhedral analysis to create parallel computations some tools that use this mathematical theory to improve code efficiency will be analyzed, focusing on what they are able to do. Since in the context where this work is inserted the often the bottleneck are memory accesses (in FPGAs the logic is free to be programmed, but the number of memory ports is fixed) in the final section some articles that suggest the use of the Polyhedral Model in order to improve memory efficiency in FPGAs will be analyzed.

2.1 Polyhedral Model

The polyhedral model is a mathematical model that can be used to represent the information needed for the execution of a program's loop nests. It is based on the following definitions:

Definition 1. ***Affine Hyperplane:***
The set X of all vectors $\vec{x} \in Z^n$ such that given $\boldsymbol{h}$ ($h_1, h_2, ...h_n$ scalar coefficients, at least one of them different from 0), $\boldsymbol{h} \cdot \vec{x} = k \in Z$

or in other words, a n-1 dimensional subspace in an n dimensional space. An example of an Affine Hyperplane is a line in a planar space.

Definition 2. ***Polyhedron:***
The set of all vectors $\vec{x} \in Z^n$ such that $A \cdot \vec{x} + \vec{b} \geq 0$, where $A \in Z^{m*n}$ and $\vec{b} \in Z^m$.

A polytope is a bounded polyhedron. For example the following system is a polytope.

$$\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} -99 \\ 0 \\ 0 \\ -99 \end{pmatrix}$$

Given as example the following code:

```
1 for (i = 100 - 1; i >=0; i--)
2   for (j = 0; j < 100; j++)
3     B[i][j] = B[i][j] + A[i][j];
4 for (i = 0; i < 100; i++)
5   for( j = 0; j < 100; j++)
6     D[i] = D[i] + B[i][j] * C[j];
```

It consist in a couple of loops in which the second has a dependence from the first, since the first writes the value in B, and the second uses it. Each dynamic instance of a statement is defined by its iteration vector, that contains the values of all the indexes of the surrounding loops. For example the vector $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ is the iteration vector for the statement at line 3 at the iteration i=0, j=0. The set of all the iteration vectors (Iteration Domain) of a statement is a polytope (and for the statement at line 3 is the polytope used above as example). The following figure (Figure 2.1) reports the domain of the two statement of the loop in the above code, that are line 3 and line 6.

$$\begin{pmatrix} -1 & 0 & 99 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 99 \end{pmatrix} * \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} \leq (\vec{0}) \qquad \begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 99 \\ 0 & 1 & 0 \\ 0 & -1 & 99 \end{pmatrix} * \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} \leq (\vec{0})$$

Figure 2.1: Domains

The domain of a statement can be described as all the constraints on the various iterators the statement depends on. Every line in the two matrices above is one of those boundaries of the loop. As it can be seen the domain is the same, but the two loops cannot be unified in a single loop, nor pipelined because the scheduling is different.

Definition 3. ***Schedule Function:***
Given a n-dimensional loop-nest, a d-dimensional schedule is the function F that maps n dimensions to fewer (d) timestamps. $F(\vec{i}) = A \cdot \vec{i} + \vec{o}$*, where A is a* $d * n$ *matrix and o is the offset vector (d-dimensional). Mapping n to d means that some of the statements can be executed in parallel.*

The schedule functions of the two statements of the above code are the following:

$$F(s_1) = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 99 \\ 0 \end{pmatrix} \qquad F(s_2) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Figure 2.2: Schedule functions

Composing one or more schedule functions in the polyhedral model the schedule of the program can be modified and improved adding more and more parallelism.

Definition 4. ***Transformation Function:***
A sequence of schedule functions applied to the same domain is called transformation function.

A transformation function for this example is reported in Figure2.3. This function is able to make the two schedules compatible, then the loops could be unified (Figure2.4) and other optimizations, such as pipelining, are applicable.

$$F(s_1) = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -99 \\ 0 \end{pmatrix}$$

Figure 2.3: Transformation function

```
for (t1=0;t1<=99;t1++) {
  for (t2=0;t2<=99;t2++) {
    B[t1][t2] = B[t1][t2] + A[t1][t2];
    D[t1] = D[t1] + B[t1][t2] * C[t2];
  }
```

Figure 2.4: The code after polyhedral transformation

The code to be compatible with the polyhedral model have to satisfy some requirements. Those requirements are:

- The piece of code to be analyzed must be closed in a SESE (Single Entry, Single Exit) region.
- The control of the part of the code that can be improved has to be static, and known at compile time. This feature gives the name to the pieces that can be improved with the polyhedral model, that are called SCoP (Static Control Part).
- To be static, the loops must have known boundaries, and they are affine expressions of constants values, parameters known at compile time or iteration variables of extern loops involved in the same SCoP.

Once it has been decided if the code satisfy those constraints, the polyhedral model is able to describe the iteration order of all loop nests that have all array accesses as affine expression of the indexes. The array accesses are also used to represent dependences between statements: if two statements access the same array locations, and at least one of those access is a write, there is a dependence.

The polyhedral model can be used to perform some code optimizations, as for example loop skewing (e.g., make the inner-loop bounds dependent on the outer-loop bounds), loop reverse (the example), loop permutation (e.g., interchange two loops) and combination of these operations. This can expose parallelism otherwise hidden because of data dependences. The dependences must be preserved, otherwise the meaning of the code would be altered (because the accesses would not be consistent).

2.2 Polyhedral Tools

This work is not interested in creating a new tool for the polyhedral analysis, but since a program able to do this kind of work is needed some existing tools have been analyzed before proceeding with the following steps. In the following sections tools that are able to perform some operations on polyhedral code are listed, divided by the kind of operation performed and the level at which they work. Section 2.2.1 will describe tools able to create code from a polyhedral representation. Section 2.2.2 will describe compilers able to perform polyhedral optimizations while compiling programs from source code to executables, or from source code to source code. Section 2.2.3 will describe tools able to perform only dependency analysis. This step is mandatory before applying polyhedral transformations, but can be separated from the others if it is the only thing needed. Section 2.2.4 will describe a protocol used by several tools to exchange information. Section 2.2.5 will describe some tools that perform front-end work (from source code to polyhedral model). Finally section 2.2.6 will describe some library useful while working with this model.

2.2.1 From polyhedral source to code

The tools described in this section are able to create code starting from a description of the SCoP made in a polyhedral representation, using different algorithms.

- CADGen
  This tool, described in [Grö08], uses Cylindric algebraic decomposition to generate loop code (allowing to generate code for semi-algebraic sets and

not only affine expressions). It has as input ClooG code or its own format, and come out with C code. It is part of HsLooPo (the Haskell modules for LooPo), but it exists only as a prototype version and it has never been updated since its release in 2009.

- ClooG
  This tool is part of the Chunky project, a research tool for data locality improvement. The official version uses ISL library and GMP to scan Z-polyhedra, finding a code that reaches each integral point of one or more parameterized polyhedra, but also has PolyLib and PPL support. It uses its own format as input, but also accept OpenScop. It is still being developed (last update July 2015) [Bas04].

- omega/ codegen and omega+/codegen+
  codegen is the tool built on top of Omega, so it uses omega format as input and generates C code. The transformations done are the ones made by the Omega Library.

2.2.2 Compilers

The tools listed and described in this section are part of compilers (often optional) or full fledged compilers, that work with source code as input and are able to create executables or new code.

- Polly
  This tool is an optional step of the LLVM compiler. It uses ISL to perform classical loop transformations, especially tiling and loop fusion to improve data-locality. It can also exploit OpenMP level parallelism, exposing SIMDization opportunities. Work has also be done in the area of automatic GPU code generation. Polly input is LLVM-IR (intermediate representation of code during LLVM work, like Gimple for gcc) and output to Code or JScop (polyhedral representation stored in a JSON file). It is a very active project (started in the end of 2009) [GGL12].

- Graphite
  It is a optional step in GCC compiler, so it goes from C to executable. It performs high level memory optimization, hosts loop-interchange, locking and flattening. It is actually on stand-by, since part of its developers moved to Polly [TCE+10].

- IBMXL
  This is a step for the IBM-XL optimizing compiler. It is focused on optimization for hardware prefetch stream buffer utilization, locality, and parallelism [BGDR10].

- ChiLL
  This tool works in three steps: It automatically derives a sequence of code transformations, that may have unbounded parameters. A transformation expresses to the code generator, at a high level, the sequence of transformations to be performed. Then the search engine takes as input those sequences of transformations, modifying the parameters with bounded ones that are the input for the third step, the loop transformation and code generation framework. The last step takes as input the original code and a transformation script with bound parameters, and generates an optimized code version [CCH08].

- LooPo
  This tool transforms source code in a parallel way, using polytope methods. Space and time are represented as dimensions of polytopes, then the tool works on the polytopes and translate back to loop code. It is an old project and it is no more supported nor developed [GL96].

- Pluto
  Pluto is both a scheduling algorithm based on polyhedral model and a tool implementing it. It is oriented towards coarse grained parallelization and data locality simultaneously. Options are provided to tune tile sizes, unroll factors and outer loop fusion. The tool works with C source (marked with pragmas) because it uses Clan as front-end, while the algorithm needs a polyhedral representation (OpenScop is used). The tool can

also work with OpenScop as input. As output the tool uses ClooG to regenerate C code. It is still in active development, and its algorithm is used in some production compilers [BHRS08].

- PoCC
  It is a collection of several state of the art tools in polyhedral analysis (Clan, Candl, Cloog, ...). The part to be optimized must be noted with pragmas (see Clan). It goes from C code to executables.

- PolyOpt
  This tool make an automatic extraction of SCoPs, then applies PoCC analysis and modifications. It works with a subset of the C and FORTRAN languages.

- PPCG
  Another part of LLVM, PPCG is a source-to-source compiler generating OpenCL or CUDA GPGPU code from sequential programs. It uses Pet for the extraction of the the polyhedral model and it has an own code generator implemented in ISL, similar to CLooG. The tool has a modular structure: it starts from the model extraction (from the source C code), then it does dependency analysis, scheduling and memory management. Its output is written with the idea of using GPUs computational power, and to achieve that objective it uses language such as CUDA or OpenCL. The tool automatically decide which part is to be executed on CPU and which on GPU and it also manages the data transfer. [VJC^{+}13].

- R-Stream
  This tool (propriety of Reservoir Labs) is a source-to-source compiler that optimizes code for parallel processors and accelerators. It can output code in a variety of formats for downstream processors, including highly optimized OpenMP and CUDA. It has C code as input and output [SLLM06].

- Alpha-Z
  This tool is able to transform, analyze and generate code. It also does memory re-mapping and complexity reduction. It goes from mathemat-

ical equations (that are represented in his own language, which is explained in the documentation) to code [YGK+12].

2.2.3 Dependency analysis

The tools in this section are the ones able to create the dependency relationships between statements. They will not optimize the code, but this is a necessary step before running the optimization algorithm because if inconsistencies are to be avoided, all the reads and writes conflicts are to be known.

- Candl
  This tool is the dependency analysis basic block of the PoCC toolchain. It computes the set of statement instances in dependence relation, it performs data dependence removal and array privatization/expansion. It uses OpenScop language for input and output.

- FADA
  The software is a C++ implementation of the Fuzzy Array Dataflow Analysis (FADA) method. This method can be applied on codes with irregular control such as while-loops, if-then-else or non-regular array accesses, and it computes exact instance-wise dataflow analysis on regular codes. The tool works with source code as input and as output it produces read/write references written in its own language [BBET10].

- Petit
  This tool performs analysis of array data dependence, working with the Omega library. It uses Omega format as input and output. It is a very old and no more developed project [KMP+95].

2.2.4 Exchange Format

The library described in this section is just an exchange format, a convention decided to support communication between polyhedral tools:

- OpenScop
  This library is used by some tools (the whole PoCC toolchain, for which

has been developed, and some others) as input and output. The library is a support for who wants to use the OpenScop format as temporary and exchange support. It saves information for each SCoP (domain, schedule functions and accesses) and more optional data. It is a relatively new project [Bas11].

2.2.5 Frontend

The front-end tools are the ones that are able to read source code and trasform it in a polyhedral model:

- Clan
  It goes from source code (C, C++, Java) to OpenScop representation. The code need to be marked with pragmas to let the program know which parts are to be analyzed. The tool is unable to detect semantical issues, so it trusts the user when a SCoP is declared.

- Pet
  This tool goes from C source to LLVM Abstract Syntax Tree. It uses pragmas to determine where are SCoPs, clang as parser and it represents sets and relations with ISL [VG12].

2.2.6 Libraries

In this section the libraries created to perform calculation on polyhedra are being listed:

- Barvinok
  The library counts the number of integer points in parametric and non-parametric polytopes, using polylib as input and output format.

- IEGenLib
  This library represents and manipulates integer sets (representing data and iteration spaces) and relations (memory access functions) that have affine and uninterpreted function constraints. It is written in python, and accept both omega and isl integer tuple set as input and output.

- ISL
  This library manipulates sets and relations of integer points bounded by affine constraints. The descriptions of the sets and relations may involve both parameters and existentially quantified variables. It works in three steps: Dependence analysis, Scheduling and Abstract Syntax Tree generation. It has a lot of options, which the user can tune to guide the whole process. It has an own format, but also accept polylib, both as input and output [Ver10].

- Omega and Omega+
  Omega permits to manipulate integer linear constraints over integer variables in first order logic, and operations on integer sets and their mappings. After the manipulation it generates code by scanning the points in a union of polytopes. It uses an own format as input and output.

- PolyLib
  It works on sets and can do the following operations: intersection, difference, union, convex hull, parameterized vertexes computation, polynomials computation. It has his own format (documented with examples), and it is an old and mature library, but still maintained [Loe99].

- PipLib
  PipLib finds min/max in the set of integer point belonging to convex polyhedron, with or without a parameter context. The input and output have to be specified in its own syntax (there are examples in the documentation). It is a very old project (started in 1988) but has been update until 2009 [Fea88].

- PPL
  This library does manipulation of numerical information that can be represented by points in some n-dimensional vector space. One of its functionalities is working with polyhedra (but is not the only one, can work with grids too). It has its own format for input and output [BHZ08].

- ZPolyTrans
  This library has two executables: the first computes the elimination of some integer variables from a union of parametric polyhedra, and the second computes the Ehrhart polynomial of a union of parametric Z-polytopes. It is based on Barvinok and PolyLib, but has an own format for input and output [SL06].

A summary of all the polyhedral tool analyzed is offered in Table 2.1, where for every tool the languages used as input and output are provided and it is highlighted if the tool is part of some bigger project.

Table 2.1: Tools recap

Name	Input	Output	Part of
CADGen	ClooG or own	C	HsLooPo
CLooG	OpenScop or own	C	Chunky project
CodeGen/ Codegen+	Omega	Omega	–
Polly	LLVM-IR	Executable or JScop	LLVM
Graphite	C	Executable	GCC
IBMXL	C	Exectutable	IBMXL optimizing compiler
ChiLL	C	C	–
LooPo	C	C	–
Pluto	C (with pragma) or OpenScop	C	Chunky project
PoCC	C (with pragma)	C	Chunky project
PolyOpt	C(with pragma)	C	Chunky project
PPCG	C	C	–
R-Stream	C	C	–
Alpha-Z	own	own	–
Candl	OpenScop	OpenScop	Chunky project
FADA	C	own	–
Petit	Omega	Omega	–
OpenScop	–	–	Chunky project
Clan	C (with pragma)	OpenScop	Chunky project
Pet	C (with pragma)	LLVM AST	LLVM
Barvinok	PolyLib	PolyLib	–
IEGenLib	Omega and ISL	Omega and ISL	–
ISL	own	own	–
PolyLib	own	own	–
PipLib	own	own	–
PPL	own	own	–
ZPolyTrans	own	own	–

2.3 Usage of Polyhedral Model

There are different way to use the information derived from the polyhedral model to improve the High Level Synthesis approaches:

1. To optimize the inter-block communication and enable intra-block parallelization and inter-block pipelining through loop transformation for FPGAs using HLS ([ZLL+13].

2. Minimize resource usage without any performance (e.g., latency) penalty [ZLC+13].

3. Identification and optimization of parts of code which could benefit from the GPU architectural characteristics, then reintegration of those parts into the user's code. It is done as step of a automatic compiler for FPGAs or GPUs (terapyps or p4a respectively) [KAI11].

4. Optimizing the performance (i.e., throughput) of the computation part of an affine program to be executed on the FPGA. [LPC14].

5. Improve memory accesses for a better performance or a lesser area consumption or a trade-off between the two. It works for (multidimensional) array accesses [WLZ+13] [CG15] [VC14] [MYO+15].

This work will focus on the last aspect.
For this reason the last section of this chapter will be focused in explaining a bit more in detail the work in this field, particularly the articles cited in the last point.

2.3.1 Generalized Memory Partitioning in High-Level Synthesis

In this article ([WLZ+13]) the authors concentrate their effort in finding a working partitioning algorithm. Since with programmable logic it is possible to duplicate the logic that is doing the computation, the number of ports for every block-ram (BRAM) is limited and this becomes the bottleneck. It is not realistic to increase the number of ports in each BRAM because that would cause a quadratic growth of complexity and area. Duplication of data in different blocks can help performance, but it creates area and consistency problems.

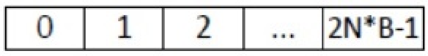
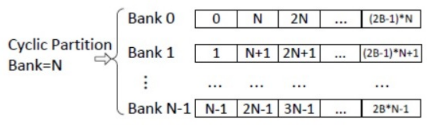


Figure 2.5: original sequential data

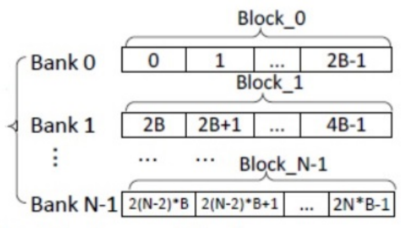
The authors start with an excursus on the actual partitioning algorithms. The original data is reported in Figure2.5, where N is the number of banks, B and the factor 2 are arbitrary numbers (to ease comprehension of the figures, B has been chosen as the size of a block for the block-cycle partitioning, and 2 has been chosen as number of blocks contained in a single array in the block cycle partitioning). The algorithms are:

cyclic partition:
The original data is split among a number N of banks, changing bank for every data. This mean that position 0 of the array will be in bank 0, position 1 in bank 1 and so on until the last bank is reached (position N-1). Then it will restart from bank 0 with position N, bank 1 with position N+1 and so on cycling until the end of the source data is reached.

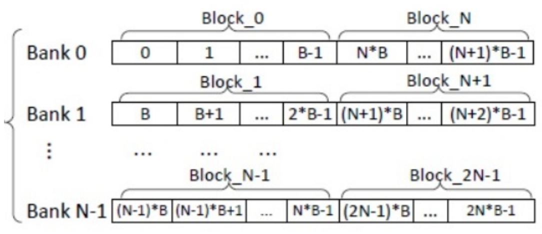


(b) Cyclic partitioning

block partition
It is similar to the cycling partition but here the data are sequentially split: every bank contains a sequential piece of the array (called block). A block has length (array_dimension/number_bank). In this algorithm the first bank contains data from 0 to (block_size-1), the second bank will contain the following up to 2*block_size-1, and so on. Note that block_size in the figure is equal to 2*B



block-cycling partition
This last partitioning algorithm is a combination of the previous two: the data are divided into blocks, that are shorter than in the block partitions (in the figure, B is the dimension of a block), and are split among banks in a cyclic way: the first bank will contain block 0, block N, the second will have block 1, N+1 and so on. This algorithm permits to have some sequential data in the bank, but it removes the limitation of having all data grouped with the same pattern (or all cyclical or all sequential, this algorithm allows a combination of the two).



The objection they move to the older partitioning algorithms is that they offer a optimal solution for mono-dimensional arrays but a suboptimal solution for multi-dimensional arrays (that are the ones involved in loop nests) because of lesser space exploration.

The suggested solution uses the polyhedral model to formulate the memory partitioning, transforming the conflict detection problem in a polytope emptiness check, and the intra-bank offset generation as a problem of counting integer points in a polytope. The suggested method has two targets: to minimize the number of banks needed to map all the array data and access them in the same cycle, and in the meantime to avoid memory dimension explosion, due to the needed padding. The objective is to have all the data that are needed in the same cycle mapped on different banks. This would allow to retrieve all of them in a parallel way using only one cycle to instantiate the reads. However building the complex mapping function would cost too much in hardware resources. A trade-off between practicality and optimality is considered by using a memory-padding based heuristic approach. The heuristic method makes each polytope P have a constant number of data elements. The main partitioning algorithm is composed of two parts: one part constructs the bank-mapping function and the other constructs the intra-bank offset function. The strategies are derived by bounded enumeration and conflict detection. The authors tested the new algorithm on a benchmark suite, and had great improvements from previous work.

2.3.2 Interplay of loop unrolling and multidimensional memory partitioning in HLS

The starting point of this article ([CG15]) is the same as the previous one, that is trying to partition multi-dimensional arrays in different memory banks to use FPGA ability to parallelize logic. The authors start their work where the previous work finished, adding the consideration of the overhead introduced by splitting the array in multiple Block-RAM, that is called bank-switching.
Bank switching symbolize all the the multiplexers and address calculation logic that have to be added to handle the parallel accesses.
They suggest that mixing unrolling with partitioning is possible to obtain a better area efficiency. This imply that the kernel of the algorithm has to be replicated, but this should not be an issue since working on FPGAs the copy only does real calculation and not just overhead.

The objective of the proposed algorithm is mapping the array along memory banks in a way that only half of the initial banks are involved. This brings to multiplexer with half the number of input wires.

```
for (i=1; i<= N; i++)
  for (j=1; j<N-1; j++)
   imageOutput[i][j] = w0*imageInput[i-1][j-1] + w1*imageInput[i-1][j] +
           w2*imageInput[i-1][j+1]+ w3*imageInput[i][j-1] +
           w4*imageInput[i][j] + w5*imageInput[i-1][j+1] +
           w6*imageInput[i+1][j-1] + w7*imageInput[i+1][j] +
           w8*imageInput[i+1][j+1];
  }
```

Figure 2.6: Before the partitioning algorithm: original loop

```
for (i=1; i<= N; i++)
  for (j=1; j<N-2; j=j+2)
   imageOutput[i][j] = w0*imageInput[i-1][j-1] + w1*imageInput[i-1][j] +
           w2*imageInput[i-1][j+1]+ w3*imageInput[i][j-1] + w4*imageInput[i][j] +
           w5*imageInput[i-1][j+1] + w6*imageInput[i+1][j-1] + w7*imageInput[i+1][j] +
           w8*imageInput[i+1][j+1];
   imageOutput[i][j+1] = w0*imageInput[i-1][j] + w1*imageInput[i-1][j+1] +
       w2*imageInput[i-1][j+2]+ w3*imageInput[i][j] +
       w4*imageInput[i][j+1] + w5*imageInput[i-1][j+2] +
       w6*imageInput[i+1][j] + w7*imageInput[i+1][j+1] +
       w8*imageInput[i+1][j+2];
  }
```

Figure 2.7: After the partitioning algorithm: unrolled loop

An example to clarify this is provided: as can be see comparing the two code snippets (Figure 2.6, Figure 2.7), they are the same code, but in the second figure the loop is unrolled (it does two iteration at time) . This unroll modifies (as visible in the accesses graphs, Figure 2.8) the position that have to be read every loop. The number of reads is increased (in the example, there are 12 reads in the unrolled loop where only 9 are present in the original one). The array is split in four banks. This unroll permits to start always from position 0 (or 2, depending from the i value) as the left lower corner of the square representing the reads. This means that it reads from bank 0 or bank 2 the value for that position. This was not happening in the not unrolled loop, that with the next iteration would have to change the input bank between 0, 1, 2 and 3. As this simple example shows the number of banks involved with the read in the left lower corner in the unrolled loop are only half of the total number of banks in which the array has been split.

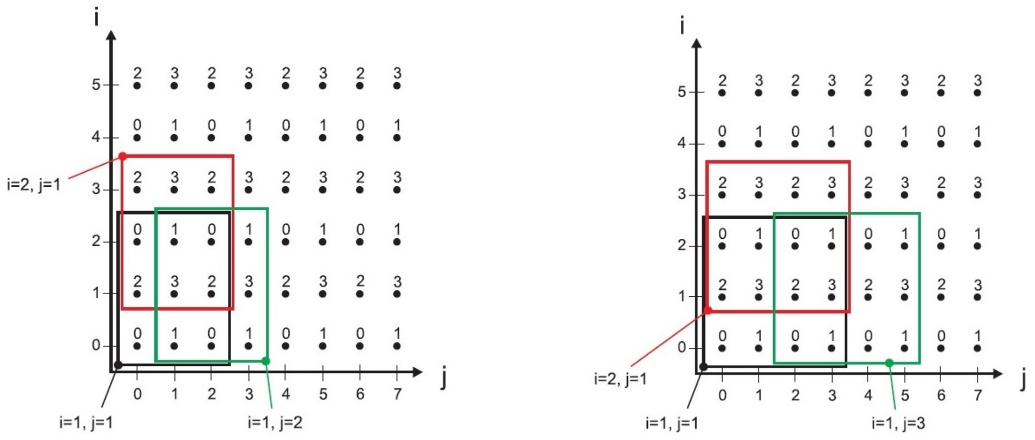


Figure 2.8: Memory accesses, before and after the algorithm

In this way the computing logic for the position corresponding to the low left corner will only have 2 wires as input (see Figure 2.10, the wires entering in the multiplexers before the processing logic (the rightmost ones) come from only two banks), instead of 4 (see Figure 2.9, where the multiplexers before the processing logic all have four wires). The same thing happens to the multiplexers that control the read from the bank, since only half of the values can be found on that bank, they will be wired in order to avoid to have all the possible values of i and j an input values but only the needed ones. This also can be see in Figure 2.9 and Figure 2.10, looking the leftmost multiplexers. This will bring to smaller and easier to control multiplexers.

Obviously this can scale, but the unrolling factor and the number of banks must be related.

In order to obtain this result they suggest a partitioning algorithm based on lattices, that is a generalization of the previous paper (hyperplanes are a particular case of lattices). They use the lattice scheduling to divide the array into banks. There are n lattices that are partitions of the array spaces, and every lat-

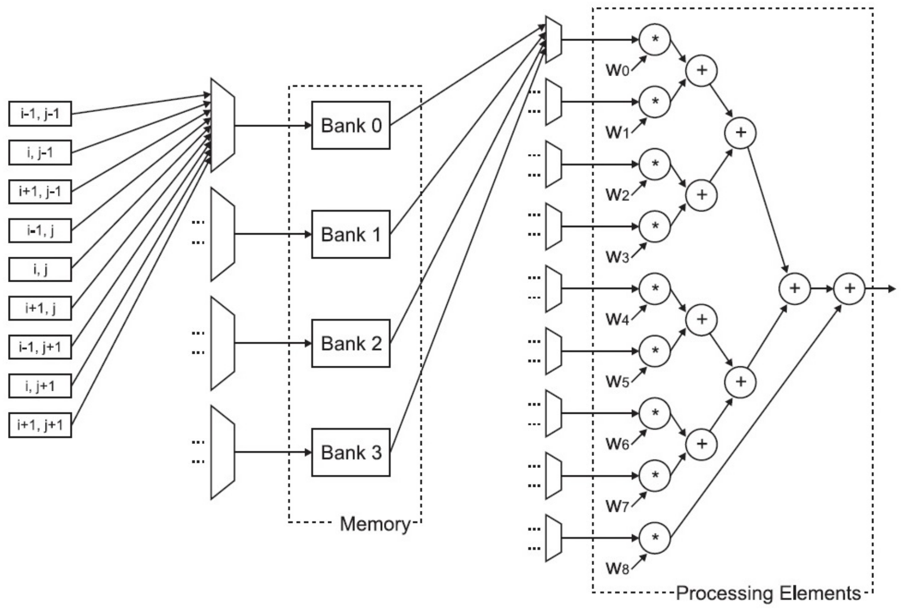


Figure 2.9: Before the partitioning algorithm: data access path

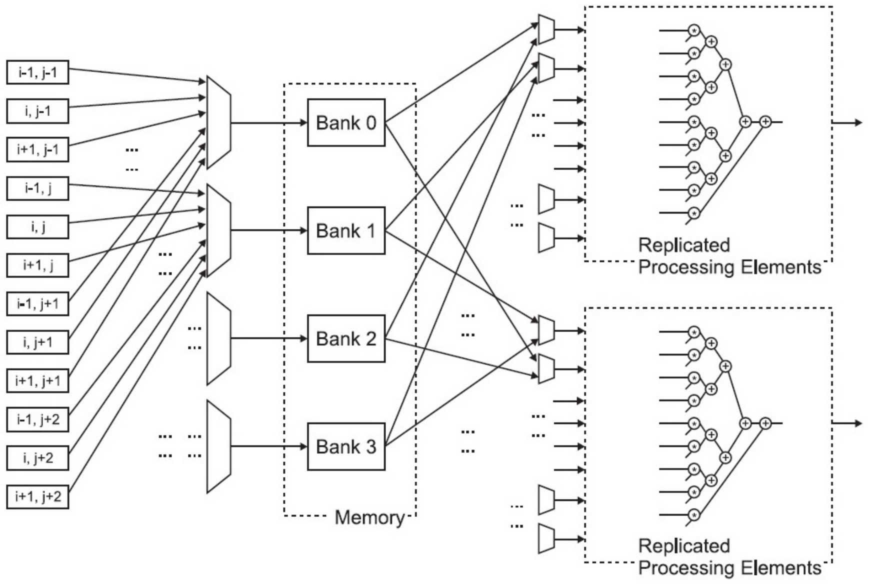


Figure 2.10: After the partitioning algorithm: datapath

tice represent the elements saved in bank n. Then they use the lattice to count the amount of bank-switching. Once the amount of bank-switching has been established they suggest to unroll the loop of a suitable amount. This is function of bank number and the previously calculated amount of bank switching. The unroll permits to avoid the overhead created by bank-switching. It is not considered the presence of data dependences (they conjecture HLS tool itself is able to find out where there is data dependence and somehow block the unrolling). They state that even if there is not real parallelism the efficiency should be better because of the avoided bank switching.
They tested this on a couple of benchmarks, and found that the efficiency of the area is effectively increased.

2.3.3 MPack: Global Memory Optimization for Stream Applications in High-Level Synthesis

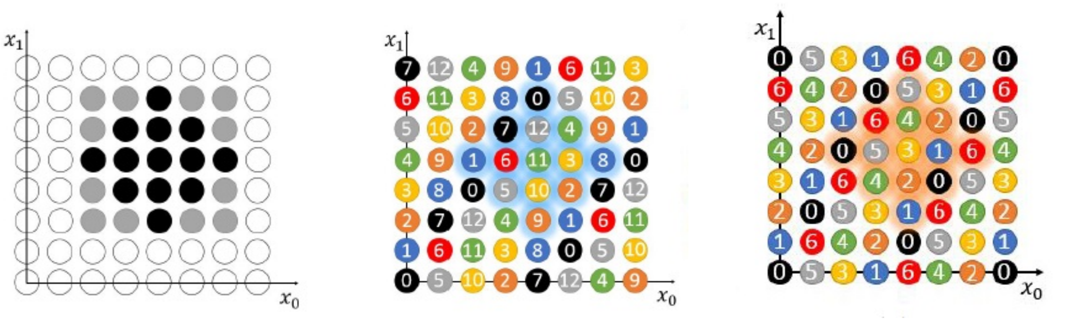
This article ([VC14]) suggests a tool that can help FPGA designers in exploration of the trade-off between memory usage and throughput. The problem they target is the complexity of fine-tuning the use of limited on-chip memory storage among many buffers in an application of stream computing.
They consider the programming model for FPGA in which the programmer focuses on designing computation kernels and relies on an automated optimization of memory, thus splitting the effort of writing the program in two parts: first the creation of the core logic, and then the organization of the memories and the datapath. The suggested tool, Mpack, performs pushbutton on-chip memory optimization for streaming applications by adding some High-level pragmas and automatically generating the optimal buffer packing approach with the highest data throughput (1st available target) and minimum memory budget (2nd available target, given the target throughput it minimizes the BRAMs used). MPack automatically generates a memory solution, with the goal to maximize throughput at the given BRAM budget. The use of Mpack allows the user to focus on optimizing computation, without concern for low-level memory optimization. This article does not use polyhedral analysis, and the suggested tool does not improve max throughput, it just help finding a fast

way to explore trade-offs between on chip memory utilization and throughput (if the max throughput is higher than the necessary, some of it can be sacrificed to lessen the area consumption).

2.3.4 Efficient Memory Partitioning for Parallel Data Access in Multidimensional Arrays

This article ([MYO$^+$15])suggests a different way to partition memory and parallelize accesses based on pattern. An access pattern is a map of the read that a statement has to do for every instance. For example if there is the statement x[i][j] = y[i][j-1]+y[i-1][j]+y[i+1][j]+y[i][j+1] it means the program has to access to the left, up, down and right cell of the matrix to determine the value of the considered position. This approach can be used only with problems that have to access more than one place in the multidimensional array.
In this work, they propose a memory partitioning algorithm with limited bank number constraint in multidimensional arrays to realize low storage overhead. They formulate the memory partitioning as a multi-objective optimization problem, which is flexible to different design considerations such as minimal memory overhead, fast accessing speed or limited memory bank number.
The proposed solution is a general framework to resolve the memory partitioning problem, which has fast speed and low complexity. They suggest the use of a linear transformation guided by the pattern instead of the use of a global iteration on all linear transformations searching the best one. The complexity in this way drops from exponential to constant (instead of iterating on all possible solution, it just create the one the pattern suggests). This algorithm can reach the optimal solution if the number of points accessed in one iteration is less or equal than the number of available memory banks, otherwise it will find the best one (i.e. with the less clock cycle possible).
They provide an example with the Laplacian of Gaussian operation. This operation has 13 memory accesses, with the pattern in fig 2.11(a).

Since it is mandatory to have all the data stored in those cells in the same instruction, an optimal solution can be found with 13 or more banks, splitting the original array as shown in Figure 2.11(b).



(a) LoG pattern

(b) LoG optimal solution with 13 banks

(c) LoG suboptimal pattern with 7 banks

Figure 2.11: Laplacian of Gaussian pattern example

If for any reason the required number of banks is not available, the algorithm can be bounded to use an inferior number of banks, and if for example is given a bound of 10 max banks, will return the suboptimal solution in Figure 2.11(c), which only uses 7 banks.

They tested their algorithm on seven benchmarks, and they state that their algorithm is a lot faster and it wastes less space (arithmetic operation amount -93.7% and execution time -96.9% and the storage overhead -31.1%) than the one proposed by Wang in [WLZ^{+}13] It is important to notice that the values are to be compared to the execution time and the number of operation of the synthesis algorithm, and not the execution time and operations of the synthesized one. The obtained result is the same (1 operation per cycle) from the throughput point of view, there is an improvement from the area point of view since less space is wasted.

2.3.5 Summary

Before starting with the methodology proposed by this work, a short summary of the state of the art will be provided, trying to highlight the differences from the proposed solution. The first article, [WLZ^{+}13], is the starting point of other two works between the ones that have been analyzed. It uses polyhedral analysis trying to optimize the mapping of the data. The reason for this is that if they are all mapped on different banks, they can be retrieved fast using only one cycle. This is also the objective of [CG15] and [MYO+15]. The first of those two articles expands the problem, considering some overheads created from the splitting of the data and trying to minimize them, always using the polyhedral analysis. The second article avoids the use of the polyhedral analysis and consider a faster methodology that is the use of patterns instead of doing many calculations to reach the same target. The fourth article, [VC14], suggests the use of a tool that can automatically map memories after the calculation kernel has been created in a way that can optimize area utilization in burst applications, but without consider data splitting. This tool is unable to increase the throughput. A quick summary of all the pros and cons of the analyzed articles is provided in Table 2.3.

Article	Pros	Cons
[WLZ+13]	It is the starting point of most of the other work.	It is computationally heavy, so an heuristic mode has to be adopted.
	It can find optimal solution for multi-dimensional arrays.	Its heuristic method cannot guarantee the optimal solution.
	It is a method to provide all the reads necessary for a write.	No data reuse.
[CG15]	It highlights the problem of bank switching that is not considered in other works.	Leave all data dependency check to the HLS tool.
	It uses unroll to do more computation in the same time.	If there is data dependency, it cannot unroll.
	It increments area efficiency since the datapath is easier to control.	

Article	Pros	Cons
	It may have data reuse, if there are no data dependences and the unroll is possible.	
[VC14]	It separates the problem of memory allocation from the kernel creation.	It cannot improve the throughput.
	It can improve area efficiency.	It does not use polyhedral analysis at all.
[MYO^{+}15]	It introduces the use of patterns.	No data reuse.
	It can find optimal solution.	No throughput improvements with regard to Wang work.
	It is a method to provide all the reads necessary for a write.	
	It is computationally light with regard to Wang work.	

Table 2.3: State of art: summary

The suggested solution deals with the problem with a slightly different point of view: where some of the analyzed articles use the polyhedral analysis just to improve the mapping of the data on the different BRAMs, this thesis tries to use polyhedral analysis to split the problem (and sequentially the data) and create some kernels that are able to to the computation in a parallel way. The previous approach has a limit: it cannot instantiate more than one computation per iteration, even if is able to retrieve all the data at the same time, and this creates a limit on the throughput. The proposed solution permits an increase of the throughput, even if it is obviously a trade-off: more area is used to create the kernels, but the FPGA can do more than one computation at a time.

Another difference from the state of the art is the data reuse. In the state of the art this issue is not being considered, because there are no parallel computations so there is no need to share data. In this work doing a *read* and sharing among kernels can bring some improvement in performances.
The general idea is to try to improve the throughput, even if this means that the area will increase. This growth of the area is anyway being reduced to the minimum with expedients like resource sharing among the cores that are state of the art in the field of High Level Synthesis.

Chapter 3

Problem Statement

The subject of this work will be explained in the following sections. The first section (3.1) will present the objective of the work, and select the fields of applicability of the described solution. The second (section 3.2)will explain the assumptions made to justify some methodology decisions. The third section (3.3) will explore in depth the methodology adopted to obtain the stated objective. Then a complete example is provided in section 3.4. The last section (3.5) will be a summary of what explained before.

3.1 Objective

The objective of this work is to improve the performance of some kind of code generated by HLS. To achieve this result the Polyhedral Analysis will be used to analyze the source code and find the SCoPs. Then from the analysis of the SCoPs certain patterns will be identified. Once this is done, the array involved in this computation will be partitioned and more cores will be created, able to perform the same computation on different data in a complete parallel way, sharing the common data. Note that pattern recognition and data partitioning are necessary for the successive step. The proposed solution is not a general algorithm for every source code, but it will work under the assumption that the piece of code to be optimized is recognized as a SCoP, and the polyhedral model of the code can be extracted. Any source code can be submitted, but if no SCoP is found the code will not be modified. If the SCoP is found, the pattern analysis can be performed. In particular the target are operations that involve the use of matrices, like matrix multiplication, or iterative stencil codes (see 3.2).

The throughput will be incremented (and execution time diminished) at the price of area consumption. This should however bring an increased overall performance because is possible that the speedup increases linearly with regard to the number of kernels, while the area will have a sub-linear increase. This result is achievable because the computation are equally split in the parallel kernels, while it is possible that not all the components have to be created for every kernel thanks to resource sharing and other optimization that are state of the art in HLS.

3.2 Assumptions

As previously stated, this work is targeting a specific architecture (FPGA) and only a specific kind of code (compute intensive loops where the same operations are performed on many different data). In order to apply the algorithm proposed in the following sections, the source code must satisfy the requirements of the polyhedral model, that are listed in 2.1. Another constraint, added on top of the polyhedral model, is that for each statement analyzed there is only one write. This constraint is needed for the pattern recognition step, but does not limit anything since a statement in the polyhedral model is an instruction. A SCoP can have more than one statement.

A particular group of SCoP identified are the so called stencils. Stencil are iterative finite-difference techniques that sweep over a spatial grid, performing nearest neighbor computations. In these kind of operations each point in a multidimensional grid is updated with weighted contributions from a subset of its neighbors in both time and space. This sort of pattern is very common in scientific applications in such diverse areas as heat diffusion, electromagnetics, and fluid dynamics [DMV$^+$08]. A subclass of this operations are the iterative stencils, that are stencil computation executed repeatedly on the same values. The proposed solution can be applied to normal stencil code, but the partitioning of the data will create a bottleneck that would probably nullify the gain since there are optimized algorithm that can read from the memory only once per data computation, that is a limit not improvable. The proposed solution lose time reading from memory during the split process, and it would perform worse than those algorithm (such for example the ones analyzed in the chapter 2. The improvement of the proposed solution is due to the fact that with iterative stencils the split is done only once and the overhead will become negligible. For this reason this work will consider the stencil as iterative and their core executed more than one time.

Another constraint adopted is the number of port on a BRAM. Actually the number of ports for every block is two [xil15]. This number has been taken as constraint in developing the algorithm. This means that if the Block RAM has less port, the algorithm will not perform optimally (some *reads* will overlap and not all the *reads* will be instantiated in one iteration).

3.3 Methodology

The proposed solution is divided into four steps:

1. **polyhedral analysis:**
   polyhedral analysis is performed and the accesses and the domain are retrieved from the code.

2. **pattern recognition:**
   The polyhedral model is analyzed. Using the information from the *reads* and the *writes* retrieved from the polyhedral model, a pattern analysis is done. The objective of this step is to find if the code can be recognized as one of the accepted patterns.

3. **data partitioning:**
   The data from the matrices are split in sub-matrices. This allows to work with more data in a single iteration, bypassing the limit of memory operation imposed by the number of port of the BRAM. It is done using the result of the pattern recognition.

4. **code rewriting:**
   The code is modified to exploit the parallelism that has been found by the polyhedral analysis. This is done by creating more computational cores that will run simultaneously and that are able to share some *reads*.

The following sections will explain in details every step.

3.3.1 Polyhedral Analysis

The code is analyzed with a polyhedral tool to retrieve some information that are the basis for the following steps. The important information retrieved during this step are:

- The presence of one or more SCoPs in the analyzed code.
- For each statement of every SCoP the following information:
  - The domain of the statement.
  - The schedule function of the statement.
  - All the accesses functions of the statement.

In order to search the SCoPs the source code is analyzed as follows. The Control Flow Graph of the code is analyzed, and the biggest Single Entry Single Exit (SESE) region is selected. Then all the instructions in that region are analyzed, trying to prove that they respect the constraints of the polyhedral model. Those constraint have already been explained in the polyhedral model section, and are:

1. Control statements are do loops with affine bounds and if conditionals with affine conditions.
2. Affine bounds and conditions depend only on outer loop counters and constant parameters.

If an instruction is not manageable by the model, that instruction is marked as difficult and more SCoPs are created of smaller dimensions, with the remaining SESE regions of the original SESE. This procedure continues reducing the considered SESE until one of the two following circumstances happens:

1. The region contains only instructions that can be handled by the polyhedral model. It can be marked as SCoP.

2. The region is composed of one instruction only. That region is not manageable by the polyhedral model.

This methodology guarantees that the biggest SCoP is found for every analyzed code.

After the SCoP recognition all the other information are extracted. The domain is easily recovered from the loop bounds, the schedule function is recovered from the Loop Syntax Tree and the accesses from the memory addressing expressions.

Domain The domain of a statement can be described as all the constraints on the various iterators the statement depends on. When the SCoP is discovered the loop that defines the boundaries are defined. All the iteration variables of all the loops that are surrounding the statement are analyzed. Starting from the extern loop, the upper and lower bound of every loop are inserted in a matrix. This matrix has a number of rows that is greater or equal to the double of the number of loops involved in the statement if there are not parameters used to calculate the boundaries of the loop. If a parameter is used, one or two rows are added to add the constraints on the parameter itself. For every boundary it is mandatory to store its dependences from the iteration variables. For the loop

```
for (x=0; x<10; x++)
  for (y=0; y<100; y++)
   for (z=0; z<50;z++)
    A[x][y]= A[x][y]+B[x][z]*C[z][y];
```

the following matrix is produced:

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \end{pmatrix} \geq \begin{pmatrix} 0 \\ -9 \\ 0 \\ -99 \\ 0 \\ -49 \end{pmatrix}$$

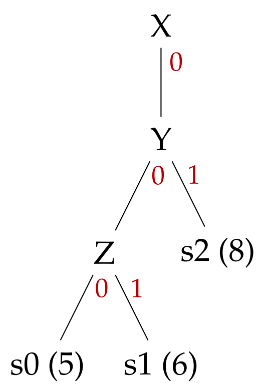
every row of the matrix represents one of the boundaries of the statement. Putting all the rows together gives all the boundaries of the statement. All the points contained in the boundaries are legal combination of values for the iteration variables.

Schedule functions The schedule function is needed to position a statement in the SCoP. It is used to represent any kind of ordering in the polyhedral model. A schedule for each instance of a given statement is provided using a function that depends on the iteration vector. Schedule functions are affine functions of the outer loop counter and the global parameters. The Loop Syntax Tree (LST) is analyzed and the position of the statement is obtained with regard to the iteration variables.

It is easier to understand with an example. Given the code:

```
1 for (x=0; x<10; x++)
2   for (y=0; y<100; y++)
3    {for (z=0; z<50;z++)
4     {
5       A[x][y]+= B[x][z]*C[z][y];
6       D[x][y]+= E[x][z]*F[z][y];
7     }
8     G[x][y]+= H[x][y]+I[x][y];
9    }
```

there is a SCoP containing three statements. Every statement has an own Domain, a schedule and his accesses function. The LST of this SCoP is given in the image that follows (the number in the parenthesis is the line of the statement in the code):



The schedule of the three statements can be easily read from the LST. To create the schedule of the statement it is enough to read the variable name in the node and associate the value on the edge that is linking the variable to the following variable or to the statement node. The statements are on the leaves of the tree. The tree can have any fan out. In the example the statement s0 (line 5 of the code) has the schedule $F_{s0}(\vec{x}) = (0, x, 0, y, 0, z, 0)^T$. That means that is the first instruction scheduled for all the variables in the loop. The statement s1 is the statement immediately after (line 6), and has the same variables. Its schedule is $F_{s0}(\vec{x}) = (0, x, 0, y, 0, z, 1)^T$. The following statement s2 (line 8) is out from the z loop, so has one less variable. Its schedule is $F_{s0}(\vec{x}) = (0, x, 0, y, 1)^T$. It is important to notice that the order is given by the numbers that follows the variables, that are always growing. If for example the last statement was situated out from the inner loop, but inside another loop with different variable as follows:

```
for (x=0; x<10; x++)
  for (y=0; y<100; y++)
   {for (z=0; z<50;z++)
    {
      A[x][y]+= B[x][z]*C[z][y];
      D[x][y]+= E[x][z]*F[z][y];
    }
    for (z=0; z<50;z++)
      G[x][y]+= H[x][y]+I[x][y];
   }
```

Its schedule would have been $F_{s0}(\vec{x}) = (0, x, 0, y, 1, z, 0)^T$.

Accesses A structure is needed to represent the accesses. The variable that are accessed in the polyhedral model are all considered as array. If a scalar variable is present in the loop, it will be considered as an array with a single cell and its index will always be 0. For every access function two information are needed: the kind of access and the place where the access is being done. The first is needed to explain if the access that is being described is (a *write* or a *read*). The second contains the information about the real memory access. This comprehend:

- The memory reference to the array
- For every dimension, the expression that is used to calculate the position being accessed, as a function of the iteration variables.

Given the following loop nest:

```
for (x=0; x<10; x++)
  for (y=0; y<10; y++)
   for (z=0; z<10;z++)
    A[x][y]= A[x][y]+ B[x][z]*C[z][y];
```

for every access to an array (A, B and C) the polyhedral tool will build a structure with two fields:

- a value that will record if the access is a *write* or a *read*.
- a structure with the following fields:
  - The memory address of the involved variable
  - For every dimension, the equation describing the position.

For example for the code of the previous loop nest, under the hypothesis that A is in position 1, B is in position 2 and C is in position 3 in the memory, these four structures will be created:

1. - WRITE
   - mem_pos = 1
   - dim_0 = x

- dim_1 = y

2. - READ
   - mem_pos = 1
   - dim_0 = x
   - dim_1 = y

3. - READ
   - mem_pos = 2
   - dim_0 = x
   - dim_1 = z

4. - READ
   - mem_pos = 3
   - dim_0 = z
   - dim_1 = y

3.3.2 Pattern Recognition

The algorithm for the pattern recognition works with some comparison between the *write* and the *reads* of a statement.
The algorithm is divided into four steps, as shown in pseudo-code 1. The first step (line 2-6) is needed to find the statement *write* and check it is unique (as per working hypothesis, see the assumptions). The second step compares the single *read* accesses with the *write* ones, searching for some properties of the code. The third step uses the properties found in the second to update some counters that will be used in the last step The last does the real pattern recognition using the counters and the order of the reads.
The recognized final patterns are the following:

1. rows only

2. row and column

3. stencils with reads on different array
4. stencils with reads and the write on the same array

and will be better described in the last step of the pattern recognition algorithm.

Algorithm 1 Pattern Recognition

```
1: procedure PATTERN RECOGNITION
2:     for $pos = 0$ to $accesses.length$ do
3:         if $write$ is not initialized && $accesses[pos]$ is a write then
4:             $write \leftarrow accesses[pos];$
5:         else if $write$ is already initialized && $accesses[pos]$ is a write then
6:             return error
7:     for $pos = 0$ to $accesses.length$ do
8:         $read \leftarrow accesses[pos];$
9:         $map[pos] \leftarrow Access\_Reduction\ (Dimension\_Reduction(write, read));$
10:    for $pos = 0$ to $accesses.lenght$ do
11:        $increment\_counters(map.[pos]);$
12:    if $!first\_recognition$ then
13:        $Check\_order\_conditions(map);$
14:    if $recognition$ then
15:        return $PATTERN;$
16:    else
17:        return $PATTERN\_NOT\_FOUND;$
```

All the steps of the algorithm 1 will now be explained more in detail in the following paragraphs.

Step1: Write research In this step all the accesses are analyzed, but only the field where the access type is stored. The loop check all the accesses found in the polyhedral analysis, until a *write* is found. When the *write* is identified, all the information related to the write access (see 3.3.1) are saved in a structure that will be used in the following step of the algorithm to do all the comparisons. The loop continues doing the same check and if another *write* is found throws error. This is done to check if working assumption that the SCoP only has one write is respected.

Step2: Access Analysis This step consists in two functions that are nested and are called for each *read*. The first of those function is the Dimension_Reduction function. It is called for every dimension of every *read*. In this function a dimension of the *read* is compared to a dimension of the *write*. The objective is to determine if those dimension accesses are equal or not. There are three possible code properties found by this step:

1. EQUAL_DIMENSION:
   This property is returned if the equation describing the dimension in the *read* is equal to the one of the *write*.

2. EQUAL_DIMENSION_BUT_CONST:
   This property is returned if the equation describing the dimension in the *read* is equal to the one of the *write*, except for a constant value.

3. DIFFERENT_DIMENSION:
   This property is returned when the difference between the two equation depends from one or more iteration variables

Using the following code as example this function is called six times, two for each *read*.

```
for (x=0; x<10; x++)
  for (y=0; y<10; y++)
   for (z=0; z<10;z++)
    A[x][y]+= B[x][z]*C[z][y];
```

The results for the two calls on array A are EQUAL_DIMENSION and EQUAL_DIMENSION.
The results for the calls on array B are EQUAL_DIMENSION and DIFFERENT_DIMENSION.
The results for the calls on array C are DIFFERENT_DIMENSION and EQUAL_DIMENSION.

The second function is the Access_Reduction. In this function the Dimension_Reduction function described above is called on all the dimensions and its results are used to compare the dimensions and find the property of the access The properties that have been found until now are:

- EQUAL
  All the dimensions have the same iteration variables and the same constants. It implies the *read* and the *write* have the same number of dimensions. It is recognized when every dimension check returns an EQUAL_DIMENSION (eg. A[x][y] = B[x][y]).

- W1_R2_EQUAL_FIRST
  The *write* is mono-dimensional, the *read* has two dimensions. The first dimension of the *read* is equal to the only dimension of the *write*. It is recognized if the dimension check between the first dimension of the *read* and the only dimension of the *write* returns an EQUAL_DIMENSION (eg. A[x] = B[x][y]).

- W1_R2_EQUAL_SECOND
  The *write* is mono-dimensional, the *read* has two dimensions. The second dimension of the *read* is equal to the only dimension of the *write*. It is recognized if the dimension check between the second dimension of the *read* and the only dimension of the *write* returns an EQUAL_DIMENSION (eg. A[x] = B[y][x]).

- W2_R1_EQUAL_FIRST
  The *write* is bi-dimensional, the *read* has only one dimension. The first dimension of the *write* is equal to the only dimension of the *read*. It is recognized if the dimension check between the first dimension of the *write* and the only dimension of the *read* returns an EQUAL_DIMENSION (eg. A[x][y] = B[x]).

- W2_R1_EQUAL_SECOND
  The *write* is bi-dimensional, the *read* has only one dimension. The second dimension of the *write* is equal to the only dimension of the *read*. It is recognized if the dimension check between the second dimension of the *write* and the only dimension of the *read* returns an EQUAL_DIMENSION (eg. A[x][y] = B[y]).

- W2_R1_EQUAL_DOUBLE
  The *write* is bi-dimensional, the *read* has only one dimension. Both the dimensions of the *write* are equal to the only dimension of the *read*. It is recognized if the dimension check between both the dimensions of the *write* and the only dimension of the *read* returns an EQUAL_DIMENSION (eg. A[x][x] = y[x]).

- EQUAL_BUT_CONST
  the *write* and the *read* have the same number of dimensions, and the same iteration variables. The only difference is a constant value added to one or more dimensions. It is recognized if all the dimension check between the dimensions of the *write* and the ones of the *read* returns an EQUAL_DIMENSION or EQUAL_DIMENSION_BUT_CONST (eg. A[x][y]=B[x+1][y-1]).

- DIFFERENT_ALL
  for matrices with more than one dimension, if all the check on the dimensions returns a DIFFERENT_DIMENSION. Notice that is not important if one of the dimension is equal to another one that is in a different position. From the example A[x][y] = B[y][z], the result is DIFFERENT_ALL even if the second dimension of the write is equal to the first of the *read*. Since this step is doing comparison dimension per dimension, the check are done between x and y (1^{st} dimension of both arrays) and between y and z ($2^{n}d$ dimension of the two).

- DIFFERENT_INDEX
  For matrices of one dimension, if the index is DIFFERENT_DIMENSION (eg. A[x] = B[y]).

- W2_R2_EQUAL_BUT_SECOND
  Both *write* and *read* are bi-dimensional. It is recognized when the check on the first dimension returns EQUAL_DIMENSION while the one on the second dimension return DIFFERENT_DIMENSION (eg. A[x][y] = B[x][z]).

- W2_R2_EQUAL_BUT_FIRST
  Both *write* and *read* are bi-dimensional. It is recognized when the check on the first dimension returns DIFFERENT_DIMENSION while the one on the second dimension return EQUAL_DIMENSION (eg. A[x][y] = B[z][y]).

- W3_R3_EQUAL_BUT_THIRD
  Both *write* and *read* have three dimensions. It is recognized when the check on the first and second dimensions returns EQUAL_DIMENSION while the one on the third dimension return DIFFERENT_DIMENSION (eg. A[x][y][z] = B[x][y][w]).

- W3_R2_EQUAL_BUT_FIRST
  The *write* has three dimensions, the *read* two. It is recognized when the check on the first and second dimensions returns DIFFERENT_DIMENSION while the one between the third dimension of the *write* and the second of the *read* returns EQUAL_DIMENSION (eg. A[x][y][z] = B[w][z]).

- W3_R2_EQUAL_BUT_SECOND
  The *write* has three dimensions, the *read* two. It is recognized when the check on the first dimension returns EQUAL_DIMENSION while the one on the second and the one between the third dimension of the *write* and the second of the *read* return DIFFERENT_DIMENSION (eg. A[x][y][z] = B[x][w]).

- SCALAR_VARIABLE
  the *read* has no dimensions. it is just a coefficient, but need to be noted and recognized because is used in the following steps.

The whole step works as follows: For each *read* the Access_Reduction function is called. In this function a couple of switch on the dimensions of the *write* and *read* governs the access to different pieces of code that have substantially the same structure but different values depending by the dimensions of the memory accesses. The structure of those is the following: first some checks are issued. Those checks call the Array_Reduction function for the dimensions that have to be compared. Some crossed checks may be needed (eg. the third dimension of a *write* with the second of a *read*) and this may cause the number of checks to be greater than the dimensions of the arrays. Then the result of those checks, combined in sequences of AND operations, are used as expression for a group of ifs. In case there is a match, that determines the final pattern of the access.

The upper bound for the number of checks is the product of the two dimensions (and up to now it has been reached only in case of a mono-dimensional array, where its dimension has to be checked against all the dimensions of the other array).

With the code below, the analysis of the A[x][y] read access will work as follows:

```
for (x=0; x<10; x++)
  for (y=0; y<10; y++)
   for (z=0; z<10;z++)
    A[x][y]+= B[x][z]*C[z][y];
```

The switches will bring to the branch where the write and *read* with two dimensions are analyzed. Then a check is issued on the first dimension, and the result is EQUAL_DIMENSION. Then a check is issued on the second dimension, and the result is EQUAL_DIMENSION. Then the ifs are evaluated. There are three ifs in this case: the one evaluating the W2_R2_EQUAL_BUT_SECOND pattern: its result is false, since it requires EQUAL_DIMENSION and DIFFERENT_DIMENSION. Then the second evaluates the W2_R2_EQUAL_BUT_FIRST pattern: its result is false, since it requires DIFFERENT_DIMENSION and EQUAL_DIMENSION. Then the third evaluates the EQUAL pattern: this is true, since it requires that all the dimensions have EQUAL_DIMENSION. For every access this work is done. The result of this step is a map with the following structure: As key it uses the identifier of the access, and as value it has the result of the Access_Reduction function.

array	value
read_1_key	EQUAL
read_2_key	W2_R2_EQUAL_BUT_SECOND
read_3_key	W2_R2_EQUAL_BUT_FIRST

Step3: Counting For every possible property of the access (the list is reported in 3.3.2, the ones returned by the Access_Reduction function), a counter is created. The map that is the output of the previous step is cycled, and every counter is incremented when an access is found with that pattern. This step is not enough for the recognition of all patterns because for some of them the order is important.
For the map produced by the example above:

array	value
read_1_key	EQUAL
read_2_key	W2_R2_EQUAL_BUT_SECOND
read_3_key	W2_R2_EQUAL_BUT_FIRST

The result of this step is counter_equal = 1, counter_equal_but_second = 1 and counter_equal_but_first= 1, and all the other counters are 0.

Step4: Pattern recognition A group of if based on the counters of the previous step detects which other checks are to do before the recognition of a pattern. Some patterns can be recognized just by counting, since the order is not important (eg. the row only: if the number of *read* - the number of SCALAR is equal to the number of EQUAL can be recognized immediately). Once those other checks are done, if something is recognized, it is returned. If no pattern is found then it returns PATTERN_NOT_FOUND.
This step uses the map produced by the second step and the counter produced in the third step as input and has the final pattern as output. As previously stated the recognized pattern are:

- **rows only:**
  This pattern is returned when all the involved matrices have the same equation describing the first dimension. It also accepts if scalar values are present. The grammar is:
  E(E | S)*
  where E is EQUAL and S is SCALAR

- **row and column:**
  This pattern is recognized every time that there is a loop where a matrix is

being read rows first, and the one immediately after is read columns first. For example it is the pattern of the matrix multiplication. Its grammar is:
$S^*E^+S^*(AS^*B)^+S^*$
where E is EQUAL, S are scalar values A and B are matrices. The couple of matrices A and B are paired. This means that if the first is found, to recognize the pattern it is important that the second is the one of the pair, and not a random matrix chosen between the set of all the possible second matrices. The recognized couples up until now are:

– W2_R2_EQUAL_BUT_SECOND and W2_R2_EQUAL_BUT_FIRST
– W1_R2_EQUAL_FIRST and DIFFERENT_INDEX
– W2_R1_EQUAL_FIRST and W2_R1_EQUAL_SECOND
– W3_R3_EQUAL_BUT_THIRD and W3_R2_EQUAL_BUT_FIRST
– W2_R2_EQUAL_BUT_SECOND and DIFFERENT_ALL
– W1_R2_EQUAL_SECOND and DIFFERENT_INDEX

where the first is the matrix A, and the second the matrix B. Scalar values can be anywhere. Some of these couples will need a little different handling during the split phase, but they are treated in the same way during the phase of code rewriting. This happens because the code rewriting where the matrices are split vertically is handled in the same way of the horizontal case, but it is important to notice that the arrays have to be divided differently in the tiling phase. Those couples will be recognized as row and column transposed (the last item of the previous itemize is the only example of this kind of pattern, up to now).

- **stencils** This pattern is recognized when the accesses are happening "near" the position of the write, with the same iteration variables. It is important to recognize how many dimensions are involved in the stencil calculation, because they are handled in different way. The grammar is:
$[E](B)^+$
where E is EQUAL and B is EQUAL_BUT_CONST

While recognizing the pattern the dimensions are counted and this information will be used both in tiling and in the core loop handling. There are two kind of stencil patterns, but the grammar is the same. They are recognized differently by checking the field of the access structure where the memory position of the array is stored: if there is at least one read that have the same memory position the stencil computation is done on the same array, so the STENCIL_SAME_ARRAY is returned. Otherwise the returned pattern is STENCIL_DIFFERENT_ARRAY.

For the following code:

```
for (x=0; x<10; x++)
  for (y=0; y<10; y++)
   for (z=0; z<10;z++)
    A[x][y]+= B[x][z]*C[z][y];
```

The first if is done on the counters, and this check is that the number of EQUAL_BUT_FIRST is equal to the number of EQUAL_BUT_SECOND. Once that this check is passed is mandatory to check the order, that is that the matrix following the EQUAL_BUT_SECOND is an EQUAL_BUT_FIRST. Since both the check are true, it recognize the rows and column pattern, without transposed.

3.3.3 Data Partitioning

To make all the kernels work in a fully parallel way it is necessary to split the data from the original array into smaller arrays, that will be located in different Block-RAM. The split functions selected are a block-cyclic partitions (see 2.2). The number of banks is the number of kernels and the block size is the product of the inner dimensions with regard to the dimension selected as guide. The dimension of the block has been chosen to have immediately successive rows or columns divided in different BRAM. This is not a requirement for all the patterns, but is necessary for the algorithm proposed for the handling of the stencils, where reading from the neighbor row is necessary and having more rows on the same BRAM can create bottlenecks. Two cases have been considered: horizontal split (where the guide dimension is the first) and vertical split

(guided by the second dimension).
It has been decided to use an easy kind of splitting because is enough to provide (as will be highlighted in the following step 3.3.4.3) all the needed data for a single iteration.

The decision of which one between the two split function is to be used is guided by the pattern recognition. Once the decision is done, all the array involved in the statement are labeled with a value that declare if that array has to be divided or not. This label is calculated working with the pattern of the whole statement and the pattern of that single access.

```
for (x=0; x<10; x++)
  for (y=0; y<10; y++)
   for (z=0; z<10;z++)
    A[x][y]+= B[x][z]*C[z][y];
```

For example, using the code above, the result was row and column without transposed. This means that the *write* and all the matrices labeled with a second level pattern EQUAL or EQUAL_BUT_SECOND have to be split horizontally while the matrices identified with the pattern EQUAL_BUT_FIRST have not to be split.
The dimensions of the split arrays are calculated in this step. It is done by taking the dimensions of the original array and dividing only the one dimension that is guiding the split by the number of kernels, the other dimensions are kept equal. A row of padding is added only to the divided dimension to avoid errors while handling arrays whose dimension is not a multiple of the number of kernels (since the integer division truncates the rest).

A detailed description and an example will be provided for all the possible split used.

Horizontal Split The original matrix is divided in blocks that have the dimension of one row. It is a simple block-cycle pattern (2.2) where the number of banks is equal to the number of parallel kernels. Every bank contains a sub-matrix. This split becomes a cyclic pattern whenever working on one-dimensional arrays. The objective of this split is to have immediately successive rows mapped onto different BRAMs.

The example is the following: given the matrix (4, 6)

$$Arr = \begin{pmatrix} a & b & c & d & e & f \\ g & h & i & j & k & l \\ m & n & o & p & q & r \\ s & t & u & v & w & x \end{pmatrix}$$

and a parallel degree of 2, the result are two arrays of dimensions (2, 6) organized as follows:

$$Arr_0 = \begin{pmatrix} a & b & c & d & e & f \\ m & n & o & p & q & r \end{pmatrix}$$

$$Arr_1 = \begin{pmatrix} g & h & i & j & k & l \\ s & t & u & v & w & x \end{pmatrix}$$

Given this split function the code that follows

```
for (x=0; x<10; x++)
  for (y=0; y<10; y++)
   for (z=0; z<10;z++)
    A[x][y]+= B[x][z]*C[z][y];
```

can be modified and the one below is produced.

```
for (x=0; x<5; x++)
  for (y=0; y<10; y++)
   for (z=0; z<10;z++)
    {
    A_0[x][y]+= B_0[x][z]*C[z][y];
    A_1[x][y]+= B_1[x][z]*C[z][y];
    }
```

This code performs the same operations than the first but is able to parallelize the computation (it can perform two multiplications, if there are enough resources). This happens because the A_0 and B_0 arrays contain the even rows of the original matrix, while the A_1 and B_1 ones have the odd rows.

Vertical Split This splitting function changes the target dimension in such a way that even operation with matrices are possible where the variable to split

is the second and not the first (eg. w[i] = w[i] + A[j][i] * z[j]). The variable to split is the second because the write has always to be split or there would not be a parallel computation. It is still a block-cycle pattern, but with less than three dimensions it is reduced to a cyclic partitioning. The objective is to have two immediately successive columns mapped onto different BRAMs. In this way data from the neighbor columns are available in the same cycle.
The example is the following with the same (4,6) matrix:

$$Arr = \begin{pmatrix} a & b & c & d & e & f \\ g & h & i & j & k & l \\ m & n & o & p & q & r \\ s & t & u & v & w & x \end{pmatrix}$$

and a parallel degree of 2, the result are two arrays of dimensions (4, 3) organized as follows:

$$Arr_0 = \begin{pmatrix} a & c & e \\ g & i & k \\ m & o & q \\ s & u & w \end{pmatrix} \qquad Arr_1 = \begin{pmatrix} b & d & f \\ h & j & l \\ n & p & r \\ t & v & x \end{pmatrix}$$

Given the following code, where the iterator of the result is equal to the second of the matrix and the horizontal split cannot be performed

```
for (x=0; x<10; x++)
  for (y=0; y<10; y++)
    A[x]= B[y][x]*C[y];
```

It can be modified with a vertical split and the one below is produced.

```
for (x=0; x<5; x++)
  for (y=0; y<10; y++)
    {
    A_0[x]= B_0[y][x]*C[y];
    A_1[x]= B_1[y][x]*C[y];
    }
```

The result is the same (two parallel computation available).

3.3.4 Core Loop Handling

Once the loop has been identified, and the data have been split, then the core instructions of the loop need to be parallelized.

This step is guided by the pattern. The general idea is that all the instructions have to be replicated and each core will have to work with different data. Another issue is linked to the buffering: especially in stencils some data are reused in the following computation. This reuse can increase if more than one computation per cycle is being done. Reusing data can speed up computation because if the data are stored in a register recovering them is much more faster than re-reading from memory.

In order to obtain this result the pattern must be unequivocally recognized, and using this information some of the *reads* to the BRAM can be substituted with register accesses.
As explained in the patter recognition section (3.3.2) there are four possible pattern to handle. The following sections will explain in detail how each identified pattern is handled.

3.3.4.1 Rows only

This pattern is the easiest one. The core can be replicated just by splitting all the involved matrices horizontally and duplicating every instruction. Every new core of the loop is working on different data, as can be seen in Figure3.1, where the lines are linking the data needed for one operation: it is evident that the lines have no common points. Looking at the example code is clear that every matrix involved in this operation is split, and every accessed cell is used only in one of the operations. There is no data sharing, just pure replication of the cores. The code is transformed as follows: from the source

```
for (i = 0; i < _PB_NI; i++)
 for (j = 0; j < _PB_NJ; j++)
  for (k = 0; k < _PB_NK; k++)
   E[i][j] += A[i][k] ;
```

to the transformed one

```
for (i = 0; i < _PB_NI; i+=4)
 for (j = 0; j < _PB_NJ; j++)
  for (k = 0; k < _PB_NK; k++)
  {
   E[i][j] += A[i][k] ;
   E[i+1][j] += A[i+1][k] ;
   E[i+2][j] += A[i+2][k] ;
   E[i+3][j] += A[i+3][k] ;
  }
```

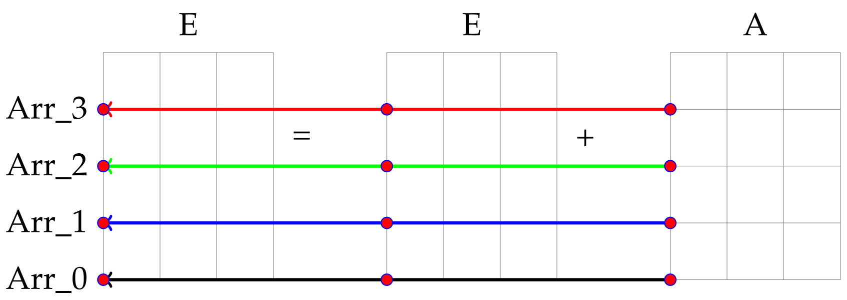


Figure 3.1: Row Only, example of accesses grids.

3.3.4.2 Rows and Columns

In this pattern, splitting all the involved matrices is not enough, because every row of the first has to interact with every column of the second.

The approach that has been used is the following, the *write* matrix and all the matrices that share the first iteration variable with the *write* are split, while all other matrices are not split at all and their *reads* are shared among the cores. In Figure 3.2 an example with four kernels: four operations are being performed during the same iteration, and every colored line link the *reads* needed to perform one of those operations. As can be seen, from the last matrix (B) only one value per iteration is read. This data can easily be shared, as the picture suggests (all the lines come from one single point in matrix B). The code is transformed as follows. From the source:

```
for (i = 0; i < _PB_NI; i++)
 for (j = 0; j < _PB_NJ; j++)
  for (k = 0; k < _PB_NK; k++)
   E[i][j] += A[i][k] * B[k][j];
```

to the transformed one:

```
for (i = 0; i < _PB_NI; i+=4)
 for (j = 0; j < _PB_NJ; j++)
  for (k = 0; k < _PB_NK; k++)
  {
   E[i][j] += A[i][k] * B[k][j];
   E[i+1][j] += A[i+1][k] * B[k][j];
   E[i+2][j] += A[i+2][k] * B[k][j];
   E[i+3][j] += A[i+3][k] * B[k][j];
  }
```

It is evident the fact that only the matrices with the first dimension iterating on the first dimension of the write (in the example the variable i) are being split, while the others are just shared.

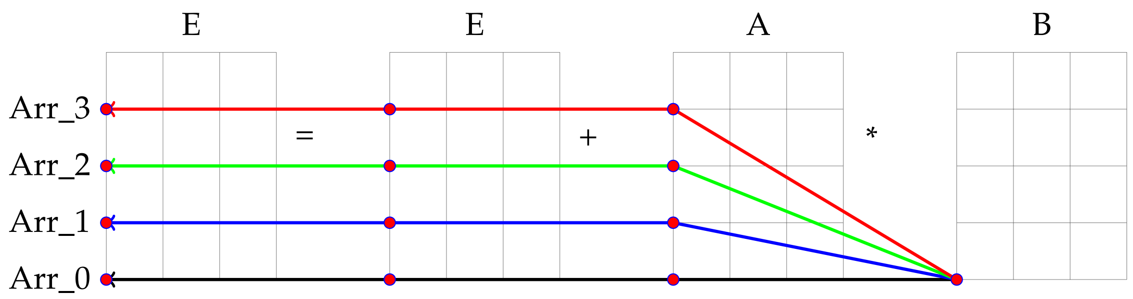


Figure 3.2: Row - Column, example of accesses grids.

The algorithm is:

- split horizontally the matrices that have the first index equal to the write index
- duplicate every instruction related to the split matrices (this means not only the instruction of the multiplication but also for example an assignment done on that variable).

In this way the operation itself will use the same second operand among all cores (doing only one *read* instead that x, where x is the parallel degree).

All the patterns involving a variant of the rows and column access to a matrix have been handled in the same way (changing the matrix to be split if there is a transposed matrix, but keeping the idea of splitting the result and the operands that have the same first iteration variable and share the other).

3.3.4.3 Stencils with reads on different arrays

The objectives for these patterns are the following:

- Reuse as much as possible the data. This happens because for a write in a stencil are needed more than one *read*, and those *reads* often come from near cells in the matrix, not always on the same row but often in the upper or lower rows.
- Limit the *read* from the same memory bank.
- Compute more than one write per cycle.

Before explaining the algorithm, a couple of definitions are needed:

Definition 5. ***Height of the stencil***
The height of the stencil is the number of rows of the matrix that are involved in a single iteration of the stencil.

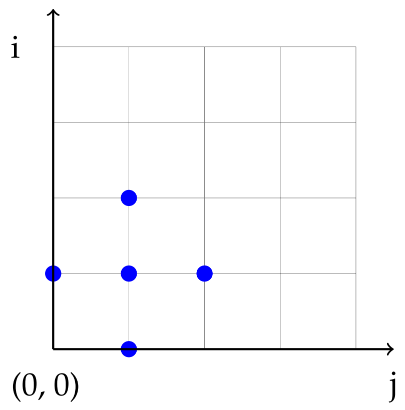


Figure 3.3: example of the reads of a stencil (a 2D-Jacobi). The five points are the reads that have to be done for the write in position 1,1. The height of this stencil is 3

An example of height of the stencil is reported in Figure 3.3, with a Jacobi 2D pattern. In order to do more than one computation per cycle, the computations on different rows are done in the same time. This will create a pattern with an altered shape (since some of the *reads* are overlapped). This "superpattern" is shown in Figure 3.4, with a parallel grade of four.

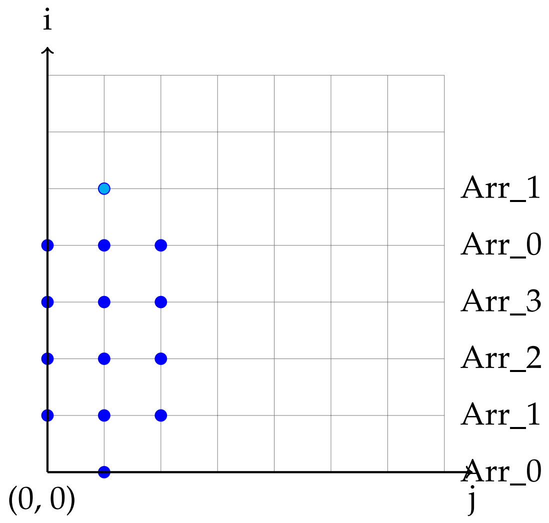


Figure 3.4: shape of the data needed to compute four parallel writes for the Jacobi 2D pattern. it can be seen that the shape of the pattern is quite different with regard to the original Jacobi pattern.

Definition 6. ***Front of advancement***
The front of advancement is defined as the line joining the cells of the matrix that have to be read in every cycle. In every row, only the rightmost cell is part of the front of the advancement. The height of the front is equal to the desired parallel degree plus the height of the stencil -1.

An example of a front of advancement is provided in Figure 3.5. The matrix is then decomposed horizontally, so that every row is mapped on a different sub-matrix (obviously as previously described the number of sub-matrices is equal to the parallel degree). Every sub-matrix is mapped on different memory bank. A copy of the instruction is created for every kernel. This will create a number of replicated kernels that are doing the computation on consecutive rows.

The algorithm that permits to obtain the three objectives listed above works in the following way: in the initialization phase reads from the memory the accesses that are not in the front of advancement that are needed in the first computation, and saves them in registers. After that, for every cycle, reads from the memory all the accesses in the front. The other accesses needed for

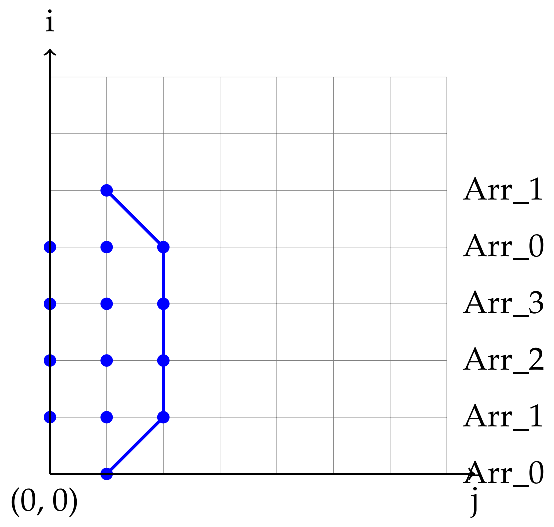


Figure 3.5: shape of the front of a four writes Jacobi 2D pattern.

the writes are saved in the registers. After the read, saves in the registers the accesses that will be needed in the next cycle. The shape of the front depends from the shape of the original stencil. To minimize the number of the accesses to the same memory bank is important the following constraint:

Definition 7. ***Constraint on the parallel degree:***
The parallel degree must be more or equal to the height of the stencil -1.

If this constraint is not satisfied there will be an overload of operations on some bank that will invalid the gain of this algorithm. In order to better explain this an example is provided. In Figure 3.6 a bigger Jacobi pattern (with height of the stencil 5) is represented. If this stencil is parallelized with a parallel degree of two, that is a violation to the constraint just explained, the result is the one in Figure 3.7(a). Three *reads* are issued for every array, and they will not be parallel if the number of port is two. If the constraint is satisfied the result is shown in Figure 3.7(b).

The last point of the algorithm is the substitution of the *reads* not on the front with registers. Since for a write all the data have to be ready in the same iteration of the loop, part of them will be stored in registers. The number of register needed can be calculated starting from the dimension of the original stencil. The formula is: $num\_register = parallel\_degree * (central\_row.length() - 1) +$

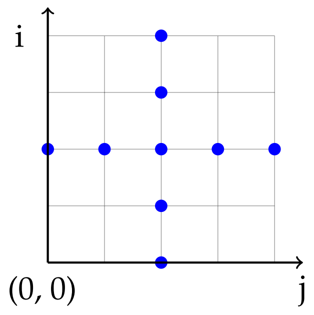


Figure 3.6: example of the reads of a stencil with height five. For every write needs to read data from five rows of the matrix.

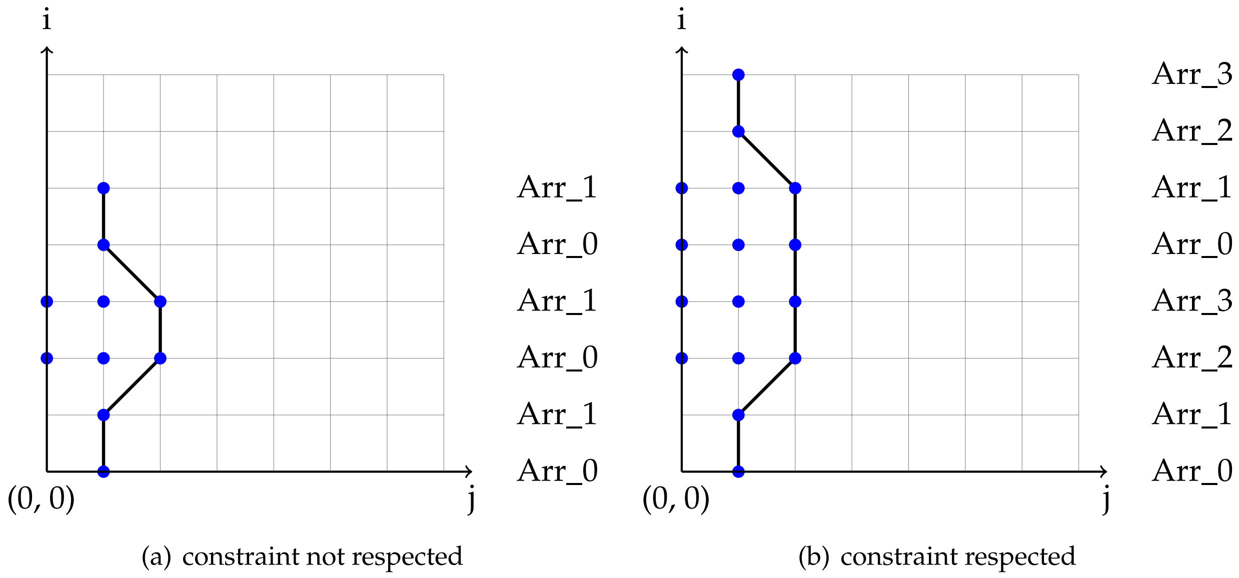
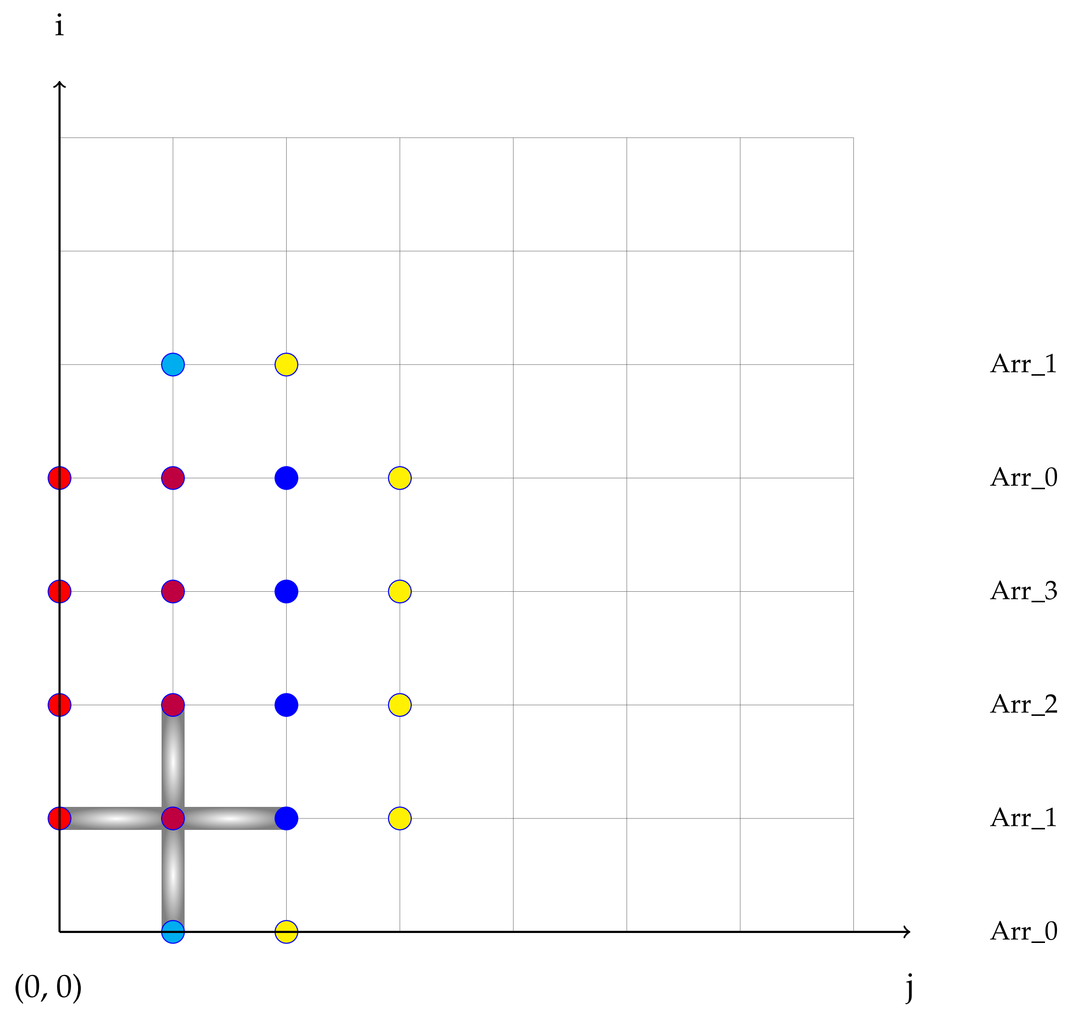


Figure 3.7: as can be seen from the pictures, the left image does not respect the constraint and has three read on every array, that cannot be instantiated in one cycle. the right image instead respects the constraint and even if it does instantiate more reads, those operations are on different arrays, with a maximum of two operations per array, that can be instantiated in only one cycle.

$sum(other\_row)(other\_row.lenght() - 1)$ where the -1 is needed because the last cell of every row is being read from the front, the central_row is the row where the stencil is centered, the other rows are the upper and the lower ones. It is important to note that the other rows must be considered only once because of the overlapping. When analyzing a *read* during the substitution step, two information are needed:

1. position of the *read* with regard to the write (this information is available from the pattern recognition).
2. number of the kernel on which the substitution is taking place.

From this information it can be understood if the *read* has to be left in that place (because is on the front) or has to be substituted because is already on a register. It is mandatory to remember the overlapping: a *read* in position (0,1) in the kernel 0, it is the same cell from the position (0,0) of the kernel 1, and so on. This must be noted because to maintain data consistency with the split of the array the needed data are never in the following row of the same array, but are mapped on another array (see 3.3.3, is explained why the for every change of row the sub-matrix is changed) and from there has to be retrieved (modular arithmetic on the number of kernel is used to perform this calculations). Often a *read* is shared, if that happen obviously it have to be instantiated only one time and then shared among kernels. The sharing operation can be handled with those two informations, the kernel that is being done and the position of the *read* in the pattern. In Figure 3.8 an example of how the *reads* are organized is provided. The upper and lower line do not need to be saved, because the length is 1 and this means that those lines *reads* are included in the front. The central row length is three, and this means that for every kernel two registers are needed. Since the kernels in the example are four, eight registers are needed. Those register needs to keep trace of the last two *reads* for each of the central row.



● read of the front, to be saved in registers
● read of the front, not to be saved in registers
● read of the next front
● read from register, not to be saved for next iteration
● read from register, to be saved for next iteration

Figure 3.8: example of the sequence of reads, with a parallel degree of four: red and purple are the position that need to be loaded before the first cycle, blue and cyan are the position read during the first run, blue and purple are the register saved after the first run, and the one needed for the second run. yellow are the position read in the second run, and so on. The gray cross highlights the five reads that have to be done for the original pattern, for the write in position 1,1.

Finally, an example on an iterative Jacobi pattern is provided. The source code follows:

```
 for (t = 0; t < _PB_TSTEPS; t++)
  {
   for (i = 1; i < _PB_NI -1; i++)
    for (j = 1; j <  _PB_NI -1; j++)
     B[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][1+j] + A[1+i][j] + A[i-1][j]);
   for (i = 1; i <  _PB_NI -1; i++)
    for (j = 1; j <  _PB_NI -1; j++)
     A[i][j] = B[i][j];
  }
```

And the output of the algorithm is the one that follows

```
int registers[2][4];
int temp_1, temp_2, temp_3, temp_4;
 for (t = 0; t < _PB_TSTEPS; t++)
  {
   for (i = 1; i < _PB_NI -1; i+=4)
   {
   registers[0][0] =A[i][0];
   registers[0][1] =A[i+1][0];
   registers[0][2] =A[i+2][0];
   registers[0][3] =A[i+3][0];
   registers[1][0] =A[i][1];
   registers[1][1] =A[i+1][1];
   registers[1][2] =A[i+2][1];
   registers[1][3] =A[i+3][1];
    for (j = 1; j <  _PB_NI -1; j++)
    {
     temp_1 =  A[i][1+j];
     temp_2 =  A[i+1][1+j];
     temp_3 = A[i+2][1+j];
     temp_4 = A[i+3][1+j]
     B[i][j] = 0.2 * (registers[1][0] + registers[0][0] + temp_1 + registers[1][1] + A[i-1][j]);
     B[i+1][j] = 0.2 * (registers[1][1] + registers[0][1] + temp_2 + regiters[1][2] + regiters[1][0]);
     B[i+2][j] = 0.2 * (registers[1][2] + registers[0][2]  + temp_3 + regiters[1][3] + regiters[1][1]);
     B[i+3][j] = 0.2 * (registers[1][3] + registers[0][3]  + temp_4 + A[4+i][j] + regiters[1][2]);
     registers[0][0] =registers[1][0];
  registers[0][1] =registers[1][1];
  registers[0][2] =registers[1][2];
  registers[0][3] =registers[1][3];
  registers[1][0] =temp_1;
  registers[1][1] =temp_2;
  registers[1][2] =temp_3;
```

```
    registers[1][3] =temp_4;
        }
      }
      for (i = 1; i <  _PB_NI -1; i+=4)
        for (j = 1; j <  _PB_NI -1; j++)
          {
          A[i][j] = B[i][j];
          A[i+1][j] = B[i+1][j];
          A[i+2][j] = B[i+2][j];
          A[i+3][j] = B[i+3][j];
          }
    }
```

It is important to note that the four involved rows are mapped on different BRAMs. As described in 3.3.3 the sequential rows are mapped into different arrays. For this figure the array has been kept the same to highlight the generated parallelism. The actual memory *reads* and *writes* are the operations with the A and B arrays, all the other variables are temporaries.

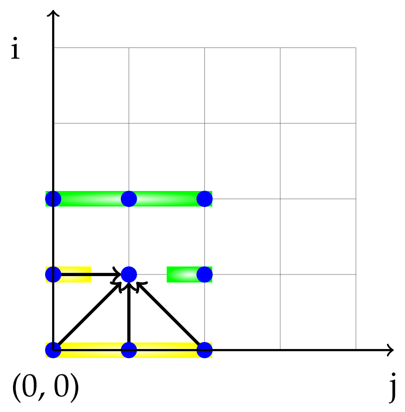


Figure 3.9: Seidel access pattern

3.3.4.4 Stencils with reads from the same array of the write

Some additional attention has to be done when dealing with a stencil that reads and writes on the same array. This because the data dependences have to be maintained and simply applying the algorithm as has been described can lead to wrong solutions because some data have to be read after the iteration and some before. For example the Seidel pattern whose code is:

```
for (t = 0; t <= _PB_TSTEPS - 1; t++)
    for (i = 1; i<= _PB_N - 2; i++)
      for (j = 1; j <= _PB_N - 2; j++)
        A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
                   + A[i][j-1] + A[i][j] + A[i][j+1]
                   + A[i+1][j-1] + A[i+1][j] + A[i+1][j+1])/9.0;
```

and its access pattern is shown in Figure 3.9. Since it is working with the same matrix, a problem with the data dependence arises. In Figure 3.9 the data dependences are drawn with black arrows. It can be seen that the points with the yellow background need to be updated before the calculation of the middle point, the ones with the green shade need not to be updated. This will not happen when proceeding with the front in a parallel way with the algorithm previously described, because the front is a vertical line. The values on the left have been updated in the previous iteration (even if some writes need to work with not updated data) and on the right they will not be updated until the next iteration (but some of the writes need to have them already calculated). Moreover the same point that is being calculated in the first kernel needs to be used by the second kernel. In Figure 3.10 can be better understood what has just been explained: the yellow and the gray shades cover the *reads* of the iteration calculating point (2,1), while the gray and the green area are the *reads* needed for the point (2,2). The blue line highlights the sequential loop updates. The points in (3,1) and (1,2) create iteration conflicts: the point in 1,2 has been updated in the previous iteration, but the calculation of (2,1) requires a not updated value, while (2,2) requires the updated value. The opposite happens for the point (3,1), which is not updated (and the calculation of (2,1) requires that point not updated) but the point (2,2) requires that that point has already been updated. Moreover the point (2,1) is needed in the calculation of (2,2) as the updated version. In order to avoid this problem a modification of the algorithm

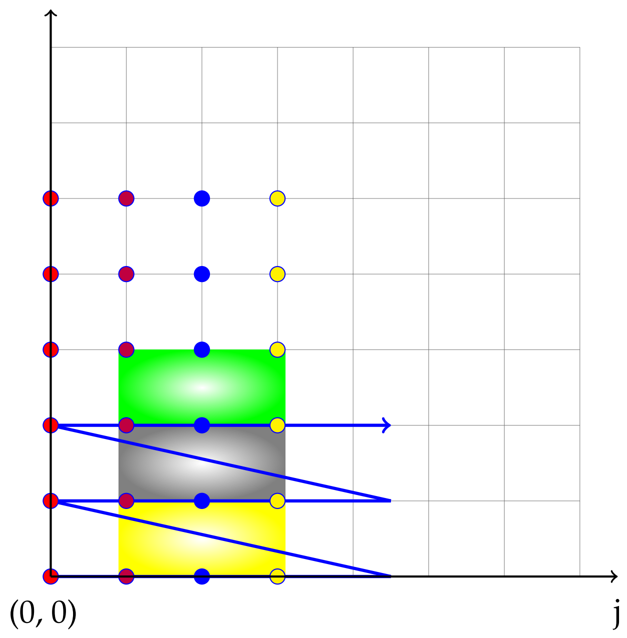


Figure 3.10: Same matrix overlap issues

has to be introduced when working with a stencil pattern that reads and writes on the same matrix. This algorithm works only with matrices (more than 1 dimensional arrays). The reason for this is that in a vector the data dependence is between a cell and the adjacent ones, and there is no way to split the single array and do parallel computations because every cell need the updated version of the previous one and the not updated version of the following one. For

matrices on the other hand the idea of working with a front composed of more than one row, limiting the number of *reads* from the banks and maximizing data reuse will be kept. This will be done moving the involved operations in order to diagonalize the front and avoiding the overlapping of iterations for the same cell. This will obviously introduce some overhead at the beginning and in the end of the loop (some operations cannot be done in a parallel way) but it is possible to apply the algorithm for the core of the loop.
The number of registers is increased, and the initialization has to be modified. Depending from the width of the upper and lower row of the stencil, some operations has to be scheduled in the initialization phase. Moreover to generalize the algorithm, an approximation on the stencil figure will be introduced. The approximation is the following: all the stencil figures are reduced to squares where the original figure is inscribed, centered in the same point. To calculate the side of the square an intermediate value (called horizontal max) is introduced. The horizontal max can be calculated, and it is the max absolute value among all the distances between the centered column of the stencil and the column value of the point. The side of the square is the max between the height of the stencil and 2*horizontal max+1. This allows to handle any strange stencil figure in the same way, but may create a number of register bigger than the needed one.

To schedule the number of registers and to handle the position of the cells it is mandatory to define a value that will be used to keep trace of the space left between two different computations:

Definition 8. ***gap of the stencil:***
The gap of the stencil is the distance between two columns where the stencils are centered in two successive rows that have to be left to avoid conflicts. It can be seen as the slope of the front. This value can be calculated as the leftmost value of a row of the square minus the central value +1 (ie. half the length of the side, rounded up).

Once calculated the gap, the number of operation that have to be computed "manually" before starting with the loop is equal to (gap)*(parallel_degree-i) where i is the number of the kernel that is being considered (1 if the first, 2 if the second, x if the last). Note that the last kernel does not have to compute any operation before starting. Note also that there is no dependence at all between the computation done in the different kernels.

For example in Figure 3.9 the gap is two: the pattern is already a square so it does not need an approximation, and its width is 3. This means that one write can be instantiated every two columns (as can be seen in Figure 3.11). The same figure shows the modified front, the increased number of registers and the division between already updated in the stencil iteration and the still not updated ones. The green line, that is the line linking the writes that are being calculated in the iteration, highlights the separation between the registers containing the updated registers (the brown ones) and the one that still have to be updated (the purple ones). The blue line is the front of advancement of this pattern. It is evident that with this solution the data dependences are respected: every green point requires that the lower points and the one of the left are already updated while the right one and the upper ones have to be the one computed in the previous iteration of the stencil. It can also be seen that the four computations are completely independent, and can be executed in the same instant.

The data that are shared are less, and there are more registers, but the number of accesses to the memory (which are the bottleneck) is the same with respect to the algorithm working with *reads* and *writes* on different matrices. The number of registers needed is symmetrical taking the center as reference, and can be calculated as two times the sum of the number of registers of the rows needed to reach the central rows plus the number of registers of the central rows. The central rows can be in odd or even number, depending by the parallel degree. If the number of kernels is lesser than the height of the stencil the central rows will not exist. This may happen when half of the number of total rows is equal to the number of rows needed to reach the central rows (that is the height of the stencil -1). It cannot be less because the constraint on the height is still valid. For the upper rows the number of registers needed is width-1+x*gap, where x is the number of that row -1 (ie. 0 if it is the first line, 1 if is the second and so on until the height of the stencil is reached.) Those rows can be seen as an initialization of the structure The central rows are all the other row from when the max size is reached to the half of the total number of lines. The number of registers for each of those rows is width-1+(height-1)*gap. Then from the half it decrease with the same pattern of how it is increased (may have some max rows, then start decreasing with lines that are symmetrical with regard to the top ones).

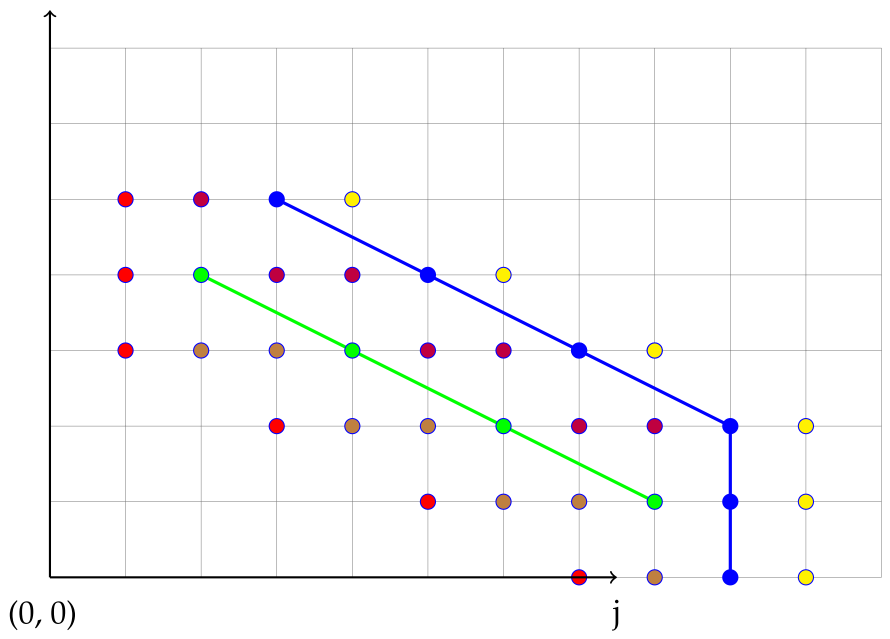
For the code used as example, reported below, the height is three.

```
for (t = 0; t <= _PB_TSTEPS - 1; t++)
    for (i = 1; i<= _PB_N - 2; i++)
      for (j = 1; j <= _PB_N - 2; j++)
        A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
                   + A[i][j-1] + A[i][j] + A[i][j+1]
                   + A[i+1][j-1] + A[i+1][j] + A[i+1][j+1])/9.0;
```

There are two rows needed to reach the central part, that have 2 and 4 registers: the width is 3, gap is 2. Applying the formula respectively are obtained from 3-1+0*2 and 3-1+1*2). The central lines are two (6 total rows, derived from height -1 + parallel degree. the number of upper lines (height-1) has to be subtracted twice) The number of registers needed in this row is width-1+(height-1)*gap (3-1+(3-1)*2) that is 6 in this example. The rows below are symmetrical so in this example their size is 4,2. The final structure is composed of 6 arrays with dimension 2,4,6,6,4,2. The core of the loop after the transformation is the following one:

```
for (t = 0; t <= _PB_TSTEPS - 1; t++)
    for (i = 1; i<= (_PB_N - 2)/4; i=i+4)
    {
    loop_initialization();
      for (j = 3*gap+1; j <= _PB_N - 2; j++)
       { A[i][j] = (reg[0][0] + reg[0][1] + A[i-1][j+1]
                   + reg[1][2] + reg[1][3] + A[i][j+1]
                   + reg[2][4] + reg[2][5] + A[i+1][j+1])/9.0;
        A[i+1][j-gap] = (reg[1][0] + reg[1][1] + reg[1][2]
                   + reg[2][2] + reg[2][3] + reg[2][4]
                   + reg[3][4] + reg[3][5] + A[i+2][j+1-gap])/9.0;
        A[i+2][j-2*gap] = (reg[2][0] + reg[2][1] + reg[2][2]
                   + reg[3][2] + reg[3][3] + reg[3][4]
                   + reg[4][2] + reg[4][3] + A[i+3][j+1-2*gap])/9.0;
        A[i+3][j-3*gap] = (reg[3][0] + reg[3][1] + reg[3][2]
                   + reg[4][0] + reg[4][1] + reg[4][2]
                   + reg[5][0] + reg[5][1] + A[i+4][j+1-3*gap])/9.0
        }
    loop_finalization();
    }
```

The first *write* will use the final part of all rows, and will have more memory accesses (see Figure 3.11, the write in 1,8). The other *writes* will only read the position in the bottom-right corner. For all the other reads, that have to be substituted with register reads, the position of the register that have to be used for the substitution is correlated to the gap and the number of the actual kernel. The position of the value that have to be read is known with respect to the center of the pattern (the write), in both directions (x and y). To obtain the position of the register containing that value, the minimum between (x_read-x_write)*gap and (parallel_degree-i-1)*gap has to be added, where x_write and x_read are the value of the line from where the read and the write are situated in the original matrix, and i is the number of the actual kernel. Looking at the previous example (Figure 3.11 when rewriting the third *write* (position 3,4) the *reads* needed are: the last registers of the fifth line (up in the figure, has length 4, the last two positions are needed) plus the memory read in (4,5), the central of the fourth line (positions [2],[3],[4]) and the first of the third line (positions [0],[1],[2]).



● read of the front, to be saved in registers

● read of the next front

● read from register, not to be saved for next iteration

● not updated read from register, to be saved for next iteration

● read from register, write to matrix and to register. has to be saved

● already updated read from register, it has to be saved for next iteration

Figure 3.11: Diagonalized front

3.3.5 Code Rewriting

To apply the suggested solution the source code has to be modified. This work is done in three steps.

Data Splitting The function that performs the partitioning is inserted in the loop, just before the beginning of the SCoP. It performs the modification described in the Data Splitting section (3.3.3). It is needed to divide the data from the source array to smaller arrays that will be mapped on different BRAMs.

Core Loop Modification The work described in Core Loop Handling section (3.3.4) is performed. In this step the source code is heavily modified as can be seen from the examples reported in that section.

Data Rebuilding This last step is needed to recreate the original array with the computed data in the correct positions. It is just a reverse operation of the split. It takes the x arrays (created in the split, and used during the computation in the different cores) and recreates the original one.

3.4 Complete Example

A complete example (from the pattern identification to the rewriting of the core) will now be presented to clarify the whole process. The chosen source code is the kernel of the Gemm test case, that is:

```
for (i = 0; i < 100; i++)
  for (j = 0; j <100; j++)
    for (k = 0; k < 50; ++k)
C[i][j] += alpha * A[i][k] * B[k][j];
```

When polyhedral analysis is performed on this code, one SCoP with one statement is found. The domain of the SCoP is:

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{pmatrix} * \begin{pmatrix} i \\ j \\ k \end{pmatrix} \geq \begin{pmatrix} 0 \\ -99 \\ 0 \\ -99 \\ 0 \\ -49 \end{pmatrix}$$

It has only one statement, and its schedule function is: $F_{s0}(\vec{x}) = (0, x, 0, y, 0, z, 0)^T$. The memory accessed produced are five, and they are:

1. - WRITE
   - mem_pos = 1
   - dim_0 = i
   - dim_1 = j
2. - READ
   - mem_pos = 1
   - dim_0 = i
   - dim_1 = j
3. - READ
   - mem_pos = 2
   - dim_0 = 0
4. - READ
   - mem_pos = 3
   - dim_0 = i
   - dim_1 = k
5. - READ
   - mem_pos = 4
   - dim_0 = k

- dim_1 = j

Then the following step, that is the pattern recognition, can start. The first pass in the pattern recognition is check that the write is unique, and is passed. During the following pass the reads are analyzed: The first read has the same equations of the write, and will be recognized as EQUAL The second read is recognized as SCALAR, since has only one dimension and it does not depends from any iteration variable but only a constant The third read has the first equation equal and the second different, and will be recognized as EQUAL_BUT_SECOND The fourth read has the first equation different and the second equal, and will be recognized as EQUAL_BUT_FIRST The output of this pass is the following:

array	value
read_1_key	EQUAL
read_2_key	SCALAR
read_3_key	W2_R2_EQUAL_BUT_SECOND
read_4_key	W2_R2_EQUAL_BUT_FIRST

Then the counters have to be updated (3rd pass) before the final recognition (4th pass). During the final recognition the branch where the counter W2_R2_EQUAL_BUT_SECOND has the same value of the counter W2_R2_EQUAL_BUT_FIRST will be taken.
Here the final check (that the W2_R2_EQUAL_BUT_FIRST is immediately after the W2_R2_EQUAL_BUT_SECOND) is performed. Since it is passed, the pattern is recognized as ROW_AND_COLUMN, without the transposed. Now the data partitioning has to be performed. The pattern recognized works with the horizontal split, in particular the arrays marked with EQUAL and W2_R2_EQUAL_BUT_SECOND have to be partitioned, while the SCALAR and the W2_R2_EQUAL_BUT_FIRST have to be untouched.

The core loop modification step can be done now. The algorithm described in 3.3.4.2 is performed. The data rebuild operation are inserted and the procedure is finished. Given a parallel degree of four, the output code is the following:

```
Array_hori_split(C, C_0,0,4);
Array_hori_split(C, C_1,1,4);
Array_hori_split(C, C_2,2,4);
Array_hori_split(C, C_3,3,4);
Array_hori_split(A, A_0,0,4);
Array_hori_split(A, A_1,1,4);
Array_hori_split(A, A_2,2,4);
Array_hori_split(A, A_3,3,4);

for (i = 0; i < 25; i++)
  for (j = 0; j <100; j++)
    for (k = 0; k < 50; ++k)
    {
C_0[i][j] += alpha * A_0[i][k] * B[k][j];
C_1[i][j] += alpha * A_1[i][k] * B[k][j];
C_2[i][j] += alpha * A_2[i][k] * B[k][j];
C_3[i][j] += alpha * A_3[i][k] * B[k][j];
    }

Array_hori_rebuild(C, C_0,0,4);
Array_hori_rebuild(C, C_1,1,4);
Array_hori_rebuild(C, C_2,2,4);
Array_hori_rebuild(C, C_3,3,4);
Array_hori_rebuild(A, A_0,0,4);
Array_hori_rebuild(A, A_1,1,4);
Array_hori_rebuild(A, A_2,2,4);
Array_hori_rebuild(A, A_3,3,4);
```

Where in the split and rebuild function calls the last two integers are the number of the array that has to be split or rebuilt, and the total number of kernels.

3.5 Conclusion

In this chapter the problem has been described and a solution to it has been proposed. It has been described how should work for the pattern identified, focusing on the iterative stencils that should be the cases where the suggested solution should perform better. A solution has also been proposed for complex cases where the write and read matrices are the same. In the following chapter how this solution has been implemented will be described.

Chapter 4

Implementation

In this chapter will be described how the methodology proposed by this thesis has been implemented. The first section (4.1) will highlight further constraints emerged during the implementation phase. This constraints are not imposed by the algorithm itself, but have to be added to use the tools that will be listed below. The second section (4.2)will explain the complete design flow. The third section (4.3) will briefly list the implemented classes and will describe the work done by them.

4.1 Constraints

The whole work is inserted in the context of the Panda Project, a framework designed to enable the research of new ideas in the HW-SW Co-Design field. More in detail, is created as a pass in the HLS framework that is *Bambu* [PF13].

Since this tool performs optimizations at a Gimple level, it has been necessary to work with the Gimple to introduce the optimizations. As has already been said, Gimple is the intermediate representation created by GCC while performing the compilation. It is necessary to extract the polyhedral representation of the source C code, and have something that is able to link the Gimple representation and the polyhedral model of the code. From all the tools previously analyzed, Graphite is the one selected to perform the polyhedral analysis on the code. The reason behind this choice is that Graphite is part of the GCC project, so it is easier to create links between the polyhedral model and the Gimple code. On the other side this choice introduce a constraint on the meaning of statement. In the previous chapter a statement has been intended as a single instruction. For Graphite a statement is a single basic block. For this reason it is possible to find a statement with more than one write. These statement have not been handled, since the algorithm is based on the comparison between the write and the reads. It is important to highlight that this limitation is only due to the implementation of Graphite and not a problem of the proposed algorithm.

As polyhedral model of a SCoP it has been chosen to use the OpenScop representation . This choice has been made because is one of the more supported formats and a C library to write and read file of this protocol is provided.

An xml file is created as link between the Gimple and the OpenScop. This file contains all the needed references between the Gimple information and its SCoP.

4.1.1 Plugins

Panda already uses GCC to extract the Gimple representation of a source C code and then performs its optimizations on that. The extraction of the Gimple is done with a plugin, that works at the end of the middle-end level of the compiler. It is the last operation in the target independent optimization flow. Another plugin for GCC that works with the already existing one has been written. This plugin implements the Graphite algorithm ([TCE+10], [SPJ+09]) for the SCoPs extraction, and produces an OpenScop file (.osl) for each SCoP identified. The plugin also produces a global xml file, containing the reference between every single osl file and the basic blocks involved in that SCoP in the Gimple. This second plugin has been written for GCC 4.9, and works at the same level of the other one.

4.1.2 Other Constraints

Another constraint introduced is an approximation on the handling of the stencil on different array (explained in section 3.3.4.3). In that chapter the figure of the stencil has been mantained and only composed in order to create a “superpattern” that can have a strange form. No approximations are needed in the algorithm. During the implementation of this part, to simplify the work, the stencil has been approximated to the smallest square containing the original stencil figure, before the composition to form the bigger figure. This is the same approximation made for the stencils that work on the same array (explained in section 3.3.4.4).

4.2 Design Flow

Two runs of *Bambu* are needed to obtain the Verilog description of the FPGA. For the first run the target file must be passed as source to *Bambu* (*Panda* HLS tool) with –enable-poly=x and –pretty-print=name_of_the_output.c, where x is the desired parallel degree. The output of this run is still C code, and is used as input for a new run of *Bambu*, in which the implemented part is not active, that will perform the HLS and produce the Verilog code.

This particular flow has been used because the output of the first step, being C code, can theoretically be used with other HLS tools.

4.3 Implemented classes

4.3.1 osl_wrapper

This class is just a wrapper around the OpenScop library. It has been created to wrap the C library in the cpp program, and to provide a class containing the SCoP and the methods needed to work on it.
The methods of this class are the ones needed to write and read the SCoP.

4.3.2 polyhedral_parallelizer

This is the class that performs all the work explained in the methodology chapter. It is a subclass of function_frontend_flow_step, the class used by Panda as basic class for every transformation of the intermediate representation. All the

work performed by the class will modify the gimple tree of the function passed as attribute, creating a different tree that will be passed to the following step of the normal *Bambu* flow.
The work of the class start with the call of the exec function, and is divided in two main steps:

- Data Retrieval
- Code Modification

The first step consist in reading all the necessary data produced by the plugin. The second step consists in performing all the work described in the methodology section. This consists in:

- New basic blocks creation.
- Identification of the old basic blocks that are part of the SCoP.
- Modification of the iteration variables and creation of the new arrays.
- Replication of the core instructions and partially dead code elimination.
- Introduction of the split-rebuild functions.

4.3.2.1 Data Retrieving

The first step of the work is the reading of the data produced by the plugin (all the SCoPs and the xml file). For every SCoP an instance of osl_wrapper is created. This item contains the SCoP, and is inserted in an ordered map where an unsigned int is the key. The number of the SCoP is decided by its name (they are called scop_x , where x is a number). This number is important because is used in the xml file as identifier for the SCoP.

After all the SCoPs are parsed the xml file is read. From the xml file the remaining information are extracted. From the root of the xml there is a list of children nodes called “function” Every function has as attribute the name and the tree node index where the function is declared in the Gimple. Inside every function there is a list of SCoP nodes that have as attribute the name of the osl file where it is described, the loop index and the involved basic blocks. these SCoP nodes have as children a list of memory identifiers, that are the correspondence between the id used in the SCoP file for an array and the tree node where that array is declared in the Gimple.

4.3.2.2 Code modification

The transform_loop function performs all the modification at a Gimple level of the code of the function. First given the function id, it is issued the retrieval of the information of the data related to that function. The whole work is based on two loops that are nested and repeated the extern one once per SCoP, and the inner one once per basic blocks of the SCoP. This means that the work described is done once per statement (statement and basic blocks are the same thing for Graphite). The first run of the inner loop starts with the modification of the basic block structure of the function. New basic blocks are created, and new iteration variables and their conditions are instantiated in this step. This new basic blocks have an onion structure: a middle block contains the “core” of the loop to be parallelized, and is surrounded by other blocks that are the needed for the start and the end of the surrounding loops, plus one more layer that will be used for the split and rebuild function calls. For example for the following code:

```
for (x=0; x<10; x++)
 for (y=0; y<10; y++)
   for (z=0; z<10; z++)
     f(x,y,z);
```

a tree is created with the following structure:

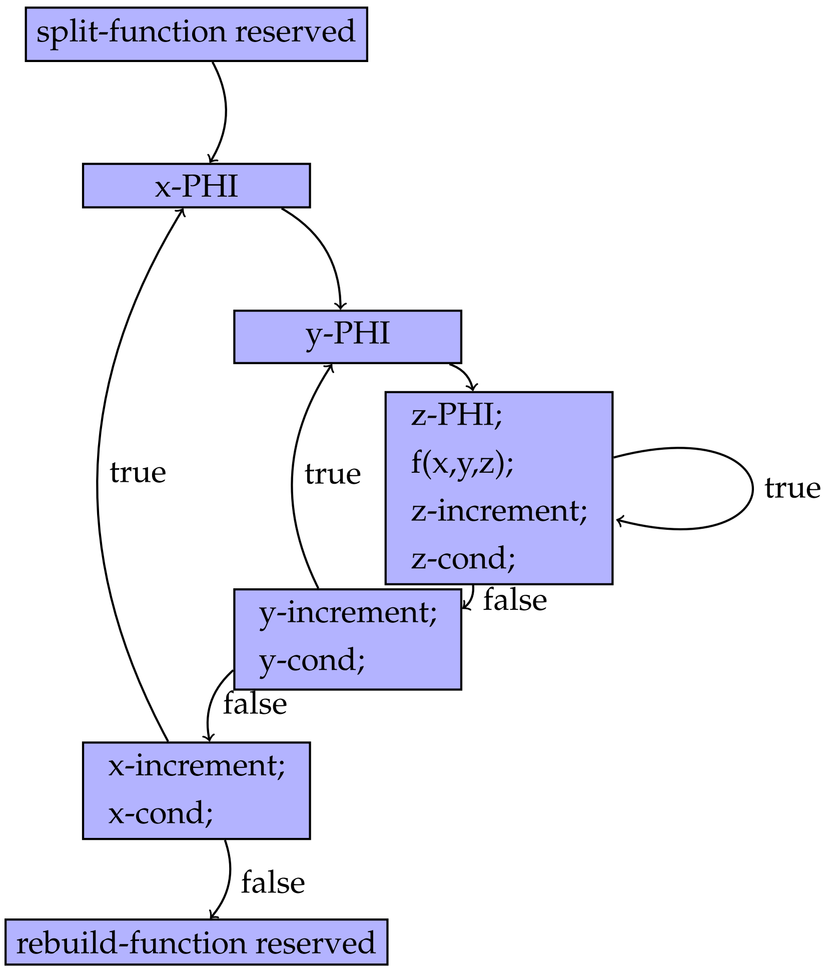


Figure 4.1: Original flow graph for one statement

The structure in Figure 4.1 is created for the first statement, then it can be modified if there are more statements since it is possible that two statements share some variable. When is created during the first iteration a structure that keeps trace of every iteration variable and its loop entry is built. If there are more statements a check is done to find which variables are shared and which ones are different, then for the different ones others basic blocks are created, and then inserted after the end of the first not-shared variable.
The list of basic blocks that have to be modified is one of the information retrieved with the xml, and it has to be crossed with the data from the SCoP:

every statement in the SCoP corresponds to a basic block in the list of the xml. In this way the instructions that have to be copied in the middle block of the newly created structure are retrieved. An example of a modified structure is provided with the following code, that has two polyhedral statements:

```
for (x=0; x<10; x++)
 {
 for (y=0; y<10; y++)
   for (z=0; z<10; z++)
     f(x,y,z)
 for (y=0; y>10; y++)
   for (z=0; z>10;z++)
     f(x,y,z);
  }
```

the flow graph created at the first iteration is modified as in Figure4.2

Another operation that has to be done is the identification and elimination of the old basic blocks that compose the polyhedral code. Since the polyhedral code is a single entry-single exit region, and the entry-exit basic blocks are an information saved in the xml, this can be done just by finding all the basic block reached by the function before the exit. They are identified in the first iteration of the loop, and are eliminated after the loop itself: some of them are needed as source if there are more statements.

The original instructions are then modified and this is the first step of the modification: the iteration variables are substituted, some dead code elimination is performed such as the elimination of instructions related to the old iteration variables. The new arrays are created, and a map to keep the reference between the original array and the new arrays is created.
Then the pattern analysis (described in section 3.3.2) is executed, using as source the data of the statement. With the result of the pattern analysis the create_parallel_stmts function is called. This function is driven by the pattern identified, and with a switch selects the transformations that have to be performed. Those transformations are the one described in the section 3.3.4 . This is the second part of the modification: the parallelization is performed now.
As last step the split and rebuild functions have to be instantiated in the first and last basic block created, because the loop contained in the code is working with the new arrays and the structure described above.

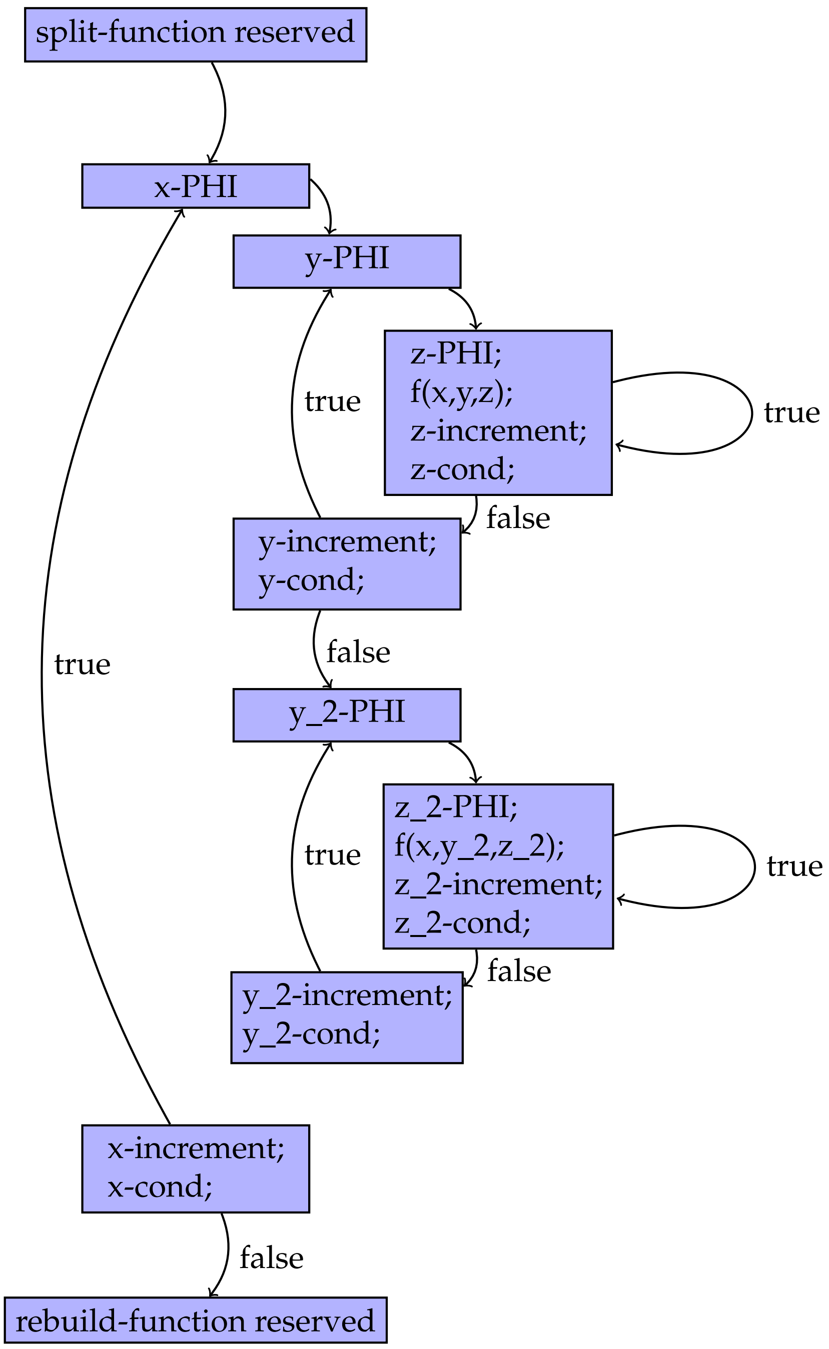


Figure 4.2: Modified flow graph for more than one statement

4.4 Conclusion

Bambu is now able to create an output file where the original loop is now parallel, since every one of the new arrays is performing an operation in the same loop. This file is still C code, and has to be re-analyzed with *Bambu* to perform HLS and obtain the target code to be run on the FPGA.

Chapter 5

Experimental Evaluation

In this chapter the benchmarks chosen will be listed. The modifications required from the original benchmark will be highlighted and the result obtained from the execution of the benchmark will be explained.

5.1 Experimental Setup

5.1.1 Experimental Tools

As it was described in section 4.2, two different runs of *Bambu* are needed to obtain the Verilog code. The first run is a C to C run where the proposed solution has been applied. For the first run *Bambu* v 0.9.3 has been configured to call GCC 4.9 with -O2. This choiche has been made to mantain the structure of the code, easing the analysis of the code and the transformations performed. The output of this is the C code where the parallel instructions have been created and that could be used as source code by any HLS tool.

For the second run *Bambu*, version 0.9.3, has been used as HLS tool. It calls GCC 4.9 with -O3. The HLS has been configured to disconnect primary ports from the IOB and use BRAM for all the data. The output of this run is the Verilog code. The Synthesis is performed by Vivado v2015.1 (64-bit) [viv15]. The target platform is the Virtex-7 690T. This platform has been chosen for the number of available BRAMs, to memorize the splitted arrays. Target applications are data intensive, and the objective is to improve the performance in terms of execution time, so area utilization is not an issue and a large FPGA has been chosen as test device. The results have been collected after the place and route step.

5.1.2 Benchmarks

All the chosen benchmarks are taken from the Polybench test suite ([Pou]), and have been adapted to create auto-regressive test. Some modifications have also been done to work with the implemented program (and those will be explained case per case). The structure for all the tests is the following:

- Function with the kernel of the test, taken from Polybench, to be optimized.
- Functions for splitting and rebuilding the arrays.
- Main, which contains:
  - Initialization of all the arrays.
  - Another kernel of the test, that works with different array that have been initialized with the same values.
  - Call to the function to be optimized.
  - Final check where the two results arrays are compared.

A set of values has been chosen to test the impact of the array dimension on the proposed solution. Three different sizes have been tested for every non-stencil test: small, default and large. For stencils, only default and large test have been considered. Some tests where the kernel of the stencil is performed more times have been introduced. This has been done to observe the impact of the split-rebuild functions. All the size that have been modified were passed to the benchmark with defines, and the values are reported in every benchmark subsection. If no defines are passed, the default test is run. Another variation that has been done was the level of parallelism. Eight different levels of parallelism have been considered. Those levels are: 2,3,4,5,6,7,8,16.

5.1.2.1 Gemm

This test case perform the following matrix multiplication: C=alpha*A*B+beta*C. The multiplication C*beta has been moved to the initialization phase, so the kernel function is only performing C+=alpha*A*B. This modification has been made because the use of the schedule function is not yet implemented, and if a SCoP has more than one statement the extern loop only may be shared (and in this benchmark it is necessary to share more than one extern loop). It is only an implementation issue, and the heavy operation in this loop is the matrix multiplication, that is the one that is being made parallel. The test has been made with three different set of ranges:

- small: _PB_NI= 20, _PB_NJ= 30, _PB_NK=40
- default: _PB_NI= 100, _PB_NJ= 200, _PB_NK= 300
- large: _PB_NI= 500, _PB_NJ= 550, _PB_NK= 600

The kernel of this test is the following.

```
for (i = 0; i < _PB_NI; i++)
  for (j = 0; j < _PB_NJ; j++)
    for (k = 0; k < _PB_NK; ++k)
C[i][j] += alpha * A[i][k] * B[k][j];
```

5.1.2.2 Gemver

This test case perform vector multiplication and matrix addition. The kernel is the same of Polybench. The sizes of the arrays for the different cases are:

- small: _PB_N= 20
- default: _PB_N= 100
- large: _PB_N= 400

The kernel of this test is the following:

```

for (i = 0; i < _PB_N; i++)
  for (j = 0; j < _PB_N; j++)
    A[i][j] = A[i][j] + u1_1_[i] * v1_1_[j] + u2_1_[i] * v2_1_[j];

for (i = 0; i < _PB_N; i++)
  for (j = 0; j < _PB_N; j++)
    x_1_[i] = x_1_[i] + beta * A[j][i] * y_1_[j];

for (i = 0; i < _PB_N; i++)
  x_1_[i] = x_1_[i] + z_1_[i];

for (i = 0; i < _PB_N; i++)
  for (j = 0; j < _PB_N; j++)
    w_1_[i] = w_1_[i] +  alpha * A[i][j] * x_1_[j];
```

5.1.2.3 Jacobi 1D

This test case is the single dimension version of the Jacobi pattern. There are two versions of this test case, the first has the same kernel of the Polybench, with the division at the end of the sum. In HLS the division is a costly operation, and *Bambu* is not able to instantiate more than one division. This generates a bottleneck and the created parallelism is wasted. In some tests the division has been substituted with a multiplication for the inverse that is calculated previously. Those tests have the string "_mul" in their name. Six test groups have been created for this benchmark, two with the original version and four with the multiplication version. The values for the parameters are the following:

- default: _PB_N= 100, _PB_TSTEPS= 10. This test is done for both the original kernel and the multiplication kernel.
- _more_steps: _PB_N= 100, _PB_TSTEPS= 25. It is performed only for the modified kernel.
- large: _PB_N= 400, _PB_TSTEPS= 10. This test is done for both the original kernel and the multiplication kernel.
- large_more_steps: _PB_N= 400, _PB_TSTEPS= 25. It is performed only for the modified kernel.

The kernel of the test is the following:

```
for (t = 0; t < _PB_TSTEPS; t++)
  {
    for (i = 1; i < _PB_N - 1; i++)
      B[i] =  (A[i-1] + A[i] + A[i + 1])/3;
    for (j = 1; j < _PB_N - 1; j++)
      A[j] = B[j];
  }
```

5.1.2.4 Jacobi 2D

This test case is the two dimensional version of the Jacobi pattern. Since the Polybench version is already implemented with a multiplication, there is no need of multiple versions. Three test groups have been created for this benchmark, with the following values for the parameters:

- default: _PB_N= 100, _PB_TSTEPS= 10.
- _more_steps: _PB_N= 100, _PB_TSTEPS= 25.
- large: _PB_N= 500, _PB_TSTEPS= 10.

The kernel of the Jacobi 2d benchmark is the following:

```
for (t = 0; t < _PB_TSTEPS; t++)
  {
    for (i = 1; i < _PB_NI -1; i++)
      for (j = 1; j <  _PB_NI -1; j++)
        B[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][1+j] + A[1+i][j] + A[i-1][j]);
    for (i = 1; i <  _PB_NI -1; i++)
      for (j = 1; j <  _PB_NI -1; j++)
       A[i][j] = B[i][j];
  }
```

5.1.2.5 2MM

This test case performs two matrix multiplications: D = alpha*A*B*C + beta*D. It is similar to the Gemm test case, it just perform one more multiplication. The beta*D has been moved out from the kernel, for the same reason as above, and has been placed in the initialization. The value of the parameters for this test case are:

- default: _PB_NI= 100, _PB_NJ= 200, _PB_NK= 100, _PB_NL= 100
- small: _PB_NI= 20, _PB_NJ= 30, _PB_NK= 40, _PB_NL= 100
- large: _PB_NI= 500, _PB_NJ= 550, _PB_NK= 600, _PB_NL= 100

The kernel of the 2mm benchmark is the following:

```
for (i = 0; i < _PB_NI; i++)
 for (j = 0; j < _PB_NJ; j++)
   for (k = 0; k < _PB_NK; ++k)
     tmp[i][j] += alpha * A[i][k] * B[k][j];
for (i = 0; i < _PB_NI; i++)
 for (j = 0; j < _PB_NL; j++)
   for (k = 0; k < _PB_NJ; ++k)
     D[i][j] += tmp[i][k] * C[k][j];
```

5.1.2.6 3MM

This test case performs the follow matrix multiplications: E=A*B, F=C*D and G=E*F. The kernel is the same of the Polybench, the only difference is that the initialization of the result matrices is done outside from the function. In the Polybench kernel every loop is done as follows:

```
for (i = 0; i < _PB_NI; i++)
 for (j = 0; j < _PB_NJ; j++)
   E[i][j]=0;
   for (k = 0; k < _PB_NK; k++)
     E[i][j] += A[i][k] * B[k][j];
```

with the initialization done inside the loop. For the same reason as above that double statement loop is not supported (it shares more than one iteration variable). The kernel of the 3mm benchmark is the following:

```
for (i = 0; i < _PB_NI; i++)
 for (j = 0; j < _PB_NJ; j++)
  for (k = 0; k < _PB_NK; k++)
   E[i][j] += A[i][k] * B[k][j];

for (i = 0; i < _PB_NJ; i++)
 for (j = 0; j < _PB_NL; j++)
  for (k = 0; k < _PB_NM; k++)
   F[i][j] += C[i][k] * D[k][j];

for (i = 0; i < _PB_NI; i++)
```

```
    for (j = 0; j < _PB_NL; j++)
     for (k = 0; k < _PB_NJ; k++)
      G[i][j] += E[i][k] * F[k][j];
```

With the following values for the kernel parameters:

- default: _PB_NI= 100, _PB_NJ= 300, _PB_NK= 200, _PB_NL= 100, _PB_NM= 200
- small: _PB_NI= 20, _PB_NJ= 30, _PB_NK= 40, _PB_NL= 100, _PB_NM= 200
- large: _PB_NI= 500, _PB_NJ= 550, _PB_NK= 600, _PB_NL= 100, _PB_NM= 200

5.1.2.7 MVT

This benchmark performs matrix vector product and transpose. The kernel is the same of the Polybench, and it is reported below:

```
  for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_N; j++)
      x[i] = x[i] + A[i][j] * y[j];
  for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_N; j++)
      w[i] = w[i] + A[j][i] * z[j];
```

With the following values for the parameters:

- default: _PB_N= 100
- small: _PB_N= 20
- large: _PB_N= 400

5.1.2.8 Seidel 2D

This benchmark performs the Seidel 2D algorithm. There are two versions of this test case, with the division and with the multiplication, as for the Jacobi 1D benchmark. The first has the same kernel of the Polybench, with the division at the end of the sum. In the second, the division has been substituted with a multiplication for the inverse (i.e the division for the factor 9 has been substituted with a multiplication for 1/9, that is calculated previously only once). Six test groups have been created for this benchmark, two with the original version and four with the multiplication version. The original kernel of this benchmark is:

```
for (t = 0; t <= _PB_TSTEPS - 1; t++)
 for (i = 1; i<= _PB_N - 2; i++)
  for (j = 1; j <= _PB_N - 2; j++)
   A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]+ A[i][j-1] + A[i][j]
              + A[i][j+1] + A[i+1][j-1] + A[i+1][j] + A[i+1][j+1])/9.0;
```

and it has been tested with the following parameters:

- default: _PB_N= 100, _PB_TSTEPS= 10. This test too is done for both the versions.
- _more_steps: _PB_N= 100, _PB_TSTEPS= 25. It is performed only for the modified kernel.
- large: _PB_N= 400, _PB_TSTEPS= 10. This test too is done for both the versions.
- large_more_steps: _PB_N= 400, _PB_TSTEPS= 25. It is performed only for the modified kernel.

5.1.2.9 Syrk

This benchmark performs symmetric rank-k operations. The kernel is equal to the Polybench case, which is:

```
for (i = 0; i < _PB_NI; i++)
 for (j = 0; j < _PB_NI; j++)
  C[i][j] *= beta;

for (i = 0; i < _PB_NI; i++)
 for (j = 0; j < _PB_NI; j++)
  for (k = 0; k < _PB_NJ; k++)
   C[i][j] += alpha * A[i][k] * A[j][k];
```

and it has been tested with the following parameters:

- small: _PB_NI= 20, _PB_NJ= 30

- default: _PB_NI= 100, _PB_NJ= 200
- large: _PB_NI= 500, _PB_NJ= 550

5.1.2.10 Syr2k

This benchmark performs symmetric rank-2k operations.

```
for (i = 0; i < _PB_NI; i++)
  for (j = 0; j < _PB_NI; j++)
    C[i][j] *= beta;

for (i = 0; i < _PB_NI; i++)
  for (j = 0; j < _PB_NI; j++)
    for (k = 0; k < _PB_NJ; k++)
      {
        C[i][j] += alpha * A[i][k] * B[j][k];
        C[i][j] += alpha * B[i][k] * A[j][k];
      }
```

and it has been tested with the following parameters:

- small: _PB_NI= 20, _PB_NJ= 30
- default: _PB_NI= 100, _PB_NJ= 200
- large: _PB_NI= 500, _PB_NJ= 550

5.2 Experimental Results

The results of the runs of the test will be now reported and discussed.

For every benchmark the results are included in two tables, performance and area, the first with the information on the wall time in terms of clock cycles and clock period and the second with information on the area used. More in detail, the first table (performance) has six columns:

- The leftmost column contains the name of the test (small, default, extended,...) that recalls the dimension of the parameter.
- The *parallel degree* column contains the number of kernel created.

- The *number of cycles split included* reports the wall time elapsed by the kernel when the external split and rebuild functions are considered part of the computation.
- The *number of cycles split excluded* reports the wall time elapsed by the kernel minus the external split and rebuild functions.
- The *clock frequency* reports the obtained frequency from the synthesis (MHz).
- The *clock slack* is the difference respect to the target clock period that is 10ns.
- The *Speedup split included* reports the speedup obtained compared to the sequential test case, when for the parallel test the time elapsed in the external split and rebuild is considered.
- The *Speedup split excluded* reports the speedup obtained compared to the sequential test case, when for the parallel test the time elapsed in the external split and rebuild is not considered.

If intermediate split-rebuild are issued between two SCoPs, they are considered as part of the calculation. The external split-rebuild instead can be avoided if the data are already divided while the copy from the external memory to the BRAMs is being executed. If the divison is done while copying the data, the split time can be neglected since the calculations that have to be made for the address of the data are masked by the cost of the copy operation.

The second table (area) has eight columns:

- The first column is the same of the previous table, with the name of the test.
- The *parallel degree* column contains the number of kernel created.
- The *LUT-FF pairs* column contains the number of LUT-FF pairs used in the synthesis. A LUT-FF pair is Look-Up Table with a dedicated Flip Flop. Of this couple both can be used, but it is not mandatory. It can happen that

only one of them is used. This column reports the total number of pairs where at least one of the two components has been used. For the target device the number of available LUT-FF pairs is 433200.

- The *LUT* column contains the number of Look-Up Table used in the synthesis. LUTs are the basic building block of an FPGA and are used for the building of the logic of the circuit. For the target device the number of available LUT as logic is 433200.

- The *Slices* columns reports the number of slices used in the synthesis. A slice is a group of LUTs and Flip Flops, as said in [xil15] there are four LUTs and 8 FFs for each series 7 Slice (the target device is one of them). This column reports the number of slices that have at least one of the internal LUT-FF pair used by the synthesis. For the target device the number of available slices is 108,300.

- The *register* contains the number of temporary storages for data. It is the sum of the FFs used as data storage. For the target device the number of available FF is 866400.

- The *DSPs* reports the number of Digital Signal Processors slices that were allocated. DSPs are used for complex math operations. Every series 7 DSP slice contains a pre-adder, a 25 x 18 multiplier, an adder, and an accumulator ([xil15]). As above the number reported here means that at least one of the elements contained in the slice has been used. For the targeted device the nuber of DSP slices is 3,600.

- The *BRAMs* column reports the number of memories used in the synthesis. For the targeted device the number of BRAMs available is 2,940.

For every column is reported between parentheses the ratio between the number of components used by the parallel test and the number of components needed in the synthesis of the sequential case.

Whenever a simulation or synthesis did not finish, the result is reported as n.a. The cause of this are two:

- The simulation takes too many cycles and is not able to end in 200,000,000 (which has been taken as upper limit) cycles.
- The synthesis is not finished in 150 minutes (timeout threshold).

Which one of the two is the reason will be explained every time.

The measurements are done after the place and route phase.

All the benchmark listed above have been clustered in five groups. This has been done because some test cases presented a pattern that was similar to other and consequently the results are similar. Those patterns are:

- Matrix Multiplication
- Matrix and Vector Multiplication
- 1D Stencil
- 2D Stencil on different arrays
- 2D Stencil on the same array

5.2.1 Benchmark Pattern: Matrix Multiplication

This pattern has been found in the following tests: Gemm, 2MM, 3MM, Syrk and Syr2k. For every one of these benchmarks the results obtained are reported in the following tables:

- Gemm: performances results are reported in Table 5.1, area results are reported in Table 5.2
- 2MM: performances results are reported in Table 5.3, area results are reported in Table 5.4
- 3MM: performances results are reported in Table 5.5, area results are reported in Table 5.6
- Syrk: performances results are reported in Table 5.7, area results are reported in Table 5.8

- Syr2k: performances results are reported in Table 5.9, area results are reported in Table 5.10

Gemm	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
default	1	*n.a.*	30,060,103	101.79	0.18	*n.a.*	*n.a.*
	2	15,135,568	15,030,053	96.72	−0.34	1.98	1.99
	3	10,325,956	10,220,437	100.82	0.08	2.91	2.94
	4	7,620,568	7,515,028	98.82	−0.12	3.94	3.99
	5	6,117,578	6,012,023	96.48	−0.37	4.91	4.99
	6	5,215,775	5,110,220	101.03	0.10	5.76	5.88
	7	4,614,583	4,509,018	94.34	−0.60	6.51	6.66
	8	4,013,396	3,907,816	100.33	0.03	7.48	7.69
	16	2,209,882	2,104,210	97.43	−0.26	13.60	14.28
small	1	*n.a.*	121,823	103.85	0.37	*n.a.*	*n.a.*
	2	64,028	60,913	103.76	0.36	1.90	1.99
	3	45,759	42,640	104.49	0.43	2.66	2.85
	4	33,598	30,458	105.75	0.54	3.62	3.99
	5	27,517	24,367	106.92	0.65	4.42	4.99
	6	27,522	24,367	104.74	0.45	4.42	4.99
	7	21,419	18,276	105.79	0.55	5.68	6.66
	8	21,480	18,276	106.18	0.58	5.67	6.66
	16	15,613	12,185	105.71	0.54	7.80	9.99
large	1	n.a.					
	2	n.a.					
	3	n.a.					
	4	n.a.					
	5	n.a.					
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					

Table 5.1: Gemm performance results (continued)

Gemm	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
	1	5,020	4,530	1,876	1,752	6	512
	2	8,456(1.68)	7,511(1.66)	3,117(1.66)	3,129(1.79)	12(2.00)	646(1.26)
	3	10,713(2.13)	9,421(2.08)	3,874(2.07)	4,269(2.44)	15(2.50)	662(1.29)
	4	10,171(2.03)	9,203(2.03)	3,711(1.98)	3,347(1.91)	18(3.00)	646(1.26)
default	5	11,183(2.23)	10,191(2.25)	4,107(2.19)	3,489(1.99)	21(3.50)	678(1.32)
	6	12,880(2.57)	11,522(2.54)	4,514(2.41)	4,628(2.64)	27(4.50)	662(1.29)
	7	13,893(2.77)	12,544(2.77)	4,979(2.65)	4,717(2.69)	30(5.00)	686(1.34)
	8	14,808(2.95)	13,447(2.97)	5,177(2.76)	4,777(2.73)	33(5.50)	710(1.39)
	16	22,061(4.39)	20,546(4.54)	7,158(3.82)	5,805(3.31)	60(10.00)	710(1.39)
	1	4,445	4,052	1,334	1,587	6	24
	2	7,577(1.70)	6,840(1.69)	2,284(1.71)	2,745(1.73)	12(2.00)	46(1.92)
	3	9,362(2.11)	8,428(2.08)	2,876(2.16)	3,693(2.33)	15(2.50)	54(2.25)
	4	8,956(2.01)	8,236(2.03)	2,670(2.00)	2,932(1.85)	18(3.00)	62(2.58)
small	5	9,718(2.19)	9,056(2.23)	2,911(2.18)	3,017(1.90)	21(3.50)	70(2.92)
	6	11,425(2.57)	10,382(2.56)	3,410(2.56)	4,090(2.58)	27(4.50)	78(3.25)
	7	14,531(3.27)	12,888(3.18)	4,411(3.31)	5,871(3.70)	27(4.50)	86(3.58)
	8	12,448(2.80)	11,406(2.81)	3,748(2.81)	4,051(2.55)	33(5.50)	94(3.92)
	16	16,957(3.81)	16,150(3.99)	5,106(3.83)	4,072(2.57)	60(10.00)	158(6.58)
	1	n.a.					
	2	n.a.					
	3	n.a.					
	4	n.a.					
large	5	n.a.					
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					

Table 5.2: Gemm area results (continued)

2MM	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
default	1	*n.a.*	18,090,203	100.30	0.03	*n.a.*	*n.a.*
	2	9,181,347	9,120,725	102.10	0.21	1.97	1.98
	3	6,286,926	6,226,299	97.49	−0.26	2.87	2.90
	4	4,658,849	4,598,201	100.83	0.08	3.88	3.93
	5	3,754,363	3,693,703	101.32	0.13	4.81	4.89
	6	3,211,677	3,151,007	95.34	−0.49	5.63	5.74
	7	2,849,897	2,789,215	100.38	0.04	6.34	6.48
	8	2,488,121	2,427,425	102.09	0.21	7.27	7.45
	16	1,402,901	1,342,109	101.80	0.18	12.89	13.47
small	1	*n.a.*	367,843	104.82	0.46	*n.a.*	*n.a.*
	2	194,107	188,865	104.67	0.45	1.89	1.94
	3	138,942	133,695	103.76	0.36	2.64	2.75
	4	102,199	96,931	104.03	0.39	3.59	3.79
	5	83,831	78,551	104.09	0.39	4.38	4.68
	6	83,851	78,561	105.78	0.55	4.38	4.68
	7	65,510	60,195	106.32	0.59	5.61	6.11
	8	65,583	60,231	105.36	0.51	5.60	6.10
	16	47,311	41,899	104.82	0.46	7.77	8.77
large	1	n.a.					
	2	n.a.					
	3	n.a.					
	4	n.a.					
	5	n.a.					
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					

Table 5.3: 2MM performance results (continued)

2MM	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
default	1	8,324	7,290	2,944	3,046	13	512
	2	13,562(1.63)	12,045(1.65)	4,856(1.65)	4,870(1.60)	22(1.69)	710(1.39)
	3	17,317(2.08)	15,238(2.09)	5,906(2.01)	6,717(2.21)	25(1.92)	662(1.29)
	4	17,739(2.13)	16,198(2.22)	6,154(2.09)	5,371(1.76)	28(2.15)	710(1.39)
	5	19,518(2.34)	17,885(2.45)	6,757(2.30)	5,637(1.85)	46(3.54)	758(1.48)
	6	22,505(2.70)	20,366(2.79)	7,324(2.49)	7,335(2.41)	58(4.46)	662(1.29)
	7	24,353(2.93)	22,212(3.05)	7,960(2.70)	7,710(2.53)	67(5.15)	686(1.34)
	8	25,993(3.12)	23,823(3.27)	8,426(2.86)	7,729(2.54)	73(5.62)	710(1.39)
	16	38,993(4.68)	36,561(5.02)	11,868(4.03)	10,041(3.30)	73(5.62)	774(1.51)
small	1	7,669	6,816	2,222	2,855	13	48
	2	12,221(1.59)	11,019(1.62)	3,679(1.66)	4,431(1.55)	22(1.69)	86(1.79)
	3	15,884(2.07)	14,168(2.08)	4,848(2.18)	6,109(2.14)	25(1.92)	102(2.13)
	4	15,900(2.07)	14,620(2.14)	4,730(2.13)	5,038(1.76)	28(2.15)	118(2.46)
	5	17,548(2.29)	16,232(2.38)	5,212(2.35)	5,192(1.82)	46(3.54)	134(2.79)
	6	21,193(2.76)	19,379(2.84)	6,236(2.81)	6,875(2.41)	58(4.46)	150(3.13)
	7	21,621(2.82)	19,886(2.92)	6,512(2.93)	6,779(2.37)	61(4.69)	166(3.46)
	8	23,446(3.06)	21,685(3.18)	7,076(3.18)	6,854(2.40)	73(5.62)	182(3.79)
	16	31,574(4.12)	29,933(4.39)	9,501(4.28)	7,797(2.73)	73(5.62)	310(6.46)
large	1	n.a.					
	2	n.a.					
	3	n.a.					
	4	n.a.					
	5	n.a.					
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					

Table 5.4: 2MM area results (continued)

3MM	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
default	1	*n.a.*	60,210,502	93.25	−0.72	*n.a.*	*n.a.*
	2	30,483,401	30,392,650	95.07	−0.52	1.97	1.98
	3	20,689,340	20,598,518	95.14	−0.51	2.91	2.92
	4	15,431,242	15,340,335	95.88	−0.43	3.90	3.92
	5	12,420,951	12,329,967	95.19	−0.51	4.84	4.88
	6	10,534,561	10,443,502	90.01	−1.11	5.71	5.76
	7	9,250,267	9,159,131	95.06	−0.52	6.50	6.57
	8	8,126,597	8,035,382	98.18	−0.19	7.40	7.49
	16	4,435,510	4,343,679	n.a.	n.a.	13.57	13.86
small	1	*n.a.*	2,752,872	105.30	0.50	*n.a.*	*n.a.*
	2	1,405,176	1,399,805	103.47	0.34	1.95	1.96
	3	952,316	946,874	103.01	0.29	2.89	2.90
	4	757,567	752,040	102.62	0.26	3.63	3.66
	5	580,010	574,406	101.99	0.20	4.74	4.79
	6	499,934	494,255	101.46	0.14	5.50	5.56
	7	482,995	477,225	103.83	0.37	5.69	5.76
	8	402,985	397,114	104.77	0.46	6.83	6.93
	16	227,123	220,672	104.98	0.47	12.12	12.47
large	1	n.a.					
	2	n.a.					
	3	n.a.					
	4	n.a.					
	5	n.a.					
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					

Table 5.5: 3MM performance results (continued)

3MM	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
default	1	10,772	9,398	4,056	3,658	10	1088
	2	18,951(1.76)	16,660(1.77)	7,127(1.76)	6,621(1.81)	28(2.80)	1512(1.39)
	3	24,263(2.25)	21,140(2.25)	8,865(2.19)	9,108(2.49)	34(3.40)	1648(1.51)
	4	24,877(2.31)	22,484(2.39)	9,032(2.23)	7,532(2.06)	52(5.20)	1512(1.39)
	5	27,466(2.55)	24,880(2.65)	9,856(2.43)	7,947(2.17)	61(6.10)	1616(1.49)
	6	32,242(2.99)	28,927(3.08)	11,360(2.80)	10,074(2.75)	76(7.60)	1648(1.51)
	7	36,462(3.38)	32,819(3.49)	12,750(3.14)	11,765(3.22)	85(8.50)	1740(1.60)
	8	39,135(3.63)	35,243(3.75)	13,241(3.26)	12,132(3.32)	49(4.90)	1640(1.51)
	16			n.a.			
small	1	9,488	8,318	3,026	3,412	10	208
	2	17,385(1.83)	15,354(1.85)	5,395(1.78)	6,243(1.83)	28(2.80)	272(1.31)
	3	22,308(2.35)	19,691(2.37)	6,856(2.27)	8,297(2.43)	34(3.40)	300(1.44)
	4	23,804(2.51)	21,388(2.57)	7,295(2.41)	8,153(2.39)	55(5.50)	312(1.50)
	5	25,523(2.69)	23,488(2.82)	7,613(2.52)	7,391(2.17)	61(6.10)	336(1.62)
	6	29,648(3.12)	27,090(3.26)	8,864(2.93)	9,446(2.77)	76(7.60)	360(1.73)
	7	32,720(3.45)	29,716(3.57)	9,968(3.29)	10,448(3.06)	82(8.20)	384(1.85)
	8	32,956(3.47)	30,271(3.64)	9,978(3.30)	9,983(2.93)	46(4.60)	408(1.96)
	16	44,906(4.73)	42,871(5.15)	14,028(4.64)	10,272(3.01)	67(6.70)	600(2.88)
large	1			n.a.			
	2			n.a.			
	3			n.a.			
	4			n.a.			
	5			n.a.			
	6			n.a.			
	7			n.a.			
	8			n.a.			
	16			n.a.			

Table 5.6: 3MM area results (continued)

Syrk	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
default	1	*n.a.*	10,070,205	103.83	0.37	*n.a.*	*n.a.*
	2	5,185,630	5,125,420	100.15	0.02	1.94	1.96
	3	3,574,401	3,514,190	101.08	0.11	2.81	2.86
	4	2,668,115	2,607,891	104.79	0.46	3.77	3.86
	5	2,164,620	2,104,390	105.80	0.55	4.65	4.78
	6	1,862,517	1,802,286	105.26	0.50	5.40	5.58
	7	1,661,128	1,600,891	104.60	0.44	6.06	6.29
	8	1,459,734	1,399,494	105.44	0.52	6.89	7.19
	16	855,650	795,358	104.25	0.41	11.76	12.66
small	1	*n.a.*	62,845	107.49	0.70	*n.a.*	*n.a.*
	2	36,043	34,196	105.15	0.49	1.74	1.83
	3	26,627	24,776	105.93	0.56	2.36	2.53
	4	20,353	18,498	106.09	0.57	3.08	3.39
	5	17,221	15,362	107.19	0.67	3.64	4.09
	6	17,231	15,368	109.05	0.83	3.64	4.08
	7	13,934	12,133	106.99	0.65	4.51	5.17
	8	14,121	12,246	107.20	0.67	4.45	5.13
	16	11,359	9,296	105.78	0.55	5.53	6.76
large	1	n.a.					
	2	n.a.					
	3	n.a.					
	4	n.a.					
	5	n.a.					
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					

Table 5.7: Syrk performance results (continued)

Syrk	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
default	1	4,110	3,703	1,433	1,425	6	256
	2	7,815(1.90)	7,046(1.90)	2,855(1.99)	2,352(1.65)	9(1.50)	454(1.77)
	3	9,845(2.40)	8,743(2.36)	3,463(2.42)	3,019(2.12)	12(2.00)	406(1.59)
	4	10,300(2.51)	9,402(2.54)	3,574(2.49)	2,689(1.89)	39(6.50)	454(1.77)
	5	11,592(2.82)	10,672(2.88)	4,068(2.84)	2,831(1.99)	48(8.00)	502(1.96)
	6	13,398(3.26)	12,308(3.32)	4,473(3.12)	3,511(2.46)	57(9.50)	406(1.59)
	7	14,753(3.59)	13,557(3.66)	4,922(3.43)	3,691(2.59)	66(11.00)	430(1.68)
	8	16,444(4.00)	15,428(4.17)	5,414(3.78)	4,000(2.81)	51(8.50)	454(1.77)
	16	25,575(6.22)	24,394(6.59)	7,854(5.48)	5,417(3.80)	99(16.50)	454(1.77)
small	1	3,823	3,503	1,106	1,339	6	16
	2	6,981(1.83)	6,408(1.83)	2,089(1.89)	2,140(1.60)	9(1.50)	46(2.88)
	3	8,846(2.31)	8,025(2.29)	2,599(2.35)	2,755(2.06)	12(2.00)	58(3.63)
	4	9,153(2.39)	8,499(2.43)	2,690(2.43)	2,464(1.84)	39(6.50)	70(4.38)
	5	10,312(2.70)	9,638(2.75)	3,076(2.78)	2,631(1.96)	48(8.00)	82(5.13)
	6	12,153(3.18)	11,205(3.20)	3,668(3.32)	3,331(2.49)	57(9.50)	94(5.88)
	7	18,382(4.81)	15,427(4.40)	5,818(5.26)	8,037(6.00)	66(11.00)	106(6.63)
	8	14,034(3.67)	13,156(3.76)	4,169(3.77)	3,608(2.69)	51(8.50)	118(7.38)
	16	22,018(5.76)	21,338(6.09)	6,481(5.86)	4,547(3.40)	99(16.50)	214(13.38)
large	1	n.a.					
	2	n.a.					
	3	n.a.					
	4	n.a.					
	5	n.a.					
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					

Table 5.8: Syrk area results (continued)

Syr2k	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
default	1	*n.a.*	12,070,205	100.08	0.01	*n.a.*	*n.a.*
	2	6,215,736	6,155,526	100.42	0.04	1.94	1.96
	3	4,284,507	4,224,296	103.51	0.34	2.81	2.85
	4	3,198,227	3,138,003	104.40	0.42	3.77	3.84
	5	2,594,735	2,534,505	104.46	0.43	4.65	4.76
	6	2,232,633	2,172,402	100.47	0.05	5.40	5.55
	7	1,991,247	1,931,010	103.18	0.31	6.06	6.25
	8	1,749,858	1,689,618	101.18	0.12	6.89	7.14
	16	1,025,798	965,506	103.08	0.30	11.76	12.50
small	1	*n.a.*	74,845	104.10	0.39	*n.a.*	*n.a.*
	2	42,967	41,120	106.10	0.57	1.74	1.82
	3	31,753	29,902	107.26	0.68	2.35	2.50
	4	24,281	22,426	106.07	0.57	3.08	3.33
	5	20,551	18,692	106.34	0.60	3.64	4.00
	6	20,563	18,700	108.12	0.75	3.63	4.00
	7	16,635	14,834	105.35	0.51	4.49	5.04
	8	16,861	14,986	110.00	0.91	4.43	4.99
	16	13,503	11,440	106.58	0.62	5.54	6.54
large	1	n.a.					
	2	n.a.					
	3	n.a.					
	4	n.a.					
	5	n.a.					
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					

Table 5.9: Syr2k performance results (continued)

Syr2k	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
	1	5,326	4,906	1,970	1,454	12	384
	2	9,778(1.84)	9,029(1.84)	3,728(1.89)	2,430(1.67)	18(1.50)	646(1.68)
	3	11,994(2.25)	10,928(2.23)	4,318(2.19)	3,150(2.17)	42(3.50)	582(1.52)
	4	12,870(2.42)	12,053(2.46)	4,709(2.39)	2,838(1.95)	54(4.50)	646(1.68)
default	5	14,683(2.76)	13,812(2.82)	5,307(2.69)	3,081(2.12)	66(5.50)	710(1.85)
	6	17,132(3.22)	16,112(3.28)	5,923(3.01)	3,961(2.72)	60(5.00)	582(1.52)
	7	19,065(3.58)	18,008(3.67)	6,442(3.27)	4,083(2.81)	69(5.75)	614(1.60)
	8	20,940(3.93)	19,954(4.07)	6,941(3.52)	4,204(2.89)	78(6.50)	646(1.68)
	16	32,535(6.11)	31,372(6.39)	10,164(5.16)	5,971(4.11)	150(12.50)	646(1.68)
	1	4,714	4,431	1,446	1,321	12	24
	2	8,697(1.84)	8,132(1.84)	2,599(1.80)	2,189(1.66)	18(1.50)	62(2.58)
	3	10,681(2.27)	9,803(2.21)	3,178(2.20)	2,928(2.22)	42(3.50)	78(3.25)
	4	11,529(2.45)	10,814(2.44)	3,426(2.37)	2,590(1.96)	54(4.50)	94(3.92)
small	5	12,974(2.75)	12,337(2.78)	3,871(2.68)	2,823(2.14)	66(5.50)	110(4.58)
	6	15,835(3.36)	14,886(3.36)	4,790(3.31)	3,697(2.80)	60(5.00)	126(5.25)
	7	21,440(4.55)	18,775(4.24)	6,913(4.78)	8,481(6.42)	69(5.75)	142(5.92)
	8	17,104(3.63)	16,268(3.67)	5,099(3.53)	3,800(2.88)	78(6.50)	158(6.58)
	16	29,263(6.21)	28,662(6.47)	8,581(5.93)	5,058(3.83)	150(12.50)	286(11.92)
	1			n.a.			
	2			n.a.			
	3			n.a.			
	4			n.a.			
large	5			n.a.			
	6			n.a.			
	7			n.a.			
	8			n.a.			
	16			n.a.			

Table 5.10: Syr2k area results (continued)

For all these benchmarks the large test fails because the simulation is too long. However analyzing the small and the default case it can be noticed that, if only one SCoP is recognized in the code, the speedup is almost equal to the number of cores, when considering the measurement without the overhead of splitting and rebuilding the matrices. It is obviously smaller if the split and rebuild operation is considered, but it is still relevant. Among all these tests, only the Gemm has a single SCoP. All the others have two or more SCoPs and this is the cause of the reduction of the speedup. Moreover even with a single SCoP with a parallelism larger than six the speedup is less than linear. From the small cases it can be seen that the size of the arrays is one of the possible reason for the reduction of the speedup. For example the Gemm tests with 5 and 6 parallel cores have the same time, since the number of executed instructions is the same. 20/5 and 20/6 rounded up both gives 4, so the whole structure has the same number of iterations and is useless to instantiate one more kernel.

The theoretical speedup can be reached, on single SCoP programs, only when the index of the extern loop is a multiple of the parallelism degree. (see the default test with parallel degree 2,4,5). A relevant result can also be obtained if the ratio between the number of iteration and parallel degree is very large (more than 10). For example, the default test with parallel degree 3 has a speedup of 2.94 even if 100 (the number of iteration of the split loop) is not a multiple of 3 (the parallel degree).

The area growth is sub-linear, as expected since only part of the kernel is replicated. It happens for some cases that increasing the parallel degree the area consumed is lower (ex. Gemm default 3-4 for LUTs, 2MM default 3-4 and 3-5 for registers) and it is probably due to the size of the array. If the number of iterations to do is a multiple of the parallelization factor there is a waste of space lesser than the cases where the number of iterations is a multiple of the parallel degree .

5.2.2 Benchmark Pattern: Matrix and vector multiplication

This pattern can be found in the remaining two algebraic test cases, the Gemver (performance is in Table 5.11, area reported in Table 5.12) and the MVT(performances are reported in Table 5.13, while area results are in Table 5.14).

Gemver	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
default	1	*n.a.*	141,106	101.16	0.12	*n.a.*	*n.a.*
	2	151,858	134,827	102.17	0.21	0.92	1.04
	3	129,407	112,368	101.16	0.12	1.09	1.25
	4	117,487	100,424	106.33	0.60	1.20	1.40
	5	110,884	93,805	103.55	0.34	1.27	1.50
	6	85,725	68,630	103.23	0.31	1.64	2.05
	7	82,750	65,639	102.36	0.23	1.70	2.14
	8	80,779	63,652	104.11	0.39	1.74	2.21
	16	73,489	56,234	102.50	0.24	1.92	2.50
small	1	*n.a.*	5,826	106.11	0.58	*n.a.*	*n.a.*
	2	6,281	5,250	103.85	0.37	0.92	1.10
	3	5,439	4,400	102.86	0.28	1.07	1.32
	4	5,092	4,029	102.97	0.29	1.14	1.44
	5	4,719	3,640	105.40	0.51	1.23	1.60
	6	4,723	3,628	104.61	0.44	1.23	1.60
	7	4,581	3,463	104.54	0.43	1.27	1.68
	8	4,703	3,576	103.66	0.35	1.23	1.62
	16	4,898	3,739	100.76	0.07	1.18	1.55
large	1	*n.a.*	2,244,406	79.72	−2.54	*n.a.*	*n.a.*
	2	2,347,108	2,099,077	80.57	−2.41	0.95	1.06
	3	1,977,207	1,729,168	84.94	−1.77	1.13	1.29
	4	1,789,312	1,541,249	84.32	−1.86	1.25	1.45
	5	1,678,744	1,430,665	85.58	−1.69	1.33	1.56
	6	1,602,629	1,354,534	*n.a.*	*n.a.*	1.40	1.65
	7	1,557,379	1,309,276	*n.a.*	*n.a.*	1.44	1.71
	8	1,515,370	1,267,243	*n.a.*	*n.a.*	1.48	1.77
	16	1,388,311	1,140,056	*n.a.*	*n.a.*	1.61	1.96

Table 5.11: Gemver performance results (continued)

Gemver	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
default	1	8,816	8,553	2,557	1,747	12	128
	2	18,336(2.08)	17,267(2.02)	5,719(2.24)	4,640(2.66)	24(2.00)	282(2.20)
	3	23,541(2.67)	21,871(2.56)	7,082(2.77)	6,921(3.96)	30(2.50)	282(2.20)
	4	24,244(2.75)	23,186(2.71)	7,334(2.87)	5,412(3.10)	48(4.00)	330(2.58)
	5	28,124(3.19)	26,768(3.13)	8,655(3.38)	6,263(3.59)	57(4.75)	378(2.95)
	6	34,050(3.86)	31,720(3.71)	9,987(3.91)	9,294(5.32)	72(6.00)	354(2.77)
	7	37,745(4.28)	35,350(4.13)	11,154(4.36)	9,785(5.60)	81(6.75)	390(3.05)
	8	39,657(4.50)	37,305(4.36)	12,079(4.72)	9,516(5.45)	90(7.50)	426(3.33)
	16	64,640(7.33)	62,355(7.29)	19,044(7.45)	12,821(7.34)	162(13.50)	714(5.58)
small	1	8,107	7,967	2,302	1,565	12	72
	2	18,996(2.34)	17,547(2.20)	5,730(2.49)	5,480(3.50)	24(2.00)	154(2.14)
	3	23,565(2.91)	22,008(2.76)	7,017(3.05)	6,821(4.36)	30(2.50)	190(2.64)
	4	25,109(3.10)	24,126(3.03)	7,387(3.21)	5,803(3.71)	36(3.00)	226(3.14)
	5	27,106(3.34)	26,002(3.26)	7,980(3.47)	6,142(3.92)	57(4.75)	262(3.64)
	6	31,180(3.85)	29,677(3.72)	9,340(4.06)	7,653(4.89)	54(4.50)	298(4.14)
	7	34,303(4.23)	32,988(4.14)	10,021(4.35)	8,031(5.13)	75(6.25)	334(4.64)
	8	37,365(4.61)	35,922(4.51)	10,937(4.75)	8,179(5.23)	90(7.50)	370(5.14)
	16	62,161(7.67)	60,954(7.65)	18,173(7.89)	10,378(6.63)	114(9.50)	658(9.14)
large	1	10,492	10,073	3,925	2,012	12	1,088
	2	22,662(2.16)	21,158(2.10)	8,528(2.17)	5,144(2.56)	24(2.00)	2,682(2.47)
	3	28,029(2.67)	25,697(2.55)	9,829(2.50)	7,689(3.82)	30(2.50)	2,322(2.13)
	4	29,417(2.80)	27,889(2.77)	10,536(2.68)	5,991(2.98)	48(4.00)	2,730(2.51)
	5	32,795(3.13)	30,933(3.07)	11,490(2.93)	6,897(3.43)	57(4.75)	2,178(2.00)
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					

Table 5.12: Gemver area results (continued)

MVT	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
default	1	*n.a.*	180,602	108.26	0.76	*n.a.*	*n.a.*
	2	137,841	121,720	108.17	0.76	1.31	1.48
	3	109,055	92,928	105.97	0.56	1.65	1.94
	4	93,539	77,388	106.37	0.60	1.93	2.33
	5	84,931	68,767	101.76	0.17	2.12	2.62
	6	58,399	42,394	104.37	0.42	3.09	4.26
	7	54,602	38,583	105.46	0.52	3.30	4.68
	8	51,815	35,784	101.80	0.18	3.48	5.04
	16	41,907	25,788	104.92	0.47	4.30	7.00
large	1	*n.a.*	2,882,402	104.34	0.42	*n.a.*	*n.a.*
	2	2,171,241	1,926,820	103.89	0.37	1.32	1.49
	3	1,696,055	1,451,628	103.92	0.38	1.69	1.98
	4	1,453,889	1,209,438	103.52	0.34	1.98	2.38
	5	1,311,391	1,066,927	105.69	0.54	2.19	2.70
	6	1,214,507	970,046	103.78	0.36	2.37	2.97
	7	1,154,891	910,408	102.86	0.28	2.49	3.16
	8	1,100,077	855,574	103.75	0.36	2.62	3.36
	16	932,903	688,296	103.00	0.29	3.08	4.18
small	1	*n.a.*	7,322	109.54	0.87	*n.a.*	*n.a.*
	2	5,013	4,206	107.75	0.72	1.46	1.74
	3	3,918	3,107	106.12	0.58	1.86	2.35
	4	3,383	2,552	103.71	0.36	2.16	2.86
	5	2,900	2,056	105.01	0.48	2.52	3.56
	6	2,871	2,026	105.04	0.48	2.55	3.61
	7	2,608	1,747	104.43	0.42	2.80	4.19
	8	2,719	1,840	102.50	0.24	2.69	3.97
	16	2,797	1,858	102.75	0.27	2.61	3.94

Table 5.13: Mvt performance results (continued)

MVT	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
default	1	7,081	6,693	2,142	1,787	12	96
	2	12,616(1.78)	11,809(1.76)	3,943(1.84)	3,746(2.10)	17(1.42)	184(1.92)
	3	16,701(2.36)	15,313(2.29)	5,103(2.38)	5,515(3.09)	19(1.58)	176(1.83)
	4	17,602(2.49)	16,745(2.50)	5,347(2.50)	4,625(2.59)	21(1.75)	200(2.08)
	5	20,325(2.87)	19,135(2.86)	6,114(2.85)	5,511(3.08)	23(1.92)	224(2.33)
	6	26,100(3.69)	23,751(3.55)	7,679(3.58)	9,088(5.09)	28(2.33)	200(2.08)
	7	27,775(3.92)	25,712(3.84)	8,089(3.78)	9,332(5.22)	30(2.50)	216(2.25)
	8	29,423(4.16)	27,387(4.09)	8,825(4.12)	9,222(5.16)	32(2.67)	232(2.42)
	16	46,233(6.53)	44,096(6.59)	13,685(6.39)	11,975(6.70)	48(4.00)	360(3.75)
large	1	8,393	8,065	3,158	1,918	12	1,056
	2	15,755(1.88)	14,653(1.82)	5,959(1.89)	4,045(2.11)	17(1.42)	2,104(1.99)
	3	19,782(2.36)	18,046(2.24)	7,150(2.26)	5,970(3.11)	19(1.58)	1,856(1.76)
	4	20,954(2.50)	19,738(2.45)	7,763(2.46)	4,968(2.59)	21(1.75)	2,120(2.01)
	5	22,947(2.73)	21,496(2.67)	8,109(2.57)	5,850(3.05)	23(1.92)	1,744(1.65)
	6	27,217(3.24)	25,410(3.15)	9,436(2.99)	7,297(3.80)	28(2.33)	1,880(1.78)
	7	29,193(3.48)	27,337(3.39)	10,048(3.18)	7,820(4.08)	27(2.25)	2,016(1.91)
	8	30,285(3.61)	28,871(3.58)	10,494(3.32)	6,805(3.55)	29(2.42)	2,152(2.04)
	16	49,949(5.95)	48,129(5.97)	16,557(5.24)	10,970(5.72)	45(3.75)	2,216(2.10)
small	1	6,566	6,222	1,965	1,787	12	96
	2	13,459(2.05)	12,287(1.97)	4,003(2.04)	4,526(2.53)	13(1.08)	80(0.83)
	3	16,742(2.55)	15,500(2.49)	5,049(2.57)	5,474(3.06)	15(1.25)	96(1.00)
	4	17,382(2.65)	16,402(2.64)	5,203(2.65)	4,909(2.75)	17(1.42)	112(1.17)
	5	19,293(2.94)	18,296(2.94)	5,734(2.92)	5,273(2.95)	19(1.58)	128(1.33)
	6	22,145(3.37)	20,923(3.36)	6,742(3.43)	6,177(3.46)	24(2.00)	144(1.50)
	7	23,790(3.62)	22,594(3.63)	7,069(3.60)	6,436(3.60)	23(1.92)	160(1.67)
	8	25,483(3.88)	24,383(3.92)	7,541(3.84)	6,817(3.81)	28(2.33)	176(1.83)
	16	41,206(6.28)	39,658(6.37)	12,183(6.20)	10,037(5.62)	45(3.75)	304(3.17)

Table 5.14: Mvt area results (continued)

The speedup obtained on these two benchmarks is smaller than the speedup obtained with the previous benchmarks. On the Gemver benchmark it was expected, since four SCoPs are recognized in this code and this implies that three inter-SCoP split-rebuild functions have been added to the code. To further investigate these results a test on the MVT benchmark with parallel degree two have been performed. In particular, the execution times of the different parts (computation and split-rebuild functions) composing the generated code have been profiled. The results are the following: both the loops take 45152 cycles, and the sum of their execution time is 90304 cycles, that is the expected result. The motivation behind the result is due to the fact that the split and rebuild operations in this test are not negligible: The vertical split of a matrix costs 15403 cycles, an horizontal split costs 7604 cycles, and they are executed several times. These data also motivate the results on the Gemver. Since it has more SCoPs, the results are worse than the MVT.

The growth of the area is still sub-linear, but greater than the matrix multiply pattern. This too is explained by the number of SCoPs, since every SCoPs has some split, and every split creates different arrays and no sharing is supported as now.

5.2.3 Benchmark Pattern: 1D stencil

The Jacobi 1D is the only test for this pattern. As already said six tests are performed on this benchmark. Performance results are in Table 5.15, and area results are in Table 5.16.

Jacobi 1D	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
default	1	*n.a.*	25,492	103.15	0.30	*n.a.*	*n.a.*
	2	20,347	19,132	105.73	0.54	1.25	1.33
	3	18,405	17,182	104.37	0.42	1.38	1.48
	4	17,503	16,272	101.57	0.15	1.45	1.56
	5	16,861	15,622	105.52	0.52	1.51	1.63
	6	16,899	15,452	102.08	0.20	1.50	1.64
	7	15,837	14,582	101.05	0.10	1.60	1.74
	8	16,415	15,152	102.17	0.21	1.55	1.68
	16	16,579	15,252	101.00	0.10	1.53	1.67

Table 5.15: Jacobi 1D performance results

Jacobi 1D	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
large	1	*n.a.*	103,492	102.17	0.21	*n.a.*	*n.a.*
	2	82,447	77,632	108.05	0.74	1.25	1.33
	3	74,005	69,182	102.24	0.22	1.39	1.49
	4	69,853	65,022	103.03	0.29	1.48	1.59
	5	67,261	62,422	102.67	0.26	1.53	1.65
	6	66,599	60,952	101.94	0.19	1.55	1.69
	7	64,157	59,302	101.13	0.11	1.61	1.74
	8	63,385	58,522	105.55	0.53	1.63	1.76
	16	60,919	55,992	103.41	0.33	1.69	1.84
large_mul	1	*n.a.*	63,693	103.93	0.38	*n.a.*	*n.a.*
	2	36,678	31,863	102.66	0.26	1.73	1.99
	3	26,126	21,303	104.96	0.47	2.43	2.98
	4	20,854	16,023	106.11	0.58	3.05	3.97
	5	17,662	12,823	103.84	0.37	3.60	4.96
	6	16,390	10,743	105.64	0.53	3.88	5.92
	7	13,998	9,143	105.10	0.48	4.55	6.96
	8	12,886	8,023	105.86	0.55	4.94	7.93
	16	9,670	4,743	106.02	0.57	6.58	13.42
large_mul_steps	1	*n.a.*	159,228	107.05	0.66	*n.a.*	*n.a.*
	2	84,468	79,653	106.46	0.61	1.88	1.99
	3	58,076	53,253	107.12	0.66	2.74	2.99
	4	44,884	40,053	107.16	0.67	3.54	3.97
	5	36,892	32,053	104.36	0.42	4.31	4.96
	6	32,500	26,853	106.69	0.63	4.89	5.92
	7	27,708	22,853	105.06	0.48	5.74	6.96
	8	24,916	20,053	104.85	0.46	6.39	7.94
	16	*n.a.*	*n.a.*	101.88	0.19	*n.a.*	*n.a.*

Table 5.15: Jacobi 1D performance results (continued)

Jacobi 1D	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
mul_steps	1	*n.a.*	39,228	106.66	0.62	*n.a.*	*n.a.*
	2	20,868	19,653	104.03	0.39	1.87	1.99
	3	14,476	13,253	101.83	0.18	2.70	2.95
	4	11,284	10,053	105.34	0.51	3.47	3.90
	5	9,292	8,053	104.72	0.45	4.22	4.87
	6	8,300	6,853	105.69	0.54	4.72	5.72
	7	6,908	5,653	104.05	0.39	5.67	6.93
	8	6,516	5,253	105.06	0.48	6.02	7.46
	16	4,180	2,853	103.73	0.36	9.38	13.74
mul	1	*n.a.*	15,693	103.67	0.35	*n.a.*	*n.a.*
	2	9,078	7,863	105.46	0.52	1.72	1.99
	3	6,526	5,303	103.18	0.31	2.40	2.95
	4	5,254	4,023	105.34	0.51	2.98	3.90
	5	4,462	3,223	106.20	0.58	3.51	4.86
	6	4,190	2,743	104.12	0.40	3.74	5.72
	7	3,518	2,263	103.17	0.31	4.46	6.93
	8	3,366	2,103	104.66	0.45	4.66	7.46
	16	2,470	1,143	103.59	0.35	6.35	13.72

Table 5.15: Jacobi 1D performance results (continued)

Jacobi 1D	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
default	1	6,376	5,945	1,870	2,347	22	16
	2	8,731(1.37)	8,150(1.37)	2,588(1.38)	3,021(1.29)	22(1.00)	36(2.25)
	3	10,972(1.72)	10,322(1.74)	3,260(1.74)	3,536(1.51)	22(1.00)	44(2.75)
	4	11,881(1.86)	11,223(1.89)	3,531(1.89)	3,730(1.59)	22(1.00)	52(3.25)
	5	13,250(2.08)	12,515(2.11)	3,965(2.12)	4,091(1.74)	22(1.00)	60(3.75)
	6	15,339(2.41)	14,493(2.44)	4,511(2.41)	4,606(1.96)	22(1.00)	68(4.25)
	7	16,399(2.57)	15,598(2.62)	4,816(2.58)	4,943(2.11)	22(1.00)	76(4.75)
	8	17,924(2.81)	17,035(2.87)	5,292(2.83)	5,295(2.26)	22(1.00)	84(5.25)
	16	29,537(4.63)	28,246(4.75)	8,637(4.62)	8,099(3.45)	22(1.00)	148(9.25)

Table 5.16: Jacobi 1D area results

Jacobi 1D	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
	1	6,454	6,003	1,926	2,347	22	16
	2	9,215(1.43)	8,671(1.44)	2,795(1.45)	3,030(1.29)	22(1.00)	36(2.25)
	3	11,064(1.71)	10,367(1.73)	3,300(1.71)	3,547(1.51)	22(1.00)	44(2.75)
	4	11,939(1.85)	11,367(1.89)	3,567(1.85)	3,740(1.59)	22(1.00)	52(3.25)
large	5	13,395(2.08)	12,699(2.12)	3,987(2.07)	4,104(1.75)	22(1.00)	60(3.75)
	6	15,373(2.38)	14,578(2.43)	4,552(2.36)	4,617(1.97)	22(1.00)	68(4.25)
	7	16,852(2.61)	15,917(2.65)	4,993(2.59)	4,958(2.11)	22(1.00)	76(4.75)
	8	17,992(2.79)	17,128(2.85)	5,316(2.76)	5,150(2.19)	22(1.00)	84(5.25)
	16	30,574(4.74)	29,302(4.88)	8,838(4.59)	7,951(3.39)	22(1.00)	148(9.25)
	1	6,196	5,771	1,823	2,226	15	16
	2	8,797(1.42)	8,205(1.42)	2,672(1.47)	2,875(1.29)	15(1.00)	36(2.25)
	3	10,963(1.77)	10,181(1.76)	3,258(1.79)	3,392(1.52)	17(1.13)	44(2.75)
	4	12,033(1.94)	11,454(1.98)	3,553(1.95)	3,589(1.61)	19(1.27)	52(3.25)
large_mul	5	13,891(2.24)	13,304(2.31)	4,081(2.24)	3,937(1.77)	21(1.40)	60(3.75)
	6	15,663(2.53)	14,897(2.58)	4,651(2.55)	4,481(2.01)	23(1.53)	68(4.25)
	7	17,163(2.77)	16,327(2.83)	5,081(2.79)	4,785(2.15)	25(1.67)	76(4.75)
	8	18,373(2.97)	17,573(3.05)	5,436(2.98)	4,985(2.24)	27(1.80)	84(5.25)
	16	31,050(5.01)	29,923(5.19)	8,799(4.83)	7,772(3.49)	43(2.87)	148(9.25)
	1	6,199	5,788	1,847	2,226	15	16
	2	8,941(1.44)	8,391(1.45)	2,639(1.43)	2,877(1.29)	15(1.00)	36(2.25)
	3	10,812(1.74)	10,082(1.74)	3,249(1.76)	3,392(1.52)	17(1.13)	44(2.75)
	4	12,095(1.95)	11,445(1.98)	3,607(1.95)	3,589(1.61)	19(1.27)	52(3.25)
large_mul_steps	5	13,906(2.24)	13,294(2.30)	4,120(2.23)	3,936(1.77)	21(1.40)	60(3.75)
	6	15,587(2.51)	14,783(2.55)	4,555(2.47)	4,453(2.00)	23(1.53)	68(4.25)
	7	17,340(2.80)	16,507(2.85)	5,091(2.76)	4,800(2.16)	25(1.67)	76(4.75)
	8	18,403(2.97)	17,583(3.04)	5,419(2.93)	4,975(2.23)	27(1.80)	84(5.25)
	16	30,962(4.99)	29,878(5.16)	8,955(4.85)	7,753(3.48)	43(2.87)	148(9.25)
	1	6,172	5,712	1,822	2,226	15	16
	2	9,029(1.46)	8,442(1.48)	2,789(1.53)	2,867(1.29)	15(1.00)	36(2.25)
	3	10,792(1.75)	10,187(1.78)	3,142(1.72)	3,380(1.52)	17(1.13)	44(2.75)
	4	11,983(1.94)	11,347(1.99)	3,537(1.94)	3,579(1.61)	19(1.27)	52(3.25)
mul_steps	5	13,794(2.23)	13,124(2.30)	4,103(2.25)	3,929(1.77)	21(1.40)	60(3.75)
	6	15,479(2.51)	14,671(2.57)	4,495(2.47)	4,439(1.99)	23(1.53)	68(4.25)
	7	17,112(2.77)	16,236(2.84)	5,035(2.76)	4,815(2.16)	25(1.67)	76(4.75)
	8	18,343(2.97)	17,501(3.06)	5,408(2.97)	5,132(2.31)	27(1.80)	84(5.25)
	16	30,598(4.96)	29,324(5.13)	9,042(4.96)	7,905(3.55)	43(2.87)	148(9.25)

Table 5.16: Jacobi 1D area results (continued)

Jacobi 1D	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
	1	6,115	5,712	1,764	2,226	15	16
	2	8,753(1.43)	8,239(1.44)	2,576(1.46)	2,870(1.29)	15(1.00)	36(2.25)
	3	10,569(1.73)	9,978(1.75)	3,109(1.76)	3,379(1.52)	17(1.13)	44(2.75)
	4	12,042(1.97)	11,398(2.00)	3,620(2.05)	3,585(1.61)	19(1.27)	52(3.25)
mul	5	13,835(2.26)	13,180(2.31)	4,098(2.32)	3,927(1.76)	21(1.40)	60(3.75)
	6	15,495(2.53)	14,729(2.58)	4,565(2.59)	4,443(2.00)	23(1.53)	68(4.25)
	7	16,920(2.77)	16,127(2.82)	5,002(2.84)	4,801(2.16)	25(1.67)	76(4.75)
	8	18,222(2.98)	17,341(3.04)	5,419(3.07)	5,130(2.30)	27(1.80)	84(5.25)
	16	30,648(5.01)	29,396(5.15)	8,982(5.09)	7,906(3.55)	43(2.87)	148(9.25)

Table 5.16: Jacobi 1D area results (continued)

From the results is clear that the division operation is a real bottleneck. The division almost nullify the effect of the parallelism, since *Bambu* does not parallelize that type of operation and all the divisions have to be executed sequentially. This leads to a little speedup for all the test done with the original kernel, that does not increase with the growth of the parallel degree. Looking to the other tests, where the division is substituted with a multiplication for the inverse, the speedup is similar to the matrix multiplication pattern test cases. It is almost equal to the parallel degree when the external split-rebuild time is not considered part of the computation. The speedup is a little worse if this overhead is taken into account, but it remains important. The tests with more steps show how a large number of iterations of the stencil kernel can reduce the impact of the split-rebuild operations on the computation. This is important because, provided that the number of iterations is significant, it means that the proposed solution can be adopted even if the split is costly.

5.2.4 Benchmark Pattern: 2D stencil on different arrays

The test adopted for this pattern is the Jacobi 2D. Its performance results are reported in Table 5.17 and its area results are in Table 5.18.

Jacobi 2D	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
default	1	*n.a.*	2,306,932	104.76	0.45	*n.a.*	*n.a.*
	2	1,186,847	1,106,432	107.45	0.69	1.94	2.08
	3	951,015	870,592	106.60	0.62	2.42	2.64
	4	739,023	658,592	104.42	0.42	3.12	3.50
	5	606,551	526,092	105.17	0.49	3.80	4.38
	6	527,039	446,592	104.61	0.44	4.37	5.16
	7	396,607	316,132	102.81	0.27	5.81	7.29
	8	421,055	340,592	104.87	0.46	5.47	6.77
	16	262,119	181,592	105.31	0.50	8.80	12.70
steps	1	*n.a.*	5,767,327	109.29	0.85	*n.a.*	*n.a.*
	2	2,846,492	2,766,077	106.92	0.65	2.02	2.08
	3	2,256,900	2,176,477	105.65	0.54	2.55	2.64
	4	1,726,908	1,646,477	105.90	0.56	3.33	3.50
	5	1,395,686	1,315,227	103.39	0.33	4.13	4.38
	6	1,196,924	1,116,477	102.71	0.26	4.81	5.16
	7	870,802	790,327	108.20	0.76	6.62	7.29
	8	931,940	851,477	105.74	0.54	6.18	6.77
	16	534,504	453,977	104.83	0.46	10.79	12.70
large	1	*n.a.*	59,530,932	99.05	−0.10	*n.a.*	*n.a.*
	2	30,532,447	28,530,432	*n.a.*	*n.a.*	1.94	2.08
	3	21,022,315	19,020,292	*n.a.*	*n.a.*	2.83	3.12
	4	18,794,623	16,792,592	*n.a.*	*n.a.*	3.16	3.54
	5	15,432,151	13,430,092	*n.a.*	*n.a.*	3.85	4.43
	6	11,512,199	9,510,152	*n.a.*	*n.a.*	5.17	6.25
	7	11,666,147	9,664,092	*n.a.*	*n.a.*	5.10	6.16
	8	10,455,655	8,453,592	*n.a.*	*n.a.*	5.69	7.04
	16	6,286,219	4,284,092	*n.a.*	*n.a.*	9.47	13.89

Table 5.17: Jacobi 2D performance results (continued)

Jacobi 2D	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
default	1	5578	5204	1806	1841	20	128
	2	10,289(1.84)	9,691(1.86)	3,256(1.80)	3,935(2.14)	30(1.50)	194(1.52)
	3	13,420(2.41)	12,559(2.41)	4,190(2.32)	4,936(2.68)	31(1.55)	178(1.39)
	4	14,094(2.53)	13,271(2.55)	4,386(2.43)	5,019(2.73)	33(1.65)	194(1.52)
	5	16,277(2.92)	15,430(2.97)	4,828(2.67)	5,558(3.02)	35(1.75)	210(1.64)
	6	18,545(3.32)	17,464(3.36)	5,661(3.13)	6,567(3.57)	37(1.85)	178(1.39)
	7	19,837(3.56)	18,665(3.59)	6,124(3.39)	7,102(3.86)	40(2.00)	186(1.45)
	8	22,272(3.99)	20,956(4.03)	6,848(3.79)	7,642(4.15)	41(2.05)	194(1.52)
	16	35,150(6.30)	33,573(6.45)	10,321(5.71)	11,971(6.50)	57(2.85)	258(2.02)
steps	1	5,603	5,228	1,784	1,845	20	128
	2	10,373(1.85)	9,683(1.85)	3,295(1.85)	3,931(2.13)	30(1.50)	194(1.52)
	3	13,286(2.37)	12,428(2.38)	4,172(2.34)	4,940(2.68)	31(1.55)	178(1.39)
	4	13,888(2.48)	13,236(2.53)	4,167(2.34)	5,018(2.72)	33(1.65)	194(1.52)
	5	16,265(2.90)	15,391(2.94)	4,997(2.80)	5,565(3.02)	35(1.75)	210(1.64)
	6	18,709(3.34)	17,636(3.37)	5,770(3.23)	6,575(3.56)	37(1.85)	178(1.39)
	7	19,929(3.56)	18,796(3.60)	6,061(3.40)	7,101(3.85)	40(2.00)	186(1.45)
	8	21,605(3.86)	20,466(3.91)	6,496(3.64)	7,639(4.14)	41(2.05)	194(1.52)
	16	35,238(6.29)	33,689(6.44)	10,313(5.78)	11,987(6.50)	57(2.85)	258(2.02)
large	1	8776	8239	4190	1916	20	2048
	2	n.a.					
	3	n.a.					
	4	n.a.					
	5	n.a.					
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					

Table 5.18: Jacobi 2D area results (continued)

The optimal result is reached when _PB_NI-2 is multiple of the degree of parallelism. In this case indeed the code reconstruction phase can avoid to add a check on the boundaries since all the calculated values have to be written. For example, the 7 kernels test with the default dimensions: 100-2 is the number of iterations that the original loop has to do. With a parallel degree of 7, only 14 iterations will be instantiated, and all those will be doing calculation on 7 rows of the original matrix. With this the proposed solution obtains a speedup a bit better than the degree of parallelism if the split-rebuild overhead is not calculated (the test with 7 kernels, time excluded, has a speedup of 7.2). As above, the test with a greater _PB_TSTEPS test is showing that the impact of the split and rebuild operations is smaller the more are the iterations of the stencil loop. The large test did not finish the synthesis, since the timeout was reached.

5.2.5 Benchmark Pattern: 2D stencil on the same array

For this pattern the selected benchmark is the Seidel 2D. This test has been run for six different cases, as the Jacobi 1D. The results are reported in Table 5.19 and Table 5.20.

Seidel 2D	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
	1	*n.a.*	4,607,472	106.44	0.61	*n.a.*	*n.a.*
	2	3,090,230	3,049,022	105.81	0.55	1.49	1.51
	3	2,432,939	2,391,522	103.86	0.37	1.89	1.92
	4	2,235,242	2,194,022	104.85	0.46	2.06	2.10
default	5	2,063,806	2,022,422	102.91	0.28	2.23	2.27
	6	1,987,291	1,945,862	*n.a.*	*n.a.*	2.31	2.36
	7	1,958,209	*n.a.*	*n.a.*	*n.a.*	2.35	*n.a.*
	8			n.a.			
	16			n.a.			
	1	*n.a.*	74,453,872	103.08	0.30	*n.a.*	*n.a.*
	2	47,495,369	46,854,562	103.75	0.36	1.56	1.58
	3	39,987,750	39,320,142	100.90	0.09	1.86	1.89
	4	35,429,847	34,789,032	102.02	0.20	2.10	2.14
large	5	32,650,451	32,009,632	*n.a.*	*n.a.*	2.28	2.32
	6	30,946,985	30,306,162	*n.a.*	*n.a.*	2.40	2.45
	7			n.a.			
	8			n.a.			
	16			n.a.			

Table 5.19: Seidel 2D performance results

Seidel 2D	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
mul_large	1	*n.a.*	58,613,473	102.72	0.27	*n.a.*	*n.a.*
	2	29,294,830	28,654,023	98.47	−0.15	2.00	2.04
	3	20,958,111	20,290,503	103.36	0.32	2.79	2.88
	4	15,961,848	15,321,033	103.33	0.32	3.67	3.82
	5	12,949,652	12,308,833	*n.a.*	*n.a.*	4.52	4.76
	6	10,993,046	10,352,223	*n.a.*	*n.a.*	5.33	5.66
	7			n.a.			
	8			n.a.			
	16			n.a.			
mul_large_steps	1	*n.a.*	146,533,678	106.42	0.60	*n.a.*	*n.a.*
	2	72,275,860	71,635,053	104.88	0.46	2.02	2.04
	3	51,393,861	50,726,253	104.70	0.45	2.85	2.88
	4	38,943,393	38,302,578	103.76	0.36	3.76	3.82
	5	31,412,897	30,772,078	101.27	0.12	4.66	4.76
	6			n.a.			
	7			n.a.			
	8			n.a.			
	16			n.a.			
mul_steps	1	*n.a.*	10,200,778	106.42	0.60	*n.a.*	*n.a.*
	2	5,421,803	5,375,578	104.88	0.46	1.88	1.89
	3	3,475,385	3,429,153	104.70	0.45	2.93	2.97
	4	2,872,590	2,826,328	103.76	0.36	3.55	3.60
	5	2,175,146	2,128,903	101.27	0.12	4.68	4.79
	6	1,989,024	1,942,753	*n.a.*	*n.a.*	5.12	5.25
	7			n.a.			
	8			n.a.			
	16			n.a.			

Table 5.19: Seidel 2D performance results (continued)

Seidel 2D	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
	1	*n.a.*	4,080,313	106.59	0.62	*n.a.*	*n.a.*
	2	2,196,458	2,150,233	107.28	0.68	1.85	1.89
	3	1,417,895	1,371,663	104.77	0.46	2.87	2.97
	4	1,176,795	1,130,533	104.00	0.38	3.46	3.60
mul	5	897,806	851,563	102.69	0.26	4.54	4.79
	6	823,374	777,103	*n.a.*	*n.a.*	4.95	5.25
	7			n.a.			
	8			n.a.			
	16			n.a.			

Table 5.19: Seidel 2D performance results (continued)

Seidel 2D	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
	1	7,116	6,469	2,092	3,571	43	64
	2	14,122(1.98)	12,722(1.97)	4,378(2.09)	7,417(2.08)	51(1.19)	102(1.59)
	3	19,870(2.79)	17,669(2.73)	6,265(2.99)	11,237(3.15)	57(1.33)	94(1.47)
	4	29,210(4.10)	24,788(3.83)	9,182(4.39)	16,804(4.71)	66(1.53)	102(1.59)
default	5	38,961(5.48)	32,772(5.07)	12,434(5.94)	23,270(6.52)	87(2.02)	110(1.72)
	6			n.a.			
	7			n.a.			
	8			n.a.			
	16			n.a.			
	1	8,493	7,569	3,216	3,588	43	1,024
	2	15,044(1.77)	13,518(1.79)	5,486(1.71)	7,076(1.97)	51(1.19)	1,542(1.51)
	3	21,140(2.49)	19,163(2.53)	7,007(2.18)	11,371(3.17)	57(1.33)	1,414(1.38)
	4	29,571(3.48)	25,262(3.34)	9,680(3.01)	16,334(4.55)	66(1.53)	1,542(1.51)
large	5			n.a.			
	6			n.a.			
	7			n.a.			
	8			n.a.			
	16			n.a.			

Table 5.20: Seidel 2D area results

Seidel 2D	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
mul_large	1	8,995	8,101	3,375	3,213	35	1,024
	2	14,535(1.62)	12,893(1.59)	5,298(1.57)	6,516(2.03)	45(1.29)	1,540(1.50)
	3	20,707(2.30)	18,478(2.28)	6,700(1.99)	10,606(3.30)	53(1.51)	1,412(1.38)
	4	28,715(3.19)	24,869(3.07)	9,594(2.84)	15,185(4.73)	64(1.83)	1,540(1.50)
	5	n.a.					
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					
mul_large_steps	1	8,907	8,086	3,440	3,213	35	1,024
	2	14,459(1.62)	13,091(1.62)	5,129(1.49)	6,497(2.02)	45(1.29)	1,540(1.50)
	3	20,844(2.34)	18,323(2.27)	6,944(2.02)	10,597(3.30)	53(1.51)	1,412(1.38)
	4	28,545(3.20)	24,806(3.07)	9,257(2.69)	15,191(4.73)	64(1.83)	1,540(1.50)
	5	n.a.					
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					
mul_steps	1	7,021	6,314	2,193	3,188	35	64
	2	13,426(1.91)	11,725(1.86)	4,220(1.92)	6,874(2.16)	45(1.29)	100(1.56)
	3	19,646(2.80)	17,114(2.71)	5,993(2.73)	10,651(3.34)	53(1.51)	92(1.44)
	4	26,801(3.82)	23,190(3.67)	8,205(3.74)	15,436(4.84)	64(1.83)	100(1.56)
	5	37,465(5.34)	32,357(5.12)	11,572(5.28)	21,970(6.89)	52(1.49)	108(1.69)
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					
mul	1	7,009	6,308	2,152	3,189	35	64
	2	13,284(1.90)	11,633(1.84)	4,152(1.93)	6,859(2.15)	45(1.29)	100(1.56)
	3	19,391(2.77)	17,238(2.73)	5,899(2.74)	10,672(3.35)	53(1.51)	92(1.44)
	4	26,883(3.84)	23,070(3.66)	8,166(3.79)	15,435(4.84)	64(1.83)	100(1.56)
	5	37,139(5.30)	32,624(5.17)	11,586(5.38)	21,923(6.87)	74(2.11)	108(1.69)
	6	n.a.					
	7	n.a.					
	8	n.a.					
	16	n.a.					

Table 5.20: Seidel 2D area results (continued)

The greatest issue with this pattern is due to HLS time. All the results that are not available are caused by reaching the HLS timeout. This timeout is due to a large number of operations that are created in the phase of the loop initialization and finalization (see 3.3.4.4). They caused the module binding step in the HLS and the synthesis to be too long. All the available results are similar to the ones obtained on the other benchmarks. Their speedup is close to the parallel degree, and they are closer to it if the number of iterations of the extern loop of the stencil is a multiple of the parallel factor.

For the area the growth is greater than the other tests, for all but the DSPs and BRAMs. This happens because in the initialization and finalization phase a lot of sums working on registers are instantiated, and the resources used to implement these operations are LUT-FF pairs (LUTs for the sums, FFs for the registers). In this phase also some multiplications are created but this does not modify the number of DSPs used (the synthesis tool performs resource sharing).

The growth of the LUT-FF pairs in this benchmark is more than linear: the initialization and finalization process is implemented in a way that is too costly with no resource sharing done.

5.2.6 Considerations on Stencils

In the benchmarks analyzed in the previous two sections the reuse technique does not provide any benefit because of the considered experimental setup. This happens because the structure of the computation is the one in Figure 5.2, where the memory accesses (the squares) are in parallel with the calculation of the sums (the ellipses). The sums are floating point operations so they take more cycles than loads from the memory, and the loads have already ended when the next sums have to start.

Instead with the sums organized as tree (see Figure 5.1) more sums can be executed in parallel (if all the data are retrieved). This bring a speedup if all the data are ready to be used, as it is visible even with the easy example shown in Figure 5.2 and Figure 5.1. The three sums, organized as a tree (Figure 5.1), need two steps to be completed while organized as a chain (5.2) they would need three. The data are all ready because of the data reuse. The GCC optimization

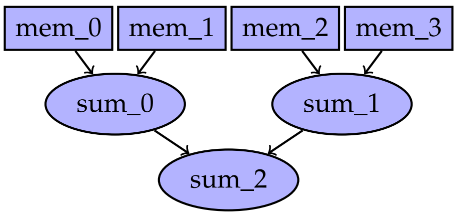


Figure 5.1: Tree schedule with four data and three additions, when all the data can be recovered in a parallel way

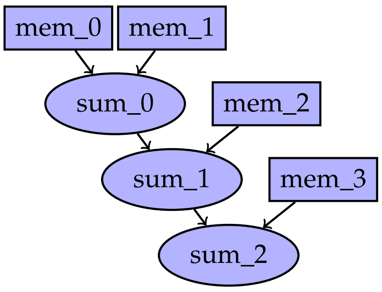


Figure 5.2: Chain schedule with four data and three additions

-funsafe-math-optimizations has been activated to try this different configuration. This optimization allow optimizations for floating-point arithmetic that assume that arguments and results are valid but it may violate IEEE or ANSI standards. In particular, in this way GCC assumes that associative property holds for floating point sums. The obtained results are reported in the following tables.

Jacobi 2D Unsafe	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
default	1	*n.a.*	2,018,812	106.28	0.59	*n.a.*	*n.a.*
	2	1,042,787	962,372	106.89	0.64	1.93	2.09
	3	853,995	773,572	105.78	0.55	2.36	2.60
	4	665,523	585,092	104.31	0.41	3.03	3.45
	5	547,731	467,292	105.21	0.49	3.68	4.32
	6	477,059	396,612	102.25	0.22	4.23	5.09
	7	355,427	274,972	103.91	0.38	5.67	7.34
	8	382,835	302,372	104.56	0.44	5.27	6.67
	16	241,539	161,012	102.50	0.24	8.35	12.53
steps	1	*n.a.*	5,047,027	106.24	0.59	*n.a.*	*n.a.*
	2	2,486,342	2,405,927	106.72	0.63	2.02	2.09
	3	2,014,350	1,933,927	106.45	0.61	2.50	2.60
	4	1,543,158	1,462,727	105.34	0.51	3.27	3.45
	5	1,248,666	1,168,227	106.09	0.57	4.04	4.32
	6	1,071,974	991,527	103.76	0.36	4.70	5.09
	7	767,882	687,427	104.31	0.41	6.57	7.34
	8	836,390	755,927	102.65	0.26	6.03	6.67
	16	483,054	402,527	103.34	0.32	10.44	12.53
large	1	*n.a.*	52,090,812	101.38	0.14	*n.a.*	*n.a.*
	2	26,812,387	24,810,372	*n.a.*	*n.a.*	1.94	2.09
	3	18,542,275	16,540,252	*n.a.*	*n.a.*	2.80	3.14
	4	16,927,123	14,925,092	*n.a.*	*n.a.*	3.07	3.49
	5	13,938,131	11,936,092	*n.a.*	*n.a.*	3.73	4.36
	6	10,272,179	8,270,132	*n.a.*	*n.a.*	5.07	6.29
	7	10,590,467	8,588,412	*n.a.*	*n.a.*	4.91	6.06
	8	9,514,435	7,512,372	*n.a.*	*n.a.*	5.47	6.93
	16	5,808,139	3,806,012	*n.a.*	*n.a.*	8.96	13.68

Table 5.21: Jacobi 2D performance results with unsafe math(continued)

Jacobi 2D	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
original	1	7,016	6,579	2,245	2,255	20	128
	2	11,032(1.57)	10,234(1.56)	3,614(1.61)	3,558(1.58)	28(1.40)	194(1.52)
	3	13,894(1.98)	12,954(1.97)	4,418(1.97)	4,559(2.02)	29(1.45)	178(1.39)
	4	16,089(2.29)	15,083(2.29)	5,027(2.24)	5,044(2.24)	31(1.55)	194(1.52)
	5	18,707(2.67)	17,722(2.69)	5,673(2.53)	5,793(2.57)	33(1.65)	210(1.64)
	6	20,803(2.97)	19,593(2.98)	6,308(2.81)	6,796(3.01)	35(1.75)	178(1.39)
	7	23,604(3.36)	22,177(3.37)	7,173(3.20)	7,514(3.33)	38(1.90)	186(1.45)
	8	25,718(3.67)	24,306(3.69)	7,742(3.45)	8,261(3.66)	39(1.95)	194(1.52)
	16	44,378(6.33)	42,341(6.44)	12,833(5.72)	14,232(6.31)	55(2.75)	258(2.02)
steps	1	7,036	6,573	2,276	2,259	20	128
	2	10,948(1.56)	10,196(1.55)	3,477(1.53)	3,562(1.58)	28(1.40)	194(1.52)
	3	13,811(1.96)	12,908(1.96)	4,380(1.92)	4,563(2.02)	29(1.45)	178(1.39)
	4	15,811(2.25)	15,018(2.28)	4,891(2.15)	5,047(2.23)	31(1.55)	194(1.52)
	5	18,685(2.66)	17,593(2.68)	5,686(2.50)	5,795(2.57)	33(1.65)	210(1.64)
	6	21,013(2.99)	19,885(3.03)	6,394(2.81)	6,786(3.00)	35(1.75)	178(1.39)
	7	23,304(3.31)	21,887(3.33)	6,993(3.07)	7,514(3.33)	38(1.90)	186(1.45)
	8	25,688(3.65)	24,209(3.68)	7,776(3.42)	8,264(3.66)	39(1.95)	194(1.52)
	16	45,055(6.40)	42,681(6.49)	13,166(5.78)	14,251(6.31)	55(2.75)	258(2.02)
large	1	10,423	9,942	4,753	2,342	20	2,048
	2			n.a.			
	3			n.a.			
	4			n.a.			
	5			n.a.			
	6			n.a.			
	7			n.a.			
	8			n.a.			
	16			n.a.			

Table 5.22: Jacobi 2D area results with unsafe math(continued)

Seidel 2D Unsafe	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
original	1	*n.a.*	2,255,232	107.48	0.70	*n.a.*	*n.a.*
	2	1,148,230	1,107,022	102.84	0.28	1.96	2.03
	3	719,579	678,162	103.75	0.36	3.13	3.32
	4	610,992	569,772	101.38	0.14	3.69	3.95
	5	503,806	462,422	102.12	0.21	4.47	4.87
	6	439,759	398,522	101.05	0.10	5.12	5.65
	7	*n.a.*	*n.a.*	100.83	0.08	*n.a.*	*n.a.*
	8			n.a.			
	16			n.a.			
large	1	*n.a.*	36,436,912	103.58	0.35	*n.a.*	*n.a.*
	2	16,560,819	15,920,012	102.61	0.25	2.20	2.28
	3	12,403,550	11,735,942	100.30	0.03	2.93	3.10
	4	9,497,847	8,857,032	100.27	0.03	3.83	4.11
	5	7,752,851	7,112,032	100.62	0.06	4.69	5.12
	6	6,618,615	5,977,792	96.50	−0.36	5.50	6.09
	7			n.a.			
	8			n.a.			
	16			n.a.			
large_mul	1	*n.a.*	36,436,913	106.35	0.60	*n.a.*	*n.a.*
	2	16,560,820	15,920,013	100.21	0.02	2.20	2.28
	3	12,403,551	11,735,943	100.93	0.09	2.93	3.10
	4	9,497,848	8,857,033	101.20	0.12	3.83	4.11
	5	7,752,852	7,112,033	99.11	−0.09	4.69	5.12
	6	6,618,616	5,977,793	101.72	0.17	5.50	6.09
	7			n.a.			
	8			n.a.			
	16			n.a.			

Table 5.23: Seidel 2D performance results with unsafe math

Seidel 2D Unsafe	Parallel Degree	Num Cycles Split Included	Num Cycles Split Excluded	Clock Frequency (MHz)	Clock Slack (ns)	Speedup Split Included	Speedup Split Excluded
mul_large_steps	1	*n.a.*	91,092,278	102.89	0.28	*n.a.*	*n.a.*
	2	40,440,835	39,800,028	103.20	0.31	2.25	2.28
	3	30,007,461	29,339,853	100.69	0.07	3.03	3.10
	4	22,783,393	22,142,578	101.01	0.10	3.99	4.11
	5	18,420,897	17,780,078	102.71	0.26	4.94	5.12
	6	*n.a.*	*n.a.*	99.27	−0.07	*n.a.*	*n.a.*
	7			n.a.			
	8			n.a.			
	16			n.a.			
mul_steps	1	*n.a.*	6,342,028	104.00	0.38	*n.a.*	*n.a.*
	2	3,154,728	3,108,503	105.40	0.51	2.01	2.04
	3	1,949,385	1,903,153	102.63	0.26	3.25	3.33
	4	1,673,790	1,627,528	101.79	0.18	3.78	3.89
	5	1,225,946	1,179,703	103.69	0.36	5.17	5.37
	6	1,160,547	1,114,303	101.30	0.13	5.46	5.69
	7	*n.a.*	*n.a.*	101.84	0.18	*n.a.*	*n.a.*
	8			n.a.			
	16			n.a.			
mul	1	*n.a.*	2,536,813	103.94	0.38	*n.a.*	*n.a.*
	2	1,289,628	1,243,403	106.12	0.58	1.96	2.04
	3	807,495	761,263	103.16	0.31	3.14	3.33
	4	697,275	651,013	102.69	0.26	3.63	3.89
	5	518,126	471,883	102.03	0.20	4.89	5.37
	6	491,967	445,723	103.32	0.32	5.15	5.69
	7	393,964	347,713	100.44	0.04	6.43	7.29
	8			n.a.			
	16			n.a.			

Table 5.23: Seidel 2D performance results with unsafe math (continued)

Seidel 2D	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
original	1	8,305	7,584	2,533	3,258	27	64
	2	17,046(2.05)	15,658(2.06)	5,163(2.04)	7,138(2.19)	38(1.41)	102(1.59)
	3	23,009(2.77)	21,073(2.78)	6,699(2.64)	10,623(3.26)	45(1.67)	94(1.47)
	4	31,947(3.85)	29,241(3.86)	9,635(3.80)	14,954(4.59)	56(2.07)	102(1.59)
	5	40,351(4.86)	36,863(4.86)	11,869(4.69)	20,267(6.22)	36(1.33)	110(1.72)
	6	50,485(6.08)	46,276(6.10)	15,150(5.98)	26,426(8.11)	51(1.89)	94(1.47)
	7	60,725(7.31)	54,034(7.12)	18,417(7.27)	33,542(10.30)	128(4.74)	98(1.53)
	8	n.a.					
	16	n.a.					
large	1	9,611	8,817	3,537	3,321	27	1,024
	2	18,241(1.90)	16,794(1.90)	6,380(1.80)	7,004(2.11)	38(1.41)	1,542(1.51)
	3	24,414(2.54)	22,454(2.55)	7,911(2.24)	10,698(3.22)	45(1.67)	1,414(1.38)
	4	33,228(3.46)	30,582(3.47)	10,392(2.94)	14,801(4.46)	56(2.07)	1,542(1.51)
	5	41,374(4.30)	37,433(4.25)	13,047(3.69)	20,056(6.04)	46(1.70)	1,350(1.32)
	6	52,725(5.49)	47,045(5.34)	18,466(5.22)	26,575(8.00)	104(3.85)	1,414(1.38)
	7	n.a.					
	8	n.a.					
	16	n.a.					
mul_large	1	10,294	9,512	3,748	3,261	26	1,024
	2	17,697(1.72)	16,314(1.72)	6,245(1.67)	6,676(2.05)	37(1.42)	1,540(1.50)
	3	23,759(2.31)	21,652(2.28)	7,664(2.04)	10,353(3.17)	44(1.69)	1,412(1.38)
	4	32,976(3.20)	30,283(3.18)	10,347(2.76)	14,462(4.43)	55(2.12)	1,540(1.50)
	5	41,458(4.03)	37,426(3.93)	13,270(3.54)	19,748(6.06)	43(1.65)	1,348(1.32)
	6	51,408(4.99)	46,222(4.86)	17,002(4.54)	26,128(8.01)	71(2.73)	1,412(1.38)
	7	n.a.					
	8	n.a.					
	16	n.a.					
mul_large_steps	1	10,256	9,473	3,878	3,265	26	1,024
	2	17,607(1.72)	16,171(1.71)	6,242(1.61)	6,669(2.04)	37(1.42)	1,540(1.50)
	3	23,648(2.31)	21,744(2.30)	7,685(1.98)	10,361(3.17)	44(1.69)	1,412(1.38)
	4	32,239(3.14)	29,762(3.14)	9,921(2.56)	14,456(4.43)	55(2.12)	1,540(1.50)
	5	40,894(3.99)	37,564(3.97)	12,840(3.31)	19,736(6.04)	82(3.15)	1,348(1.32)
	6	52,511(5.12)	47,136(4.98)	17,968(4.63)	26,168(8.01)	59(2.27)	1,412(1.38)
	7	n.a.					
	8	n.a.					
	16	n.a.					

Table 5.24: Seidel 2D area results with unsafe math

Seidel 2D	Parallel Degree	LUT-FF pairs	LUTs	Slices	Registers	DSPs	BRAMs
mul_steps	1	8,350	7,651	2,559	3,206	26	64
	2	16,285(1.95)	15,030(1.96)	4,842(1.89)	6,816(2.13)	37(1.42)	100(1.56)
	3	22,417(2.68)	20,320(2.66)	6,714(2.62)	10,399(3.24)	44(1.69)	92(1.44)
	4	31,336(3.75)	28,621(3.74)	9,112(3.56)	14,701(4.59)	55(2.12)	100(1.56)
	5	39,177(4.69)	35,767(4.67)	11,718(4.58)	19,979(6.23)	65(2.50)	108(1.69)
	6	49,869(5.97)	45,189(5.91)	15,250(5.96)	26,113(8.15)	97(3.73)	92(1.44)
	7	60,355(7.23)	53,260(6.96)	19,338(7.56)	33,303(10.39)	130(5.00)	96(1.50)
	8	n.a.					
	16	n.a.					
mul	1	9,315	8,413	2,783	3,786	28	64
	2	16,134(1.73)	14,657(1.74)	4,917(1.77)	6,807(1.80)	37(1.32)	100(1.56)
	3	22,289(2.39)	20,319(2.42)	6,613(2.38)	10,393(2.75)	44(1.57)	92(1.44)
	4	31,557(3.39)	28,808(3.42)	9,246(3.32)	14,705(3.88)	55(1.96)	100(1.56)
	5	39,706(4.26)	36,285(4.31)	11,716(4.21)	19,978(5.28)	65(2.32)	108(1.69)
	6	49,325(5.30)	44,979(5.35)	14,534(5.22)	26,236(6.93)	93(3.32)	92(1.44)
	7	60,672(6.51)	54,434(6.47)	19,973(7.18)	33,250(8.78)	59(2.11)	96(1.50)
	8	n.a.					
	16	n.a.					

Table 5.24: Seidel 2D area results with unsafe math (continued)

The activation of the -funsafe-math-operations has the effect of improving the execution time of a single kernel, and this can be seen by the improvement of the sequential test case. The better speedup is caused by the data reuse: some data are already in registers, and the other data needed can be read in the same time from the memories. This cannot happen with the sequential case, because it can read only two data per iteration and the tree is modified as shown in Figure 5.3.

Analyzing the results of the Seidel 2D it can be noticed that the standard test (the one with the division) with this GCC optimization is improved. GCC is able to recognize the division for a constant and it substitutes that operation with a multiplication. This allows to exploit the parallelism. The results are similar to the "mul" test case, in which the division has been substituted by hand. The original test has a better result with a low degree of parallelism, while the modified one behave better with high degree of parallelism. This is

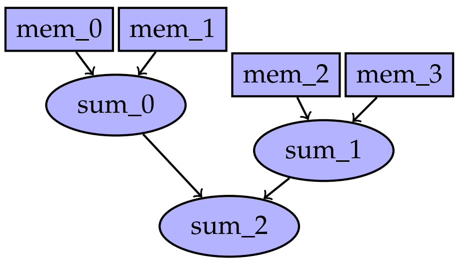


Figure 5.3: Tree schedule with four data and three additions, when all the data can not be recovered in a parallel way

probably due to some optimizations performed while substituting the division in the -funsafe-math-optimization pass, that does not scale well with the parallel structure.

However the gain from the data reuse is limited, and this is explained with the presence of all of the data on the BRAMs: working with floating point operations the reads from these memories are faster than the calculations, so they are hidden. With this setup it seems that having most of the data in registers does not provide large benefits. With a different experimental setup, where memories are slower than floating point functional units, the advantage of having data in the registers can emerge. Looking to Figure 5.2, if a memory read takes 2 cycles, and a sum takes 4 cycles, all the reads after the first two are ended when the next operation is scheduled, so that sums will never have to wait for data. If instead a read to the external memory takes 10 cycles, the chain would be greatly slowed, and every sum after the first would be stalled for 6 cycles before all the data are ready, if the read and the previous sum are issued in the same time. The same thing happens with the tree.

Having the tree structure however is more costly in area, since the single kernel is bigger than the one created with the operation issued as chain, and this is visible comparing the area tables (eg. Table 5.18 and Table 5.22).

Chapter 6

Conclusion and Future Work

From the obtained results it can be seen that for the algebraic test cases the theoretical optimal result is reached only when there is one single SCoP and the overhead for the split and rebuild operation is not considered. It is however also evident that the overhead introduced is not always significant, especially when the data have to be split only once. It can also be seen that in certain circumstances the proposed algorithm is not useful, and this happens where the number of iteration of the loop itself are comparable to the number of iteration of the split-rebuild operations. Another issue is the behavior of the algorithm for the stencil on the same array: the solution works, and the results (on timing) are good but it takes just too much time to perform HLS. This happens because many instructions are inserted and the number of instructions created depends on the level of desired parallelism. This makes the HLS long since it has to schedule a always bigger number of instructions for the header and tail of the loop to be parallelized. This happens (and also causes the area super-linear growth) because as it is now the algorithm schedules all the instructions for the head and the tail of the loop as single instructions. This can be improved and this waste of area can be reduced or eliminated.

However as it can be seen comparing the results in the tables with the resources of the targeted device, the occupied space percentage is low. The only components that are heavily used are the BRAMs, and that happens when some of the large test are considered. All the other components are lightly used and this is important because when the fraction of used device is low the timing is less constrained.

It is important to notice that the proposed methodology does not have an impact on the timing of the design. From the tables it can be noticed that the slack of the parallel tests is always in line with the one obtained by the sequential test.

The results are mostly in line with what was expected. The only unexpected result is the fact that the data sharing is mostly masked. Only with the instruction issued as a tree the results show the usefulness of the reuse created by the algorithm. This is caused, as has already been said, by the setup used for the experimental evaluations. The BRAMs are too fast compared to floating point calculations, and having the data on a BRAM or in a register creates no difference. In a real world application this would probably not happen, so the speedup can be even greater than the one shown in the previous chapter.

This work still have room for improvement. A better handling for the schedule functions and SCoPs with more than one statement has still to be done. This can reduce the overhead created by the inter-SCoP split and rebuilds, with selective split-rebuilds (avoid inter SCoP rebuild and split process if no operation are issued on that variable between the two scops) and reuse of space (avoid to reserve new BRAMs if the same array is being split and rebuilt in two different scops). Another option that can be implemented is the extension of the support for the stencils with more than two dimensions.

References

[Bas04] Cédric Bastoul, *Code generation in the polyhedral model is easier than you think*, PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques (Juan-les-Pins, France), September 2004, pp. 7–16.

[Bas11] Cédric Bastoul, *Openscop: A specification and a library for data exchange in polyhedral compilation tools*, Tech. report, Paris-Sud University, France, September 2011.

[BBET10] Marouane Belaoucha, Denis Barthou, Adrien Eliche, and Sid-Ahmed-Ali Touati, *Fadalib: an open source c++ library for fuzzy array dataflow analysis*, Procedia Computer Science **1** (2010), no. 1, 2075–2084.

[BGDR10] Uday Bondhugula, Oktay Günlük, Sanjeeb Dash, and Lakshminarayanan Renganarayanan, *A model for fusion and code motion in an automatic parallelizing compiler*, International Conference on Parallel Architectures and Compilation Techniques (PACT), 2010, pp. 343–352.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan, *A practical automatic polyhedral parallelizer and locality optimizer*, SIGPLAN Not. **43** (2008), no. 6, 101–113.

[BHZ08] Roberto Bagnara, Patricia M Hill, and Enea Zaffanella, *The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems*, Science of Computer Programming **72** (2008), no. 1, 3–21.

[CCH08] Chun Chen, Jacqueline Chame, and Mary Hall, *A framework for composing high-level loop transformations*, Tech. report, USC, 2008.

[CG15] Alessandro Cilardo and Luca Gallo, *Interplay of loop unrolling and multidimensional memory partitioning in HLS*, Proceedings of the 2015

Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015, 2015, pp. 163–168.

[Che12] Chun Chen, *Polyhedra scanning revisited*, Conference on Programming Language Design and Implementation (New York, NY, USA), ACM, 2012, pp. 499–508.

[DMV$^+$08] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick, *Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures*, Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Piscataway, NJ, USA), SC '08, IEEE Press, 2008, pp. 4:1–4:12.

[Fea88] P. Feautrier, *Parametric integer programming*, RAIRO Recherche Opérationnelle **22** (1988), no. 3, 243–268.

[GCT$^+$14] L. Gallo, A. Cilardo, D. Thomas, S. Bayliss, and G.A. Constantinides, *Area implications of memory partitioning for high-level synthesis on fpgas*, Field Programmable Logic and Applications (FPL), 2014 24th International Conference on, Sept 2014, pp. 1–4.

[GGL12] Tobias Grosser, Armin Größlinger, and Christian Lengauer, *Polly – performing polyhedral optimizations on a low-level intermediate representation*, Parallel Processing Letters **22** (2012), no. 04.

[GL96] Martin Griebl and Christian Lengauer, *The loop parallelizer loopo*, Proc. Sixth Workshop on Compilers for Parallel Computers, vol. 21, Citeseer, 1996, pp. 311–320.

[Grö08] A. Größlinger, *Scanning index sets with polynomial bounds using cylindrical algebraic decomposition*, no. MIP-0803, 2008.

[KAI11] Dounia Khaldi, Corinne Ancourt, and François Irigoin, *Towards automatic c programs optimization and parallelization using the pips-pocc integration*, 2011.

[KMP$^+$95] W Kelly, V Maslov, W Pugh, E Rosser, T Shpeisman, and D Wonnacott, *New user interface for petit and other interfaces: user guide*, University of Maryland (1995).

[Loe99] Vincent Loechner, *Polylib: A library for manipulating parameterized polyhedra*, 1999.

[LPC14] Peng Li, Louis-Noël Pouchet, and Jason Cong, *Throughput optimization for high-level synthesis using resource constraints*, 2014.

[MYO$^+$15] Chenyue Meng, Shouyi Yin, Peng Ouyang, Leibo Liu, and Shaojun Wei, *Efficient memory partitioning for parallel data access in multidimensional arrays*, Proceedings of the 52Nd Annual Design Automation Conference (New York, NY, USA), DAC '15, ACM, 2015, pp. 160:1–160:6.

[PF13] Christian Pilato and Fabrizio Ferrandi, *Bambu: A modular framework for the high level synthesis of memory-intensive applications*, 23rd International Conference on Field programmable Logic and Applications, FPL 2013, Porto, Portugal, September 2-4, 2013, 2013, pp. 1–4.

[Pou] L. N. Pouchet, *PolyBench: The polyhedral benchmark suite*.

[SL06] Rachid Seghir and Vincent Loechner, *Memory optimization by counting points in integer transformations of parametric polytopes*, Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (New York, NY, USA), CASES '06, ACM, 2006, pp. 74–82.

[SLLM06] Eric Schweitz, Richard Lethin, Allen Leung, and Benoit Meister, *R-stream: A parametric high level compiler*, Proceedings of HPEC (2006).

[SPJ$^+$09] Jan Sjödin, Sebastian Pop, Harsha Jagasia, Tobias Grosser, and Antoniu Pop, *Design of graphite and the Polyhedral Compilation Package*, GCC Developers' Summit (Montréal, Canada), June 2009.

[TCE$^+$10] Konrad Trifunović, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta, *GRAPHITE two years after: First lessons learned from eal-world polyhedral compilation*, 2nd GCC Research Opportunities Workshop (GROW), 2010.

[VC14] Jasmina Vasiljevic and Paul Chow, *Mpack: Global memory optimization for stream applications in high-level synthesis*, Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays (New York, NY, USA), FPGA '14, ACM, 2014, pp. 233–236.

[Ver10] Sven Verdoolaege, *isl: An integer set library for the polyhedral model*, Mathematical Software (ICMS'10) (Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama, eds.), LNCS 6327, Springer-Verlag, 2010, pp. 299–302.

[VG12] Sven Verdoolaege and Tobias Grosser, *Polyhedral extraction tool*, Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12) (Paris, France), January 2012.

[viv15] *Vivado design suite*, 2015, http://www.xilinx.com/products/design-tools/vivado.html.

[VJC$^+$13] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor, *Polyhedral parallel code generation for CUDA*, ACM Transactions on Architecture and Code Optimization **9** (2013), no. 4, 54:1–54:23.

[WLZ$^+$13] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong, *Memory partitioning for multidimensional arrays in high-level synthesis*, Proceedings of the 50th Annual Design Automation Conference (New York, NY, USA), DAC '13, ACM, 2013, pp. 12:1–12:8.

[xil15] *7 series fpgas overview, product specification*, May 2015, v 1.17.

[YGK$^+$12] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye, *AlphaZ: A system for design space exploration in the polyhedral model*, Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing, 2012.

[ZLC$^+$13] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong, *Improving polyhedral code generation for high-level synthesis*, Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (Piscataway, NJ, USA), CODES+ISSS '13, IEEE Press, 2013, pp. 15:1–15:10.

[ZLL$^+$13] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong, *Improving high level synthesis optimization opportunity through polyhedral transformations*, Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (New York, NY, USA), FPGA '13, ACM, 2013, pp. 9–18.