Architectural Exploration and Scheduling Methods for Coarse Grained Reconfigurable Arrays

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Giovanni Ansaloni

under the supervision of
Prof. Laura Pozzi

September 2011

Dissertation Committee

Prof. Matthias Hauswirth	Università della Svizzera Italiana, Switzerland
Prof. Mariagiovanna Sami	Università della Svizzera Italiana, Switzerland
Prof. Nikil Dutt	University of California - Irvine, USA
Prof. Paolo Ienne	Ecole Polytechnique Fédérale de Lausanne, Switzerland
Prof. Marco Platzner	Universität Paderborn, Germany

Dissertation accepted on 9th September 2011

Prof. Laura Pozzi
Research Advisor
Università della Svizzera Italiana, Switzerland

Prof. Michele Lanza
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Giovanni Ansaloni
Lugano, 9th September 2011

To Lugano, for being there. Twice.

Sará un bel souvenir

L. Ligabue

Abstract

Coarse Grained Reconfigurable Arrays have emerged, in recent years, as promising candidates to realize efficient reconfigurable platforms. CGRAs feature high computational density, flexible routing interconnect and rapid reconfiguration, characteristics that make them well-suited to speed up execution of computational kernels.

A number of designs embodying the CGRA concept have been proposed in literature, most of them presenting specific, ad-hoc solutions. This thesis instead takes a more general approach, focusing on techniques that can be adapted to arrays having different architectural features, enabling experimental comparisons and exploration of the design space.

At the hardware level, a template virtual prototype, named *Expression Grained Reconfigurable Array*, is introduced. Instances can be derived from the template at design time, varying the architecture structure as well as the characteristics of its different blocks. Presented design space explorations include the search of an efficient multi-ALU structure to be used as computational cell and exploration of heterogenous elements to support parallel execution of whole kernels.

EGRA instances are defined at the RTL level. This feature makes it possible to simulate their behavior employing a digital simulation environment. The thesis illustrates a co-simulation framework to evaluate whole applications compiled for EGRA-accelerated systems, taking into account the impact of non-kernel code and reconfiguration overhead.

Success of an architecture paradigm strongly depends on the availability of automated strategies to map applications onto it. The thesis tackles mapping issues at two levels: it proposes a recursive algorithm to partition kernels according to available hardware resources, and it presents a novel modulo scheduling framework, which considers combinatorial chains of computation and routing operations.

The thesis touches the areas of architectural exploration, automated kernel mapping and system integration of CGRAs, providing a quantitative analysis of the efficiency of proposed methods over State of the Art ones. It also outlines a comprehensive hardware/software co-exploration framewok, able to investigate opportunities and pitfalls of coarse grained reconfiguration.

Acknowledgements

Four years of hard work have borne their fruits. One of them is, of course, this thesis, a compendium of the efforts that challenged me along the way. Others are less tangible, but not less precious: the experience I gathered on identifying worthwhile research questions and investigating them is maybe the single most important result of my studies.

During all of the PhD, my advisor Laura Pozzi has been invaluable in leading me by example; for her guidance, and for constantly being present with the right advice, she will always have my gratitude.

Many people have helped me progress on my research path, either through collaborations, informal discussions or by lifting my morale in time of need. I won't try to list them, with the risk of leaving someone out, but to everyone goes my appreciation for making USI-Lugano such a great work environment.

Nikil Dutt and his group at UC Irvine were fundamental in opening new horizons and better my research. Their kindness and their help have been a great gift.

Because the apple never falls far from the tree, I wish to thank Dante and Antonietta.

Finally, no words are enough to thank Simona for her love, and for making my life beautiful.

Contents

Figures

Tables

Chapter 1

Introduction

Reconfigurable (or field-programmable) arrays are modular architectures that can perform execution of applications in a spatial way, much like a fully custom integrated circuit, but retain the flexibility of programmable processors by providing the opportunity of reconfiguration.

Commercially available reconfigurable arrays, like FPGAs (Fiels Programmable Gate Arrays), are designed as a matrix of look-up tables and flip-flops that can be programmed on site to perform different functionalities, dictated by a configuration bitstream. FPGAs have enjoyed growing success as technology scalings made them suited for increasingly complex tasks, and it is now not uncommon for them to embed a whole microcontroller system (System on a Programmable Chip, SOPC). Commercially available FPGAs, proposed by vendors such as Altera [2011] and Xilinx [2011], are today used in a wide range of applications, spanning from simple glue-logic replacement, to digital signal processing, to run-time reconfigurable SOPCs.

The ability to support application-specific features that are not "set in stone" at fabrication time would suggest reconfigurable architectures as good candidates for being integrated in computing systems as customizable accelerators. Nonetheless, the bit-level reconfigurability they present results, for most applications, in a huge performance penalty with respect to fixed implementations.

Coarse Grain Reconfigurable Arrays (CGRAs) narrow the performace gap of fine grained arrays with respect to ASICs by employing coarser basic elements, thus minimizing the overhead due to reconfiguration and the unavoidable difference in efficiency of reprogrammable architectures with respect to single-function ones.

They propose a paradigm shift with respect to FPGAs: instead of considering boolean formulas (in turn representing gates and connections of a digital circuit) and mapping them onto a mesh composed of look-up tables, CGRAs accept as input a Data Flow Graph (DFG) describing *operations* and their dependencies, usually extracted from the single-assignment intermediate representation of a software loop. The application DFG is then mapped on an array of computing elements, embedding one or more ALUs. Figure 1.1 graphically shows similarities and differences between fine and coarse grained reconfigurable arrays.

Research has been fervent in the coarse grained field in recent years; nonetheless many aspects are still to be investigated, partially due to the complex, multi-faceted task of devising both an efficient CGRA structure *and* an automated strategy to map applications on it. The two aspects are tightly intertwined, and the thesis presents contributions on both of them.

The first contribution, described in Chapter 4, is the introduction of a hardware template

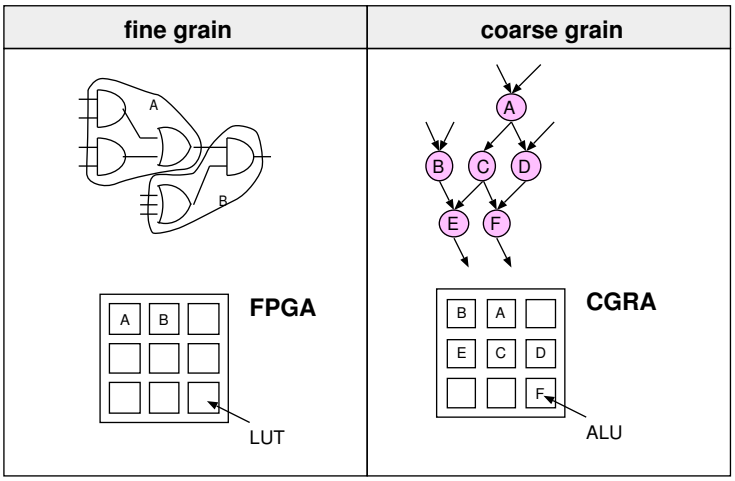


Figure 1.1. CGRAs are inspired by fine grained reconfigurable arrays. They are tailored to map computational kernels, instead of boolean functions.

(the *EGRA, Expression Grained Reconfigurable Array*), developed to investigate the CGRA architectural design space. The chapter reports comparative studies of different CGRA configurations, derived from the template: its first part deals with the evaluation of the performance of diverse computing cells to be employed as CGRA tiles, varying their coarsenss from single ALUs to complex ALU clusters. Entire, heterogeneous meshes and their performance when executing benchmark kernels are then considered. A third study compares different arrangements to support embedded memories in a coarse-grained mesh, using either a multi-ported dedicated memory on the side of the array, or memory cells scattered inside it.

The CGRA concept is especially promising in scenarios where intensive, well defined loops (kernels) dominate execution time[1]. The second thesis contribution, described in Chapter 5, presents a novel methodology to schedule kernels on CGRAs. The technique allows for fully heterogeneous meshes to be targeted; additionally, it supports combinatorial chaining of computation and routing, increasing schedulability and run-time performance of mapped kernels. Also in Chapter 5, a branch-and-bound approach is introduced to partition computational kernels under the tight architectural constraints typically present in the CGRA scenario, so that complex DFGs can be divided into cuts, whose size matches the capabilities of the underlying hardware.

The last thesis contribution, detailed in Chapter 6, investigates system-level performance of platforms embedding a CGRA accelerator. It describes a hardware/software co-simulation framework, instrumental in investigating speedups obtained by CGRA-enabled systems over entire benchmarks. The framework integrates the RTL model of a coarse grained reconfigurable array in a System-on-a-chip (SoC), for which standard C code can be compiled and resulting execution accurately simulated. In this way, whole computing systems can be evaluated when running complete applications, including their non-kernel (non-accelerated) parts, and overheads due to configurations and data transfers.

[1]The notion of whether a loop is "well defined" in this context is not univocal, but always implies that it does not contain recursions and calls to functions that are not in-lined.

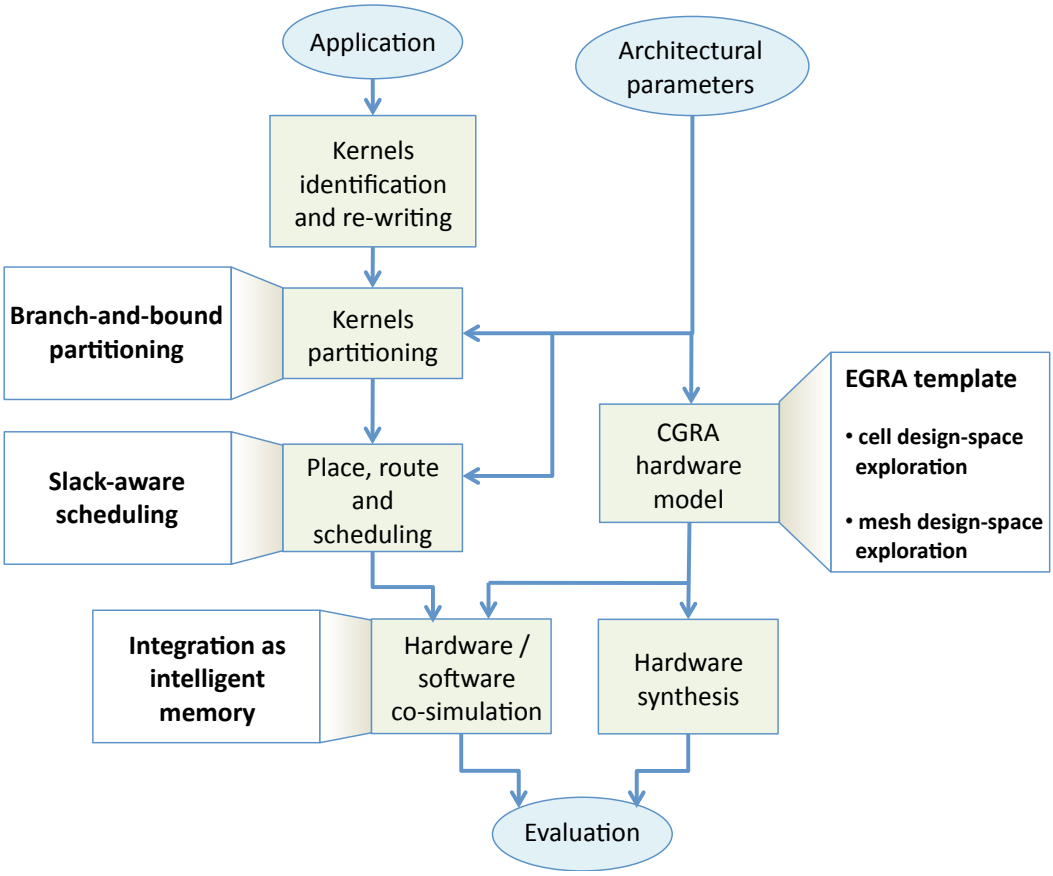


Figure 1.2. Shaded boxes illustrate the conceptual flow of architectural evaluation and application mapping for coarse grained meshes. The thesis proposes novel approaches on multiple aspects in the field (white boxes).

The different challenges addressed by the thesis conceptually belong to a unified landscape; Figure 1.2 illustrates it, highlighting the thesis contributions and their mutual relations. Work exposed in this thesis builds on previous research (Bonzini and Pozzi [2008], Bonzini and Pozzi [2007]) on instruction set extension identification and rewriting. Techniques introduced in these papers are adapted and expanded in the context of CGRAs in Bonzini et al. [2008]; the interested reader can refer to these works for a detailed description of the front-end compilation steps targeting the proposed EGRA template.

Novel research is here introduced and detailed at the compilation back-end level, addressing application partitioning and scheduling, and at the architectural level, leveraging the EGRA template parametric nature. The approach is cross-fertilizing: hardware features, and software techniques exploiting them, can be evaluated when applied to each other, either exploring the efficiency of a compilation framework on a target CGRA platform, or evaluating diverse hardware solutions when executing compiled kernels. This holistic view was essential in driving the research exposed in Chapters 4 - 6.

Content of the thesis is organized as follows: Chapter 2 acknowledges the increasing interest in the research community for the opportunities and challenges of explicitly parallel architectures, a broad field of which coarse grained reconfigurable arrays are part. It justifies why the software-level sequential abstraction has become untenable in recent years, and how different approaches have tackled explicit parallelism, highlighting conceptual differences and similarities. Chapter 3 narrows its focus to the CGRA research field, giving an overview of proposed architectures and application mapping methods, relating research presented in the thesis with the State of the Art.

Chapter 4 details the developed architectural template and related cell- and mesh- level architectural explorations, while Chapter 5 describes novel strategies for kernel partitioning and scheduling. Chapter 6 proposes a hardware/software co-simulation flow for CGRA-accelerated systems evaluation. Finally, concluding remarks are given in Chapter 7, wrapping up the thesis content and suggesting future directions to expand the presented work.

Some of the research strategies and related experimental evidence being part of the thesis have been published in peer-reviewed conference proceedings and journals: the coarse grained architectural template has been the focus of Ansaloni, Bonzini and Pozzi [2008a], Ansaloni et al. [2009] and Ansaloni, Bonzini and Pozzi [2011], while the scheduling framework was introduced in Ansaloni, Tanimura, Pozzi and Dutt [2011]. The CGRA partitioning strategy is a recent development, a related paper describing it has been submitted (Ansaloni and Pozzi [2011]). Also relevant to this thesis are two non-peer reviewed papers: Ansaloni, Bonzini and Pozzi [2008b] and Ansaloni, Najvirt and Pozzi [2008], the latter exploring system simulation issues.

Chapter 2

Background and Motivation: Explicitly Parallel Architectures

2.1 The end of frequency scaling

In the past decades, the performance of sequential computing has increased at an exponential pace, achieving a 100 billion speed-up in a sixty years span. Performance has doubled roughly every 18 months, thanks to technology advancements that made available faster and smaller transistors at each technology node. Increased transistor switching speed increased ICs clock frequency, while greater integration made possible for complex control circuitry to mask execution parallelism at the micro-architectural level, maintaining the illusion of sequential execution at the application level.

Asanovic et al. [2009] argue that, during these years of exponential growth, the major architectural concern was to maintain this abstract sequential interface toward applications above all other considerations. According to Asanovic, increased transistor count and power dissipation were secondary matters, as long as the programming model remained unchanged. Superscalar machines presenting out-of-order execution and speculative execution, prefetching and deep cache hierarchies were common design choices to realize high-performance microprocessors, all in an effort to increased performance while preserving the sequential programming model, even at the expenses of computational efficiency.

The trend was particularly evident in the general purpose domain (comprising CPUs for desktop and laptop computers) but present even in the embedded systems one, as applications evolved from simple controllers to complex multimedia machines. In fact, even if power efficiency is always been an important factor for embedded and mobile appliances, the complexity and computing power of high-end embedded processors has risen sharply in the past years, forcing designers to adopt complex strategies to constrain power and energy consumption.

The sequential programming model could be stretched until two fundamental limits were reached: the power an integrated circuit can dissipate in a practical way, and the ever increasing cost of design and verification of complex digital circuitry. The first limit is often referred to as the "power wall" in literature, while the second is termed the "verification wall". The two walls forced an abrupt change in the way computation is performed by hardware ICs and programmed in software, terminating the race to higher and higher clock frequencies that

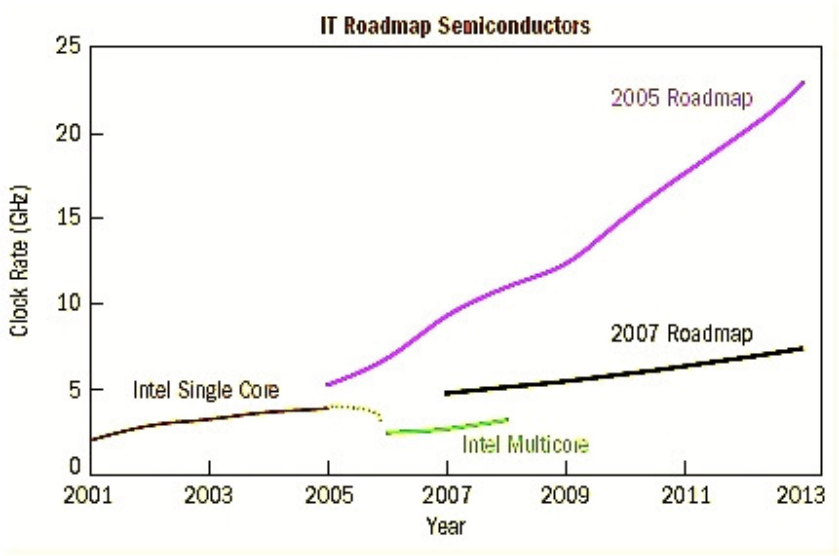


Figure 2.1. Microprocessor clock rates of intel products vs. projects from the international roadmap for semiconductors in 2005 and 2007 (ITRS [2007]).

dominated the industry until the last decade.

Figure 2.1 reflects this shift, plotting the projected microprocessor clock rates of the International Technology Roadmap for Semiconductors in 2005 and in 2007 (data from ITRS [2007]). The 2005 prediction was that clock rates should have exceeded 10GHz in 2008, reaching 15GHz in 2010. Note that Intel products were in 2008 far below even the conservative 2007 prediction.

2.1.1 Power wall

Up until about 2003, exponential increase in processor performance has been fed by constant electric field frequency scaling. In those years, as process geometries scaled downward, the capacitance of transistors also scaled down, in turn driving in the same direction voltage supplies and transistor threshold voltages. These effects combined meant that smaller transistors were invariably faster ones, without sacrificing power consumption.

As manufacturers have shrunk chip features below the 90 nm scale, however, the technique began to reach its limits as thresholds could not be scaled indefinitely without hampering the ability of transistors to block current when in the inactive state, resulting in more and more current leaking through digital circuits, even when no transistor activity is performed.

Leakage currents put a hard limit in reduction of threshold voltages, and this in turn meant the supply voltage could not be reduced correspondingly. Processors soon started hitting the power wall, as adding more transistors to a core gave only incremental improvements in serial performance, while adding up in the total power budget[1].

The issue of power density has become the dominant constraint in the design of new processing elements, and ultimately limits clock frequency growth for future microprocessors. The

[1] Some chip designs, such as the Intel Tejas, were ultimately cancelled due to power consumption issues.

direct result has been a stall in clock frequency that is reflected in the flattening of the performance growth rates *of individual cores* starting in 2002 (as noted by Shalf [2007]). This contributed to a significant course correction in the IT semiconductor roadmap (see again Figure 2.1) reflecting the reduced prominence of single-core performance respect to the overall computing capability of systems comprising multiple, parallel cores.

2.1.2 Verification wall

Technology issues, like current leakage, are one factor forcing the IC industry to embrace explicit parallelism; but design cost considerations are as much as important. In the era of constant electric field frequency scaling, single core designs became increasingly complex, adding hardware control features to hide parallelism below the instruction set: out-of-order execution (and related reordering buffers), sophisticated caching mechanism and other latency-hiding circuitry fueled the increase in core transistor count and the design cost of new ICs.

The number of transistor that can be integrated on a single die and the efficiency in designing them both grew exponentially in the past years, but automation in the design process followed a slower curve than Moore's law, leading to what is commonly known as *design-productivity gap*. Consequence of the gap was skyrocketing cost for new cores (Kahng [2001]), as design and verification efforts made ever more sophisticated single cores a less viable solution to increase performance, leading the way for multi-core designs whose single elements can be independently designed and verified, leaving only the on-chip interconnect study as a global problem.

Finally, as transistors physical dimensions continue to shrink, random defects are becoming more common, due to random process variations. The trend favors systems composed of simple, modular elements over ones composed of bigger and more complex ones, because single points of failure are reduced in the former case, resulting in more resilient ICs whose performance is less impacted by low chip yield.

2.2 Beyond single cores

The power and verification walls force a change in the traditional programming model, the only way out being explicit parallelism: breaking the sequential programming abstraction as was taken for granted in the past decades. It is however not clear what kind of computing architecture should take its place. Considerable effort has been undertaken by the research community as well as commercial vendors to go beyond the single processing core paradigm.

Three trends can be identified:

- Thanks to shrinking fabrication technologies, multiple, fully featured cores can be realized and interconnected on the same die. The strategy is one of incremental evolution of single core designs.
- A more radical approach is the emergence of many-cores. These parallel architectures derive from fixed function Graphic Processing Units, but present flexible computing capabilities. They are increasingly successful in speeding up execution of applications that can be decomposed in a large number of independent threads.

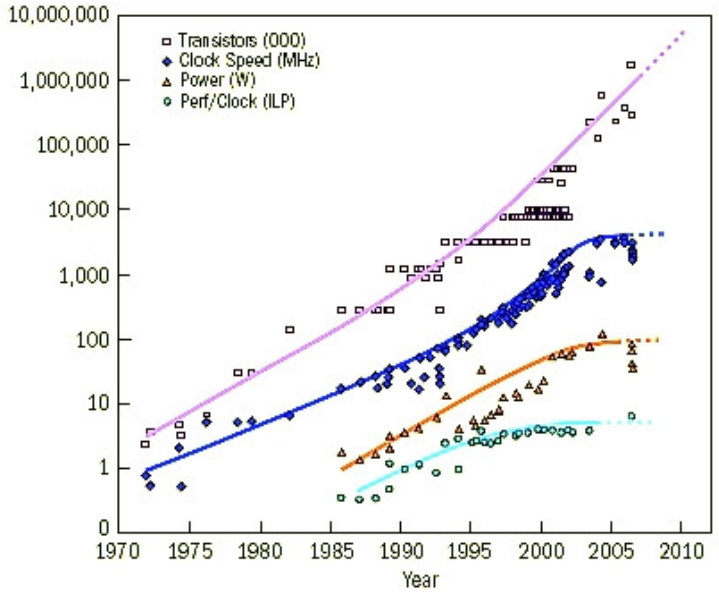


Figure 2.2. Moore's law keeps progressing, as smaller fabrication technologies allow for more and more transistors to be integrated on the same die, but traditional sources of performance improvements (ILP and clock frequencies) have reached a plateau (ITRS [2007]).

- An even more disruptive path derives its roots from reconfigurable architectures such as Field Programmable Gate Arrays (FPGAs). Research and commercial efforts in this field is two-pronged: on one side, FPGA-like structures are embedded inside microprocessors to perform fast bit-width data manipulation in a parallel way. On the other, studies have focused on specialized arrays for speeding up arithmetic computations, maintaining a flexible connection scheme akin to traditional FPGAs, but employing a sea of ALUs (instead of look-up tables) to parallelize execution. ICs realizing this last paradigm are termed *Coarse Grained Reconfigurable Arrays (CGRAs)*.

All three strategies aim at boosting execution performance through parallelization, going beyond micro-architectural Instruction Level Parallelism, which has indeed flattened since 2000 (Figure 2.2). Nonetheless, they achieve parallelization at different levels: multi-core architectures maintain backward compatibility, allowing for legacy serial applications to be run on a single core, and mostly delegating load balancing on multiple cores to the Operating System or leveraging specialized APIs to realize application level parallelism. Multi-cores are well-suited for a general-purpose computing, and this path is indeed followed by major manufacturers of desktop microprocessors, like Intel (Intel Corp. [2006]) and AMD (AMD Corp. [2005]).

Many-core architectures present quite different characteristics. The closest industrial embodiment of the many-core concept is the General Purpose Graphics Processing Unit (GPGPU). It is constituted by hundreds of processing elements, sharing memory space and each executing private threads. GPGPUs are an evolution of fixed-function GPUs, employing programmable processing elements instead of dedicated pipelines. Their increased flexibility (with respect to earlier GPUs) make them attractive solutions for a wide range of massively parallel applica-

tions.

Embedded FPGAs are small reconfigurable meshes, that can be programmed to realize special purpose bit manipulations, either as reprogrammable functional units or as reprogrammable peripherals. Usually, net-lists of functionalities to be implemented are defined through Hardware Description Languages (HDLs) or high level synthesis, and can be changed at run-time, adding flexibility at the hardware level and offering support for realizing dynamic instruction set processors.

Similarly to GPGPUs, CGRAs parallelize computation using distributed hardware resources, but they are organized quite differently. Instead of assigning each thread to a single element, execution of every computing flow on CGRAs is a collaborative effort by many elements, properly interconnected at run-time. Coarse grained arrays adopt the flexible routing network typical of their fine grained siblings (the FPGAs), but at a much coarser granularity, usually employing data channels between four and thirty-two bits wide. Thanks to their coarse grained nature, CGRAs can achieve short reconfiguration times, while the parallel arrangement of their computing units is well suited for spatial execution of computational kernels. These characteristics make them apt to accelerate embedded and Digital Signal Processing (DSP) applications, which are mainly composed of repetitive and computationally intensive loops.

The reminder of this chapter gives some insight on the characteristics of multi-cores (Section 2.3) and GPGPUs (Section 2.4), highlighting the differences in execution model and, consequently, in the programming one. Section 2.5 briefly describes the architectural structure of FPGAs and their evolution.

Coarse Grained Reconfigurable Arrays, as well as their evolution from FPGAs, are introduced in Subsection 2.5.2. The State of the Art in proposed architectures and programming models implementing the Coarse Grained Reconfigurable Array concept, focus of this thesis, is discussed in detail in Chapter 3.

2.3 Multi-cores

Once the power and verification walls put a limit to the complexity of individual cores, major microprocessors manufacturers shifted their focus to the integration of multiple, smaller and power efficient cores on the same die, considering it the only way to take advantage of ever increasing numbers of transistors available on chip and sustain performance scaling.

The frequency scaling trend has been substituted by core count scaling, with doubling the number of processors, or cores, at each technology generation. The incremental path towards *multi-core* chips (two, four, or eight cores) has forced the software industry to expose parallelism to application developers explicitly. Applications developed for multi-cores aim at maximizing performance by distributing their loads on multiple units; legacy code can still be supported on single units, even if their run-time performance is far from optimal.

OS-managed load balancing, in fact, becomes increasingly inefficient as the number of cores scales, because granularity applied at task level is just too coarse to make an efficient use of resources in many cases; consequently, efforts have been undertaken to expose parallelism management to application programmers. Two different approaches have been pursued towards this goal: either based on explicit APIs, or relying on code annotation.

MPI (HP Corp. [2007]) is an implementation of the first strategy; it defines library function to support development of parallel code. It is highly successful in the high-end technical computing community, where clusters of multi-core processors are common, but requires consider-

Figure 2.3. A typical CPU and GPU block scheme: the GPU devotes more transistors to data processing (from NVIDIA Corp. [2010b]).

able effort in deriving efficient implementations when applications must be ported to multiple architectures. Even when the target architecture is fixed, application parallelization is far from trivial, as care must be taken to manage communication and access to shared resources to avoid concurrent programming errors like deadlocks.

Goal of OpenMP (Chapman et al. [2008]) is to assist programmers in the parallelization process allowing for incremental code transformations (starting from a sequential version). It relays on pragma annotations to state which sections should be execute parallely, which data should be private and how execution should be synchronized.

2.4 GPGPUs

Multi-cores break the sequential paradigm exposing explicit parallelism to software, but retain fully featured processing elements, able to execute general purpose code; the approach maintains compatibility with serial applications and does not restrict their generality with respect to single core microprocessors. In multi-cores, core number is restricted by their complexity, as a large amount of control logic makes impossible to effectively lower transistor count per core more than a (rather high) threshold, in turn limiting multi-cores performance when executing applications that expose massive parallelism. Integration of hundreds or thousands of cores on a single die is possible only if dedicated architectures executing special purpose applications are considered; to distinguish them from multi-cores, this family of ICs are termed *many-cores*.

Many-core architectures use much simpler, lower-frequency cores than the fully-featured processors used in multi-cores, resulting in more substantial power and performance benefits. Moreover, the presence of many simple computing units, lacking complex control and caching mechanisms, makes it possible to devote a larger portion of resulting ICs to actual computation than is feasible with multi-cores (Figure 2.3).

In many-cores, hundreds to thousands of computational threads per chip are possible, each executing on a small data set in parallel and allocated to individual cores with little inter-core communication. Long-latency loads and stores to main memory can then be masked by shuffling threads execution on the computing units, instead of deep cache hierarchies. Ultimately, these architectures enable an exponential growth in explicit parallelism, and their performance increase is greatly outpacing general purpose microprocessors, as shown in Figure 2.4.

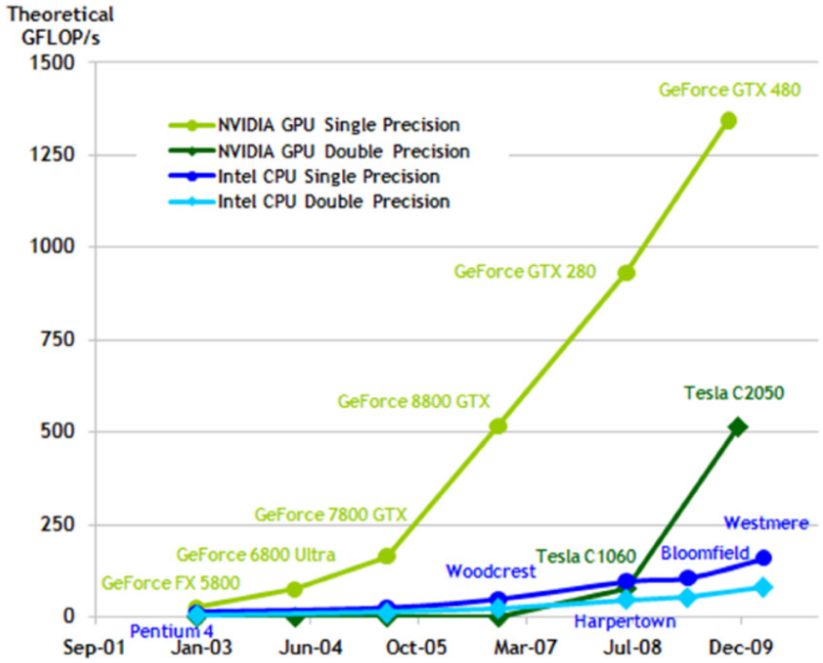


Figure 2.4. Floating-Point Operations per Second for the CPU and GPU (data from NVIDIA Corp. [2010b]).

Obtaining maximum speedup out of an application mapped on a many-core platform requires for application developers knowledge of the underlying architecture. In particular, problems must be properly parallelized and care must be taken to adapt them to many-cores' tricky memory structure, which is explicitly exposed to the programmer and is more akin to software-managed scratchpad memories than traditional caches.

The industrial incarnation of the many-core paradigm is the General Purpose Graphics Processing Unit. GPGPUs evolved from graphic processors in the last decade, when GPU manufacturers replaced fixed custom pipelines of previous generation GPUs with a mesh of more general-purpose processing cores. Whereas traditional GPUs only specialized in drawing an image data to the screen, modern GPGPUs cores can be programmed, using a variant of C Code. The most popular software environment for GPGPU programming are C for CUDA (NVIDIA Corp. [2010a], a proprietary framework developed by NVIDIA) and OpenCL (NVIDIA Corp. [2010b], a collaborative effort managed by the Khronos Group). Both frameworks enable application developers to directly interface with many-core hardware and execute parallel applications on it.

GPGPUs are able to address problems presenting high data parallelism and arithmetic intensity, which is the ratio of arithmetic operations with respect to control and memory ones. Cluster of cores in a GPGPU execute in a SIMD (Single Instruction Multiple Data) fashion, minimizing the required control flow logic. Many applications that process large data sets can use a data-parallel programming model to speed up computations: in traditional GPU applications, like 2D and 3D image processing, large sets of pixels and vertices are mapped to parallel threads; added flexibility in GPGPUs made possible for many algorithms outside the field of image rendering and processing to be parallelized.

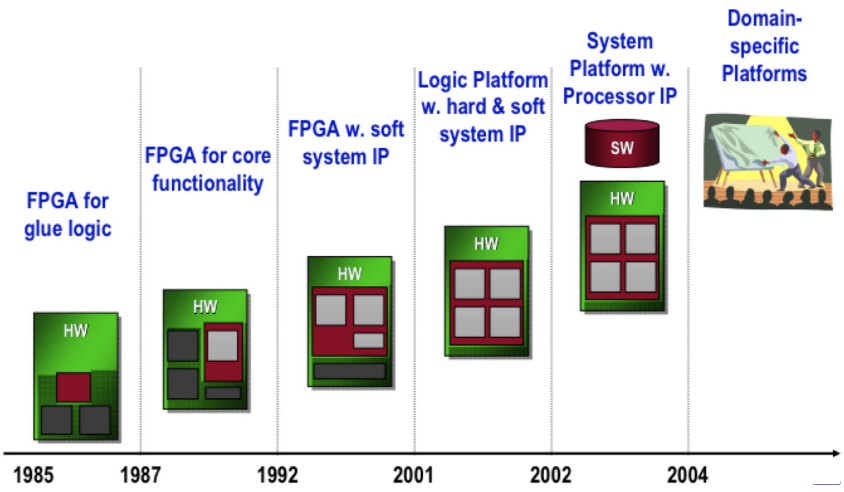


Figure 2.5. Field Programmable Gate Array Evolution (from Xilinx Corp.).

Increasing interest in porting non-graphics applications to GPGPUs is a consequence of the growing gap in peak performance between microprocessors and graphic units. Even if extracting maximum performance from GPGPU hardware requires considerable effort, encouraging results have been reported in accelerating scientific workloads, like molecular dynamics (Elsen et al. [2006]) and Monte Carlo simulations (Preis et al. [2009]).

2.5 Field Programmable Gate Arrays

FPGA architectures are meshes of logic elements, each embedding look-up tables (LUTs) and flip-flops, connected by configurable switch-boxes. The actual boolean function implemented by each LUT and how logic elements are connected with each other determines the desired functionality, as dictated by a program bitstream. Functionalities can be defined by application designers using HDLs or system-level tools like SOPCBuiler (Altera [2010]), and can be changed on the field, or even remotely.

FPGAs have enjoyed a tremendous success in the past decade, as integration allowed for increasingly complex circuits to be mapped onto them. Moreover, specialized units (hard macros) have been added on the side of the sea of logic elements: among them, embedded memories, DSP units and whole microprocessors are the most common.

As illustrated in Figure 2.5, this "bigger and better" evolution have greatly enlarged the possible application fields of FPGAs: historically, reconfigurable chips were first introduced to substitute discrete elements on PCB boards, performing glue logic among other ICs. The availability of greater number of resources have enabled them to implement more complex tasks, like the integration of custom peripherals. Since dedicated hardware blocks have become available, even some performance-critical functionalities, like Digital Signal Processing (DSP), can be efficiently mapped on FPGAs.

Starting around 2002, processors embedded into FPGAs, and the possibility to reconfigure only portions of high-end meshes, made possible for powerful SoCs to be realized on a single chip, tightly integrating software and (reconfigurable) hardware tasks. Today FPGAs present

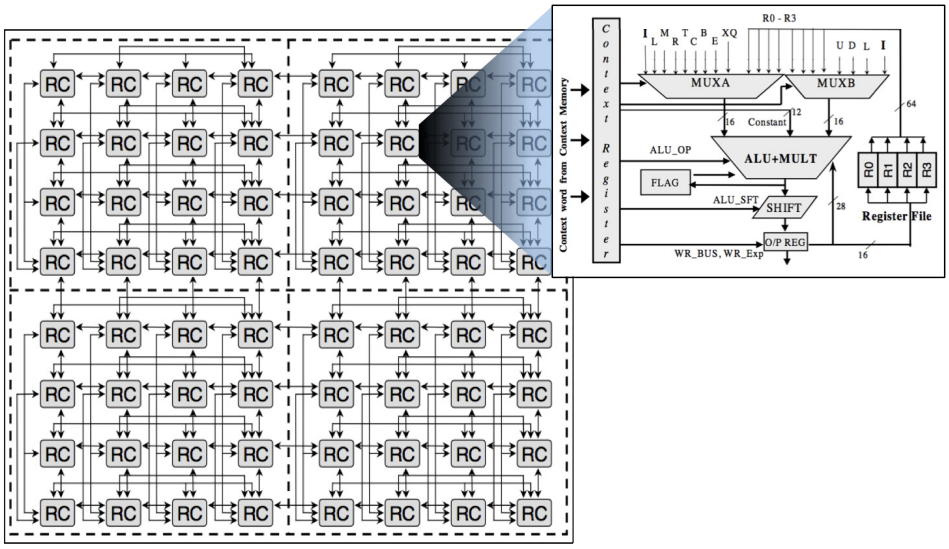


Figure 2.6. Morhosys: an example CGRA mesh Lee et al. [2000].

many complex functionalities, and vendors are targeting different markets with specialized IC families, being them either logic, DSP or memory intensive.

2.5.1 Embedded FPGAs

As commercial FPGAs became bigger and added more capabilities, some research efforts focused instead on small reconfigurable IP blocks, to be encapsulated in strategic positions *inside* traditional SoCs (Wilton and Saleh [2001]). Systems with hardware programmable parts, in fact, can implement flexible peripherals, bus interfaces and/or processing units inside cores, becoming more adaptable to different applications (Magarshack and Paulin [2003]).

Embedded FPGAs (*eFPGAs*) have different constraints with respect to traditional ones: in particular, to maintain performance level comparable to the SoC of which they are part, energy consumption must be minimized, utilizing specialized architectural organizations as the one proposed by Kusse and Rabaey [1998].

Reconfiguration times are also an issue, especially when eFPGAs are employed close to the computational core. In this setting, overlapping computation and configuration to mask overhead becomes mandatory, an example being the Chimaera configurable functional unit introduced by Hauck et al. [1997], which provides eight independently reconfigurable cores able to enrich a host instruction set with custom operations.

2.5.2 CGRAs

While flexibility is the major driver for FPGA adoption in different scenarios, they nonetheless present other desirable features. In particular, their parallel nature makes them good candidates for implementing reconfigurable accelerators; indeed some tasks, relaying on bit-width manipulation (like cryptographic algorithms), present high performance when mapped

Architecture	Application Field	Parallel	Efficient	Evolution	Application Field
Single-core	General Purpose	no	no	Multi-core	General Purpose
GPU	Graphic Processing	yes	yes	Many-core	Parallel Computing
FPGA	Reconfigurable Digital Circuits	yes	no	eFPGA	Embedded Systems (Reconf. IP Macros)
				CGRA	Embedded Systems (Comput. Kernels)

Table 2.1. Comparison of explicitly parallel computing platforms and their evolution

onto them. Other applications, like DSP, can make good use of hardwired macros to increase throughput.

In the general case, however, execution of arithmetic operations on FPGA meshes is quite inefficient, due to their bit-width granularity. This application scenario would be especially interesting to implement parallel execution of small, compact loops characterizing embedded systems applications; to support it, dedicated arithmetic units, small reconfiguration time and streamlined control logic is needed.

Architectures embodying the concept are named Coarse Grained Reconfigurable Arrays. They are composed of meshes of ALUs (instead of FPGAs' logic elements) and provide word- (instead of bit-) level routing. These characteristics have important consequences, as they increase their efficiency when mapping computational kernels derived from software applications. Figure 2.6 provides the architectural block scheme of Morphosys (Lee et al. [2000]), a representative CGRA mesh.

CGRAs can be either considered as VLIW machines employing a large number of functional units and presenting a sparse interconnection scheme, or as special-purpose FPGAs, targeting arithmetic-intensive operations. A third perspective is to regard them as many-core architectures whose units are not assigned one or more thread each, but instead collaborate in the execution of an application using a dynamic local interconnect.

2.6 A conceptual comparison

Multi-cores, many-cores and CGRAs are all resulting from the strive for parallelism that superseded the strive for higher clock frequencies once the power and verification walls were hit. It can be argued that a convergence of quite different types of ICs is undergoing, each family of architectures being inspired from the strong points of the others.

Table 2.1 highlights this evolution: single core microprocessors are well-suited for general purpose computing, but can't exploit massive parallelism. On the other hand, GPUs are very efficient number-crunching architectures when applied to graphics processing, but, until the introduction of more flexible capabilities, were specialized to one particular problem. FPGAs, finally, expose to application developers an extreme degree of flexibility and parallelism. Nonetheless, flexibility comes at a cost, especially in area, run-time performance and power consumption; moreover, FPGA reconfiguration time is often not acceptable if the execution model allows, and the application requires, different functionalities to be shuffled at run time.

To derive more efficient FPGA-like structures, one approach is to reduce their size and embed them inside SoCs as IP macros, taking advantage of FPGAs ability to manipulate data at

the level of individual bits either to increase flexibility at the system periphery or to implement custom functional units at its core. A different strategy is to increase the *coarseness* of basic elements, targeting arithmetic operations (as in many-cores), but retaining a spatial interconnect, so that computational intensive tasks can be mapped and executed in a parallel way utilizing many computation elements at once. This approach has marked the introduction of Coarse Grained Reconfigurable Arrays.

The convergence is conceptual, as each architecture family retains its specificity: while many-cores have greatly expanded the possible applications with respect to GPUs, they still focus on massively parallel applications. In the same way, CGRAs remain special purpose architectures, targeting acceleration of embedded systems applications, where the great majority of execution time is spent on relatively small, compact computational kernels.

CGRAs are novel architectures, and many of their aspects remain to be studied, in particular regarding their architectural features, integration in wider systems and application mapping. The thesis proposes advances over the State of the Art on these three fields.

Chapter 3

Coarse Grained Reconfigurable Arrays: State of the Art

Coarse Grained Reconfigurable Arrays have attracted considerable research in the past two decades, as partially summarized by Hartenstein [2001]. This is due, on one side, to their promise to provide flexibility and efficient computing, making them an interesting solution for accelerating computational kernels in the domains of embedded systems applications and digital signals processing. On the other, on the novel challenges they presented at the architectural and application mapping level.

An implementation of the coarse grained reconfigurable paradigm is multi-faceted. The capability of computational elements composing a CGRA have to be defined and elements have to be connected efficiently. A variety of architectures have been proposed, presenting different characteristics, especially in terms of the *coarseness* of the array: proposed tiles vary from multi-bit look-up tables or single small-width ALUs to fully featured RISC processors or rich multi-ALU clusters. Figure 3.1 provides a conceptual comparison of the architectures described in Section 3.1, considering two dimensions: coarseness of constituting tiles and their heterogeneity, which is the presence of different cell types specialized for different functions.

A hardware platform is only useful as long as a sensible strategy can be envisioned to map applications onto it. Compilation and scheduling for CGRAs are tricky, because high computational density must be aimed for in an environment presenting a sparse connection topology. To address them, ideas in literature are borrowed from distant fields, spanning from mapping for multi-context FPGAs to scheduling for Very Long Instruction Word (VLIW) processors. These efforts are described in Section 3.2.

One last important aspect regards the integration of a CGRA accelerator in a computing system. Well-thought integration is especially important to minimize the overhead due to data transfers between the accelerator and the host and to minimize overhead due to reconfiguration. Solutions proposed by related works are discussed in Section 3.3.

Comparisons between related works and novel concepts investigated in the thesis are briefly introduced here. Thesis contributions are also put in perspective, citing relevant literature, in separate sections of each chapter.

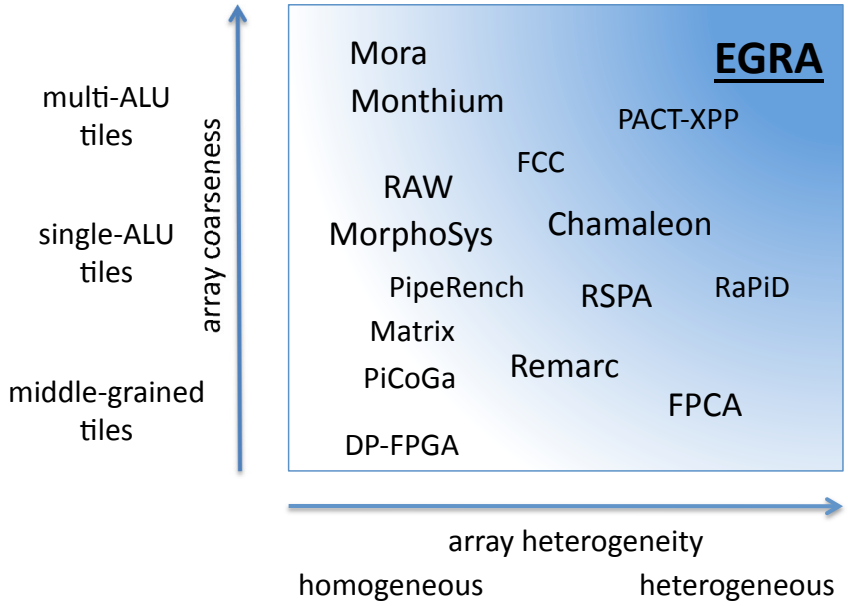


Figure 3.1. Conceptual comparison of different CGRAs, with respect to granularity of tiles and mesh heterogeneity. The EGRA template introduced in Chapter 4 presents a high degree of coarseness and heterogeneity.

3.1 CGRA architectures

3.1.1 Linear arrays

The most apparent characteristic of a coarse grained array is the arrangement of its tiles. While CGRAs draw inspiration from FPGAs, some of the proposed implementations depart from two-dimensional meshes typical of their fine grained siblings, positioning elements in a one-dimenensional linear configuration.

RaPiD (Ebeling et al. [1996]) is composed of a linear array of diverse functional units, implementing ALUs, multipliers and memories. A collection of functional units forms a RaPiD cell, that is replicated sixteen times to constitute the complete array. Units are connected using segmented buses (fourteen are present in the RaPiD-1 prototype) that can be chained and buffered to connect distant functional units. Proper connection of functional units can efficiently implement regular datapaths for DSP applications.

PipeRench (Goldstein et al. [1999]) consists of a reconfigurable datapath that enables execution of many virtual pipelines stages through few physical ones. The function of each physical stage is assigned at each clock cycle according to the desired virtual function by rapidly rotating configurations.

The **FCC** (Flexible Compute Accelerator, proposed by Galanis et al. [2006]) is a linear array of elements allowing for combinatorial chaining of operators. As opposed to the above-mentioned designs, each FCC cell is not composed of a single ALU but by a two-by-two cluster of computing elements, allowing for added flexibility, as a rich set of *expressions*, instead of single operations, can be supported. Employment of complex computing cells is a recurring

theme in CGRA designs; the concept of expression grained reconfigurable arrays is investigated in Chapter 4, using a two-dimensional mesh template.

3.1.2 Mesh arrays

The most used arrangement of tiles in proposed CGRAs is a two-dimensional mesh or thorus, presenting a mix of nearest-neighbor and long-distaince lines, in the form of either point-to-point dedicated connections or shared buses.

One of the first mesh-based CGRA designs is the **Data-Path FPGA** (Cherepacha and Lewis [1996]), which strongly resembles a FPGA array, but employs a four-bit width for routing. By considering groups of bits instead of single ones, the DP-FPGA can reduce the control logic employed in switch-boxes and Look-Up Tables (LUTs), increasing efficiency when regular data-paths are executed on it. The term *middle grained array* is probably more apt to describe this array, given the small bit-width and the use of LUTs in its basic cells.

PiCoGa-III by Campi et al. [2007] also works on four-bit slices. It is inspired by the DP-FPGA, but differentiates itself by employing an ALU and a LUT in each cell in parallel, so that both boolean and arithmetic operations can be efficiently hosted. It relies on fast, hardware managed reconfiguration to achieve high computational density in a small area, using a scheme similar to the one implemented in the proposed EGRA template, which is described in Chapter 6.

Another middle grained example is the **FPCA** (Field Programmable Counter Array, Brisk et al. [2007]), that proposes specialized cells employing compressor trees to speed up execution of arithmetic operations. The design has two distinct areas: one presents standard LUT-based tiles for single bit operations, the other compressor tree cells for arithmetic ones.

Matrix (Mirsky and DeHon [1996]) embeds features of both coarse and fine grained architectures, employing a simple ALU and fine grained logic in every cell. Matrix cells are connected through nearest neighbour connections and horizontal-vertical long lines, in a similar way to the EGRA template.

The majority of CGRAs are instead specialized toward execution of arithmetic operations, not including fine grained support for mapping boolean functions. One example is the **REMARC** array, proposed by Miyamori and Olukotun [1999], which is composed of an eight-by-eight mesh of nanoprocessors, each including a 16-bits ALU and small input, output and data register files. It targets cryptographic and video compression applications.

A similar structure is realized in the **Morphosys** array (Lee et al. [2000]), where reconfigurable cells consisting of a 28-bit ALU, a multiplier and a barrel shifter are embedded in a planar arrangement and connected using a rich point-to-point scheme. Small register files, on the side of each cell, are used to store scalar data generated during execution.

A trend in the evolution of CGRAs has been to enlarge the datapath width and to employ more capable and complex computational elements. In the context of single-ALU cells, the most striking example is **RAW**, introduced by Waingold et al. [1997], which employs fully featured RISC cores connected through a Network-on-a-Chip (NoC), representing a point of contact between CGRAs and multi-core architectures.

Another, complementary evolutionary path increased the number of ALUs in each reconfigurable cell, so that complex *expressions*, instead of single operations, can be executed by CGRA tiles at each clock cycle. In this way, one more degree of flexibility is added to these architectures, retaining a simple control logic and high computational density. This category of CGRAs is well represented by the **Montium** array, introduced by Heysters and Smit [2003], which

presents a complex datapath embedded in each tile. The datapath is specialized to efficiently support DSP applications like signal correlation and filtering.

A more general tile is employed by Lanuzza et al. [2007] in **MORA**, having dedicated logic and arithmetic subsections, and supporting the most common operations utilized in the embedded systems domain. Another example of clustered cells design is **PACT-XPP** (Baumgarte et al. [2003]), which uses an 8-ALU datapath in some elements to support execution of expressions. MORA and PACT-XPP cells represent fixed solutions, in contrast to the parametric Reconfigurable ALU Cluster (RAC), the novel computational cell introduced in this thesis.

3.1.3 Homogeneous and heterogeneous arrays

CGRA designs that were firstly introduced presented a homogeneous structure, with identical elements replicated throughout the mesh. The previously mentioned Remarc and Morphosys arrays are examples of this category. To increase computational density and adaptability, heterogenous CGRAs have been proposed in recent years, breaking away from the concept of a general-purpose cell, able to execute all operations supported by a mesh, and proposing instead specialized ones. In heterogenous CGRAs, operations that statistically are more frequently executed can take advantage of more hardware support than ones that are seldom recurring.

Specialized ALU and multiplier cells are the most common example: **Chameleon** (Tang et al. [2000]) utilizes clusters of cells, each one having seven ALUs and two multipliers and forming, in their parlance, a tile that is then replicated to form a 108-cells array. A different layout is considered by Kim et al. [2005]. In their work, the described **RSPA** mesh presents multipliers external to the reconfigurable mesh, shared by the reconfigurable tiles as needed by the application.

PACT-XPP also consideres cells with different memory and computation capabilities. Scattering memory cells *inside* the reconfigurable array leads to higher bandwidth with a smaller area footprint than it is possible utilizing an external memory. The thesis further investigates the concept in Chapter 4; the EGRA template described there completely decouples storage and computing support by employing dedicated type of tiles inside EGRA instances.

3.1.4 CGRA design space and its exploration

Previously illustrated designs leverage empirical engineering expertise to devise efficient solutions for intended application fields. As CGRAs diversify and become more complex, possibly employing clustered cells and/or heterogenous elements, the need arises for quantitative evaluation of architectural choices.

One notable effort in this field is the paper of Bouwens et al. [2007], exploring different interconnection topologies and register files arrangements for the **ADRES** reconfigurable array. Results show that a moderately rich interconnection among cells and small local register files is the most performing setting for executing a number of benchmark applications.

A CGRA modeling framework has been presented by Chattopadhyay et al. [2008]. Starting by a user-defined high level description, the framework can model tiled architectures with possibly heterogeneous elements inside tiles. An HDL implementation can be derived and synthesized to enable performance comparisons.

On the other side of the spectrum, the **Morpheus** integrated circuit, developed by Kuhnle et al. [2008], presents a physical platform comprising reconfigurable IP elements with different granularities: a fine-grained embedded FPGA, a middle grained PiCoGa and a PACT-XPP, all

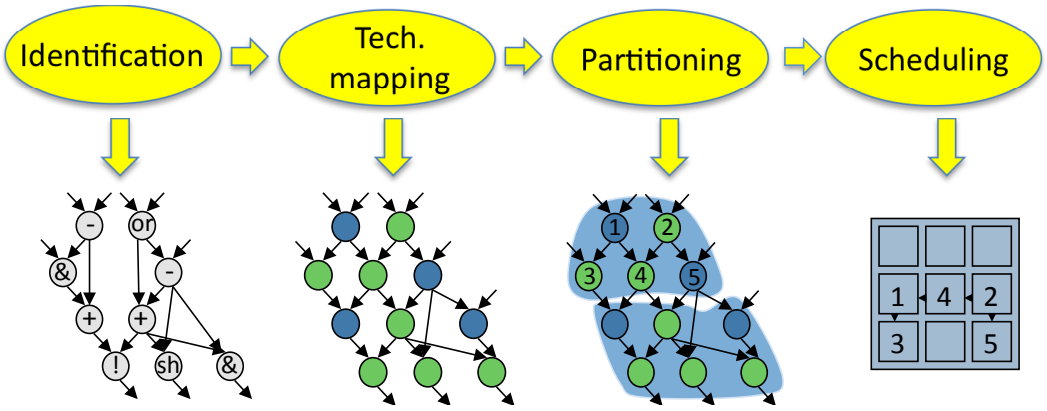


Figure 3.2. Application mapping on CGRAs: kernel identification, technology mapping, partitioning, scheduling.

instantiated on the same chip and communicating with a Network-on-Chip. The platform enables rapid comparative evaluations of these three technologies.

The major difficulty of developing high-quality design exploration tools is to envision hardware templates that are flexible enough to mimic a wide variety of possible structures. In exploring the performance of complex CGRA cells presented in Chapter 4, we took our inspiration from the parametric **Custom Compute Accelerator** introduced by Clark et al. [2005]. The CCA is intended as a stand-alone functional unit; in the thesis a similar circuit was instead used as a replicable element in a coarse grained mesh.

3.2 Computational kernels processing and scheduling

As illustrated in Figure 3.2, mapping of kernels on coarse grained array requires multiple compilation steps. Kernels have to be extracted from applications and expressed as Data Flow Graphs (DFGs), using some sort of intermediate representation. DFGs have then to be transformed, to adapt them to the technological features of a specific hardware platform, and partitioned so that each part doesn't overuse hardware resources. Finally, operations must be associated with computational elements and routing must be performed. This section presents research efforts investigating each of these tasks; related advances over the State of the Art introduced by the thesis are described in Chapter 5.

3.2.1 Identification and technology mapping

First step in scheduling an application on a CGRAs is to *identify* its intensive loops (*kernels*), whose execution is to be performed by a reconfigurable mesh. Kernels have to be well formed: they must not present recursion, irregular exits and function calls that cannot be inlined. Most often, only loops whose iteration count is constant-bound, such as "for" loops, are supported, in order to minimize the amount of hardware control logic.

While these constraints somehow restrict the applicability of application acceleration using CGRAs, they are usually satisfied in the DSP domain and in the embedded systems field. One

further restriction is that only innermost loops are usually considered (as exemplified in the paper by Mei et al. [2002]); standard loop transformations such as loop unrolling can be applied to increase the number of nodes of a data flow graph up to a desired size.

Identification of application sections to be accelerated has been the target of various research efforts in the instruction set extension field, where tight micro-architectural constraints are present, such as input-output number and forbidden operations. Exact and approximate algorithms to solve this problem has been proposed by Pozzi et al. [2006]. Kernels identification for CGRAs is a simpler problem, because possible candidates are restricted to loops. Identification is then either done using manual code annotation, as in the **PACT-VC** compiler by Cardoso and Weinhardt [2002b], or every innermost loop is considered for CGRA execution. This last strategy is pursued by the **DRESC** compiler for the ADRES array (Mei et al. [2002]).

Technology mapping is the process of transforming kernels to express them in terms of features supported by the coarse grained mesh. This pass is crucial to both make a kernel schedulable and to increase its execution performance. Technology mapping transformations are usually performed at the intermediate representation (IR) level, such as the single assignment **lcode** employed by DRESC (Chang et al. [1991]). Using a proper IR, branches present in a kernel, e.g. if-than-else statements, can be converted into multiplexer operations, supported by the vast majority of CGRAs.

The **FELIX** framework, developed by Morra et al. [2005], uses a term rewriting technique, inspired by the model checking field, to derive alternative, more efficient implementations of a given kernel. The above-mentioned PACT-VC compiler performs loop unrolling and vectorization to increase a kernel size; limited support for kernel splitting via loop disserving is also supported.

Bonzini et al. [2008] presents a set of ten graph rewriting rules to adapt the GCC intermediate representation of loop bodies to EGRA instances features. Clustered cells, such as the ones present in the EGRA template, pose an additional challenge to technology mapping, which is to appropriately group operations into expressions that can be executed by a multi-ALU cell, maximizing utilization. In the same paper, a solution to this problem, using a branch-and-bound algorithm, is also proposed. Later chapters of the thesis assumes the availability of properly identified and technology mapped data flow graph representations of computational kernels; the interested reader can refer to the work of Bonzini for a description of these steps in the EGRA scenario.

3.2.2 Partitioning

CGRAs attractiveness resides in their computational efficiency, in terms of throughput per unit time, unit area and/or unit power. To achieve high efficiency, coarse grained meshes employ simple support logic and small internal memories, limiting the size and complexity of data flow graphs that can be mapped onto them. Kernels exceeding the constraints imposed by a given CGRA architecture must than be *partitioned* into sub-kernels to obey such constraints.

Techniques to perform partitioning include the approach illustrated by Kaul and Vemuri [1998], which assumes applications that are naturally divided into tasks that can be grouped to form partitions. Partitioning is then formalized as a non-linear programming problem. Given the exponential complexity of the formulation, solutions are provided up to a limited number of tasks and partitions.

The Kernighan-Lin network flow algorithm (Kernighan and Lin [1978]) provides an efficient heuristic to accomplish graph partitioning, but only considers undirected graphs. Its

extension to acyclic data flow graphs is presented by Liu and Wong [1998]. The implementation, intended for circuit mapping on time-multiplexed FPGAs, performs subsequent graph min-cuts until constraints are met.

Purna and Bhatia [1999] introduce two heuristic algorithms for partitioning kernels: a level-based approach groups together nodes presenting the same depth[1], while a cluster-based one tends instead to group together nodes with common predecessors. The first methodology aims at maximizing parallelism regardless of bandwidth requirements, the second tries to strike a balance between achieved parallelism and memory footprint size. Nodes are assigned in a single DFG traversal, achieving linear convergence time but also solutions distant from optimality in many cases.

Binary recursion over abstract search trees, illustrated in Pozzi et al. [2006] in the context of instruction set extension identification, is a promising methodology to perform partitioning, too. Appropriate recursion bounding can limit algorithm run-time, and the partitioning can be performed either exactly or, more efficiently, in a locally exact way. Chapter 5 presents and discusses such implementation targeted to CGRA partitioning, formalizing algorithms and related constraints.

3.2.3 Scheduling

Data flow graph mapping on CGRAs presents similarities both with FPGA place and route and VLIW compilation. As in FPGAs, functional units are spatially distributed, routing resources are limited, and tiles can be used either for computation or as pass-through to route data. As in VLIWs, execution is dynamic at run time, with cells performing different operations in different clock cycles.

Some mapping techniques only consider spatial placement of operations on a reconfigurable mesh. Lee et al. [2003] propose decomposing the mapping problem in row placement and mesh placement, assuming dedicated connections to a multi-ported memory bank for each row of CGRA elements. In the first phase, clusters of operations that can be mapped in a single CGRA row are identified, while in the second different clusters are properly assigned to different rows. In this way, dimensions of the problem are reduced from two to one, resulting in fast convergence.

The papers by Ahn et al. [2006] and Kim et al. [2005] expand on the previous work by including support for shared resources performing heavy computation, like multipliers. In the targeted RSPA, pipelined multipliers are shared among tiles belonging to the same row, the added constraint being managed by the clustering step. Lee, Ahn and Kim works are restricted to CGRAs composed of identical rows.

The ability of CGRA tiles to perform both computation and routing is addressed by Yoon et al. [2008]. The split-push kernel mapping scheme employed in this work considers routing nodes as well as operation nodes when mapping a DFG on a mesh; multiple routing nodes can be collapsed on the same cell if they connect operations placed at a short distance from each other. Investigated mappings on a homogeneous mesh are of comparable quality with respect to the ones obtained using an exact ILP formulation.

Other research efforts consider both spatial and temporal dimensions. Goal of these works is to take advantage of parallelism present on a CGRA mesh by partially overlapping execution of different iterations of a kernel. The technique, called modulo scheduling (illustrated by Rau

[1] Depth of a node is its distance to the furthest node without predecessors, as defined by Rau [1996]

[1996]), is an established compiler optimization originally devised for VLIW architectures. Research in the field assumes registered connections between cells, so that computing or passing through them always takes one clock cycle. Improvements obtained by relaxing this constraint are present in Chapter 5, targeting fully heterogenous EGRA instances.

Mei et al. [2003] adapted modulo scheduling to CGRAs by representing the software application and the hardware architecture (replicated for each time step) as directed graphs. Simulated annealing iterations are used to converge to a solution from an initial, possibly invalid, DFG placement. The employed detailed architecture representation leads to lengthy compilation time; the issue is addressed by Hatanaka and Bagherzadeh [2007], that simplify the hardware representation decoupling resource reservation and scheduling. An additional feature introduced in their work is the support for heterogeneous reconfigurable arrays and shared communication resources.

Park et al. [2006] propose a heuristic, called modulo graph embedding, inspired by graph layout. By placing nodes greedily, modulo graph embedding is able to speedily schedule complex DFGs onto homogenous meshes. The scheduler places nodes in order of increasing depth, trying to minimize both distance to predecessors and distance to nodes with common successors. Scheduling slots in the leftmost row are utilized first, and the scheduling space is appropriately skewed to acknowledge non-utilized resources at each scheduling pass.

In most modulo scheduling adaptations to the CGRA scenario operation placement is done before data routing. The methodology illustrated by Park et al. [2008], instead, promotes routing as the major scheduling concern. The edge-centric approach exposed in their work route DFG connections on a scheduling space until all inputs of an operation are available in one cell, which is the position (in space and time) where the operation is executed.

All the above-mentioned works focus on maximizing utilization of computing resources during scheduling. A different perspective is described in a recent work authored by Kim et al. [2010], which assumes a multi-banked memory on the side of computing resources. The resulting scheduler tends to place operations accessing the same data on cells sharing the same memory bank connection, thus reducing data duplication. Memory constraints are also considered in the research illustrated in Chapter 5, in the different scenario of CGRA meshes embedding memory cells.

3.3 System Integration

CGRAs are almost never conceived as stand-alone units. The necessity then arises to interface them efficiently with various other components in a SoC, especially with a general purpose host. The execution model is one where a microprocessor acts as master, on one side executing non-kernel code, on the other managing configuration of the CGRA and transfer of datasets of kernels to and from the accelerator.

Integrating a CGRA in a system is a more challenging that integrating a single-core Reconfigurable Functional Unit. RFUs, in fact, execute small code segments, usually the size of a basic block, as exemplified by the CCA (Clark et al. [2005]) and Chimaera (Ye et al. [2000]) architectures. In this context, granting access to the host register file usually suffice to avoid a bandwidth bottleneck; nevertheless addition of a small number of shadow registers private to the RFU have been shown to be beneficial by Cong et al. [2005] to support complex instruction set extensions.

Some coarse grained arrays are also interfaced at the register file level. Bouwens et al.

[2007] utilize a clever dual VLIW-CGRA view to execute kernel and non-kernel code on the ADRES architecture. In their layout, the first row of functional units can be used as a VLIW machine, while the whole eight-by-eight mesh is activated to parallelize execution of computational kernels.

A more general solution decouples the host processor from the reconfigurable array by employing dedicated memory buffers. PiCoGa, developed by Campi et al. [2007], presents multiple memory banks to store processed arrays and dedicated storage for fast access to scalar variables; individual PiCoGa tiles can access data through a programmable crossbar interconnect. A similar solution is proposed in Morphosys (Lee et al. [2000]), which embeds a frame buffer divided in two banks to allow for overlapping of computation, executed by the reconfigurable array, and data transfers, managed by a DMA unit.

In many embedded systems applications data is sequenced serially. To leverage this characteristic, the **Zippy** array introduced by Plessl and Platzner [2005] employes FIFO queues (instead of RAM blocks) to provide input and store output data, and a bus interface to communicate with the host. FIFOs require simpler control logic with respect to RAMs, resulting in a streamlined implementation.

A different research direction investigates Networks-on-a-Chip (NoCs) as a viable interconnection solution in a CGRA context. Research has been undertaken to integrate them both for local communication inside a reconfigurable array and for data transfers to external SoC elements. The **Tartan** architecture (Mishra and Goldstein [2007]) is organized in a hierarchical way, having switch-boxes managing low-level interconnects and NoCs routers deployed to steer global routing. The Morpheus chip, designed by Kuhnle et al. [2008], uses instead a NoC to connected the host and multiple reconfigurable elements presenting different flexibility/efficiency tradeoffs, from an embedded FPGA to a coarse grained array.

Works previously illustrated in this section are computation-centric, their focus being on integration of host and CGRA computation capabilities. Proposed Intelligent Memory solutions (like the **IRAM** described in Patterson et al. [1997]) have reversed this paradigm, adopting a memory-centric approach. Works on intelligent memories investigate opportunities offered by embedding computational elements inside RAM banks, allowing for data manipulations with limited main processor interference.

Belonging to this field is also **FlexRAM** (Kang et al. [1999]). A FlexRAM chip includes intelligent elements at different levels: many small integer engines and a single low issue superscalar microprocessor are embedded in each memory bank, while a conventional cache hierarchy and a host processor is provided for non-accelerated portions of applications. A simple ring network is adopted for inter-chip communication.

Given the distributed nature of processing on intelligent memories and the intensive, repetitive tasks associated with in-memory computations, reconfigurable hardware appears to be a good candidate to implement data manipulation inside memory banks. **ActivePages** (Oskin et al. [1998]) investigated the concept by enriching a standard DRAM chip with small embedded FPGAs connected to each memory bank. Authors report considerable speedups and small area overhead due to the reconfigurable fabric.

Interfacing CGRAs as intelligent memories is an attractive solution, given the fast reconfiguration times and the computing densities attainable by coarse grained arrays. Chapter 6 explores this opportunity, introducing a hardware/software co-simulation framework for evaluating execution of whole applications on systems comprising a coarse-grained reconfigurable accelerator.

Chapter 4

The EGRA template: CGRA Architectural Design Space Exploration

4.1 Introduction

Reconfigurable (or field-programmable) arrays are flexible architectures that can perform execution of applications in a *spatial* way—much like a fully-custom integrated circuit—but retain the flexibility of *programmable* processors by providing the opportunity of reconfiguration.

These features would suggest reconfigurable architectures as particularly good candidates for being integrated in customizable processors. Unfortunately, other drawbacks have kept reconfigurable arrays from becoming a largely adopted solution in this field. Among different factors, the performance and area gap that still exists between reconfigurable and hardwired logic is certainly one of the most important. The problem of bridging this gap has been the focus of much research in the last decades, and important advances have been made. Research exposed in this chapter goes in the direction of decreasing such gap further.

A walk through related historical background will help stating this chapter's aims and contributions. In the earliest examples of reconfigurable architectures such as the PLA (Programmable Logic Array), mapping of "applications" (boolean formulas in sum-of-product form) is immediate. In fact, each gate in the application is mapped in a *1-to-1* fashion onto a single gate of the architecture (Figure 4.1a).

However, this organization does not scale as applications to be mapped get more complex. For this reason, CPLDs and FPGAs instead use elementary components—PLAs themselves, or look up tables—as building blocks, and glue them with a flexible interconnection network. Then, programming one cell corresponds to identifying a *cluster of gates* in the boolean function representation (Figure 4.1b). Introducing this additional level is a winning architectural choice in terms of both area and delay.

An orthogonal step was the introduction of higher granularity cells (Figure 4.1c). Fine grain architectures provide high flexibility, but also high inefficiency if input applications can be expressed at a level coarser than boolean (e.g. as 32-bit arithmetic operations). Coarse Grain Reconfigurable Arrays provide larger elementary blocks that can implement such applications

more efficiently, without undergoing gate-level mapping.

A variety of CGRA architectures exist (Chapter 3 proposes a survey of the field), but the process of mapping applications to current CGRAs is usually not very sophisticated: a single node in the application intermediate representation gets mapped onto a single cell in the array (again, *1-to-1* mapping). Instead, the architecture described in this chapter (Figure 4.1d) employs an array of cells consisting of a group of ALUs with customizable capabilities. We consider this the equivalent of the evolution from single-gate cells to LUT-based ones observed in the fine grain domain.

This complex, coarse grained cell is termed *RAC (Reconfigurable ALU Cluster)*, and the architecture that embeds it *EGRA (Expression Grained Reconfigurable Array)*. The first idea explored in this paper is therefore a mesh of RACs, as shown in Figure 4.1d. This is described and evaluated in Section 4.3.

As a second step we consider the EGRA at the architecture level and focus our attention on the mesh. Again we make a parallel with historical background through Figure 4.2. Reconfigurable architectures' design evolved at the mesh level also, as FPGA meshes changed from a homogenous sea of logic elements (as in Figure 4.2a) to complex heterogenous arrays, embedding dedicated hardwired blocks for memory storage and for specific computations (as in Figure 4.2b). The presence of dedicated memory and computational blocks boost FPGAs performance for certain classes of applications, like digital signal processing. Heterogeneity, together with technology scaling, has been instrumental in expanding FPGA application fields form simple implementations of custom peripherals to whole systems on a programmable chip.

Moreover, including local memory elements into custom functional units has been shown to be beneficial by previous studies on automatic identification of instruction-set extensions (Biswas et al. [2006]). Starting from these assumptions, we improve on the homogeneus EGRA instance studied in Section 4.3 and represented in Figure 4.2d, investigating the benefits of integrated storage and multiplication units inside the EGRA (Figure 4.2e). This is studied and evaluated in Section 4.4.

Heterogeneity indeed introduces non-obvious benefits in the coarse grained scenario: if a reconfigurable mesh is able to execute all operations in a loop (memory loads, arithmetic calculations, and memory stores) it is possible to go beyond the customizable processor model, and to offload execution control to the accelerator in order to aggressively pipeline the loop, an execution paradigm first proposed by Mei et al. [2002]. Therefore, still in Section 4.4, we outline how EGRA instances can efficiently support the execution of modulo-scheduled loops.

As a third step we explore memory interfaces. Local memory can be interfaced in a variety of configurations, and different solutions have been proposed in CGRA research project as well as commercial FPGAs, sometimes with different solutions coexisting on the same architecture (as in Figure 4.2c). We make the same step in EGRA evolution (Figure 4.2f) and study the EGRA memory interface in Section 4.5.

The three studies carried out in this chapter (cell level, heterogeneus architecture level, and heterogeneous memory interface level) have in common a strong focus on design space exploration. In order to understand the tradeoffs involved at the three design levels we don't propose a single array implementation that are solely the result of the designer's expertise; rather, the EGRA enables a systematic and comparative study, by virtue of it being a *template* from which a number of architectural instances can be derived—either generic, or tailored to implement the computational kernels of a particular benchmark. The case studies presented throughout the chapter use this approach to show how scratchpad memories, arithmetic clusters and multipliers offer together enough flexibility to implement a wide variety of kernels.

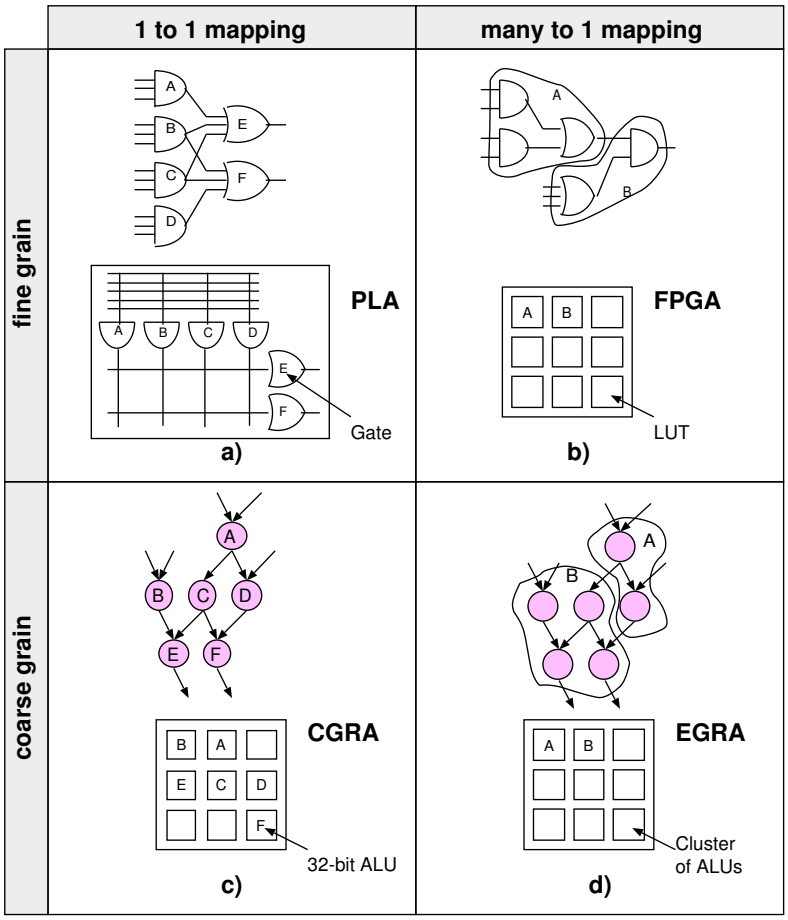


Figure 4.1. Parallel between the evolution of fine grained architectures' cells from simple gates to FPGAs' LUTs (a and b), and the evolution of CGRAs' cells from single ALUs to RACs, proposed here (c and d).

To sum up, in this chapter a new architecture, the EGRA, is introduced; its design is investigated at the cell, array, and memory interface level through evaluation and comparison of several different instances of the template. Ultimately, this work can then be seen as an explorative step towards the design of an efficient, compact accelerator, that can be reconfigured to suit a wide range of applications.

Presented contributions are:

- A new design for the combinational part of coarse grain reconfigurable arrays is envisioned. The Reconfigurable ALU Cluster, the complex cell at the heart the EGRA's arithmetic, supports efficient computation of entire subexpressions, as opposed to single operations. In addition, RACs can generate and evaluate branch conditions and be connected either in a combinational or a sequential mode.

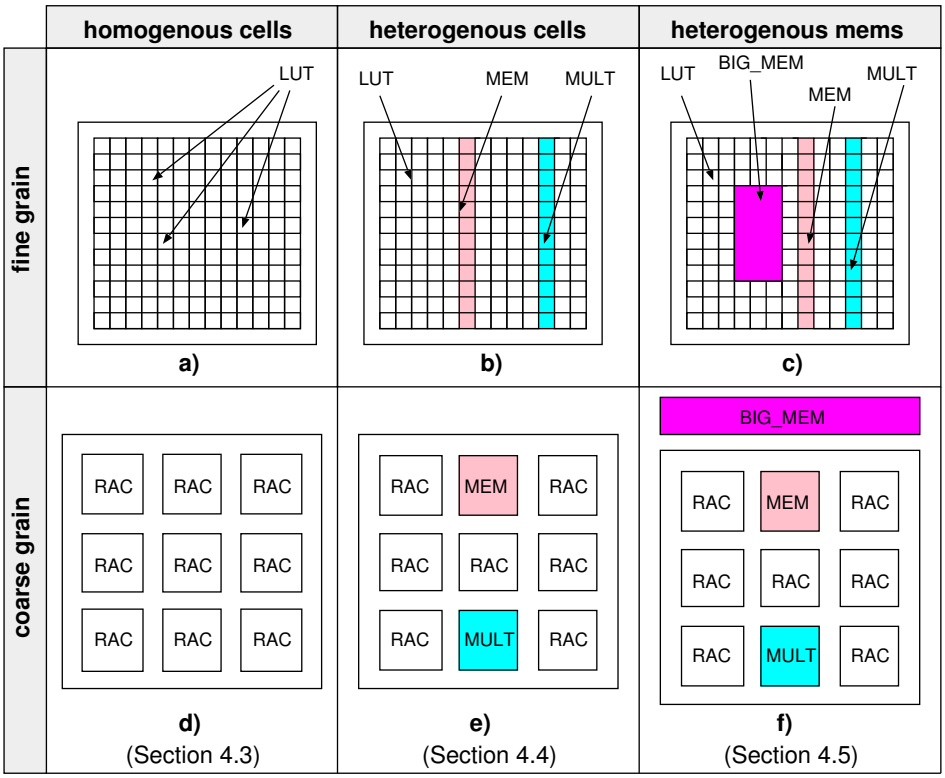


Figure 4.2. Parallel between the evolution of fine grained architectures' meshes from homogenous to heterogenous (a, b and c), and the evolution of CGRAs' meshes to the EGRA proposed here (d, e and f).

- A template architecture for heterogeneous CGRA design is described, accommodating different types of computational element and local storage of data, and able to execute modulo scheduled computational kernels. EGRA instances can be derived to perform different design space explorations: it is applied here at three different level, to investigate clustered computational cells, heterogeneous meshes and memory interfaces.

The remainder of this chapter is structured as follows. Section 4.2 relates the presented content with related works; Section 4.3 details the structure of the EGRA's RAC cell and presents synthesis results for different instances of the architecture; Section 4.4 introduces the heterogeneous array, describing the different types of cells employed and how they operate to execute pipelined computational kernels; Section 4.5 explores different interfaces for local storage of data, giving trade-offs between complexity of the memory structure and achieved speed-up of applications.

4.2 Related work

In the past years, several CGRAs have been proposed (Hartenstein [2001]). The definition is broad, and includes designs that differ widely even in the very "coarseness" of the cell. For example, the cell will usually implement a single execution stage, but may also include an entire execution unit (Rapid, Fisher et al. [2001]) or can even be a general purpose processor (RAW, Waingold et al. [1997]).

The work that most resembles the proposed RAC structure is probably the Flexible Computational Component proposed by Galanis et al. [2006] which, while targeted more specifically to DSP kernels, is similar to the RAC in size and set of allowed operations. However, the authors do not present an exploration methodology to explain quantitively their choices.

ADRES (Mei et al. [2002], Mei et al. [2004]) also features a complex VLIW cell. Even though it lacks the ability to perform multi-stage computations within a cell, it features strong instruction-level parallelism capabilities that can be exploited by the compiler through software pipelining.

The architectural choices that drove the above-mentioned designs are usually the result of the designer's expertise, more than of systematic, quantitative exploration of the design space. Therefore, the resulting designs have a fixed structure. Even when some flexibility is present (as in ADRES or Rapid), results for exploration are presented only for high-level cycle-base exploration, or not given at all. The work of Bouwens et al. [2007] is somehow an exception as it demonstrates design space exploration at the synthesis level for the ADRES architecture. However, it focuses on CGRA high level topology, without investigating the structure and coarseness of the processing elements as done here.

The RAC cell design is inspired by the *Configurable Computation Accelerator* structure (CCA) proposed by Clark et al. [2004]. The CCA is used as a stand-alone accelerator, while RACs are replicated in a mesh structure. Also, RACs introduce several other novel aspects, such as the ability to build combinational functions from multiple expressions and support for if-conversion. For the latter, we use a peculiar "flag" design that is inspired by the program status word of a microprocessor and more powerful than the carry chains available in many reconfigurable architectures (e.g.: Stretch, Rupp [2003]).

Most CGRAs (as, for example, Morphosys introduced by Lee et al. [2000] and MORA by Lanuzza et al. [2007]) present a homogeneous structure, with identical processing elements replicated in a planar arrangement; nonetheless some works did investigate heterogeneous CGRAs: Mei et al. [2005] present some results over multipliers placement in the ADRES architecture, while PACT-XPP (Baumgarte et al. [2003]) employs two types of cells with different storage and computation capabilities. EGRA instances differs from the above-mentioned architectures by decoupling groups of ALU operations from multiplications and by separately supporting computation and memory on different cells of the reconfigurable mesh. The approach leads to a better equalization of critical paths over different elements and to a flexible design space.

A common trait of reconfigurable architectures is the presence of local storage connected to logic or computational elements by high-bandwidth lines. StratixII FPGAs from Altera embed different sized memories to support different requirements, while in the coarse grained domain a wide range of solutions have been proposed: the memory component can be a small general-purpose scratchpad, as in DREAM (Campi et al. [2007]) and ADRES (Mei et al. [2004]), or a buffer interface sitting between the computational cells and the system RAM, as in MorphoSys, which also implements a small register file local to every cell.

The EGRA template differs from DREAM in that scratchpad memories can be scattered around the array, in the form of specialized memory cells. By giving the possibility to distribute the memories on the array, their I/O requirements (number of ports) can be tightly limited. Furthermore, the array interconnect can be reused as a data/address bus, removing the need for a separate connection between the computational and storage elements of the array.

The proposed EGRA template can store values at three hierarchical levels: inside the cells, in specialized memory cells and in a multi-ported scratchpad outside the mesh; this flexibility allows for data to be stored and retrieved efficiently and for novel comparative studies, like the one presented in Section 4.5, to be performed.

Effective pipelining of the loop, enabling different parts of multiple iterations to execute simultaneously, is key to extract performance from CGRAs. EGRA instances explicitly support modulo scheduling (Rau [1996]), a software pipelining technique apt at coping with resource constraints, such as the number of I/O ports on a scratchpad memory or the number of multipliers in the mesh, which has proven well suited in the CGRA domain (see, for instance, DRESC, Mei et al. [2002]). The contribution of this thesis in advancing the State of the Art in the field of application mapping is illustrated in Chapter 5.

4.3 RAC architecture

This section presents the investigation of the EGRA at the *cell* level: it describes the RAC, its function and the exploration and evaluation of various RAC instances.

4.3.1 Cell architecture

The RAC datapath consists of a multiplicity of ALUs, with possibly heterogeneous arithmetic and logic capabilities, and can support efficient computation of entire subexpressions, as opposed to single operations. As mentioned in Section 4.2, it is inspired by the Configurable Computation Accelerator (CCA) proposed by Clark et al. [2004]. However, this structure is used as a *replicable* element within a reconfigurable array architecture; this has important consequences. First of all, it opens up the possibility of creating combinational structures (in Clark's design, a CCA has a fixed multi-cycle latency) chaining multiple RACs; this favours designs featuring a smaller number of rows. Furthermore, it removes the limit on the number of inputs and outputs, because a pipelining scheme (along the lines of the one proposed by Pozzi and Ienne [2005]) can be used to move data in and out of local memories connected to or embedded in the array; this allows scheduling of more complex applications and consequently higher gains.

ALUs are organized into rows (see Figure 4.3) connected by switchboxes. It is important to have flexible routing between ALUs on neighbouring rows, because subexpressions extracted from typical embedded applications often have complex connections that are not captured well by simpler topologies. This organization allows the usage of a simple array topology (as described in Section 4.4) without incurring high penalties on place-and-route.

The inputs of the RAC (see again Figure 4.3) are taken from the neighbouring cells' outputs, from the outputs of the cell itself, or (optionally) from a set of constants; the inputs of the ALUs in subsequent rows are routed from the outputs of the previous rows or again from the constants.

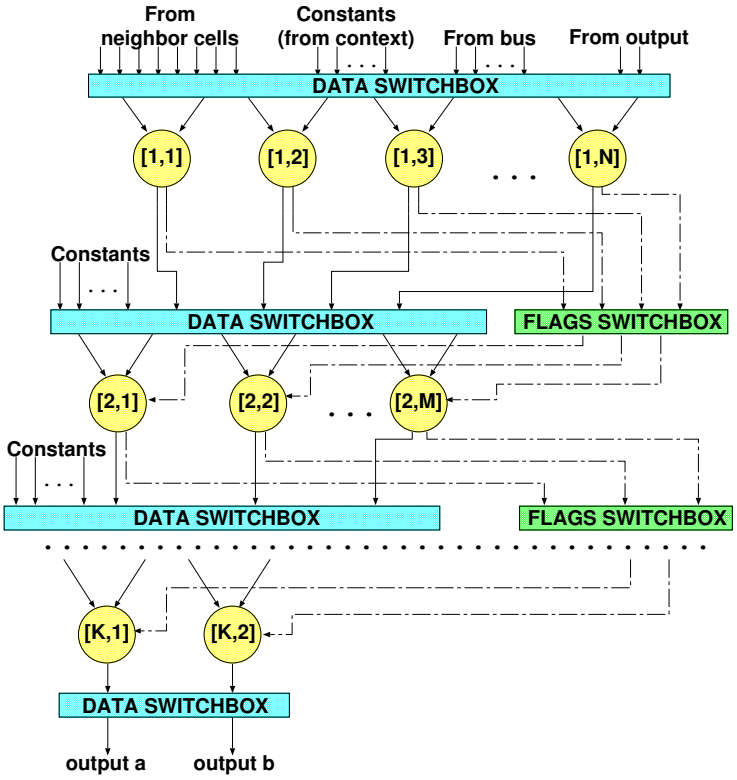


Figure 4.3. Datapath of the Reconfigurable ALU Cluster.

The number of rows, the number of ALUs in each row and the functionality of the ALUs is flexible and can be customized by the designer. In fact, they constitute the RAC *exploration level* explained in Section 4.3.2. The number and size of the constants is also defined at exploration time. If the datapath is wider than the constants, their value is sign- or zero- extended.

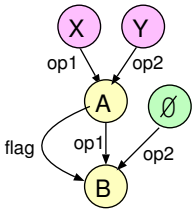
Being a reconfigurable design, the processing element includes not only a datapath, but also a *context memory*, which stores a number of possible configuration words and can be programmed according to the desired functionality of the cell at configuration time. Configurations controls the switch-boxes routing data through each RAC and between them, operations performed into datapath elements and values of embedded constants.

As in other CGRAs, the basic processing element of our cell design is an ALU. Unlike in the fine grain domain, it is not possible to define a generic component that can implement arbitrary functions, as is the case of the PLD or the LUT. Therefore, expressions are realized in our architecture by clustering more than one elementary unit (ALU) in one cell. Four types of ALUs can be instantiated. The simplest one is able to perform bitwise logic operations only; the other three add respectively a barrel shifter (with support for shifts and rotates), an adder/subtractor, and both the shifter and adder. The list of operations in a full-featured unit is in Table 4.1.

Each operation can generate three 1-bit flags: a *zero flag*, an *unsigned* $\geq$ *flag* (equivalent to the carry flag of general-purpose processors), and a *signed* $<$ *flag* (equivalent to $N \oplus V$, where

Table 4.1. List of supported opcodes

data opcodes	flag opcodes
$out = A \,\&\, (B \oplus flag_{sext})$	0
$out = A \mid (B \oplus flag_{sext})$	1
$out = A \oplus (B \oplus flag_{sext})$	$=$
$out = flag\ ?\ A\ :\ B$	$\neq$
$out = A + B + flag_{zext}$	signed $<$
$out = A + \overline{B} + flag_{zext}$	signed $\geq$
$out = A \ll B$	unsigned $<$
$out = A \ll_{rot} B$	unsigned $\geq$
$out = A \gg_{arith} B$	
$out = A \gg_{logical} B$	
$out = A \gg_{rot} B$	



node	opcode	op1 source	op2 source	flag source
A	op1 + $\overline{\text{op2}}$ + flag	BUS input 1	BUS input 2	1
B	flag ? op1 : op2	dataout(A)	constant 0	GEU(A)

Figure 4.4. Programming a RAC. This example shows how two ALUs can be connected to compute an unsigned subtract with saturation, `(X >= Y) ? X - Y : 0`. The node computing the subtraction also performs the comparison. The multiplexer node B uses both the data output and the *unsigned* $\geq$ flag of the subtraction node A.

N and V are the sign and overflow flags). Other conditions can be tested by complementing the flag, and/or exchanging the operands of the comparison. Dually, each operation has *three* operands, two being 32-bit values and the third being a 1-bit value that will be sign- or zero-extended depending on the ALU opcode.

The third operand can be hardcoded to zero or one, or it can be taken from another ALU's output flags, possibly complemented. This gives a total of eight possible *flag opcodes*, also listed in Table 4.1. Figure 4.4 illustrates and example of how complex expressions can be supported by RACs composing flag and arithmetic operations. A framework to automate expression mapping is presented in Bonzini et al. [2008].

Table 4.2. Datapath area and delay for different RAC configurations

# of rows	ALU type	ALUs per row					
		1		2		3	
		area (μm^2)	delay (ns)	area (μm^2)	delay (ns)	area (μm^2)	delay (ns)
1	log	7 141	0.45	15 001	0.51	28 926	0.55
	log+sh	11 695	0.62	30 627	0.66	48 029	0.71
	log+add	9 125	0.63	22 802	0.75	35 124	0.85
	log+sh+add	12 438	0.71	35 105	0.76	53 837	0.86
3	log	10 586	0.75	30 971	0.98	57 384	1.06
	log+sh	21 740	1.29	66 490	1.51	113 648	1.66
	log+add	14 926	1.54	44 054	1.88	71 716	2.18
	log+sh+add	27 672	1.57	77 472	1.89	125 552	2.32
5	log	12 458	1.05	43 793	1.49	86 560	1.68
	log+sh	32 455	1.93	100 165	2.43	168 760	2.68
	log+add	20 186	2.13	65 134	2.67	114 034	2.97
	log+sh+add	40 583	2.37	123 294	2.78	202 633	3.18

Flags enable efficient implementation of if-conversion—important when automatically mapping software representations onto hardware. In fact, ALUs can act as multiplexers, choosing one of the two 32-bit inputs based on another ALU's flags. This way, cells can evaluate both arms of a conditional, and choose between the two via a multiplexer.

4.3.2 Architectural exploration

Choosing the configurable architectural features—RAC granularity, number of constants in a RAC, number of contexts in the array to mention a few—is not at all an obvious task and should be guided by performance evaluation. Therefore we define an *exploration level* where a number of cell and array features can be automatically varied and evaluated in different experiments.

Design-space exploration was made feasible by the availability of a compilation flow that can speedily evaluate many different design choices, whose implementation is detailed in Bonzini et al. [2008]. Currently, this compilation flow is available for homogeneous arrays only, such as the ones considered in this section. For the more complex, heterogeneous arrays considered in the next two sections, considerable manual effort was involved in the mapping process, instead.

The flow automatically extracts graphs from frequently executed loops; graph nodes are clustered into groups fitting a single RAC (these groups are the EGRA *expressions*), and placed on the array by (optimistically) considering that no pass-through cell is needed on the critical path. The structure of the RAC is defined in a *machine description*, shared by the hardware and compilation flows.

In order to investigate area and delay figures of the RAC datapath, we synthesized different versions using Synopsys Design Compiler and TSMC 90nm front-end libraries. This has been instrumental in achieving two goals: on one hand, collected data is used by the compiler to compute the performance of Instruction Set Extensions (ISEs) mapped onto the array; additionally, it gives insights on the efficiency of various EGRA configurations as a digital circuit, both in term of occupied silicon area and clock speed.

Table 4.2 gives area and delay results for some of the different datapath configurations explored. All numbers refer to a datapath without embedded constants and with an equal number of ALUs on every row—neither of these, however, are actual limitations of the RAC template.

4.3.3 Experimental Results

Evaluation of RAC designs. In order to evaluate different RAC designs, DFGs were gathered from four MiBench benchmarks (Guthaus et al. [2001]) using a GCC-based compiler front-end. The graphs were then automatically tested with 872 different RAC configurations, employing RACs of one to three rows; the largest one had 5 ALUs on the first row, 4 ALUs on the second, and 2 on the third. The register file bandwidth is limited to 2 reads and 1 write; higher bandwidth values would yield higher speedups.

A single kernel is identified by the compiler in the case of the two audio benchmarks (rawcaudio and rawdaudio), performing respectively ADPCM encoding and decoding, while the two crypto benchmarks des and sha use four.

Estimated clock cycle savings are plotted in Figures 4.5 to 4.8. Speedup is calculated as follows:

$$speedup = \frac{tot\ cycl}{tot\ cycl - \sum_{all\ ISEs}(cycl_{sw} - cycl_{hw}) \cdot freq}$$

where *freq* is the number of times the ISE is executed, $cycl_{hw}$ is the latency of the ISE on the EGRA, and $cycl_{sw}$ is the cost of executing the ISE without custom instructions (both measured in clock cycles). Because $cycl_{hw}$ is integer and bounded by $cycl_{sw}$, the plotted speedups can take only a few discrete values, as observed in the figures.

Obtained speedup results show that multi-ALU cells outperform single-ALU cells found in more traditional CGRA designs. In fact, cells consisting of only one row correspond to the low-area points in the plottings, and have barely noticeable speedups.

Figure 4.9 shows four RAC designs. The first two represent the configuration of the maximum-speedup Pareto point, i.e. achieves the maximum speedup at minimal area cost, for each of the audio benchmarks; the third achieves maximum speedup on both crypto benchmarks; the last finally performs well on all benchmarks but costs noticeably more area than specialized cells. It is important to note that trivially merging the features of the cells in Figures 4.9a and 4.9b would use more area than Figure 4.9d, without improving performance.

All three solutions are 2-row RACs. It is interesting that, despite the apparent similarity between the design of the RAC and the CCA, they are much smaller than the examples of *Configurable Computation Accelerator* presented by Clark et al. [2004]. The reason is that RACs can be connected to form combinational structures. This features puts smaller cells to an advantage, since they will usually have higher utilization rates without sacrificing speed.

Evaluation of EGRA versus CGRA. To further investigate RACs' architectures, EGRA meshes were compared to a more traditional CGRA; i.e., the use of Clusters of ALUs as array cells, as opposed to using a single ALU. The latter solution has been proposed by the vast majority of CGRA designs like, among others, Morphosys (Lee et al. [2000]) and ADRES (Mei et al. [2004]).

We mapped computational kernels from rawdaudio, rawcaudio and sha benchmarks. For each kernel, three experimental settings were considered: 1-ALU meshes, meshes composed by best performing RACs from the given kernel and meshes composed by the “generic" RAC

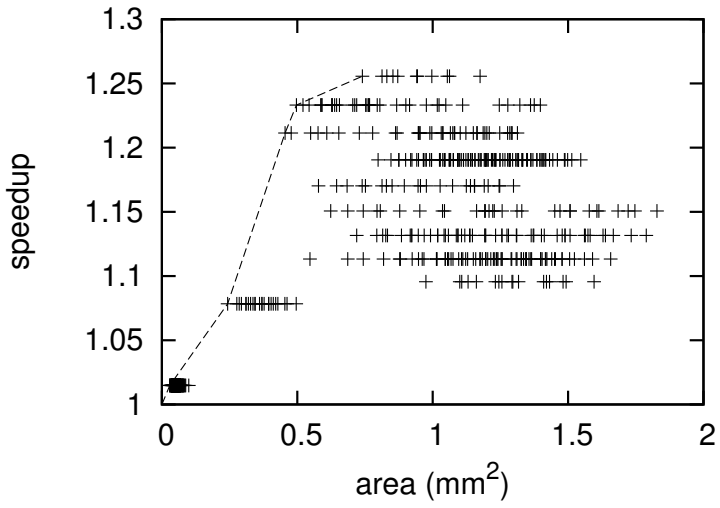


Figure 4.5. Speedups obtained by 872 RACconfigurations on rawcaudio

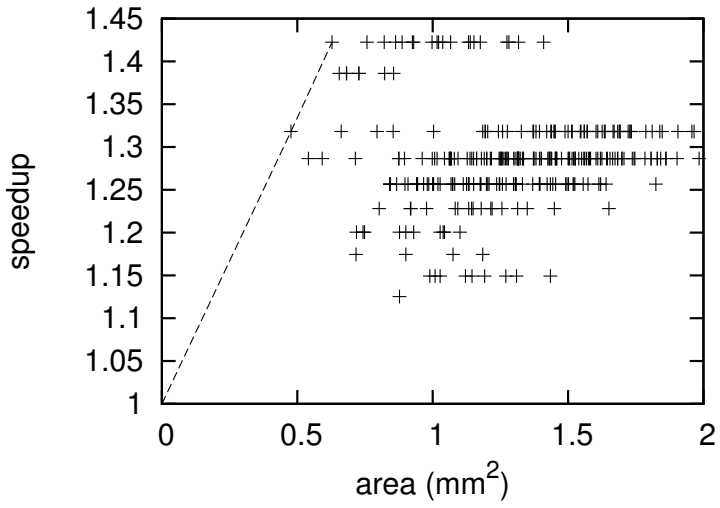


Figure 4.6. Speedups obtained by 872 RACconfigurations on rawdaudio

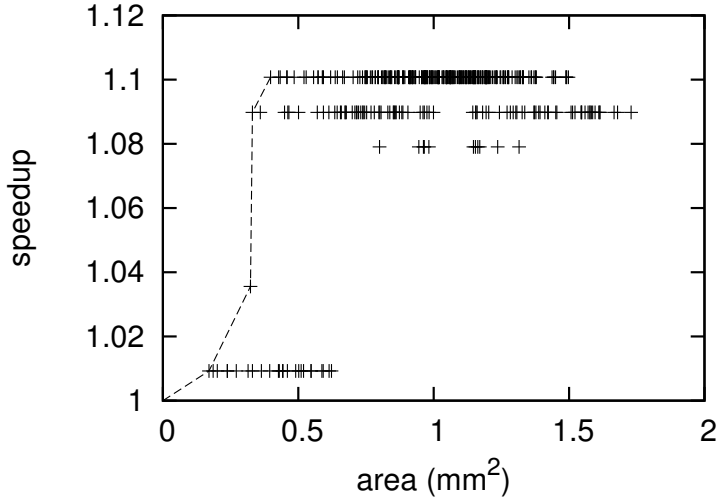


Figure 4.7. Speedups obtained by 872 RACconfigurations on des

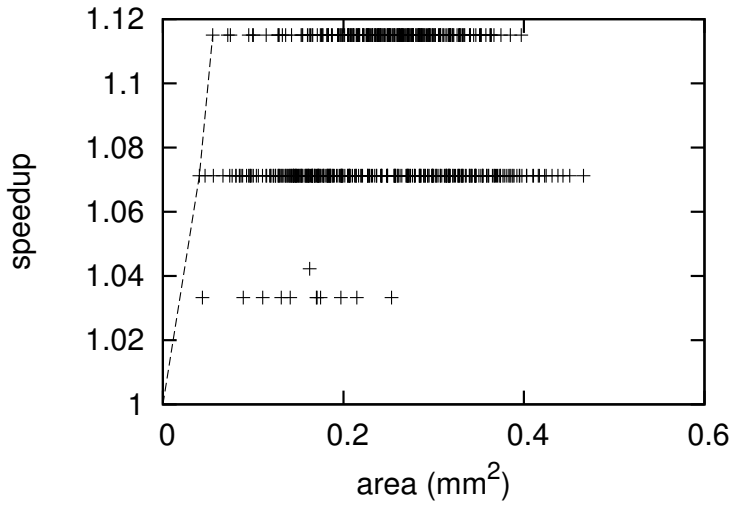


Figure 4.8. Speedups obtained by 872 RACconfigurations on sha

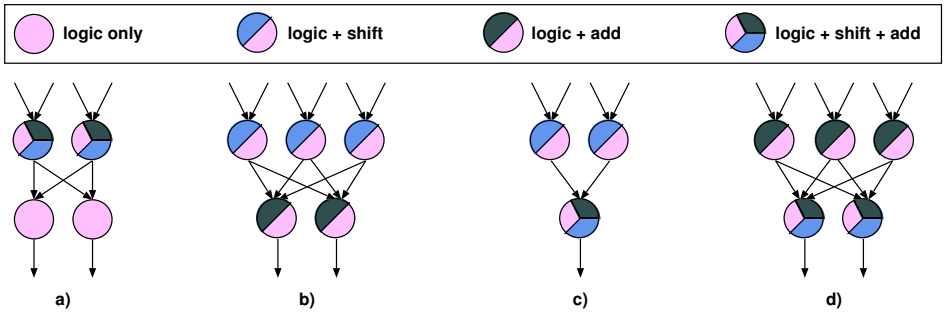


Figure 4.9. RAC design of the maximum-speedup Pareto point configuration, for a) rawdaudio; b)rawcaudio; c) crypto benchmarks (des, sha); d) all four benchmarks.

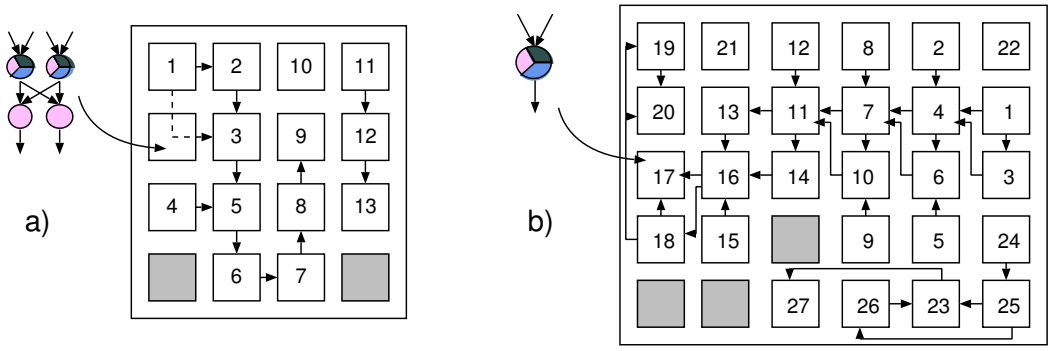


Figure 4.10. Manually derived spatial place-and-route of *rawdaudio*, for two cell designs. a) RACs as in Figure 4.9a. b) Each RAC only has one ALU. Straight arrows represent nearest neighbour connections, angled arrows refer to bus lines.

described in Figure 4.9d; this last case being considered to avoid over-tuning of the cell design. Figure 4.10 gives an example of resulting mappings in the *rawdaudio* case using the best performing RAC (a) and a single ALU (b).

As shown in Tables 4.3 through 4.5, RACs leads to faster execution on all three computational kernels, even when a generic RAC structure is considered; meshes composed by RACs achieved a speedup of up to 26% compared to single-ALU ones. Area penalty is not huge, and in some cases, as in rawcaudio, complex cells even lead to a smaller implementation, due to the smaller number of cells needed to map the kernel (as each cell is able to execute more than one operation).

4.4 EGRA array architecture

This section presents the investigation of the EGRA at the *array* level: it describes the EGRA heterogeneous mesh and architecture, the coexistence of various types of cells (RACs, multipliers, memories) and their performance, and the ability to perform modulo scheduling.

The EGRA structure is organized as a mesh of cells of three different types: RACs, memories

Table 4.3. Performance of placed and routed rawdaudio kernel on EGRA instances

	rawdaudio		
	RAC[a]	**generic RAC**[b]	**ALU**
Cell area (μm^2)	45 943	64 585	21 860
Cell delay (ns)	1.36	1.64	0.93
Array size	4x4	4x3	6x5
Array area (μm^2)	735 088	775 020	655 800
Iteration time (ns)	**10.88**	**11.48**	**12.09**

[a] as in Figure 4.9a
[b] as in Figure 4.9d

Table 4.4. Performance of placed and routed rawcaudio kernel on EGRA instances

	rawcaudio		
	RAC[a]	**generic RAC**[b]	**ALU**
Cell area (μm^2)	46 129	64 585	21 860
Cell delay (ns)	1.35	1.64	0.93
Array size	5x4	5x4	9x7
Array area (μm^2)	922 580	1 291 700	1 377 180
Iteration time (ns)	**16.20**	**19.68**	**20.46**

[a] as in Figure 4.9b
[b] as in Figure 4.9d

Table 4.5. Performance of placed and routed sha kernel on EGRA instances

	sha[a]		
	RAC[b]	**generic RAC**[c]	**ALU**
Cell area (μm^2)	47096	64 585	21 860
Cell delay (ns)	1.46	1.64	0.93
Array size	3x2	3x2	4x4
Array area (μm^2)	282 576	387 510	349 760
Iteration time (ns)	**7.30**	**8.2**	**8.37**

[a] first kernel only
[b] as in Figure 4.9c
[c] as in Figure 4.9d

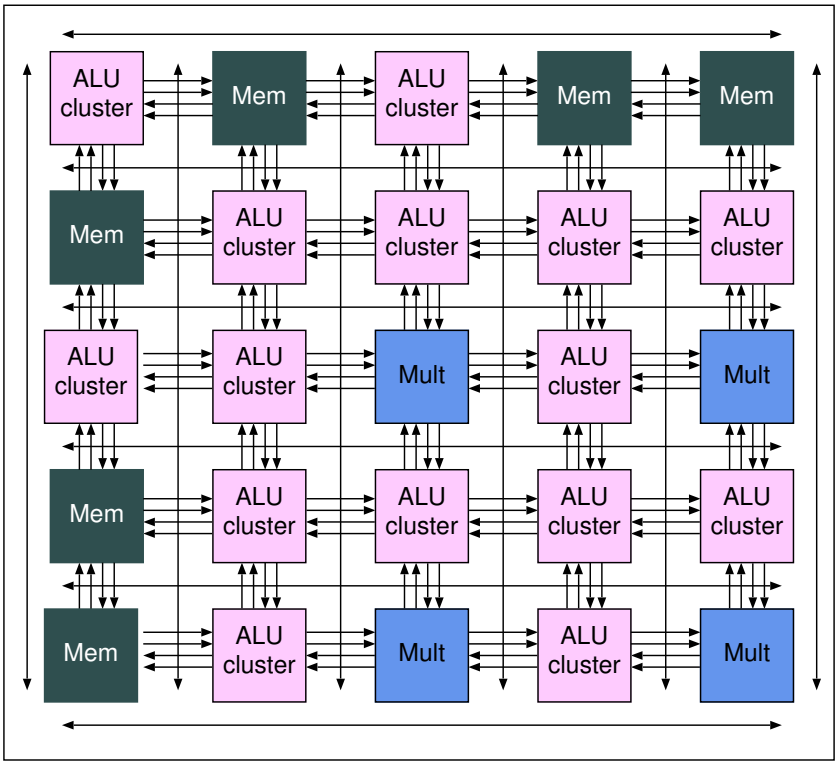


Figure 4.11. EGRA instance example: a 5x5 mesh with 15 RACs, 6 memory cells and 4 multipliers.

and multipliers (see Figure 4.11). The number and placement of cells for each type is part of the architecture parameter space. As such, it is decided at design time and can vary for different instances of the EGRA. Cells are connected using both nearest-neighbour connections and horizontal-vertical buses, with one such bus per column and row of the array. External to the mesh, a control unit orchestrates execution over the EGRA and a multi-ported scratchpad memory can be instantiated (see Section 4.5).

Every EGRA cell present an I/O interface, identical for all cell types, a context memory, and a core, that implements the specific functionality of the cell.

The interface part takes care of the communications between cells, and between each cell and the control unit. It is in charge of connecting the datapath with the outside of the cell by sending inputs to the datapath, providing values from the registers and the datapath to the neighbours, and placing them on the row-column buses if requested.

The context memory is where cells store their configuration; depending of the cell type (RAC, memory, multiplier) the word size and format of the context memory will vary. For example, multipliers have a fixed-function unit, and consequently the configuration word mostly deals with the routing of data for the interface part.

Each cell can store several configuration words, so that the entire array can hold several *contexts*; the number of contexts is also part of the architectural specification that is given at design time. Each context is composed of a configuration word per cell; the control unit can

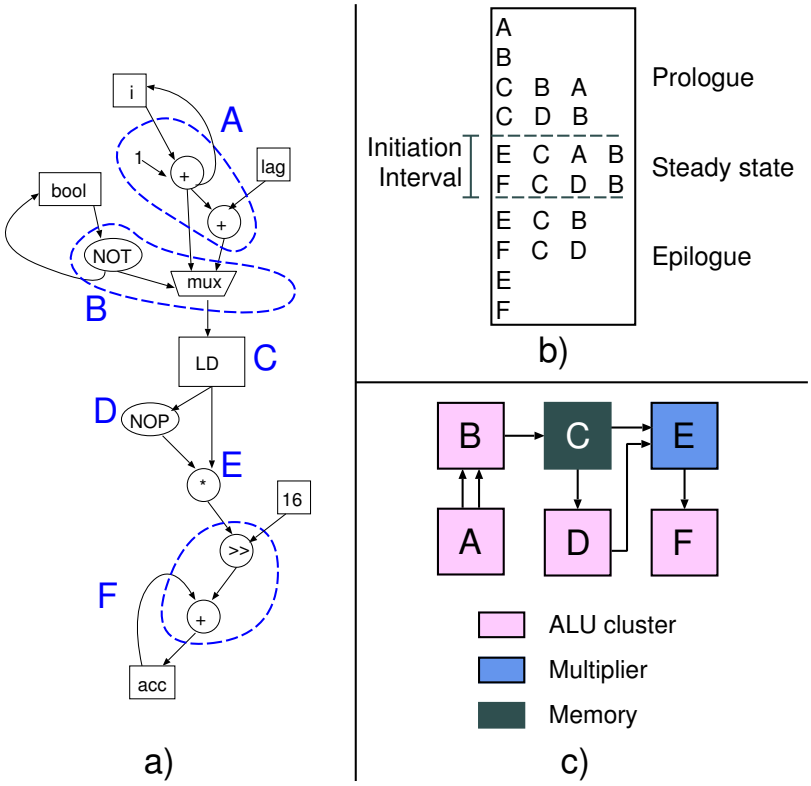


Figure 4.12. autcor loop kernel DFG: a) clustering; b) scheduling; c) place and route.

activate a different context on every clock cycle.

The rest of the cell is the actual implementation of the datapath and/or storage. For memory cores, this also include an address generation unit, so that multiple arrays can be mapped in the same memory cell in different areas; different data widths are supported (from 8- to 32-bit accesses) and are selected at reconfiguration time, while memory size is instead specified at design time.

4.4.1 Control unit

The control unit is explicitly designed to support modulo scheduling, a software pipelining technique that increases inter-iteration parallelism and cell utilization by allowing different iterations of the loop to partially overlap (Rau [1996]).

Modulo-scheduled loops present a prologue part, an iterated steady state part and an epilogue part. The host processor communicates the desired number of iterations to the accelerator and the length of prologue, steady state and epilogue. It also dictates if some scalar inputs or outputs, global to the kernel, have to be connected to array cells.

The control unit program counter is incremented on every clock cycle until it completes the prologue and the first iteration of the steady state, then it goes back at the start of the steady state, looping through it for the requested number of iterations. Execution is terminated with

Table 4.6. Characteristics of EGRAs optimized for different benchmarks

Benchmark	mesh size	cells type[a]			ALU cluster type[b]		memory (bits)
aifirf	3x3	6 C	1 MU	2 M	3-AL	2-SL	1 024 x 2
autcor	2x3	4 C	1 MU	1 M	3-ASL	2-AL	128
fbital	3x3	6 C		3 M	3-AL	2-ASL	4 096 x 3
fft	5x3	9 C	4 MU	2 M	2-AL	2-ASL	4 096 x 2
life	4x4	15 C		1 M	3-ASL	2-AL	4 096
rawdaudio	5x3	14 C		1 M	2-ASL	1-L	2 048
viterb	5x3	9 C		6 M	3-ASL	2-ASL	512 x 6
generic	5x5	15 C	4 MU	6 M	3-ASL	2-ASL	4 096 x 6

[a] C: ALU clusters; MU: multipliers; M: scratchpad memories

[b] ALUs in each row and supported operations. A: arithmetic; S: shift-rotate; L: bitwise logical. For example, 2-AL represent a row with 2 ALUs supporting arithmetic and logic operations, but no shifts/rotates.

a single activation of the epilogue.

On every clock cycle, generated data must be exchanged between the processing elements and loaded or stored in memory resources. To this end, the control unit distributes the id of the context to be executed to the cells, orchestrating execution. As the control unit limit itself to activate different control words inside the tiles, it can be made agnostic their internal implementation and architectural parameters, leading to a small and efficient design.

Some conditions must be verified for a loop to be mapped on the array, the most stringent of them is that the number of iterations must be known before the loop starts; this condition is usually met for kernels in the embedded systems domain. Loops must also employ only operations supported by the instance elements (for example, no multiplications are allowed if no multipliers are instantiated). Finally, resources must not be overused; this last constraint is in fact less hard then it seems, as memory footprint can be lowered by software transformations like loop tiling, and complex loops can be partitioned in simpler ones, employing a technique such as the one described in Chapter 5.

4.4.2 EGRA operations

The EGRA can be set in two operational modes: DMA mode and execution mode. DMA mode is used to transfer data in bursts to the EGRA, and is used both to program the cells (including initial content of the cells' output registers) and to read/write from scratchpad memories. Scratchpad memory transfers can happen either around a loop, or at program initialization if the scratchpads are used to store read-only look-up tables. In execution mode, the control unit orchestrates the data flow between the cells as explained earlier in this section.

Chapter 6 describes how EGRA instances con be integrated in a computing systems. Most extensible processors, like Altera NiosII or Tensilica Xtensa, support variable-latency custom instructions; in this case, after execution mode is triggered by invoking a special instruction on the host processor, the host processor can stall until the loop is completed and the EGRA asserts a "done" signal. By embedding input and output vectors entirely in scratchpad memories, a kernel can be run with a single special instruction, possibly surrounded by DMA transfers.

Table 4.7. Synthesized EGRA instances area and critical path

Benchmark	**area (μm^2)**	**crit. path (ns)**
aifirf	600 454	2.06
autcor	648 734	2.03
fbital	974 547	1.89
fft	1 136 287	2.15
life	1 332 020	1.71
rawdaudio	750 951	1.59
viterb	1 018 894	1.98
generic	2 699 577	2.16

Table 4.8. Initiation Interval (II) and parallelism achieved by loop kernels executing on the EGRA

Benchmark	**II**	**avg. active cells/cycle**	**avg. ops/cycle**
rawdaudio	15	2.6	5.2
fft	3	5	8
life	9	8.25	16.25
fbital	3	3.33	5
autcor	2	4	5.5
aifirf	2	4.5	7.5
viterb	3	5.6	10

4.4.3 Experimental results

To analyze the performance of the architecture template mesh, the parametric RTL design for the cells and control unit was implemented and validated. Various EGRA instances performance figures were retrieved using Synopsys Design Compiler to synthesize them on TSMC 90nm front-end cell libraries.

Loop kernels from seven benchmarks were considered: aifirf (fir filter), autcor (autocorrelation calculation), fbital (bit allocation alogorithm), viterb (convolutional packet decoding), fft (Fourier transform) from the EEMBC automotive and telecommunications suites (Halfhill [2000]), rawdaudio (audio decode) from MiBench (Guthaus et al. [2001]), life (game of life) from MIT bitwise benchmarks (Bitwise [1999]).

Choosen kernels were simple enough to be scheduled by hand, and yet illustrative of applications from different fields and with different memory and computational requirements. For example, fft requires the highest number of multipliers, viterb makes good use of multiple memories, while rawdaudio and life include mostly arithmetic operations.

For each benchmark, a custom-tailored EGRA was designed; Table 4.6 shows the characteristics of each of these specific architectures: the number of cells in the array, their type (ALU clusters, multipliers or memories), the type of ALU cluster used, and the total amount of memory present in the array. Table 4.7 reports total area and critical path of the different designs after synthesis. These architectures were instrumental in validating the EGRA model and to obtain initial indications on the EGRA capabilities.

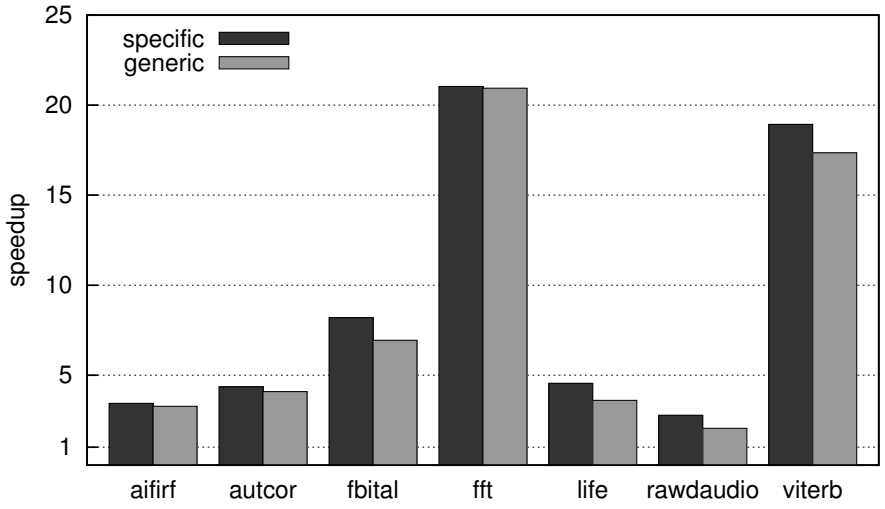


Figure 4.13. Speedup obtained on custom-tailored and generic EGRA instances executing benchmark kernels.

The last row of Table 4.6 and Table 4.7 show the characteristics of a *generic* EGRA mesh, i.e., an array that was designed so that all benchmarks kernels analysed could be mapped onto it. Such array configuration is actually the one shown in Figure 4.11 and consists in a 5x5 mesh, with an area of 2.7mm^2.

To map kernels into specific and generic architectures, their most intensive loops were extracted; these represented 80% to 99% of the execution time of the whole application in the test cases considered. Arithmetic-logic operations of kernels were then grouped so that every group could be mapped onto one ALU cluster cell, and modulo scheduling was performed (Figure 4.12). While done manually in this study, algorithms exist for these steps to be automated (e.g.: Mei et al. [2002] and the one illustrated in Chapter 5).

The effectiveness of the acceleration is heavily dependent on the degree of parallelism of the benchmarks. This in turn depends on the presence of dependencies in the loop, as well as on the distribution of the different operation types. The amount of parallelism obtained is shown by the average number of cells or operations that are active at any clock cycle. This data, and the initiation interval (II) obtained in each mapping, is summarized in Table 4.8.

To obtain speedups of EGRA-accelerated execution, as opposed to microprocessor-only, each benchmark was run through SimpleScalar/ARM, tuned to simulate an XScale processor with a 624 MHz clock[1]. The number of cycles needed to run the kernels on the XScale was then compared with those needed on various EGRA architectures. Figure 4.13 shows the speedup obtained by running benchmarks on a processor powered either with the application-specific EGRAs of Table 4.6 (labeled *specific*) or with the *generic* one of Figure 4.11.

As it can be observed, speedups as high as 21x can be obtained in the best case. Speedups on the generic architectures are marginally lower for benchmarks that do not need multiplications, due to the slightly bigger critical path associated with multiplier cells. A strategy to leverage

[1] 624 MHz is the maximum working frequency of the XScale PXA310 processor, also fabricated in 90nm technology

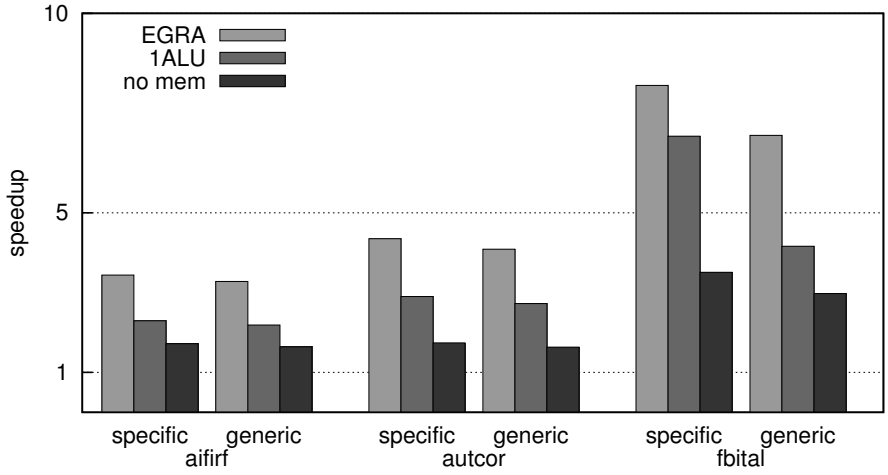


Figure 4.14. Comparison of speedups obtained by EGRAs with different characteristics: fully-featured instances, single-ALU instances and EGRAs without embedded memories.

differences in critical paths to ease routing by combinatorially chaining cells is presented in Chapter 5.

For three specific kernels, fbital, autcor and aifirf, we additionally compared the architectures of Table 4.6 with limited ones, in order to assess the advantage provided by the characteristics of the EGRA. In particular, we substituted ALU clusters with single-ALU cells, and avoided the usage of memory cells. The former shows the advantages of evaluating whole expressions in a cell as opposed to single operations; the latter causes all communication to go through the host processor and assesses the benefit of embedding memory in the array.

Results, plotted in Figure 4.14, show that both ALU clusters and scratchpad memories are beneficial to increase speedups on benchmark applications. In the absence of those, the net speedup is, in some cases, minimal. On the other hand, the two features are substantially orthogonal. Clustered cells have the advantage of allowing complex operations to be executed in a single clock cycle; complementarily, memory-capable EGRAs fare better compared to memory-less templates thanks to the parallelization of load/store operations on different scratchpads. Therefore, depending on the benchmark, either or both features can improve the speedups obtained by the EGRA. This is true either when kernels are mapped to custom-tailored instances or to the generic one represented in Figure 4.11.

4.5 EGRA memory interface

This section presents the investigation of the EGRA at the *memory interface* level: it describes the various interfaces that can be implemented by the EGRA template, and their exploration and performance.

4.5.1 Memory architecture

As discussed in Section 4.2, an aspect of paramount importance in designing an efficient accelerator is the implementation of its storage requirements. The EGRA template enables exploration of different solutions: a data register is provided at the output of computational cells, memory cells can be scattered around the array (an approach similar to the PACT-XPP architecture, Baumgarte et al. [2003]), and a scratchpad memory can be instantiated outside the reconfigurable mesh (as has been proposed by DREAM, Campi et al. [2007]). Memory solutions can be freely mixed to better suite requirements, and comparative studies can be carried out with different loads.

Different memory instantiations are better suited for different tasks, as registers of computational cells are usually employed to pipeline computation, while memory cells and the external scratchpad can store input/output data array and intermediate results. Data can be either read-only during execution, implementing look-up table or substitution boxes, or it can be writable by computational elements as directed by configuration words, which drives the write-enable and read-enable signals during loop executions. Data can be exchanged from and to the host system around execution via DMA transfers.

Both memory cells and the scratchpad are configurable in size and supported addressing modes (from 8 to 32 bits), and can store both signed and unsigned values. The scratchpad can have between one and four read-write ports. Multiple arrays can be placed in a single memory block by providing base addresses in the memory control word. This strategy streamlines the interface towards the computational cells, as only the index of the required data is to be presented to the memory, which transforms it according to its configuration to obtain the local memory address. At every clock cycle, sources for array index and data inputs, as well as addressing mode and base address, can be changed as needed by the mapped application.

4.5.2 Architectural Exploration

To evaluate design choices on test cases, kernel mapping of the applications described in Section 4.4.3 were further investigated. For each kernel, ad-hoc EGRA instances were designed, presenting different memory structures: either memory cells scattered around the mesh, or a single scratchpad with a number of read and write ports between one and four. The architectural choices, along with the data-flow graph dependencies of the kernels, determine the II (Initiation Interval) achievable by modulo scheduling kernels iterations, as they limit the number of loads and stores being performed in a single clock cycle. Table 4.9 summarizes the number and type of employed cells, as well as resulting II of mapped computational kernels; Table 4.10 shows resulting area and loop initiation time of the configurations, determined as $II * critical\_path$.

Figures 4.15 and 4.16 plot the occupied silicon area of the instances and their performance, expressed in time used to execute a kernel loop iteration, the latter being calculated by multiplying the maximum operational clock frequency and the loop II. The two graphs, therefore, represent the costs and benefits of the architectural choices.

A number of insights can be derived by analyzing the results. The area cost of adding ports depends on two factors: the area of the memory itself and that of the computational cells used to provide the indexes to access memory locations. Memory area in turn is affected by memory size and by the number of cells it is connected to, the second factor impacting the width of multiplexers connecting data to and from the mesh. Port addition is, for example, particularly

Table 4.9. Scratchpad and memory-cells based EGRA instances characteristics, tailored to different benchmark kernels.

Benchmark	Memory arrangement	Mesh structure	Kernel Initiation Interval (II)
autcor	memory cells	4 C 1 MU 1 M	2
	scratchpad - 1port	3 C 1 MU	2
	scratchpad - 2ports	3 C 1 MU	1
	scratchpad - 3ports	3 C 1 MU	1
	scratchpad - 4ports	3 C 1 MU	1
fft	memory cells	9 C 4 MU 2 M	6
	scratchpad - 1ports	5 C 4 MU	10
	scratchpad - 2ports	5 C 4 MU	6
	scratchpad - 3ports	5 C 4 MU	6
	scratchpad - 4ports	5 C 4 MU	5
viterbi	memory cells	14 C 6 M	3
	scratchpad - 1ports	12 C	6
	scratchpad - 2ports	12 C	5
	scratchpad - 3ports	12 C	4
	scratchpad - 4ports	12 C	4
life	memory cells	14 C 2 M	9
	scratchpad - 1ports	15 C	10
	scratchpad - 2ports	20 C	5
	scratchpad - 3ports	28 C	3
	scratchpad - 4ports	32 C	3
aifirf	memory cells	6 C 1 MU 2 M	2
	scratchpad - 1ports	5 C 1 MU	3
	scratchpad - 2ports	5 C 1 MU	2
	scratchpad - 3ports	5 C 1 MU	2
	scratchpad - 4ports	5 C 1 MU	2
fbital	memory cells	6 C 3 M	3
	scratchpad - 1ports	6 C	5
	scratchpad - 2ports	6 C	3
	scratchpad - 3ports	6 C	3
	scratchpad - 4ports	6 C	3
rawdaudio	memory cells	14 C 1 M	15
	scratchpad - 1ports	15 C	15
	scratchpad - 2ports	15 C	15
	scratchpad - 3ports	15 C	15
	scratchpad - 4ports	15 C	15
histogram	mem cells + 1port scratch	3 C 1 M	3

Table 4.10. Achieved performance of scratchpad and memory-cells based EGRA instances executing benchmark kernels.

Benchmark	**Memory arrangement**	**Instance critical path delay (*ns*)**	**Loop initiation time (*ns*)**	**Instance area (μm^2)**
autcor	memory cells	1.7	3.40	648 734
	scratchpad - 1port	1.75	3.50	648 142
	scratchpad - 2ports	1.75	1.75	701 170
	scratchpad - 3ports	1.76	1.76	710 994
	scratchpad - 4ports	1.81	1.81	962 802
fft	memory cells	2.15	12.90	1 136 287
	scratchpad - 1ports	2.14	21.40	915 871
	scratchpad - 2ports	1.98	11.88	930 067
	scratchpad - 3ports	2.01	12.06	995 157
	scratchpad - 4ports	2.12	10.60	1 360 855
viterbi	memory cells	1.98	5.94	1 018 894
	scratchpad - 1ports	1.61	9.66	874 416
	scratchpad - 2ports	1.75	8.75	1 172 240
	scratchpad - 3ports	1.77	7.08	1 236 547
	scratchpad - 4ports	1.79	7.16	1 299 307
life	memory cells	1.71	15.39	1 332 020
	scratchpad - 1ports	1.72	17.20	1 240 715
	scratchpad - 2ports	1.77	8.85	1 707 500
	scratchpad - 3ports	1.79	5.37	2 383 406
	scratchpad - 4ports	1.86	5.58	2 727 323
aifirf	memory cells	2.03	4.06	567 507
	scratchpad - 1ports	2.06	6.18	492 584
	scratchpad - 2ports	2.11	4.22	512 270
	scratchpad - 3ports	2.17	4.34	539 797
	scratchpad - 4ports	2.19	4.38	554 112
fbital	memory cells	1.83	5.49	974 547
	scratchpad - 1ports	1.84	9.20	770 995
	scratchpad - 2ports	1.84	5.52	826 374
	scratchpad - 3ports	1.85	5.55	961 702
	scratchpad - 4ports	1.87	5.61	1 060 623
rawdaudio	memory cells	1.59	23.85	750 951
	scratchpad - 1ports	1.61	24.15	795 286
	scratchpad - 2ports	1.62	24.30	824 635
	scratchpad - 3ports	1.64	24.60	875 082
	scratchpad - 4ports	1.65	24.75	888 178
histogram	mem cells + 1port scratch	1.33	3.99	5 630 657

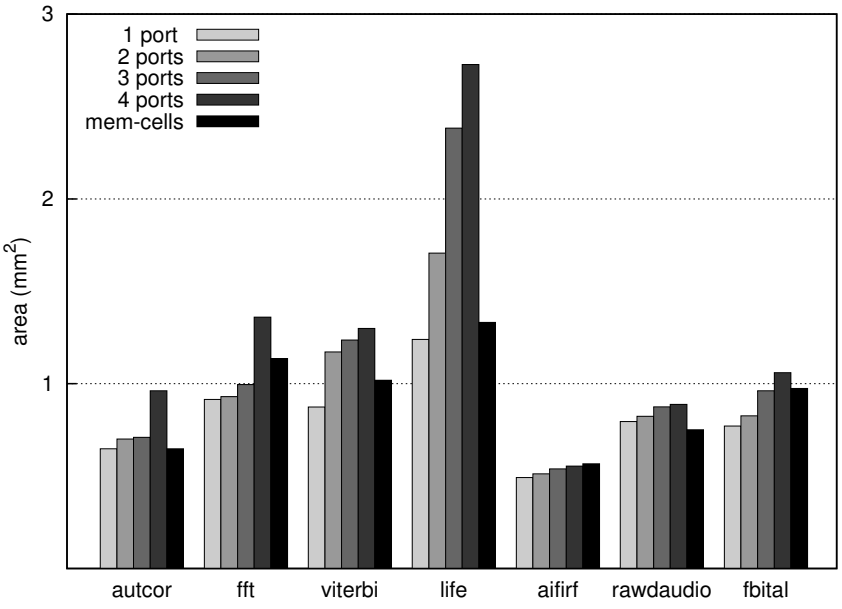


Figure 4.15. Area occupation of EGRA instances used to execute the benchmark kernels.

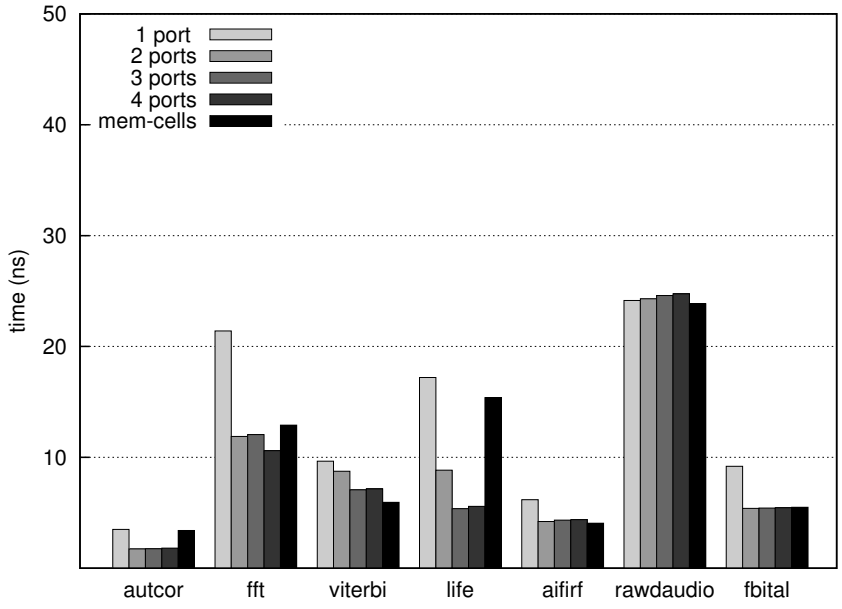


Figure 4.16. Kernel iteration execution time of benchmark application over EGRA instances.

costly in the life case, that employs a quite large memory (1 KB) and mesh. Required computational elements tend to increase as the number of ports grow, if multiple indexes are needed to address the memory ports with maximum parallelism, as in the case or the life benchmark; on the other hand, the autcor benchmark requires one more cell when a single ported scratchpad is considered to store an intermediate result lacking two simultaneous reads.

Adding memory ports to the scratchpad memory speeds up execution as long as memory accesses are the bottleneck of the system; after that point, the added memory ports present no execution time benefit (actually, the critical path worsens slightly due to increased circuitry complexity); in the examples provided, going from three to four ports was detrimental on all test cases.

Confronting memory-cells based instances and scratchpad based ones shows how different kernels are better suited to different strategies: kernels presenting a high number of accesses to small data arrays benefit from the parallelism offered by multiple memory cells like in the viterbi kernel, while bigger data collections are better supported by an external memory. Adding multiple ports to the scratchpad ultimately offsets the benefit of scattering the memory in most benchmarks, but with an increased area cost of the resulting array.

Even if memory cells and multi-ported scratchpads are comparatively studied in this work, EGRA instances can freely mix them, as the implementation of the MIT-bitwise histogram kernel shows. The kernel presents a small look-up table array and a bigger data array; the former naturally fits in memory cell, while the latter is better placed in the scratchpad memory. Characteristics of the instance tailored to this kernel can be found in the last row of Table 4.9, while its performance is indicated in Table 4.10.

4.6 Conclusion

This chapter describes the EGRA, and outlines different ways to explore the design space of this architectural template. The EGRA template features complex cells' design (RACs), various modes of memories implementations and efficient support for modulo-scheduled loops, enabling designers to investigate choices at multiple levels of the architectural hierarchy. Given the EGRA parametric nature, embodied in its machine description, different design aspects can be speedily evaluated, as the *machine description* parameters provide a simple, unified interface for EGRA instances generation.

The EGRA flexibility is exploited in this chapter to investigate performance of complex RACs, showing how they outperform simpler ones on a set of embedded system benchmarks; its support of heterogeneous cells is instrumental in developing instances mapping whole computational kernels; finally, its capability to instantiate multiple memory types gives insights on trade-offs in implementing storage requirements.

The EGRA template can be further expanded to accomodate other kinds of fixed-function units, and to include layout considerations. For example, irregular mesh topologies could be devised automatically in the presence of different-size processing elements.

The evolution of coarse-grained architectures should not happen in isolation; in particular, the support of an automated mapping methodology of computational kernels onto architectural instances explored in the next chapter presents a natural complementarity to the architectural exploration presented here.

Chapter 5

Application Mapping: Branch-and-bound Partitioning and Slack-aware Scheduling on Coarse Grained Arrays

5.1 Introduction

An effective methodology to map applications onto CGRAs has to consider multiple constraints, due to the parallel execution paradigm, sparse interconnection topology and, in some cases, heterogenous computational elements featured by these architectures. The first part of the chapter introduces a scheduling framework that automates modulo scheduling on heterogeneous CGRAs, considering an EGRA instance as the target architecture. The scheduler copes with limitations typically present on coarse grained meshes; more than that, it leverages differences in delays of various operations, which a CGRA always exhibits at run-time, to appropriately and effectively route data. We call this ability "slack-awareness". Slack-aware scheduling is beneficial in a coarse grained reconfigurable environment, as more complex applications can be mapped for a given mesh size and more efficient schedules can be achieved, compared to State of the Art methods.

CGRAs present streamlined control logic and memory capabilities, which in turn tightly constrains the complexity of computational kernels that can be successfully mapped on hardware resources. In order to handle complex applications, it is important to devise efficient strategies to partition kernels into pieces that can be scheduled and processed on such reconfigurable accelerators. The second part of this chapter describes an exact and an iterative solution to the partitioning problem, based on recursive searches over abstract trees. Experimental evidence suggests that the iterative method is computationally feasible even for fairly large kernels. It also achieves a partition quality close to optimality, and of much higher quality with respect to greedy algorithms.

Contributions of this chapter are:

- A novel framework to automate scheduling of computational kernels on coarse grained reconfigurable arrays is described. The slack-aware scheduler here presented is able to chain computation and routing operations to better exploit available architectural capabilities.
- A recursive partitioning algorithm is illustrated. The algorithm can cope with architectural constraints typically present on CGRA meshes: limited computation, control and memory resources.

5.1.1 Kernels scheduling on CGRAs

Many strategies have been proposed to accomplish application mapping on CGRAs; however, all previous efforts consider time in discrete chunks, assuming that each operation executed on a CGRA tile consumes a full clock cycle. Contribution to the research field exposed in this chapter focuses instead on exploiting *slack*, the difference between the clock period and the critical path of execution of an operation, to combinatorially chain computation and routing. Careful utilization of slack time makes it possible to increase routability on a reconfigurable mesh, leading to higher quality schedules of applications without impacting the working clock frequency.

The intuition behind the approach is presented in Figure 5.1: if communication between cells must be registered, operation B must be executed three cycles after operation A; on the other hand, if cycle time allows it, *and* unregistered communication is supported, B can be executed immediately after A. This strategy presents no penalty in maximum clock frequency if operation A, and routing data through cells, is fast enough compared to the slowest operation performed on the mesh.

This chapter presents a novel scheduling strategy that considers both registered and unregistered communication among tiles, resulting in an efficient utilization of computational resources, thus allowing the mapping of more complex kernels, and with a better execution performance, than is done by the State of the Art, which consists of slack-oblivious methodologies.

Slack-aware scheduling is evaluated when targeting an Expression Grained Reconfigurable Array instance, as described in Chapter 4, showcasing how scheduling can be adapted to an architectural template of which widely different instances can be derived parametrically, comprising heterogeneous cells and without restrictions on their arrangement. Investigated concepts are anyway more general and applicable to any CGRA architecture featuring registered and unregistered connections among tiles.

5.1.2 Kernels partitioning

CGRAs are able to extract loop-level parallelism from loops typical of embedded and DSP applications, and indeed research efforts have been undertaken to automate the application mapping process, notable examples being the works of Mei et al. [2003], Lee et al. [2003] and the one proposed here. A problem generally overlooked by proposed scheduling frameworks is how to deal with computational kernels whose size exceeds available hardware resources. Exceeding of resources can result from the limitation of three physical entities in the array: the first, obvious one, is the number of cells perfoming computation (ALUs, or cluster of ALUs); the

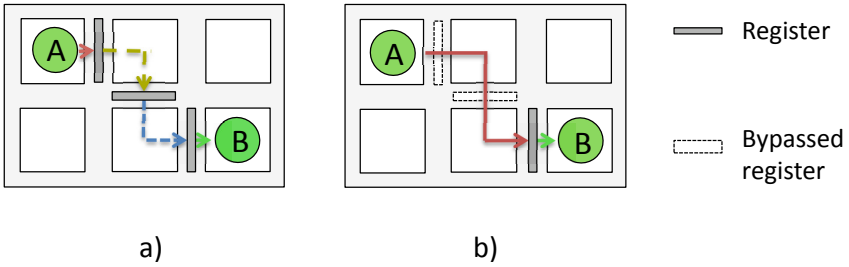


Figure 5.1. a) registered and b) unregistered routing through a CGRA mesh.

second, the number and size of internal memories; and the third, less obvious, is the number of configurations that the array can hold.

Loop fission, a compilation technique developed to improve data locality and described by Kennedy and Kinley [1994], is a useful approach in this context. In fact, a well-conceived partition of kernel computation can produce smaller pieces, which a scheduler can map in sequence on CGRA hardware resources. On the other side of the coin, kernels fission introduces issues of its own, as it may add pressure on the limited memory resources present on CGRAs accelerators.

As a simple illustrative example, consider the pseudo-code of a kernel, and its DFG representation, in Figure 5.2-a. If the computational requirement of the kernel exceeds that allowed by the underlying platform, the kernel can be partitioned, as it is done in the example along the dashed line.

The resulting partition includes two sub-kernels (Figure 5.2-b), each of which has a decreased operation count and depth, but an increased memory footprint. In fact, all edges crossing partition boundaries (corresponding to scalar variables c and d in the example) must be promoted to arrays, so that the values produced at each iteration can be stored in internal memory, and later passed on from one sub-kernel to the other.

The storage capacity needed by a sub-kernel is therefore the sum of 1) the memories already referenced by the computation of the sub-kernel itself and 2) those created by the partitioning. In the example, the storage capacity needed by the first sub-kernel is: the size of $aArr$, plus the size of the two newly created memories (each needing $Iter$ items, $Iter$ being the loop iteration count).

The second part of this chapter presents an efficient algorithm for partitioning loops to be executed on CGRAs. The inspiration from the proposed algorithm is taken from one published in a different field, that of Instruction Set Extensions identification. In particular, the employed branch-and-bound methodology is mutated by the one introduced by Atasu et al. [2003] and refined by Pozzi et al. [2006]. The algorithm is adapted to fit our slightly different needs, it is applied for the first time to loop partitioning, and its efficiency is investigated.

Sections 5.5 and 5.6 illustrate and evaluate an exact and an iterative technique to perform kernel partitioning under constraints present in CGRA architectures: limited computation, memory and configuration resources. Results are compared with a state of the art algorithm, the cluster-based greedy algorithm described by Purna and Bhatia [1999]. We show that the proposed algorithm performs partitionings of tangibly better quality than Purna and Bhatia [1999], while still scaling gracefully as problem size increases.

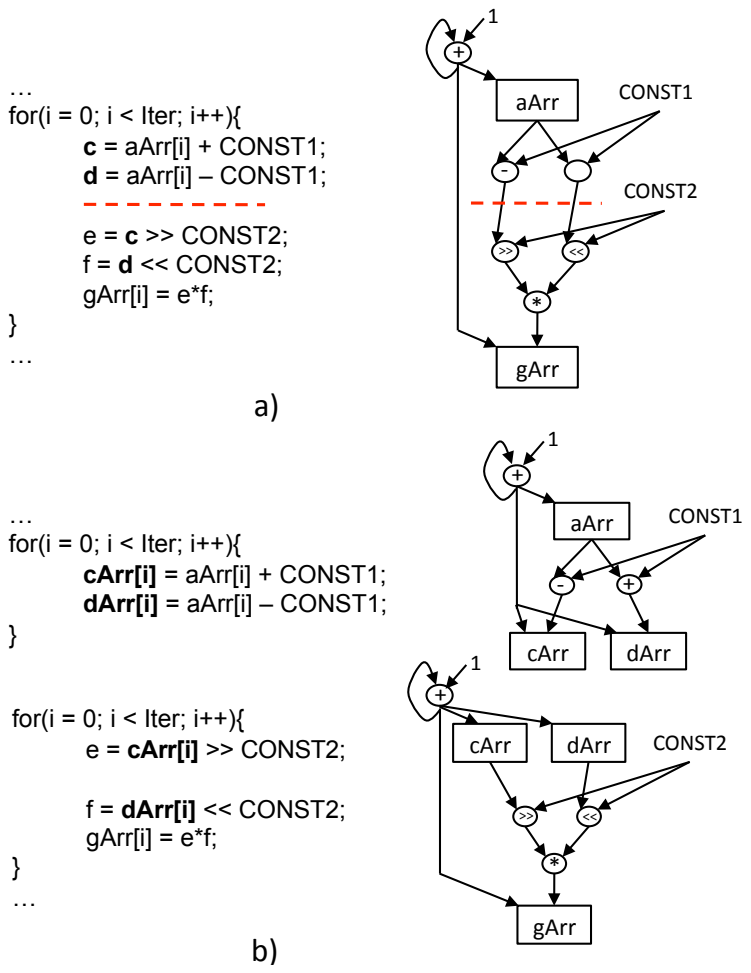


Figure 5.2. Pseudo-code and related DFGs of a small computational kernel, for the sake of an example, (a) before and (b) after partitioning. Partitioning causes a decrease in the size and depth of the graphs to be mapped onto hardware, but it increases their memory needs, as each edge crossing partition boundaries requires memory to store data passed between sub-kernels.

The present chapter details novel strategies to perform partitioning and scheduling of kernels. Comparisons with related efforts proposed in literature are provided in Section 5.2. Content of the chapter than illustrates the slack-aware scheduling strategy in Sections 5.3, providing experimental evidence of its performance in Section 5.4. A novel technique for kernel partition, based on branch-and-bound recursion, is detailed in Sections 5.5 and 5.6.

To better highlight effectiveness of the two frameworks, partitioning and scheduling are evaluated separately. Nonetheless, they are conceptually part of a unified framework to automate mapping of computational kernels onto CGRA accelerators. A comprehensive evaluation of the two interfaced environments it therefore a natural extension of the research presented in this chapter, and is planned as future work.

Table 5.1. CGRA scheduling methodologies.

Scheduling strategy	
Spatial	**Modulo**
SPKM Yoon et al. [2008] SMP Ahn et al. [2006]	DRESC, Mei et al. [2002] Hatanaka and Bagherzadeh [2007] Graph Embedding, Park et al. [2006] Resource Pipeline, Kim et al. [2005]

5.2 Related Work

5.2.1 Scheduling

An evolution pattern among proposed CGRAs has brought researchers to consider increasingly complex computational cells. Early proposed CGRAs, in fact, used tiles made of single ALUs (as exemplified by Morphosys, Lee et al. [2000] and ReMarc, Miyamori and Olukotun [1999]), while later designs employed more complex building blocks, able to evaluate *expressions* (groups of operations), as in PACT-XPP (Baumgarte et al. [2003]) and Montium (Heysters and Smit [2003]). The difference in computation time among operations supported by tiles increases with their complexity, as some expressions can be evaluated faster than others.

The transition from homogeneous to heterogenous structures has been another recent development; notable implementations of heterogenous CGRAs include RSPA, introduced by Kim et al. [2005] and Chameleon (Tang et al. [2000]). Heterogeneity is another factor increasing differences in computation time, as slower and faster cells have to cohabit on the same array.

It is worth mentioning that the present, slack-aware, approach is not only important in presence of heterogeneous and/or complex cells. Granted that heterogeneity in tiles' computation times increases in these scenarios, an important point to note is that *reconfigurable architectures, even when homogeneous, always exhibit heterogeneous computation times in tiles*, depending on operations executed on cells. If a tile containing an ALU is configured to perform an addition, while another is configured to perform a boolean operation, their delay will vary greatly, even though the two tiles are identical (as in homogeneous architectures).

Slack awareness aims at exploiting this imbalance to improve routability. To do so, the output register of each tile in a mesh must be by-passable, so that combinatorial (same-clock) chains of operations can be executed in different cells. While standard in FPGAs' logic elements, this mechanism is not typically present in CGRAs. It is instead a feature of the EGRA architectural template described in Chapter 4, target of the mapping methodology introduced here.

Slack-aware scheduling improves on approaches previously proposed in literature. Some of these efforts, taking inspiration from FPGA placing and routing, have considered spatial mapping as a way to maximize execution parallelism. Examples of this strategy are SPKM, authored by Yoon et al. [2008] and SMP, Ahn et al. [2006]. These works acknowledge the dual function (computation and data routing) of CGRA cells but neglect the opportunity to combinatorially chain cells to speed up execution.

A modulo scheduling approach has instead been taken by Mei et al. [2002], Hatanaka and Bagherzadeh [2007] and Park et al. [2006]. In these papers, both space and time dimensions are considered during mapping—as opposed to a spatial approach—borrowing from

the modulo scheduling technique employed on VLIW architectures (described by Rau [1996]). However, proposed modulo scheduling algorithms for CGRAs also overlook critical paths issues by assuming only registered connections between tiles, so that each operation consumes an entire clock cycle. This presents a negative performance impact, as shown in the experiments section, in both cases of spatial and modulo scheduling.

Another approach is Kim et al. [2005] research on resource pipelining, investigating how slow tiles can be pipelined and integrated with faster ones. We go one step further, as slack-aware scheduling can be applicable even when the divide between “slow" and “fast" tiles is not clear-cut and, again, when the execution time on a given tile depends on the operation scheduled onto it, dictated by configuration.

5.2.2 Partitioning

Application partition to cope with limited hardware resources is also the focus of many research efforts, as it presents itself in a variety of scenarios.

The first scenario is that of partitioning methodologies targeting FPGAs. Kaul and Vemuri [1998] propose an NLP formulation to optimally solve temporal partitioning of applications for time-multiplexed FPGAs. This approach assumes an application is split in well-formed tasks beforehand, and does not scale above a limited number of tasks. The same problem is tackled by Liu and Wong [1998] by adapting the Kernighan-Lin (KL, Kernighan and Lin [1978]) network-flow based algorithm to directed graphs; however the KL approach cannot directly guarantee that the number of edges in each sub-graph, and therefore the memory requirements of a sub-graph, is within given bounds.

In the context of high-level synthesis, Purna and Bhatia [1999] propose a cluster-based heuristic to map DFGs to multi-FPGA boards while minimizing communication bandwidth. The algorithm has a linear complexity but, as highlighted in Subsection 5.5.5, often fails to identify good candidate partitions, especially when dealing with fairly complex computational kernels.

Instruction Set Extension (ISE) identification aims at identifying groups of operations to be implemented as custom functional units with constrained inputs and outputs. Research in these direction has been undertaken by Yu and Mitra [2004] and Pozzi et al. [2006] among others. Given the high similarity between the ISE problem and the one tackled here, the pursued strategy is to adapt an efficient ISE algorithm, the one proposed by Pozzi et al. [2006], to the scenario of loop partitioning. The algorithm illustrated in Section 5.5 presents a different set of constraints to the one in the paper by Pozzi, and it is here proposed for a different aim, that of loop partitioning, for the first time.

Previous works on kernels scheduling for CGRAs for the most part assume the problem size is small enough, compared to available resources, and present methodologies to map applications. In this context, partitioning can be seen as a necessary pre-processing step, producing computational kernels to be mapped.

In the proposed methodology, based on loop fission, sub-kernels execute until completion for has many iterations as needed, with reconfiguration happening only at their *boundaries*. An alternative approach is loop disserving, described by Cardoso and Weinhardt [2002a] as applied to the PACT-XPP CGRA, in which the underlying hardware is reconfigured *inside* loop bodies many times at each kernel iteration. Loop disserving does not need temporary arrays to store intermediate data, but presents a much higher configuration overhead.

5.3 Slack-Aware Scheduling Framework

Goal of a CGRA scheduler is to modulo map a Data Flow Graph (DFG), representing an iteration of a computational kernel, onto the target architecture, i.e., onto a scheduling space representing a computational mesh. This problem is known to be NP-Complete (Shields [2001]), therefore a heuristic is devised to solve it.

Figure 5.3a shows an example DFG to be mapped, while Figure 5.3b exemplifies a possible valid mapping achieved using a slack-aware methodology on a 3x2 EGRA instance with nearest-neighbour connections and composed of four ALU clusters and two memory cells (a detailed overview of the architecture is given in Chapter 4). The graph can be executed in 3 cycles, because it combinatorially chains routing through two tiles between operations 1 and 3 in a single clock cycle.

In a nutshell, the proposed scheduling algorithm starts from an initial mapping that is high performance but possibly invalid, and iterates in search of a valid solution via a simulated annealing strategy. If a valid solution is not found with the currently sought high performance, the performance is lowered and the iteration starts again. Each step of the proposed algorithm will now be explained.

5.3.1 Expansion of the input DFG

To account for the dual use of CGRA cells (computation and routing), the input DFG is expanded by inserting routing nodes. On each edge, the number of routing nodes must be sufficient to completely traverse the scheduling space, whose time dimension is bounded by the maximum as-late-as-possible (ALAP) among operations to be mapped (as defined by Rau [1996]), while its space dimension corresponds to the physical size of the reconfigurable mesh. This approach is an extension of Yoon et al. [2008] from a spatial to a modulo-constrained environment. For each edge:

$$routing\_nodes\_number_{(edge)} =$$

$$max(\ N\_ROWS + N\_COLS - 3,\ max(ALAP(op))\ )$$

In the case of the considered example, this amounts to 2 routing nodes, and the graph is expanded accordingly (Figure 5.4).

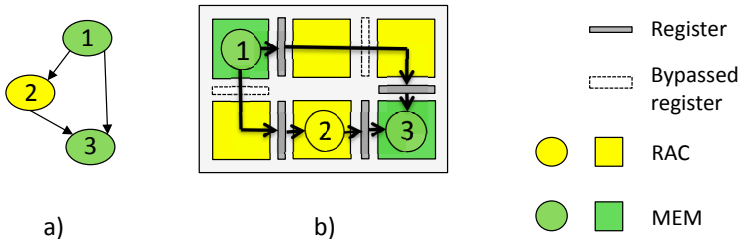


Figure 5.3. a) An example DFG and b) Its slack-aware mapping on a 3x2 heterogeneous EGRA.

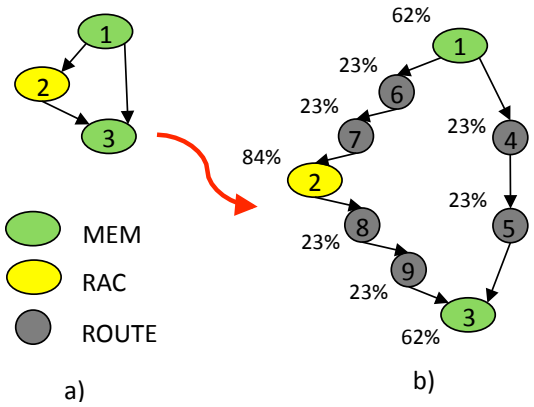


Figure 5.4. Routing nodes insertion on a DFG, with the annotation of the critical path length relative to the clock period.

5.3.2 Generation of an initial schedule.

The scheduling space is a three dimensional graph replicating the CGRA structure size, cells' types and topology $max(ALAP(op))$ times (3 times in the example). The graph edges connect to both the same time plane (representing unregistered connections) and to the following plane (representing registered connections).

To generate an initial schedule, three steps are performed (and illustrated in Figure 5.5a-b): first, operations are placed in the scheduling space on cells that support them and respecting their precedence constraints. Then, routing nodes are mapped to connect such operations, by employing the A* algorithm (formalized by Hart et al. [1968]). Finally, redundant routing nodes are deleted; routing nodes can be redundant either because they carry the same data of a node already scheduled on the same position (node 4 or 6, Figure 5.5a-b) or if they are placed at the position of their successor operation node (node 7 and node 9, Figure 5.5a-b).

Figure 5.5c shows the mapped DFG, decorated with registers among planes, and annotated with delays.

In the following, details are given of the second step, routing nodes mapping. Routing nodes on the cells between a scheduled predecessor operation ($cell_{pred}$) and a successor one ($cell_{succ}$) is handled as a problem to find the least costly path between them on the scheduling space. The cost of a routing cell ($cell_{rout}$) is defined as:

$$\begin{aligned} g(cell_{rout}) &= distance(cell_{pred}, cell_{rout}) \\ &\quad + max(distance(cell_{pred}, cell_{succ}) \\ &\qquad , (-t_{pred} + t_{succ})) \\ &\quad \times \#overused\_cells\_in\_a\_path \end{aligned}$$

$$h(cell_{rout}) = distance(routing\_node, cell_{succ})$$

$$cost(cell_{rout}) = g(cell_{rout}) + h(cell_{rout})$$

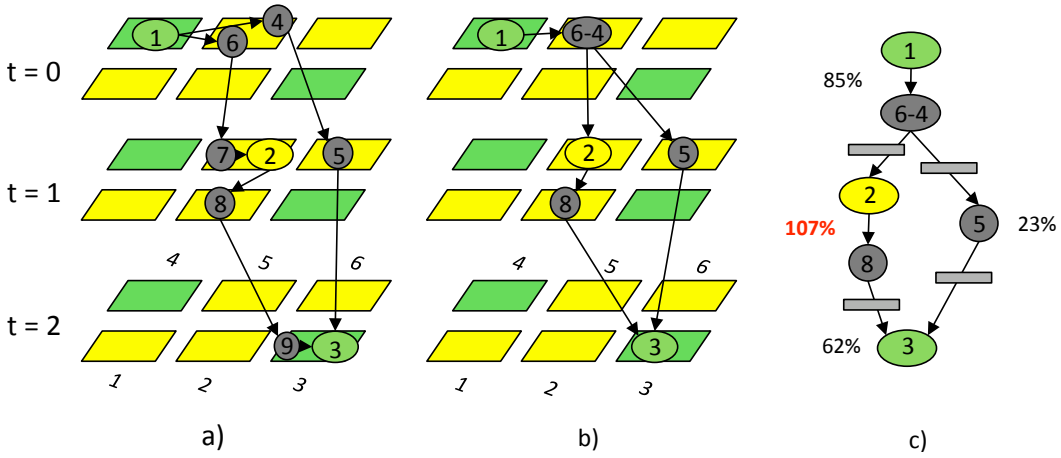


Figure 5.5. a) Expanded DFG mapping on the scheduling space. b) After redundant nodes deletion. c) Resulting DFG with annotation of routing times. The combinatorial chain of nodes 2 and 8 violates the timing constraint.

operation node	1	2	3
1	-	0	0
2	-	-	**1**
3	-	-	-

a)

	cell index					
	1	2	3	4	5	6
occupancy	0	1	1	1	**2**	1

b)

Figure 5.6. a) Slack Violation Table and b) Modulo Resource Table derived from the scheduling space in Figure 5.5.

where g is a path-cost function between the predecessor cell and a mapped routing node cell and h a heuristic function that estimates the remaining hops to the successor cell. t_{pred} and t_{succ} represent clock cycle planes of predecessor and successor operations, respectively.

5.3.3 Calculating the cost of a schedule.

The initial schedule, constructed in the previous step and shown in Figure 5.5, is not valid, as explained in the following. To check whether a schedule is valid or not, a *Slack Violation Table (SVT)* and a *Modulo Resource Table (MRT)* are derived, the former keeping track of timing violation, the latter of resource overuse.

Timing violation occurs when a path from register to register exceeds cycle time. Delays over paths are calculated, and a table is kept that indicates the amount of violation on each

edge. In the example, see Figure 5.5c and 5.6a, the SVT indicates a violation between nodes 2 and 3. Indeed, delay from 2 to 3 accounts to 107% of cycle time, as node 2 is computed at t = 1, and its output is routed to the cell below it without registering the result.

Resource overuse occurs when more than what can be supported by a cell is mapped onto it. This can happen in two cases: 1) when a cell is being used to route more than a single value, 2) when a cell is being used to compute an operation, *and* to route a different value.

Information on resource overuse is stored in the MRT, and a note is needed here on modulo scheduling, to explain the MRT. Modulo scheduling aims at maximizing parallelism by pipelining successive iterations of kernels execution; the distance (in clock cycles) between two iterations is defined as the *Initiation Interval* (II). To account for pipelining, the scheduling space must be folded according to the II when considering resource overuse; the resulting MRT is composed of exactly II rows, and contains the usage of each resource added modulo II. Figure 5.6b illustrates the Modulo Resource Table for the initial placement in Figure 5.5, considering II = 1. It can be noticed that cell 5 is overused, as it hosts node 6-4 at t = 0 and node 2 at t = 1.

This scheme can be easily extended to more complex topologies, including shared communication links, modeled as resources able to accommodate routing cells only. Indeed, results presented in Section 5.4 do consider local buses.

Once the MRT and the SVT are computed, a placement cost can be derived by adding up overuse and timing violations:

$$placement\_cost = \sum_{cells,buses} (\max((MRT_{i,t} - 1), 0)) + \alpha * \sum_{edges} (SVT_{op})$$

where $MRT_{i,t}$ are the elements of the modulo resource table, SVT_{op} the elements of the slack violation table and α a parameter trading off the importance given to each violation type (an $\alpha = 0.3$ was empirically determined as a good balance in the experiments presented in Section 5.4).

5.3.4 Iterating in search of a valid solution.

If the current schedule is not valid, a new one is created: an operation node is unscheduled together with its successor and predecessor routing nodes, freeing up related resources; the operation node is then remapped and related routing is performed to and from the node; a new cost value is computed and the move is accepted depending on its cost and the current (ever-decreasing) $temperature$. The process is repeated until a valid mapping is found (with $placement\_cost = 0$) or if the maximum number of tries has been reached.

If a valid solution has not been found after a number of iterations, a less aggressive mapping, of lower performance, is tried. This can be obtained by either increasing nodes mobility by augmenting their ALAP, or by increasing the Initiation Interval. The former can be beneficial to overcome timing violations, the latter to alleviate resource overuse.

5.4 Slack-aware Scheduling evaluation

5.4.1 Test architectural parameters

Target platform for evaluation of the benefits of slack-awareness is, in this study, an EGRA instance parametrized as a 5 x 4 mesh, composed by 14 RACs, 4 memory cells and two mul-

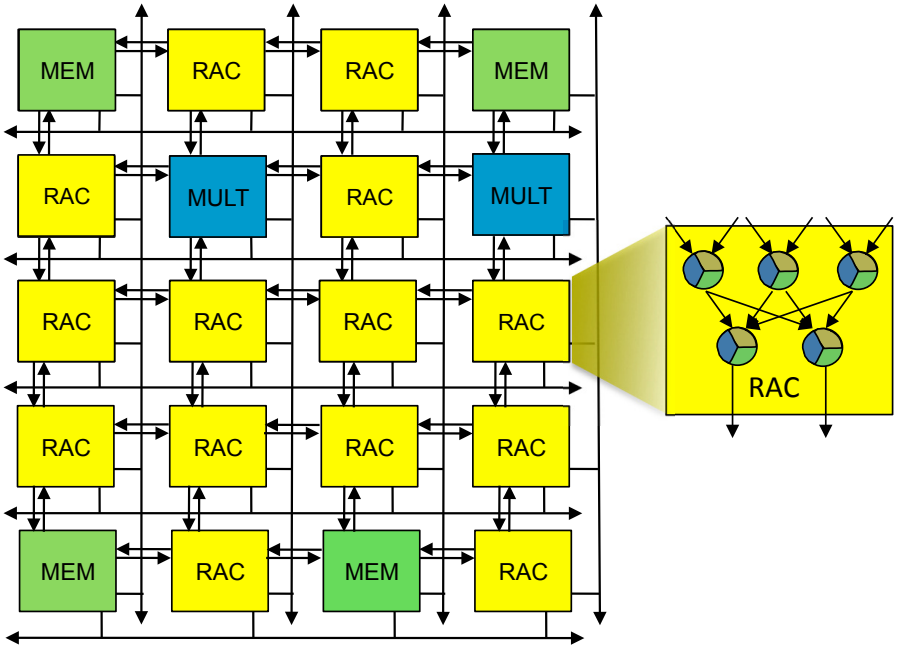


Figure 5.7. Example EGRA instance composed of 2 multipliers, 4 memory cells and 14 RACs, used for slack-aware scheduling evaluation.

tipliers. Considered RACs are composed of five ALUs in two rows, while multiplier cells are capable of both signed and unsigned multiplication, and memory cells are single-ported, 1kB in size and presenting 32-bits data addressing. Cells are connected using both dedicated nearest-neighbour connections and horizontal/vertical local buses. The EGRA instance is illustrated in Figure 5.7; features of the EGRA template detailing its architectural parameters are described in Chapter 4.

Data shown in Tables 5.2 and 5.3, first row, highlight the different critical path of different array tiles, depending on their type. In the case of RAC tiles, the critical path also depends on the operation performed on the cell. Results were obtained synthesizing the EGRA instance RTL implementation employing Design Compiler from Synopsys and TSMC 90nm libraries.

Assuming a clock period equal to the critical path of the slowest cell (the multiplier), Tables 5.2 and 5.3 show, in the second row, the percentage of clock period taken by each cell performing their supported operations. When this percentage is low, some routing hops can be accomodated in the same cycle, as shown in the motivational example of Figure 5.1b. The third row of Tables 5.2 and 5.3 show how many routing hops can be performed *after* computation and in the same clock cycle, without violating timing constraints. For example, a RAC configured to execute two boolean operations can chain two routing hops before exhausting cycle time, while a memory operation just one; results of a RAC performing two additions in two subsequent rows must be immediately registered.

This data highlights how heterogenous computation times can be leveraged to increase schedulability of kernels without increasing the clock period, and is used to derive scheduling results presented in this section.

Table 5.2. Critical path delay of different RAC operations.

	RAC					
	bool-bool	**bool-sh**	**bool-add**	**sh-sh**	**sh-add**	**add-add**
critical path (*ns*)	0.67	0.85	0.98	1.03	1.16	1.29
% of a 1.37 *ns* clock period	49	62	71	75	84	94
routing hops	2	1	1	1	0	0

Table 5.3. Critical path delay resulting from data routing, multiplication and memory cells read/write operations.

	route	**mult**	**mem**
critical path (*ns*)	0.31	1.37	0.85
% of a 1.37 *ns* clock period	23	100	62
routing hops	3	0	1

5.4.2 Experimental methodology

Performance of slack-aware scheduling was evaluated by applying it to two approaches employed in the State of the Art: spatial mapping and modulo scheduling.

In spatial mapping, the scheduling space is two-dimentional, and execution of just one operation on each cell is allowed for every kernel iteration, which is the approach of Yoon et al. [2008] and Ahn et al. [2006]. In modulo scheduling, the scheduling space has both a space *and* a time dimension, so that different operations can be executed at different times on the same cell (the strategy used by Mei et al. [2002], Hatanaka and Bagherzadeh [2007] and Park et al. [2006]). In each case, each operation consumes an entire clock cycle (slack-obliviousness), or slack-awareness can be applied to allow for combinatorial chains of operations. This leads to the four scenarios depicted in Table 5.4.

5.4.3 Automatically generated data flow graphs

Automatically generated DFGs were scheduled on the EGRA instance presented in Figure 5.7, using the four above-mentioned methodologies. DFGs ranged from 6 to 15 operation nodes, and one hundred of them were considered for each DFG size.

DFGs presented diverse shapes and characteristics: nodes were set to have one or two predecessors, with 50% probability in each case; nodes' types were randomly assigned with a probability matching the composition of the target mesh (10% multiplications, 20% memory operations, 70% RAC operations). The clock period was set to be equal to 1.37ns (the time used by the slowest operation, multiplication); consequently, allowed unregistered routing for the slack-aware scheduler followed data presented in Tables 5.2 and 5.3. Three thousand simulated annealing cycles were performed before increasing the Initiation Interval; application mapping failed when II reached $max(ALAP)$, a situation where loops are not pipelined at all.

Data plotted in Figure 5.8 shows the percentage of successful mappings for each DFG size

Table 5.4. Experimental framework.

	slack-oblivious	slack-aware
Spatial	**A**	**B**
Modulo	**C**	**D**

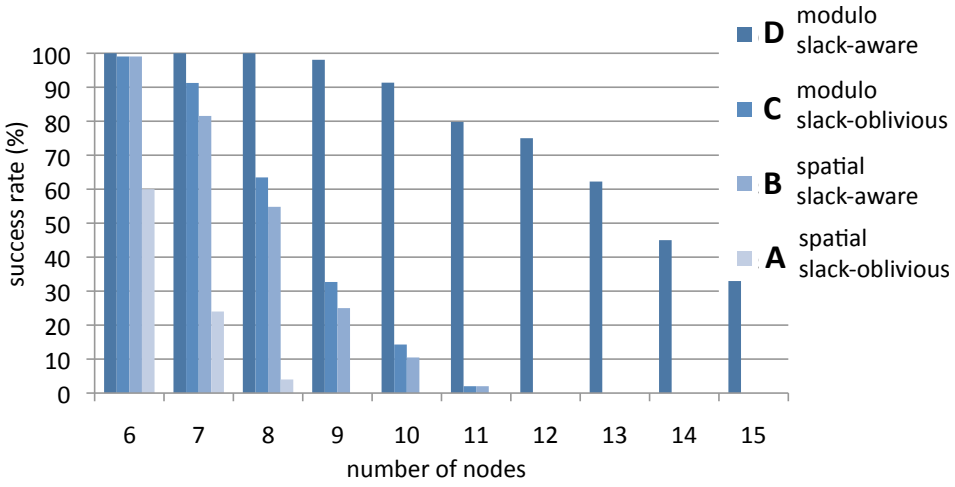


Figure 5.8. Slack-aware vs. slack-oblivious using modulo and spatial scheduling strategies: success rate of test DFGs.

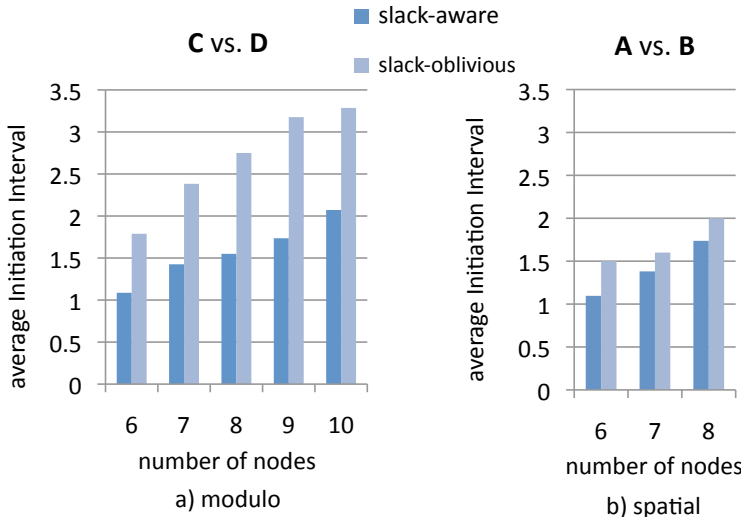


Figure 5.9. Slack-aware vs. slack-oblivious using a) modulo and b) spatial scheduling strategies: achieved Initiation Interval.

Table 5.5. Schedulability and performance of benchmark DFG kernels scheduled using different methods.

Benchmark	DFG nodes	Exp. DFG nodes	Scheduling method	Success (%)	II
autocorr	5	35	modulo-sl.aware	100	1.04
			modulo-sl.obl.	100	1.59
			spatial-sl.aware	100	1.04
			spatial-sl.obl.	98	2.09
conven	5	35	modulo-sl.aware	100	1.00
			modulo-sl.obl.	92	2.32
			spatial-sl.aware	100	1.00
			spatial-sl.obl.	77	2.32
aifirf	6	48	modulo-sl.aware	100	2.00
			modulo-sl.obl.	51	2.25
			spatial-sl.aware	100	2.01
			spatial-sl.obl.	40	2.98
mpegcorr	8	62	modulo-sl.aware	100	2.66
			modulo-sl.obl.	0	-
			spatial-sl.aware	0	-
			spatial-sl.obl.	0	-
iquant	9	69	modulo-sl.aware	100	2.08
			modulo-sl.obl.	0	-
			spatial-sl.aware	0	-
			spatial-sl.obl.	0	-
fbital	9	81	modulo-sl.aware	100	3.22
			modulo-sl.obl.	0	-
			spatial-sl.aware	0	-
			spatial-sl.obl.	0	-

using slack-aware and slack-oblivious mappings, both in a modulo and spatial setting. Modulo scheduling presents a better schedulability than a spatial alternative, as resources can be better utilized; more importantly, the graph highlights that *slack-aware scheduling maps more DFGs*. For example, 90% percent of 10-nodes DFGs where successfully mapped using slack-aware in modulo scheduling, while the same figure is around 10% for the corresponding slack-oblivious strategy.

In addition to being able to map more DFGs, slack awareness also *improves performance of mapped applications*. Figure 5.9a compares the average Initiation Interval of DFGs which were successfully mapped, in modulo scheduling, with both a slack-aware and a slack-oblivious strategy. The slack-aware scheduler achieves on average a 33% smaller II, corresponding to a 33% faster execution of the mapped kernel. A similar comparison in a the spatial environment (Figure 5.9b) is less clear-cut because the spatial mapping doesn't take full advantage of increasing II, so that in most cases either a DFG is mapped right away or is not mapped at all.

5.4.4 Kernels from benchmark applications

DFGs of computational kernels extracted from the EEMBC [1997] benchmark suite were also considered, again employing the EGRA instance described in Figure 5.7 and retaining the simulated annealing parameters used for the previous experiments. Each kernel was mapped one hundred times starting from different initial conditions. Table 5.5 illustrates the size of the DFGs before and after graph expansion, the percentage of successful mappings and the average Initiation Interval achieved with a slack-aware method compared to slack-oblivious one using modulo and spatial scheduling methods.

Results are in line with the ones obtained for randomly generated DFGs: slack-aware modulo scheduling is able to map all six benchmarks, while simpler strategies fail in the three most complex kernels. In all cases, slack awareness improves the chance of a kernel to be successfully mapped and the performance of obtained solutions.

5.5 Kernels partitioning framework

Efficient scheduling strategies, like the one described in the previous sections, increase the complexity of kernels that can be successfully mapped on constrained hardware resources. Nonetheless, applications whose size exceed the capabilities of target architectures must be split into multiple sub-kernels to be properly handled. In the following part of the chapter, a formalization of such partitioning problem is given, together with exact and iterative branch-and-bound strategies to solve it. Obtained solutions are evaluated in comparison with a State-of-the-Art greedy algorithm.

5.5.1 Problem formalization

Let $G\{V, E\}$ be a Direct Acyclic Graph (DAG), where nodes V represent operations executed in an iteration of a computational kernel and E dependencies among operations. Nodes $v \in V$ can be computation nodes or memory-access nodes. In the latter case nodes have an attribute m_v that has an index unique for each array that the kernel processes. Different memory-access nodes can have the same m_v attribute if they read/write on the same array.

A cut S is a sub-graph of G, where $S \subseteq G$, containing the nodes assigned to a sub-kernel. A partition P of G is a set of non-overlapping S_i cuts covering all nodes of G. Let $IN(S_i)$ be the set of predecessor nodes of those edges crossing the cut boundary into S_i, and $OUT(S_i)$ the set of predecessor nodes of edges crossing the cut boundary out of S_i.

Goal of partitioning is to assign each node of G to a cut, such that each cut does not violate memory, size, depth and convexity constraints. A merit function is then used to discern lower- and higher- quality solutions among the valid ones. Constraints and merit function are detailed as follows.

Cut Size. To cope with the limited number of computation elements present in a CGRA mesh, the size of each cut should not exceed a threshold. Schedulability for a give mesh size, in fact, strongly correlates with the application DFG size, as shown in the works of Yoon et al. [2008] and Ansaloni, Tanimura, Pozzi and Dutt [2011] (the latter illustrated previously in this Chapter) among others.

Cut Depth. During execution, CGRAs activate a control word at each clock cycle, so that cells can perform the proper operation at the proper time. CGRAs support a limited number of control words, in turn limiting the maximum depth of DFGs that can be mapped onto them.

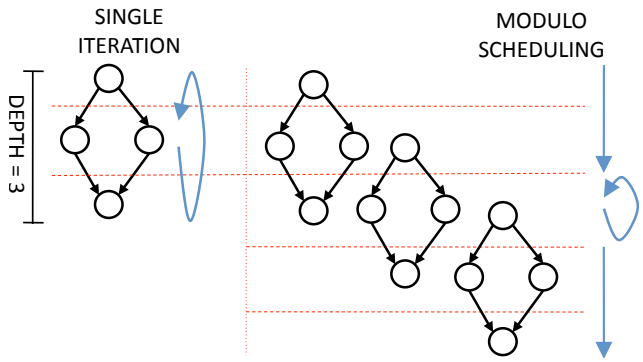


Figure 5.10. Sub-kernel depth is limited by available control words: $Depth(DFG)$ words are necessary if iterations are mapped in sequence (left), while $2*Depth(DFG)-1$ are used when employing modulo scheduling (right).

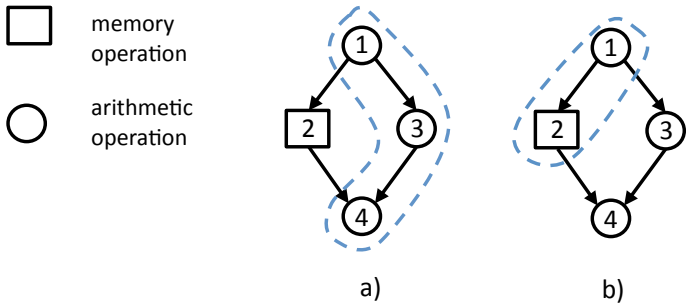


Figure 5.11. Single cut identification: a non-convex cut (a) and a cut with one memory reference and two outputs (b).

Two scenarios can be envisioned, as illustrated in Figure 5.10: if kernel iterations are subsequently executed on a mesh, $maxDepth(S_i) = (nCtrlWords - P)$, were P is a user-defined relaxation parameter to ease DFG scheduling on the reconfigurable mesh. If instead iterations are modulo scheduled (as considered, for example, by Mei et al. [2003]), we have that $maxDepth(S_i) = (((nCtrlWords + 1)/2) - P)$ in the case of explicit prologue and epilogue.

Cut memory footprint. As discussed in Subsection 5.1.2, after kernel fission a sub-kernel might require excessive data storage with respect to underlying hardware. Given a cut S_i, M_{S_i} is defined as the set of all distinct m_v attributes of each $v \in S_i$. The cardinality of M_{S_i} indicates the internal memory requirement of a cut S_i. In addition, $IN(S_i)$ indicates the number of temporary arrays which are input to the sub-kernel, while $OUT(S_i)$ the number of arrays that are generated by executing the sub-kernel. The contribution of these three elements must not exceed the amount of memory available in the hardware.

Local and global convexity. Local convexity is a property of a cut, imposing that no path through G exists that exits and re-enters a valid cut S_i. The constraint ensures that all inputs to a cut can be ready when the cut is to be executed. Figure 5.11-a shows a non locally-convex cut.

Global convexity is instead a property of a partition; it states that once all cuts of a partition are collapsed into single nodes, the resulting graph is acyclic. A globally convex partition allows for at least one order in which sub-kernels can be scheduled in sequence, as each cut S_i can be either a predecessor or a successor of another cut S_j of the partition, but not both.

Merit Function. Kernel fission forces a sequential barrier in execution of sub-kernels on a CGRA accelerator. To ensure maximum parallelism given the above-mentioned constraints, the number of sub-kernels should be minimized. A solution employing a small number of large sub-kernels also minimizes the overhead due to reconfiguration and transfer of the data-set in and out of the accelerator.

Problem Formulation. The kernel fission problem can now be formalized as follows.

Given a DFG representation of a kernel $G\{V,E\}$ and indexes of its memory references m_{v_i}, find a partition $P = \{S_1, S_2, ..., S_n\}$ of G such that

1) $\forall$ S_i, size of $S_i < MaxSize$
2) $\forall$ S_i, depth of $S_i < MaxDepth$
3) $\forall$ S_i, $|M_{S_i}| + IN(S_i) + OUT(S_i) < MaxMems$
4) all S_i are locally convex, P is globally convex
5) $|P|$ is minimized

The partitioning problem is solved either exactly or iteratively, employing the algorithm described in Atasu et al. [2003] and later refined in Pozzi et al. [2006], adapting it to the loop partitioning problem. The explanation of such methodologies is reprised here, for completeness and clarity, stating differences when needed. Firstly, an algorithm is detailed identifying the single largest valid cut given a kernel DFG and architectural constraints. The problem is then expanded to consider a complete DFG partitioning; an exact, but computationally expensive, solution is proposed in Subsection 5.5.3, while much less-expensive iterative one is illustrated in Subsection 5.5.4. In Subsection 5.5.5, instead, the linear-time greedy approach adopted in Purna and Bhatia [1999] is summarized, used for comparison purposes in Section 5.6.

5.5.2 Single cut identification

The single cut identification algorithm starts by a topological sorted graph G, where a node u precedes v in the order if $Depth(u) < Depth(v)$.

The algorithm uses binary recursion to span an abstract search tree. At each bisection of the tree, two branches are considered, respectively including or excluding a node $v \in G$ from a cut S, considering each v_i in topological order.

The size of the search tree thus constructed is obviously exponential, but effective pruning can be performed to restrict the search space. Two straightforward pruning conditions, which are not present in Pozzi et al. [2006] as they are specific to this problem formulation, examine the depth and the size of the cut when nodes are added to it: if $MaxDepth$ or $MaxSize$ are exceeded, adding further nodes to S by expanding the underlying search branch cannot result in a valid solution (it violates underlying-platform resources).

Other pruning conditions are related to subgraph convexity and input-output count, and apply to graphs that are topological sorted. If a cut is non-convex, there is no way to recover local convexity by adding nodes that come later in the topological order (Figure 5.11-a). Moreover, the number of inputs $IN(S)$ present in a cut can only increase by adding nodes with greater depth, as is the number of referenced arrays.

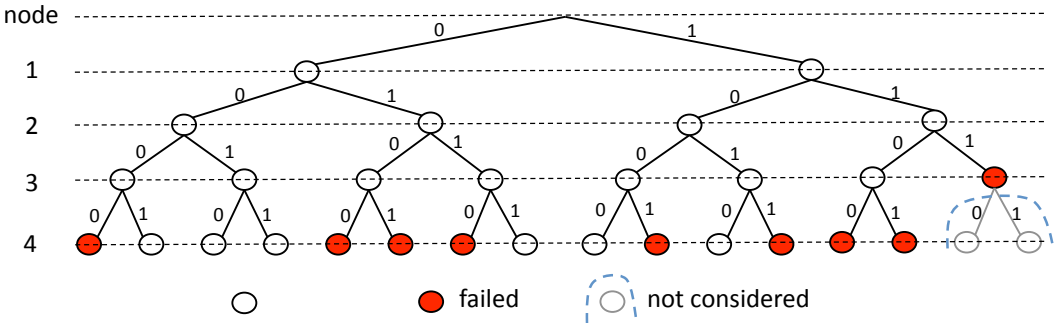


Figure 5.12. Single cut identification: abstract search tree of the DFG in Figure 5.11, considering $MaxSize = 3$, $MaxMems = MaxDepth = 2$.
0 → node not included in cut, 1→ node included.

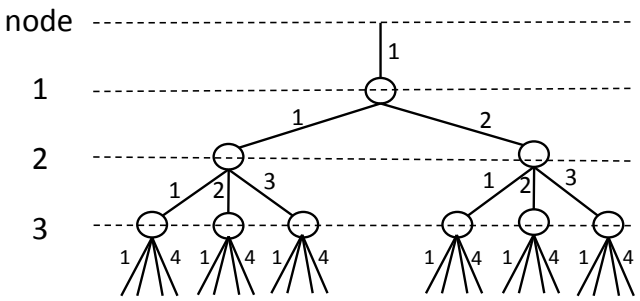


Figure 5.13. Abstract search tree for multiple cuts identification.

Finally, outputs of a cut S become non-recoverable, i.e. they cannot exit $OUT(S)$, once a successor of a node $v \in S$ is excluded from S. Consider Figure 5.11b: if node 3 is not part of S, adding any further node to the cut will not remove node 1 from $OUT(S)$; these outputs are named *permanent* and the set containing them $OUT_P(S)$. The pruning condition due to memory constraints is therefore derived as:
$|M_S| + IN(S) + OUT_P(S) < MaxMems$

The use of permanent outputs is also described in Pozzi et al. [2006]. The pruning condition is here slightly different as it is the sum of inputs, outputs and internal memories that must be constrained. Figure 5.12 shows the abstract search tree for the simple 4-nodes DFG of Figure 5.11. It can be noticed that just six of sixteen possible cuts are valid (the empty cut is never considered). More importantly, some search region can be pruned away before a leaf is reached.

5.5.3 Exact multiple cuts identification

It is possible to extend the previously illustrated algorithm to multiple cuts, substituting binary recursion with an N-ary one, where each branch corresponds to assigning a node to one of N cuts.

Benchmark	Nodes	Edges	Arrays	Array accesses
fft	22	31	2	8
rgbcmyk	24	34	2	7
rgbhpg	30	37	2	10
rgbiq	33	40	2	6
viterbi	36	51	5	14
dct	93	126	3	16
idct	94	145	3	24

Table 5.6. Benchmarks characteristics

Without loss of generality, symmetric solutions can be avoided by assigning node v_1 to cut S_1 unconditionally, let v_2 be part of either S_1 or S_2 and so on, increasing by one the number of possible choices, at each level. The resulting abstract search tree is shown in Figure 5.13.

The lower bound on the number of cuts in a partition is $K = \lceil (|V|/MaxSize) \rceil$, while the upper bound is $|V|$, which corresponds to assigning every node in G to a different cut. As discussed in Subsection 5.5.1, partitions employing the smallest number of cuts are desirable. Therefore, a search for solutions employing the smallest possible number of cuts, K, is performed first. If no valid partition with cardinality K is found, the algorithm proceeds by seeking a valid partition with $K+1$, $K+2$, ... , $|V|$ cuts.

The exact algorithm is guaranteed to find an optimal solution to the partitioning problem if said solution exists within constraints. However, its exponential complexity makes it intractable in some cases.

This algorithm is very similar to the one presented in Atasu et al. [2003] and Pozzi et al. [2006]. The difference lays in the figure of merit of the problem, which, in turn, guides the search in a different way. In the ISE problem, the merit to be maximised is the collective gain of selected ISEs, i.e. selected subgraphs, and graph nodes that are left-out (branch at 0, in the exact algorithm) do not influence the merit function—they are to be executed in software. In other words, a partition made of few large subgraphs, but also many single-node subgraphs (those left-out), can be an optimal solution in the problem formulation of Atasu et al. [2003] and Pozzi et al. [2006]. However, for the problem tackled here, a winning partition must have minimum cardinality. Correspondently, a branch at 0 is not considered: each node must belong to a subgraph, and a valid partition's goodness is measured in terms of minimising its cardinality.

In practice, the Exact partitioning algorithm can deal with slightly larger subgraphs than that of Atasu and Pozzi. Graph sizes up to 36 nodes are dealt with in the experimental results, while a size of less than 30 is reported by Atasu et al. [2003].

5.5.4 Iterative multiple cuts identification

The iterative strategy performs single cut identification multiple times, until either all nodes are assigned or no valid partition is found. At the first iteration, the largest cut obeying constraints is optimally identified, and its nodes marked accordingly. Successive iterations optimally search for other valid cuts *that exclude already assigned nodes*.

This iterative strategy differs from the exact one as it does not guarantee that the partition with smallest cardinality is found, but the solution space is greatly reduced, as the same search tree as in Subsection 5.5.2 is employed multiple times, preserving high efficiency.

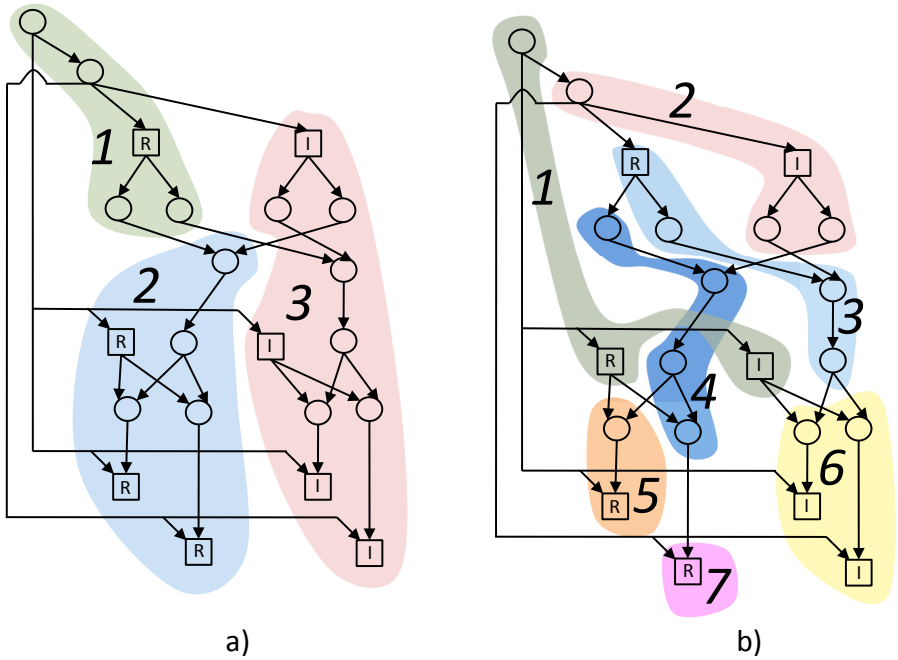


Figure 5.14. fft partition using a) iterative/exact and b) greedy methodologies, considering $Maxsize = 11$, $MaxMems = 5$, $MaxDepth = \infty$. Iterative and exact partitioning result in 3 cuts, greedy partitioning in 7.

5.5.5 Greedy partitioning

An even faster strategy is the greedy one, which assigns each node exactly once according to some metrics. In particular, a cluster-based partitioning algorithm, published by Purna and Bhatia [1999], is described here.

Greedy algorithms can achieve linear complexity; the above mentioned cluster-based algorithm is $O(|E|+|V|)$. It is the best performing algorithm among the ones investigated by Purna and Bhatia [1999].

Cluster-based partitioning performs a top-down sweep of the application DFG, and schedules to a cut S_i the “ready" node with maximum depth. At each iteration, the ready list is updated, adding those nodes whose predecessors have already been scheduled. The algorithm adds nodes to a cut until constraints are not violated; when this happens, the violating node is assigned to a new cut.

5.6 Partitioning experimental evaluation

To compare the proposed partitioning methodologies, kernels were extracted from the EEMBC benchmark suite (EEMBC [1997]); their characteristics are summarized in Table 5.6.

Three round of experiments were conducted, varying the requirement relative to 1) maximum storage, 2) control logic and 3) computation capability, respectively (Figures 5.15 – 5.17). Executing its implementation on a standard computer, the iterative algorithm converged at

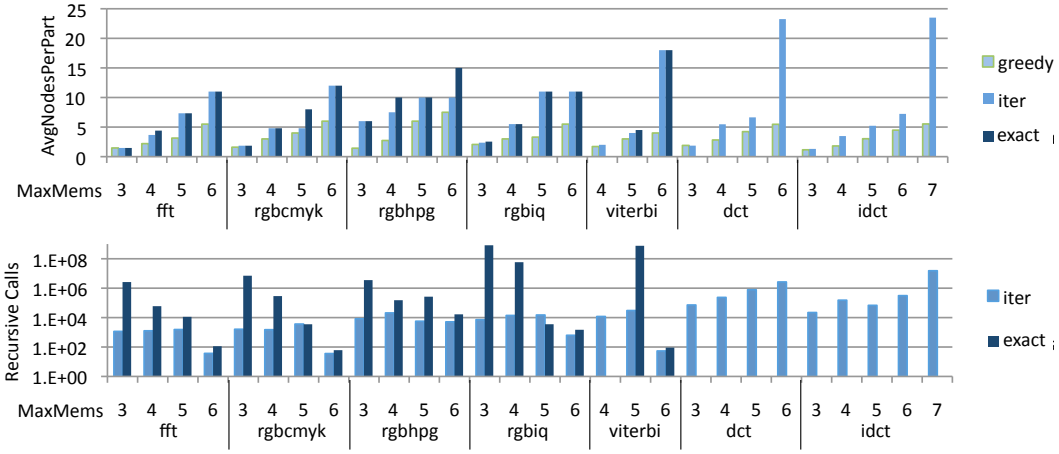


Figure 5.15. Top: partitioning quality using exact, iterative and greedy algorithms. Bottom: efforts required to reach solution using iterative and exact algorithms. Varying $MaxMems$ with $Maxsize = |V|/2$, $MaxDepth = \infty$.

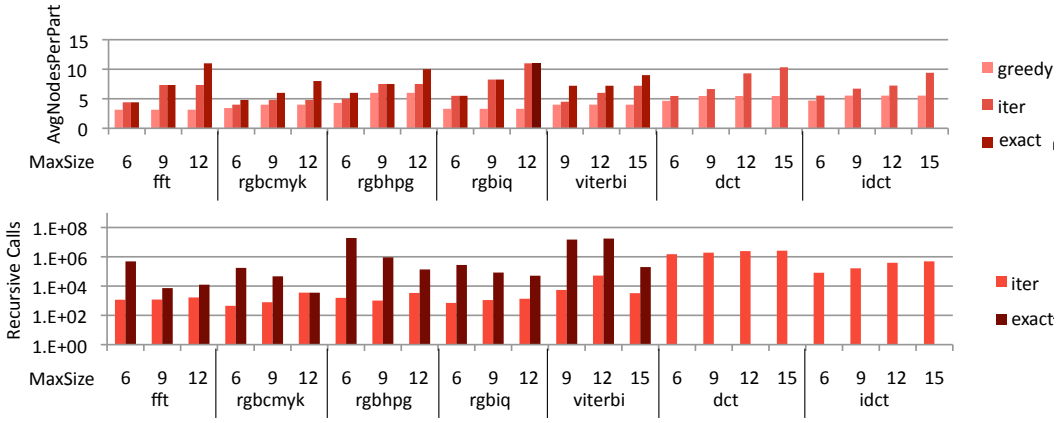


Figure 5.16. Top: partitioning quality using exact, iterative and greedy algorithms. Bottom: efforts required to reach solution using iterative and exact algorithms. Varying $Maxsize$ with $MaxMems = 5$ for all benchmarks except viterbi, dct ($MaxMems = 6$) and idct ($MaxMems = 7$). $MaxDepth = \infty$.

most in a matter of seconds; on the contrary, it was not possible to obtain exact solutions in a reasonable time for the two most complex kernels (dct and idct).

When both exact and iterative did complete, results were similar, in many cases identical; identified partitions where along the lines of solutions an expert programmer would identify, as the dct partition obtained by the iterative algorithm presented in Figure 5.18 illustrates.

On the other hand, in all but the simplest cases, the greedy methodology trailed well behind the ones based on recursive searches, resulting in smaller and more numerous sub-kernels. A graphical comparison of the methods, presented in Figure 5.14, shows how the lack of flexi-

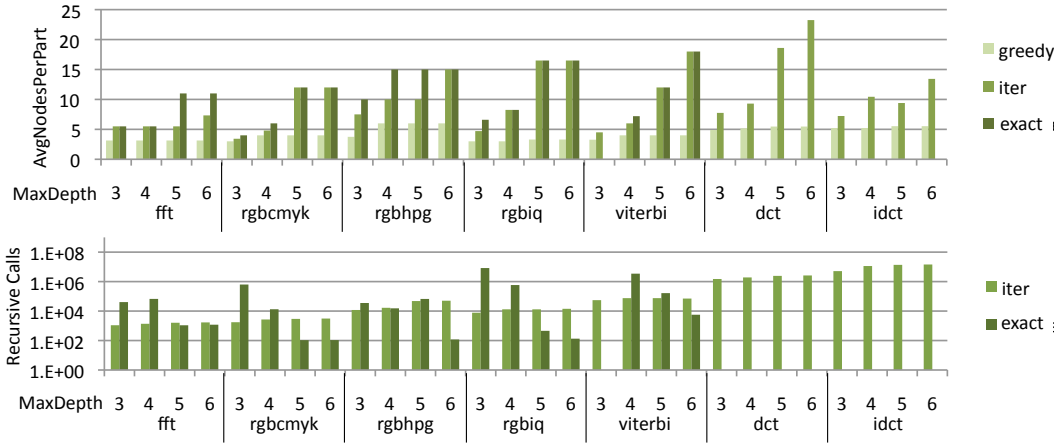


Figure 5.17. Top: partitioning quality using exact, iterative and greedy algorithms. Bottom: efforts required to reach solution using iterative and exact algorithms. Varying $MaxDepth$ with $MaxMems = 5$ for all benchmarks except viterbi, dct ($MaxMems = 6$) and idct ($MaxMems = 7$). $Maxsize = \infty$.

bility of the greedy approach leads to a much worse partition given the same constraints with respect to the iterative and exact solutions. The greedy cluster-based algorithm was particularly ineffectual when big, complex cuts could be identified and exploited, as is the case of the viterbi, dct and idct kernels.

An interesting consideration can be made regarding computation time: while exact partitioning converged quite fast in a few selected cases, it was not able to do it *consistently*, presenting a hugely different required effort in different settings. Particularly demanding are searches presenting a big gap between the upper threshold in number of cuts ($\lceil(|V|/MaxSize)\rceil$) and the actual cuts necessary for a valid solution. In Figure 5.15, the experiments relative to rgbiq with $MaxMems = 3$ exemplifies this effect.

Table 5.7, second column, compares the relative size of cuts obtained by the greedy and the iterative methodologies, subdivided by benchmark and aggregated on all performed experiments. The metric is computed as

$$(Avg_{S\_size}(iter) - Avg_{S\_size}(greedy))/Avg_{S\_size}(greedy).$$

It can be noticed that cuts obtained by iterative partitionings are on average twice the size of the ones identified by a cluster-based one, and as much as 172% bigger in the case of rgbiq. Comparing in a similar way exact and iterative partitionings results in just 12% difference in average size of cuts (and only 3% in the best case).

5.7 Conclusion

This chapter introduced methodologies to map complex data-flow graph, extracted from computational kernels, onto coarse grained reconfigurable arrays. The problem is tackled from two points of view: on one side, slack-aware scheduling is introduced to allow for better utilization of resources, leveraging registered and unregistered connections among CGRA tiles.

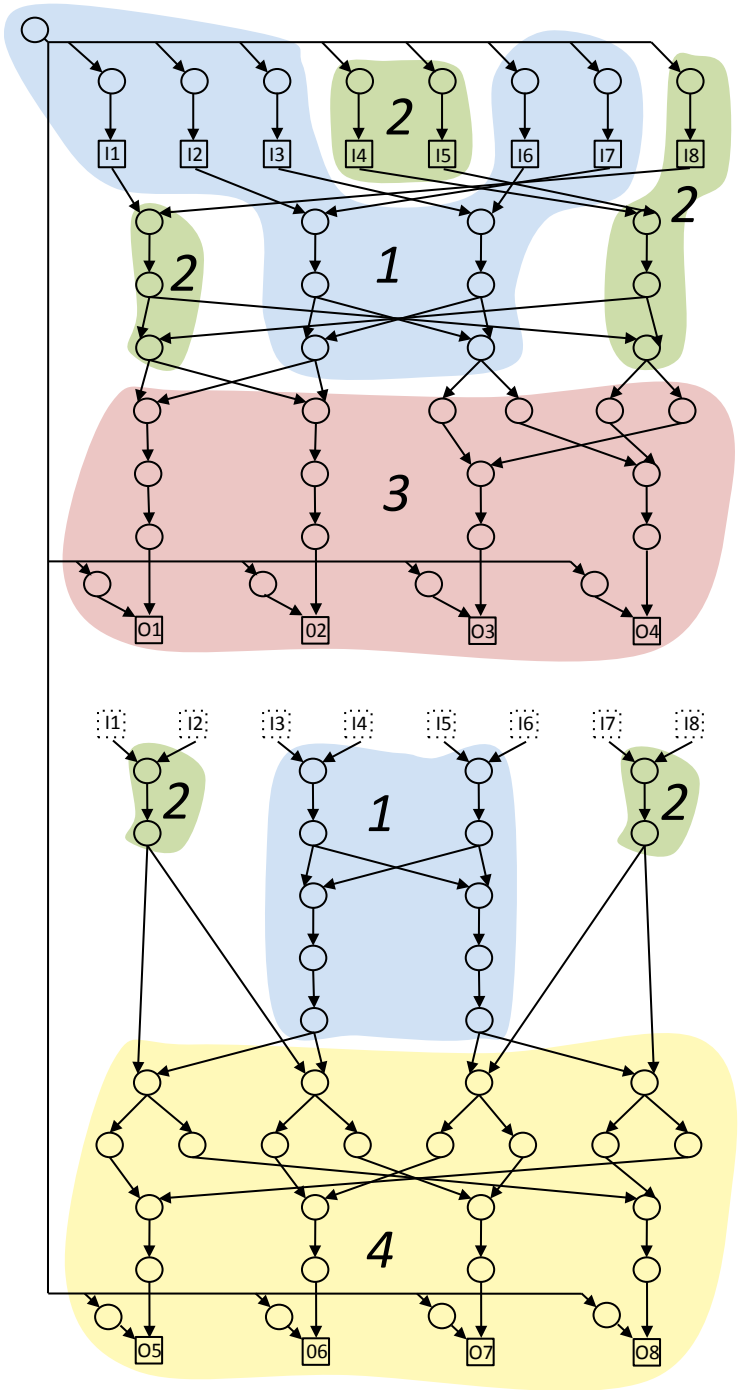


Figure 5.18. Partition of dct, iterative strategy, with $Maxsize = 30$, $MaxMems = 7$, $MaxDepth = 10$.

Benchmark	iterative/greedy(%)	iterative/exact (%)
fft	89	-12
rgbcmyk	59	-15
rgbhpg	89	-17
rgbiq	172	-3
viterbi	113	-12
dct	100	-
idct	66	-
Average	**98**	**-12**

Table 5.7. Relative cut size comparison aggregated by benchmark

On the other, a novel loop fission technique is detailed to partition complex kernels into cuts according to architectural constraints.

Slack-awareness leverages differences in computation time to allow for computation and routing operations to be chained in the same clock cycle, increasing schedulability and execution performance for coarse grained meshes supporting either spatial mapping or modulo scheduling. It is particularly beneficial in case of meshes composed by heterogeneous elements and/or complex cells, which most likely presents differences in actual critical path depending on cell type and performed operation.

The partitioning strategy is based on recursive searches over abstract trees, in which each branch considers including or excluding an operation from a sub-kernel. The employed approach is inspired by a previous work on instruction set extensions (Pozzi et al. [2006]), modified to tackle the different scenario of efficient loop fission of kernels, in the context of systems comprising a CGRA accelerator. The problem of single sub-kernel identification is solved optimally, taking into account architectural bounds derived by limited computation, control and memory resources present in coarse grained meshes.

Sub-kernel identification is then extended, both exactly and iteratively, to kernel partition: completely covering the application DFG assigning every kernel operation to sub-kernels. Experimental evidence shows that iterative partitionings result in average sub-kernel size that is only marginally smaller than in the exact case, and twice the size than the one resulting by applying a State-of-the-Art cluster-based greedy algorithm (Purna and Bhatia [1999]). Moreover, the low computational complexity of the iterative partitioning, with respect to the exact one, makes it applicable to more complex cases.

Chapter 6

System Integration: EGRA as Intelligent Memory

6.1 Introduction

Among the factors to be taken into account when devising a compact, efficient coarse grained reconfigurable accelerator, tight system integration is of paramount importance. The solution proposed in this chapter interfaces the coarse grained reconfigurable array template (the EGRA) with a host general purpose microprocessor, employing both a custom functional unit interface, to configure the accelerator, and memory mapped ports, to store kernels' datasets.

The approach combines low reconfiguration overhead and computing performance with the flexibility needed for an architectural template – as opposed to a single architecture – to be embedded in a computing system. Experimental evidence shows that RTL-level Hw-Sw co-simulation of EGRA-accelerated systems results in up to 13x speedups over non-accelerated benchmark applications.

Integration is achieved by granting the host visibility of internal EGRA memories. The host knowledge of the EGRA is, in fact, mostly limited to its memory content, an abstraction named "intelligent memory" in literature. Research on intelligent memory aims at embedding computing elements in a memory hierarchy and offloading execution of parallel tasks to them. Intelligent memories are treated as standard RAMs by the host system, and can indeed default to them for non-accelerated tasks.

While small processors and embedded FPGAs have both been proposed to support distributed computation in intelligent memories, CGRAs can be a more promising candidate for this task. This class of architectures combines flexible hardware execution, typical of FPGAs, with fast reconfiguration time and computational density, due to the arithmetic elements that constitute its cells. These characteristics are desirable in the intelligent memory scenario, in which small and computationally intensive kernels of applications are sped up.

The combination of arithmetic and storage capabilities makes the EGRA able to efficiently execute computational kernels in a parallel way, while its smart memory interface avoids data transfers to and from the reconfigurable mesh and its related overhead. An EGRA instance interfaced in this way can be seen as comprising two views: the reconfigurable mesh view composed by heterogeneous coarse grained cells executing computational kernels, and the

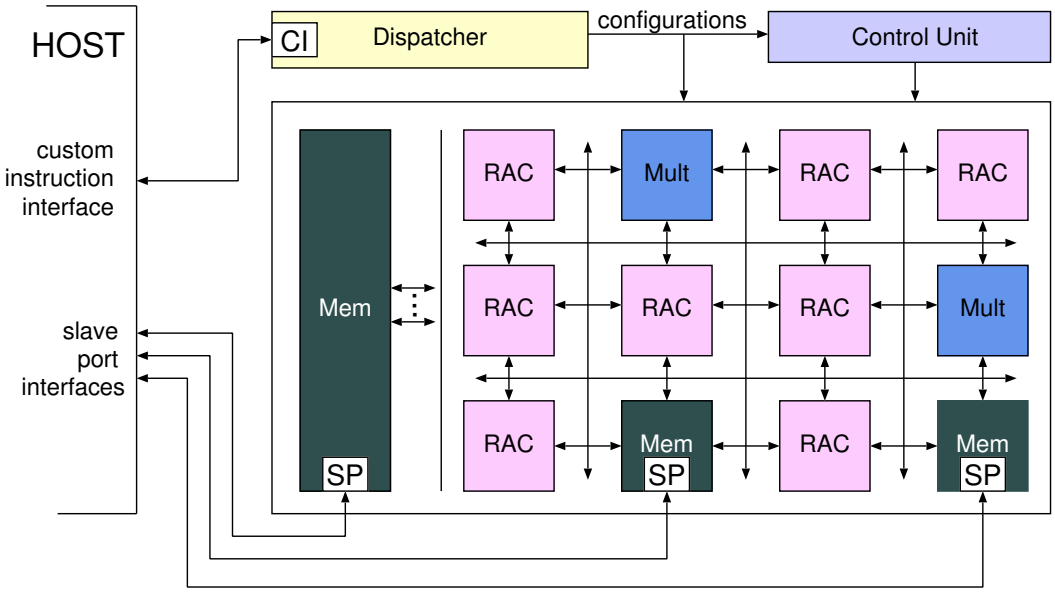


Figure 6.1. Example of an EGRA instance architecture and interface

intelligent memory view as seen by the host processor.

In this scenario, kernel execution requires minimal intervention from outside the reconfigurable array: the microprocessor host being aware only of "intelligent" memory regions where data manipulations can take place according to some reprogrammable functionality, that can be streamed in (and activated on) the reconfigurable mesh.

The chapter presents a study of expression grained arrays integration as intelligent memories, made feasible by the EGRA template flexibility and the developed hardware/software co-simulation environment. The template features and the architectural parameters, customizable at design time, are detailed in Chapter 4, with integration and co-simulation solutions being the focus of this chapter. The framework allows for RTL simulation of systems comprising a RISC host and an EGRA instance, embedded as components in a System on a Chip (SoC).

The contribution of the chapter is two-fold:

- An intelligent memory scheme is proposed for integrating a Corse-Grained Reconfigurable Array template with a host system. The approach combines low-overhead reconfiguration and the flexibility needed to interface a wide variety of possible EGRA instances.
- A hardware/software co-simulation framework is detailed, able to investigate systems comprising EGRA accelerators and their performance when executing whole benchmark applications, including reconfiguration overhead and non-kernel parts executed by the host.

The reminder of this chapter is structured as follows: Section 6.2 compares the proposed methodology with related research in the field, Section 6.3 details the proposed EGRA-host

interface, Section 6.4 describes the overall framework developed to evaluate accelerator instances. Section 6.5 shows results on achieved speedups, and Section 6.6 concludes the chapter.

6.2 Related Work

Integration of reconfigurable architectural elements to speed up computation in SoC is a fervent research topic. In particular, different solutions have been proposed to overcome the intrinsic limitation of read and write ports available on host processors, which constitutes a bottleneck with respect to the parallel nature of hardware execution.

Cong et al. [2005] have proposed the use of shadow registers to make more inputs available to ad-hoc functional units; the solution does not scale over a very limited number of registers and does not address multiple outputs, making it nonetheless viable for mapping basic blocks. A similar strategy is applied by the Chimaera architecture described by Hauck et al. [2004], that replicates a subset of the host register file to make many-inputs operations possible.

In the CGRA field, the host-accelerator interface is usually implemented at the register file level (as in ADRES, Mei et al. [2004]) or with dedicated communication buffers (MorphoSys, Lee et al. [2000], PiCoGa, Campi et al. [2007]). The former approach does not scale well as the register file size and port count increases with application complexity, while the latter one suffers from unavoidable time and area overheads as data is transferred between the host and the accelerator memory.

A different perspective is considered by intelligent memory architectures, in which some computational capabilities are directly embedded in RAM structures, as in the IRAM case (Patterson et al. [1997]). This memory-centric approach eliminates the need for dedicated buffers and related data transfers. FlexRAM (Kang et al. [1999]) uses small integer processors to be integrated in memory banks, while ActivePages (Oskin et al. [1998]) employs embedded FPGAs. We suggest that CGRAs are instead better candidates for this type of task if arithmetic operations have to be performed, as they marry efficient hardware execution with a much faster reconfiguration time and a smaller area than fine grained arrays. In fact, CGRAs ability to map arithmetic functions (instead of boolean ones) in their building blocks enables efficient execution of Data Flow Graphs of computational kernels.

CGRAs proposed so far tend to have a fixed structure; the EGRA template instead, thanks to its parametric nature, enables to quantitatively explore many aspects of CGRAs' architectural design space, as is done by the studies presented in Chapter 4. Given the widely different EGRA instances that can be possibly generated, the integration methodology introduced here aims at being suitable for accelerators with varying degrees of complexity.

To investigate performance of EGRA instances when integrated as intelligent memories in a SoC, features of the hardware/software co-simulation environment provided by SOPCBuilder (Altera [2010]) and Menthor Graphics Modelsim (Altera [2008]) is leveraged. The scope of the concepts presented is anyway not limited to a specific vendor toolchain, and can be adapted to any system whose host that can be expanded with variable-latency custom functional units.

6.3 EGRA-host communication

Envisioning a unified strategy to interface EGRA instances into a computing system presents numerous challenges. First of all, the interface scheme must be flexible enough to accommo-

EGRA elements	Design parameters
Mesh	Size (rows and columns) Cell type in each location Number of contexts
RAC cells	Datapath depth Number of ALUs in each stage ALUs supported operations in each stage Flags support (per stage) Number of embedded constants
Multiplier cells	Sign-Unsigned multiplication support
Memory cells	Size Number of read-write ports Supported addressing modes
Scratchpad memory	Size Number of read-write ports Supported addressing modes
Control Unit	Control lines number

Table 6.1. Synopsis of machine description parameters

date widely different configurations to cope with the template parametric nature. Secondly, communication between the sequential non-kernel part of an application, running on a host system, and its parallelized kernels, executing on an EGRA instance, must be addressed, minimizing data transfers from both sides. Finally, as discussed below, the difference in access patterns between configuration transfers and data transfers must be considered.

To configure an EGRA instance, cells must be programmed with the appropriate functionality to be executed at each clock cycle. Cells provide a local configuration memory to store this information, which contains a number of context words, activated in the proper order during kernel execution (the architecture of a generic cell is illustrated in Figure 6.2). Context words include control of the interface part of a cell (e.g.: which are its inputs, if row/column buses must be driven), and that of its internal datapath, specific for each cell type. For RAC cells, this part dictates operations to be executed by each ALU and the routing among RAC rows performed by switchboxes, while in the case of memory cells, it states if a read or write operation should be executed. It can be observed that the lenght of a context word varies depending on the cell type and the values of related architectural parameters, which are described in Table 6.1. Moreover, only some contexts (of some cells) actively participate in a kernel execution and thus need to be configured.

These two factors lead to a scattered access arrangement of context memories during an EGRA instance configuration phase. Accesses to embedded memories, on the other hand, present a more regular behaviour, their elements being usually read or written in sequence by the host. The scenario calls for a *differentiation between configuration transfers and data memory accesses*.

In the proposed implementation, illustrated graphically in Figure 6.1, EGRAs reconfiguration is inspired by FPGAs bitstream download: configurations are transferred serially, using a

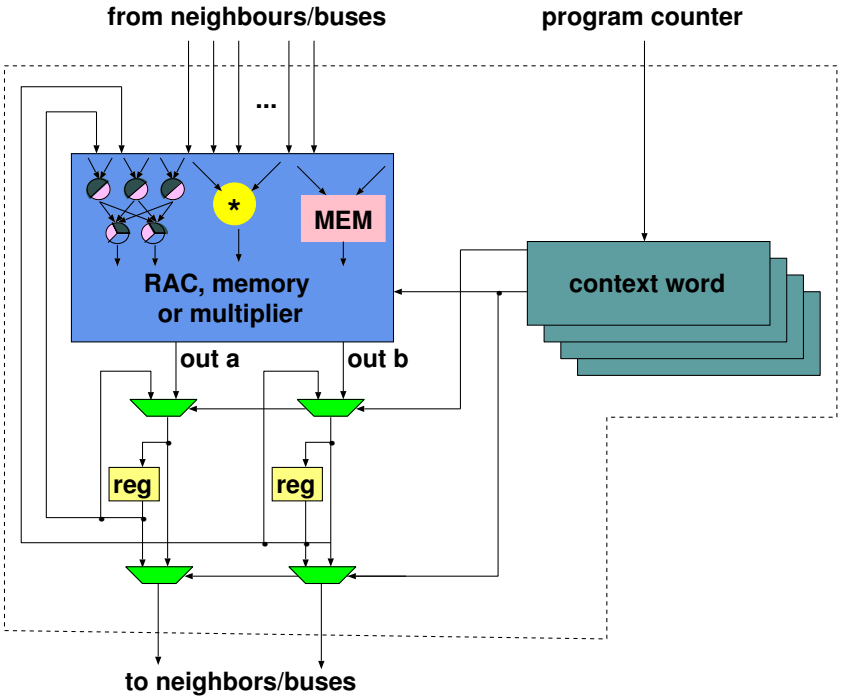


Figure 6.2. Generic EGRA cell block scheme

custom instruction interface, to a hardware dispatcher. The dispatcher in turn routes them to the proper location in the cells' configuration memories. By embedding the destination context word index in the program bitstream, contexts can be programmed independently, allowing for partial reconfiguration, ultimately reducing configuration time.

Working data sets (input/output arrays, look-up tables) are instead interfaced with slave ports, mapped in the host system addressing space. Software applications can then access data directly. In this respect, *the host views the reconfigurable accelerator as an intelligent memory*, able to transform data as opposed to just store it. RISC processors can access a single data item at a time; instead, during EGRA execution, multiple values can be read and written from memories, greatly contributing to overall speedup. Memories can be interfaced with data width of 8, 16 and/or 32 bits, corresponding to arrays of char, short or long values in C language.

6.4 Hardware-Software platform

To test performance of EGRA instances, a hardware/software co-simulation framework was developed, its block scheme being illustrated in Figure 6.3. Commercially available tools are leveraged whenever possible: SOPCBuilder from Altera is used to generate test platforms (comprising custom CGRA accelerators), while the NIOSII software development environment, also from Altera, is employed to compile applications for the systems. Menthor Graphics Modelsim is used to simulate execution, exploiting the SOPCBuilder-Modelsim interface described in Altera [2008]. Finally, critical path and area occupation of EGRA instances are derived with

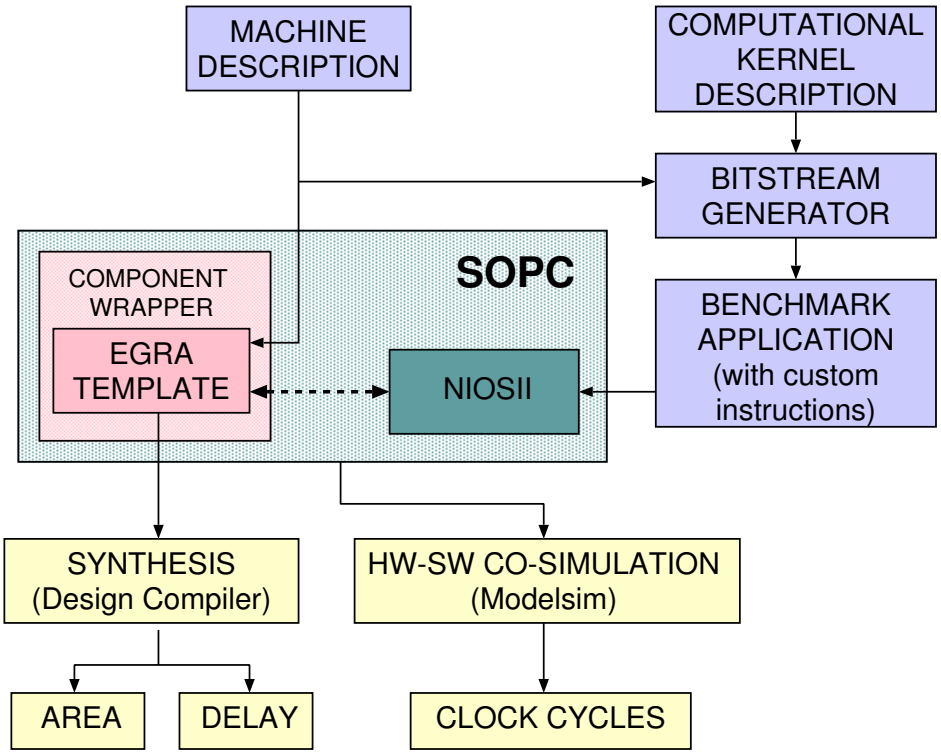


Figure 6.3. Block scheme of the EGRA Hw-Sw co-simulation framework

Design Compiler from Synopsys.

Here the custom hardware components and the software ones, developed to generate and test EGRA-enabled systems, are described.

6.4.1 Software components

As outlined in Figure 6.3, inputs of the framework are the machine description of the instance to be evaluated and the description of the scheduled computational kernel(s) to be mapped. Scheduling can be either hand-optimized, as it is done for this study, or derived from a scheduling framework such as the one described in Chapter 5. The description is parsed and transformed in a C header file containing the appropriate bitstream, implemented as a series of control words to be transfered into the instance, controlling execution of the desired functionality.

The header file is then included in the benchmark application, and computational kernels are substituted by configuration code and by a single custom variable-latency instruction firing execution. Figure 6.4a illustrates the original code of a kernel contained in the histogram benchmark, Figure 6.4b the code controlling the same functionality when mapped on an EGRA instance. It can be noticed that the latter does not contain an explicit loop, as looping is managed and parallelized by the reconfigurable hardware; indeed, the actual loop execution is composed by just one instruction in this scenario.

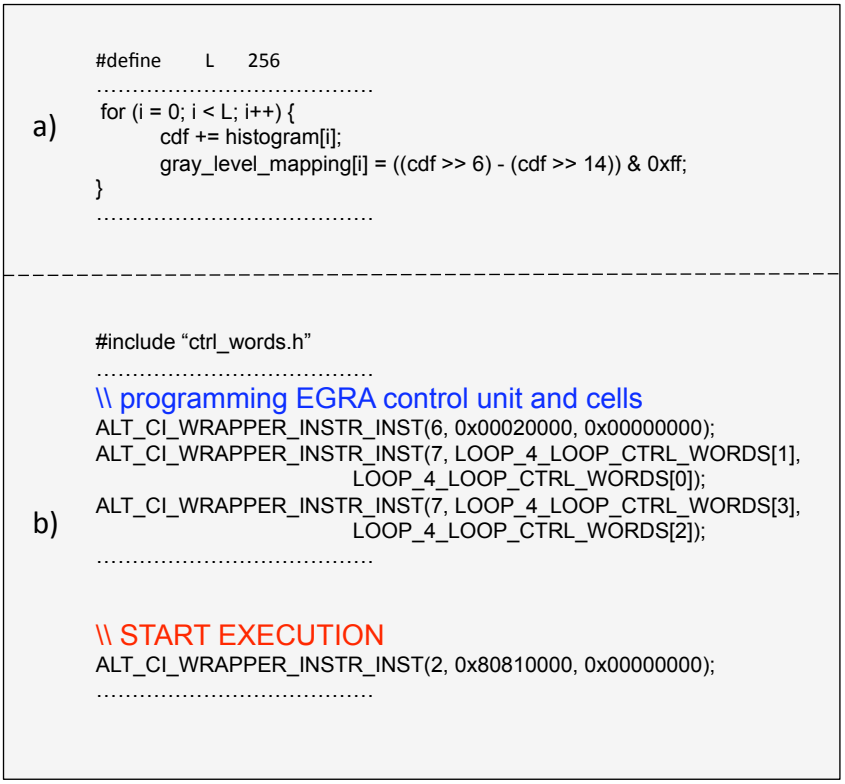


Figure 6.4. Example of kernel computation (from histogram benchmark): original C code (a), execution on EGRA instance (b).

6.4.2 Hardware components

EGRA instances are interfaced as components in a SoC, instantiating them inside a component wrapper which connects the EGRA outer signals with the SoC bus and the custom instruction interface. Role of the wrapper is to make deeper architectural levels of instances agnostic of the system conventions; in the present study, the wrapper is the only component that has to be manually edited to adapt it to different instances. Its structure is anyway trivial, mostly dealing with signal renaming, and its generation could be possibly automated in a future refinement of the co-simulation framework.

Apart from the EGRA instance, all other components used in the SoC are standard SOPC-Builder ones: NIOSII/f high-end RISC processor is used to host non-kernel, non-accelerated computation and to program and control the EGRA, while on-chip memory with a single-cycle latency hosts program code and data; finally, a performance counter component is used to collect the number of cycles spent in different phases of execution.

Benchmark	mesh size	cells type[a]	RACs structure[b]	memory cells size (bytes)	scrathpad memory size (bytes)
histogram	2x3	4 C 2 M	2-ASL 2-AL	512 x 2	16k
bubblesort	2x4	5 C 1 MU 2 M	3-ASL 2-AL	512 x 2	0
median(1)	2x4	7 C 1 M	3-AL 2-L	64	0
median(2)	2x4	7 C 1 M	3-AL 2-L	64	32k
generic	**2x5**	**7C 1 MU 2M**	**3-ASL 2-AL**	**512 x 2**	**32k**

[a] C: RACs; MU: multipliers; M: memory cells

[b] ALUs in each row and supported operations. A: arithmetic; S: shift-rotate; L: bitwise logical. For example, 2-AL represents a row with 2 ALUs supporting arithmetic and logic operations, but no shifts/rotates.

Table 6.2. Characteristics of EGRAs optimized for different benchmarks

benchmark		Sw execution Cycles	EGRA execution Cycles	EGRA execution Config. overhead	Speedup
histogram	Whole b.mark	810 751	82 854	332	9.79
	Loop1	115 613	16 390	53	7.05
	Loop2	2 059	262	39	7.86
	Loop3	344 089	32 777	88	10.50
	Loop4	4 883	265	77	18.43
	Loop5	344 083	32 774	75	10.50
bubblesort	Whole b.mark	3 805 301	524 121	288	7.26
	Loop1	4 117	262	79	15.71
	Loop2	3 798 075	523 271	135	7.26
	Loop3	3 090	263	74	11.75
median(1)	Whole b.mark	773 342	77 132	171	10.03
	Loop2	748 671	52 281	171	14.32
median(2)	Whole b.mark	773 342	59 854	338	12.92
	Loop1	15 408	3 084	105	5.00
	Loop2	748 671	55 360	159	13.52
	Loop3	9238	1 032	74	8.95

Table 6.3. Speedups over kernels execution

6.5 Experimental results

To measure performance of systems embedding EGRA instances, three benchmark applications from the MIT-bitwise suite (Bitwise [1999]) were studied: bubblesort, histogram and median. For every application, intensive loops were extracted—five are present in the histogram benchmark, while bubblesort and median have three. Kernels were then mapped on EGRA instances and simulated to compare the number of cycles it takes to execute benchmarks on a NIOS-only system, with respect to a NIOS+EGRA system.

Following the experimental methodology employed for EGRA design space exploration

Benchmark	mesh area (mm^2)	scratchpad area (mm^2)	max. clock freq. (MHz)
histogram	0.45	6.53	507
bubblesort	0.72	—	565
median(1)	0.35	—	598
median(2)	0.35	13.07	598
generic	**1.08**	**17.54**	**505**

Table 6.4. Area and critical path of EGRAs optimized for different benchmarks

(Chapter 4), considered instances included a custom-tailored one for each application, and additionally a single EGRA instance (called *generic*) to accommodate *all* applications. The characteristics of these instances can be seen in Table 6.2. Since the median benchmark requires an expensive 32kb memory to map two non-critical loops, two implementations are actually proposed and shown in the table: median(1) accelerates just the most critical loop, while median(2) employs a scratchpad and accelerates all three loops.

Looking at the first four rows of Table 6.2, one can see the mesh size employed by each benchmark, the distribution of cell types (RACs, multipliers, memories), the RACs' structure (note that all RACs featured in these istances consist of 4 to 5 ALUs, arranged in two levels), and the size of memories. The last row corresponds to the generic instance. Its characteristics are a superset of the instances above, but still fitting in a very contained area (instance areas are shown later in this section).

Applications were run on a fully featured NiosII/f system simulated with Menthor Graphics Modelsim. Execution cycles were counted, and compared, for NIOS-only versus NIOS+EGRA, and are reported in Table 6.3. The table shows resulting execution clock cycles, as well as speedup, of EGRA-accelerated execution over software one for every loop and for the three whole benchmarks, as well as the reconfiguration overhead (in cycles). It is interesting to note that substantial speedups are achieved, with a negligible reconfiguration overhead. This highlights the importance of coarse-grain versus fine-grain reconfiguration: a reduced flexibility (leading to extremely small bitstreams) is traded for an increased speedup.

To measure the benefit of an intelligent memory approach (referred as IMem in Figures 6.6 and 6.5), speedups obtained by two less capable solutions were also investigated: in the first case (appl_dma) data was transferred at application boundaries, while in the second (loop_dma) data was explicitly copied in and out of the reconfigurable fabric at every kernel execution. Resulting overheads greatly reduced performance in histogram and median(2), while bubblesort and median(1) are less affected, being dominated by computation intensive loops with small memory footprints. Speedups for every loop, in the intelligent memory and memory buffer settings, are presented in Figure 6.5.

Finally, since comparing clock counts while ignoring latencies can be a pitfall, the template RTL implementations were synthesized using Synopsys Design Compiler and TSMC 90nm front-end libraries. Results are shown in Table 6.4. This table complements the speedup results given earlier in this section, by showing that an execution frequency of over 500 MHz was achieved on all instances.

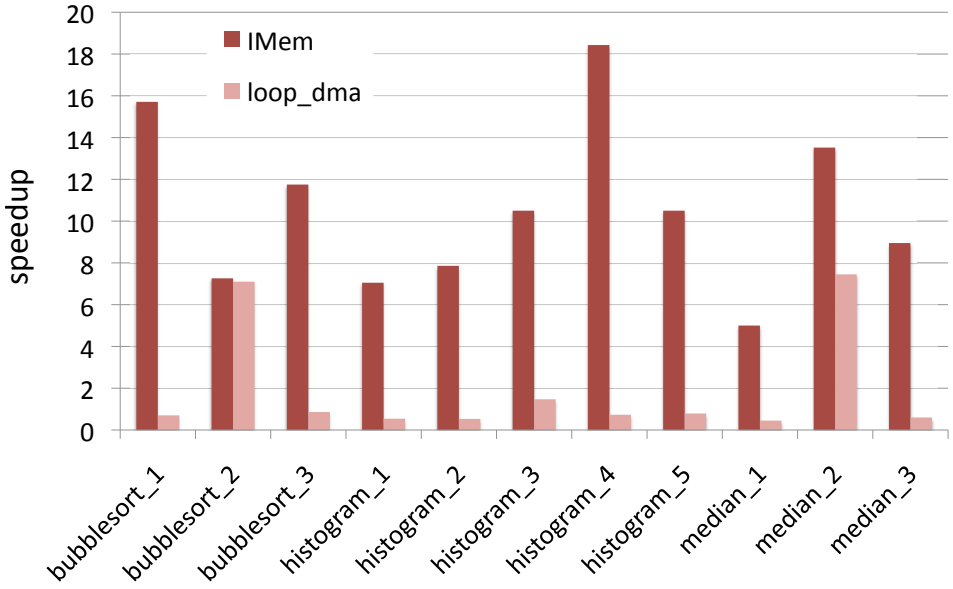


Figure 6.5. IMem vs. explicit data transfer speedup over benchmark kernels

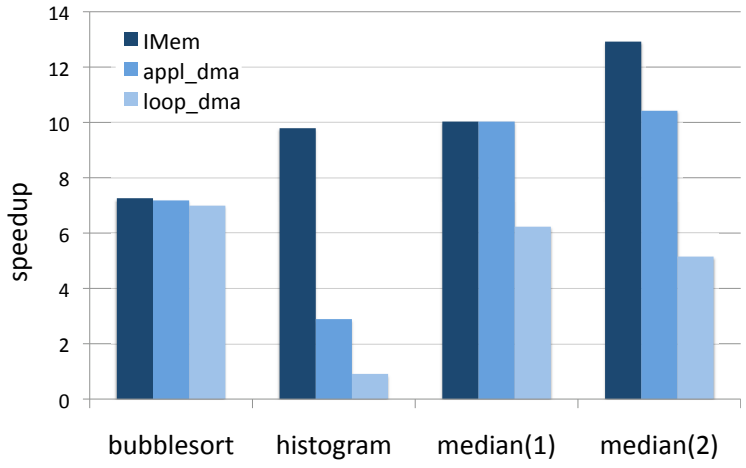


Figure 6.6. IMem vs. explicit data transfer speedup over benchmark applications

6.6 Conclusion

This chapter presented and investigated the integration of a Coarse Grained Reconfigurable Architecture template in a computing system as an intelligent memory.

The proposed approach decouples the internal operation of the reconfigurable mesh from its external view, as the accelerator transparently transforms data sets on smart memory regions. The host-accelerator interface differentiates between configuration operations, where data is transferred serially to mesh elements, and execution of accelerated functions, happening transparently in the shared host-accelerator memory. The presented scheme both minimizes configuration overhead and minimizes the need for expensive data transfers to and from the accelerator.

Moreover, the scalability of the approach allows for different architectural instances to be seamlessly integrated, matching the flexibility of the developed architectural template and enabling architectural exploration over different design dimensions.

To investigate performance of benchmark applications over systems embedding EGRA instances, a comprehensive hardware/software co-simulation framework has been developed, enabling inspection of application execution at the RTL level, and allowing precise quantification of execution times.

Accelerated systems allowed for speedups (over whole applications) of up to 13x, while maintaining aggressive critical path and area footprint of parametrically generated EGRA instances.

Chapter 7

Concluding Remarks and Possible Extensions

The Coarse Grained Reconfigurable Array paradigm poses novel challenges in many aspects, spanning the whole hardware stack needed to generate meshes and execute computational kernels, and the software stack that automatically compiles applications onto the reconfigurable hardware.

The thesis presented the following innovations in the field:

- A CGRA architectural template (the *Expression Grained Reconfigurable Array*) was introduced. Thanks to its parametric nature, the EGRA template enables exploration of the CGRA design space. Three such explorations were presented, the first one focusing on comparative evaluations of different implementations for multi-ALU computational cells, the second on integration of heterogeneous elements with diverse computation and storage capabilities, and the third on solutions for embedding memories inside meshes and/or on their side.

- Novel application mapping techniques were described. On one side, slack-aware scheduling was presented; the scheduler leverages heterogenous critical paths of different operations to combinatorially chain computation and routing during execution, resulting in better schedulability and run-time performance of applications. On the other, a novel partitioning algorithm was detailed, able to efficiently divide complex computational kernels into cuts, schedulable on constrained hardware resources.

- Integration of EGRA instances in complete computing systems was illustrated. The presented hardware/software co-simulation framework allows for system-wide evaluation of SOCs embedding a CGRA accelerator, including the impact of non-kernel code and overhead due to reconfiguration. By decoupling configuration and data transfers, the integration scheme allows for minimization of overheads.

The EGRA architectural template was instrumental in performing the research here described. Every step of the EGRA development, and that of its surrounding ecosystem, opened up new possibilities for further research. This *enabling* aspect has been one of the most exciting aspects of my PhD studies. The process is never-ending, and I often found myself in the difficult position of deciding which directions to pursue. In the following paragraphs I describe some

of the open research opportunities made possible by the EGRA and by the environment built around it; I leave them as possible future work.

7.1 Local register files

As data is routed through a reconfigurable mesh, temporary values have to be stored in registers private to each cell. Registers are not fundamentally different from any other memory structure, but they present a much smaller size and they support different functionalities with respect to dedicated memory cells and/or scratchpad memories: registers contain scalar data generated by computation, while scratchpads and memory cells store arrays being processed by a computational kernel. The appropriate size of local register files, private to each cell, has been investigated by Bouwens et al. [2007] in the context of the single-ALU ADRES reconfigurable array. The local output registers present in EGRA cells takes inspiration from this study, nonetheless its applicability to a mesh including complex cells like the Reconfigurable ALU Cluster, and supporting combinatorial chaining, would need further investigation.

7.2 Energy and power consumption

The thesis focuses on execution time as a metric to evaluate different design solutions, with the assumption that execution speed can be transformed into energy savings if clever techniques like clock and power gating or dynamic frequency/voltage scaling are applied. Nonetheless, energy and power consumption can also be explicitly explored leveraging the proposed framework. As the EGRA template is synthesizable, switching activities of transistors can be annotated before synthesis, retrieving the power/energy consumed by execution of a kernel for evaluation. Trade-offs using different architectural choices or transistors libraries can than be explored and compared. Architectural features implementing power management policies can be also explored in this way.

7.3 Configurations and data transfer overhead

Relative speedup of computational kernels executing on a CGRA mesh with respect to execution on a host processor is of paramount importance to highlight CGRA effectiveness, but other factors have to be accounted for to have a complete system-wide picture. Specifically, overheads due to data and configuration transfers have to be taken in consideration, as well as the impact of code sections that are kept on the host. In this perspective, the overall speed-up obtained by an application executing on a system that embeds a CGRA is:

$$speedup = \frac{execTime(nonKernel) + configTime + dataTransferTime + execTime(kernel)}{execTime(SW)}$$

where *execTime(nonKernel)* is the execution time of non-kernel, non-accelerated code, *configTime* and *dataTransferTime* account for the overhead due to configuration and input/output data transfer respectively, and *execTime(kernel)* is the execution time of kernels executing on the CGRA; *execTime(SW)* is the execution time of the application when running on the host system only.

The hardware/software co-simulation framework described in this thesis can be used as a virtual prototype to investigate this issue at the architectural level, as it makes possible to compile and simulate applications on systems comprising a microprocessor and an EGRA instance, observing the impact of the different execution phases and different interface mechanisms. Such a study would be a generalization of the one proposed in Chapter 6.

Data transfers minimization should be the goal of any efficient system. The problem is not trivial especially when applications are split into sub-kernels, as proposed in Chapter 5. Effective strategies to schedule sub-kernels to minimize pressure on the memory systems, and avoid as much as possible data transfers out of CGRA internal memories, is a natural extension of the thesis work.

7.4 Architectural meta-model

The EGRA is an architectural template defined at the RTL level. This characteristic allows for a detailed exploration of its instances and for their accurate simulation. The drawback of the approach is the long turnaround time needed to explore the design space. Hardware synthesis, employed to extract area and critical paths figures, is especially time consuming, so that an EGRA meta-model (similar in principle to the one proposed by Mariani et al. [2009]) could greatly broaden the design dimensions that can be explored in a given amount of time, at the price of reduced accuracy of results. A first, fast pruning of a design space using the meta-model followed by a more detailed one using the RTL description could result in both accurate and rapid explorations.

Bibliography

Ahn, M., Yoon, J., Paek, Y., Kim, Y., Kiemb, M. and Choi, K. [2006]. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, European Design and Automation Association 3001 Leuven, Belgium, pp. 363–368.

Altera [2008]. *Simulating NiosII Embedded Processor Designs*.
URL: *http://www.altera.com/literature/an/an351.pdf*

Altera [2010]. *SOPC Builder User Guide*.
URL: *www.altera.mobi/literature/ug/ug_sopc_builder.pdf*

Altera [2011]. Altera website: www.altera.com.
URL: *http://www.altera.com/*

AMD Corp. [2005]. *AMD Multi-core White Paper*.
URL: *www.sun.com/emrkt/innercircle/newsletter/0505multicore_wp.pdf*

Ansaloni, G., Bonzini, P. and Pozzi, L. [2008a]. Design and architectural exploration of expression-grained reconfigurable arrays, *Proceedings of the 6th Symposium on Application Specific Processors*, Anaheim, CA, pp. 26–33.

Ansaloni, G., Bonzini, P. and Pozzi, L. [2008b]. Evaluating flexible CGRA cells, *Aether-Morpheus Workshop*, Lugano, Switzerland.

Ansaloni, G., Bonzini, P. and Pozzi, L. [2009]. Heterogeneous coarse-grained processing elements: a template architecture for embedded processing acceleration, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Nice, France, pp. 542–547.

Ansaloni, G., Bonzini, P. and Pozzi, L. [2011]. EGRA: a Coarse Grained Reconfigurable Architectural Template, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **19**(6): 1062–1074.

Ansaloni, G., Najvirt, R. and Pozzi, L. [2008]. Interfacing a CGRA template as an intelligent memory, *Technical report*, University of Lugano, Switzerland.

Ansaloni, G. and Pozzi, L. [2011]. An efficient loop partitioning algorithm for coarse grained reconfigurable arrays, submitted paper, *Proceedings of the 9th Symposium on Application Specific Processors*.

Ansaloni, G., Tanimura, K., Pozzi, L. and Dutt, N. [2011]. Slack-aware scheduling on coarse grained reconfigurable arrays, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Grenoble, France, pp. 1–4.

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D. and Yelick, K. [2009]. A view of the parallel computing landscape, *Commun. ACM* **52**(10): 56–67.

Atasu, K., Pozzi, L. and Ienne, P. [2003]. Automatic application-specific instruction-set extensions under microarchitectural constraints, *Proceedings of the 40th Design Automation Conference*, Anaheim, CA, pp. 256–61.

Baumgarte, V., Elhers, G., May, F., Nuckel, A., Vorback, M. and Weinhardt, M. [2003]. PACT-XPP - A Self-Reconfigurable Data Processing Architecture, *Journal of Supercomputing* **26**(2): 167–184.

Biswas, P., Dutt, N., Ienne, P. and Pozzi, L. [2006]. Automatic identification of application-specific functional units with architecturally visible storage, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich, Germany, pp. 212–217.

Bitwise [1999]. Bitwise benchmarks.
URL: *http://www.cag.lcs.mit.edu/bitwise/bitwise_benchmarks.htm*

Bonzini, P., Ansaloni, G. and Pozzi, L. [2008]. Compiling custom instructions onto expression-grained reconfigurable architectures, *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, Atlanta, GA, pp. 51–60.

Bonzini, P. and Pozzi, L. [2007]. Polynomial-time subgraph enumeration for automated instruction set extension, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Nice, France, pp. 1331–36.

Bonzini, P. and Pozzi, L. [2008]. Recurrence-aware instruction set selection for extensible embedded processors, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **16**(10).

Bouwens, F., Berekovic, M., Kanstein, A. and Gaydadjiev, G. [2007]. Architectural exploration of the ADRES coarse-grained reconfigurable array, *Reconfigurable Computing: Architectures, Tools and Applications*, Vol. 4419 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, pp. 1–13.

Brisk, P., Verma, A. K., Ienne, P. and Parandeh-Afshar, H. [2007]. Enhancing FPGA performance for arithmetic circuits, *Proceedings of the 44th Design Automation Conference*, ACM, New York, NY, USA, pp. 334–337.

Campi, F., Deledda, A., Pizzotti, M., Ciccarelli, L., Rolandi, P., Mucci, C., Lodi, A., Vitkovski, A. and Vanzolini, L. [2007]. A dynamically adaptive DSP for heterogeneous reconfigurable platforms, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 1–6.

Cardoso, J. M. P. and Weinhardt, M. [2002a]. XPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture, *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, Montpellier, France.

Cardoso, J. and Weinhardt, M. [2002b]. Xpp-vc: A c compiler with temporal partitioning for the pact-xpp architecture, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, Vol. 2438 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 207–226.

Chang, P. P., Mahlke, S. A., Chen, W. Y., Water, N. J. and Hwu, W.-m. W. [1991]. IMPACT: An architectural framework for multiple-instruction-issue processors, *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Toronto, Canada.

Chapman, B., Jost, G. and van der Pas, R. [2008]. *Using OpenMP: portable shared memory parallel programming*, The MIT press, Cambridge, Massachusetts.

Chattopadhyay, A., Chen, X., Ishebabi, H., Lupers, R., Ascheid, G. and Meyr, H. [2008]. High-level modelling and exploration of coarse-grained re-configurable architectures, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich, Germany, pp. 1334–39.

Cherepacha, D. and Lewis, D. [1996]. DP-FPGA: an FPGA architecture optimized for datapaths, *VLSI Design* **4**(4): 329–343.

Clark, N., Blome, J., Chu, M., Mahlke, S., Biles, S. and Flautner, K. [2005]. An architecture framework for transparent instruction set customization in embedded processors, *Proceedings of the 32st Annual International Symposium on Computer Architecture*, Madison, Wisconsin.

Clark, N., Kudlur, M., Park, H., Mahlke, S. and Flautner, K. [2004]. Application-specific processing on a general-purpose core via transparent instruction set customization, *MICRO 37: Proceedings of the 37th Annual International Symposium on Microarchitecture*, IEEE Computer Society, Washington, DC, USA, pp. 30–40.

Cong, J., Fan, Y., Jagannathan, A., Reinman, G. and Zhang, Z. [2005]. Instruction set extension with shadow registers for configurable processors, *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays*, pp. 99–106.

Ebeling, C., Cronquist, D. C. and Franklin, P. [1996]. RaPiD: Reconfigurable Pipelined Datapath, *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications*, Springer, Darmstadt, Germany, pp. 126–35.

EEMBC [1997]. EEMBC website.
URL: *http://www.eembc.org/*

Elsen, E., Houston, M., Vishal, V., Darve, E., Hanrahan, P. and Pande, V. [2006]. N-Body simulation on GPUs, *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*.

Fisher, C., Rennie, K., Xing, G., Berg, S. G., Bolding, K., Naegle, J. H., Parshall, D., Portnov, D., Sulejmanpasic, A. and Ebeling, C. [2001]. An emulator for exploring RaPiD configurable computing architectures, *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications*, pp. 17–26.

Galanis, M., Theodoridis, G., Tragoudas, S. and Goutis, C. [2006]. A high-performance data path for synthesizing DSP kernels, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **25**(6): 1154–1162.

Goldstein, S., Schmit, H., Moe, M., Budiu, M., Cadambi, S., Taylor, R. R. and Laufer, R. [1999]. PipeRench: A coprocessor for streaming multimedia acceleration, *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 28–39.

Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T. and Brown, R. [2001]. MiBench: A free, commercially representative embedded benchmark suite, *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, pp. 3–14.
URL: *http://www.eecs.umich.edu/mibench/Publications/MiBench.pdf*

Halfhill, T. R. [2000]. EEMBC releases first benchmarks, *Microprocessor Report* .

Hart, P., Nilsson, N. and Raphael, B. [1968]. A formal basis for the heuristic determination of minimum cost paths, *IEEE IEEE Transactions on Systems Science and Cybernetics* **4**(2): 100–107.

Hartenstein, R. [2001]. A decade of reconfigurable computing: A visionary retrospective, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 642–649.

Hatanaka, A. and Bagherzadeh, N. [2007]. A modulo scheduling algorithm for a coarse-grain reconfigurable array template, *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium, 2007*, pp. 1–8.

Hauck, S., Fry, T., Hosler, M. and Kao, J. [2004]. The Chimaera reconfigurable functional unit, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **12**(2): 206–217.

Hauck, S., Fry, T. W., Hosler, M. M. and Kao, J. P. [1997]. The Chimaera reconfigurable functional unit, *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, pp. 87–96.

Heysters, P. and Smit, G. [2003]. Mapping of DSP algorithms on the MONTIUM architecture, *Parallel and Distributed Processing Symposium*, p. 6pp.

HP Corp. [2007]. *HP-MPI User's Guide*.
URL: *docs.hp.com/en/B6060-96024/B6060-96024.pdf*

Intel Corp. [2006]. *Intel Multi-Core Processors*.
URL: *www.intel.com/technology/architecture/downloads/quad-core-06.pdf*

ITRS [2007]. International technology roadmap for semiconductors. executive summary, 2005 and 2007.
URL: *http://public.itrs.net/*

Kahng, A. [2001]. Design technology productivity in the dsm era, *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 443 –448.

Kang, Y., Huang, W., Yoo, S.-M., Keen, D., Ge, Z., Lam, V., Pattnaik, P. and Torrellas, J. [1999]. FlexRAM: toward an advanced intelligent memory system, *Proceedings of the International Conference on Computer Design*, pp. 192–201.

Kaul, M. and Vemuri, R. [1998]. Optimal temporal partitioning and synthesis for reconfigurable architectures, *Proceedings of the 35th Design Automation Conference*, Paris, France.

Kennedy, K. and Kinley, K. [1994]. Maximizing loop parallelism and improving data locality via loop fusion and distribution, *Lecture Notes in Computer Science* **768**: 301–320.

Kernighan, B. and Lin, S. [1978]. An efficient heuristic procedure for partitioning graphs, *IEEE Transactions on Computers* pp. 1064–1068.

Kim, Y., Kiemb, M., Park, C., Jung, J. and Choi, K. [2005]. Resource Sharing and Pipelining in Coarse-Grained Reconfigurable Architecture for Domain-Specific Optimization, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich, Germany, pp. 12–17.

Kim, Y., Lee, J., Shrivastava, A. and Yoon, J. [2010]. Memory-aware application mapping on coarse-grained reconfigurable arrays, *International conference on High-Performance Embedded Architectures and Compilers*, pp. 171–185.

Kuhnle, M., Hubner, M., Becker, J., Deledda, A., Mucci, C., Ries, F., Coppola, A. M., Pieralisi, L., Locatelli, R., Maruccia, G., DeMarco, T. and Campi, F. [2008]. An interconnect strategy for a heterogeneous, reconfigurable soc, *IEEE Design and Test of Computers* **25**: 442–451.

Kusse, E. and Rabaey, J. [1998]. Low-energy embedded FPGA structures, *Proceedings of the 1998 International Symposium on Low Power Electronics and Design.*, pp. 155–159.

Lanuzza, M., Perri, S. and Corsonello, P. [2007]. MORA: A New Coarse-Grain Reconfigurable Array for High Throughput Multimedia Processing, *Proceedings of the 7th Workshop on Simulation, Architectures and Modeling of Systems*, Samos, Greece, pp. 159–168.

Lee, J., Choi, K. and Dutt, N. [2003]. An algorithm for mapping loops onto coarse-grained reconfigurable architectures, *Proceedings of the 2003 ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 183–188.

Lee, M.-H., Singh, H., Lu, G., Bagherzadeh, N., Kurdahi, F. J., Filho, E. M. C. and Alves, V. C. [2000]. Design and implementation of the MorphoSys reconfigurable computing processor, *Journal of VLSI Signal Processing Systems* **24**(2–3): 147–164.

Liu, H. and Wong, D. [1998]. Network flow based circuit partitioning for time-multiplexed FPGAs, *Proceedings of the International Conference on Computer Aided Design*, New York, NY, USA, pp. 497 – 504.

Magarshack, P. and Paulin, P. [2003]. System-on-chip beyond the nanometer wall, *Proceedings of the 40th Design Automation Conference*, pp. 419–424.

Mariani, G., Palermo, G., Slivano, C. and Zaccaria, V. [2009]. Meta-model assisted optimization for design space exploration of multi-processor systems-on-chip, *Proceedings of 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD09*, pp. 383–389.

Mei, B., Lambrechts, A., Mignolet, J.-Y., Verkest, D. and Lauwereins, R. [2005]. Architecture exploration for a reconfigurable architecture template, *Design and Test of Computers* **22**(2): 90–101.

Mei, B., Vernalde, S., Verkest, D., De Man, H. and Lauwereins, R. [2002]. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures, *Proceedings of the IEEE International Conference on Field-Programmable Technology*, pp. 166–173.

Mei, B., Vernalde, S., Verkest, D., De Man, H. and Lauwereins, R. [2003]. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling., *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 255–261.

Mei, B., Vernalde, S., Verkest, D. and Lauwereins, R. [2004]. Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 1224–1229, vol.2.

Mirsky, E. and DeHon, A. [1996]. MATRIX: a Reconfigurable Computing Architecture With Configurable Instruction Distribution and Deployable Resources, *Proceedings of the 4th IEEE Symposium on Field-Programmable Custom Computing Machines*, IEEE, Napa Valley, CA, USA, pp. 157–166.

Mishra, M. and Goldstein, S. [2007]. Virtualization on the tartan reconfigurable architecture, *Proceedings of the 17th International Conference on Field-Programmable Logic and Applications*, pp. 323 –330.

Miyamori, T. and Olukotun, K. [1999]. REMARC: Reconfigurable multimedia array coprocessor, *IEICE Transactions on Information and Systems* **82**(2): 389–397.

Morra, C., Becker, J., Ayala-Rincon, M. and Hartenstein, R. [2005]. FELIX: using rewriting-logic for generating functionally equivalent implementations, *International Conference on Field Programmable Logic and Applications*, pp. 25–30.

NVIDIA Corp. [2010a]. *CUDA Programming Guide*.
URL: *developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_Programming_Guide.pdf*

NVIDIA Corp. [2010b]. *OpenCL Programming Guide*.
URL: *developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA OpenCL Programming Guide.pdf*

Oskin, M., Chong, F. and Sherwood, T. [1998]. Active pages: a computation model for intelligent memory, *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 192–203.

Park, H., Fan, K., Kudlur, M. and Mahlke, S. [2006]. Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures, *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pp. 136–146.

Park, H., Fan, K., Mahlke, S., Oh, T., Kim, H. and Kim, H. [2008]. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures, *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques*, pp. 166–176.

Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R. and Yelick, K. [1997]. A case for intelligent RAM, *IEEE Micro* **17**(2): 34–44.

Plessl, C. and Platzner, M. [2005]. Zippy - a coarse-grained reconfigurable array with support for hardware virtualization, *Proceedings of the 16th International Conference on Application-specific Systems, Architectures and Processors*, pp. 213 – 218.

Pozzi, L., Atasu, K. and Ienne, P. [2006]. Exact and approximate algorithms for the extension of embedded processor instruction sets, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **CAD-25**(7): 1209–29.

Pozzi, L. and Ienne, P. [2005]. Exploiting pipelining to relax register-file port constraints of instruction-set extensions, *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, San Francisco, CA, pp. 2–10.

Preis, T., Virnau, P., Paul, W. and Schneider, J. J. [2009]. GPU accelerated Monte Carlo simulation of the 2d and 3d ising model, *Journal of Computational Physics* **228**(12): 4468–4477.

Purna, K. and Bhatia, D. [1999]. Temporal partitioning and scheduling data flow graphs for reconfigurable computers, *IEEE Transactions on Computers* **48**(6): 579–590.

Rau, R. B. [1996]. Iterative Modulo Scheduling, *International Journal of Parallel Processing* **24**(1): 2–64.

Rupp, C. R. [2003]. Multi-scale Programmable Array, U.S. Patent 6633181.

Shalf, J. [2007]. The new landscape of parallel computer architecture, *Journal of Physics Conference Series* **78**(1): 1–15.

Shields, C. [2001]. *Area efficient layouts of binary trees in grids*, PhD thesis, University of Texas at Dallas.

Tang, X., Aalsma, M. and Jou, R. [2000]. A compiler directed approach to hiding configuration latency in chameleon processors, *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, Vol. 1896 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 29–38.

Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S. and Agarwal, A. [1997]. Baring it all to software: Raw machines, *IEEE Transactions on Computers* **30**(9): 86–93.

Wilton, S. J. E. and Saleh, R. [2001]. Programmable logic IP cores in SoC design: opportunities and challenges, *Proceedings of the IEEE Custom Integrated Circuit Conference*, pp. 63–66.

Xilinx [2011]. Xilinx website: www.xilinx.com.
URL: *http://www.xilinx.com/*

Ye, Z. A., Moshovos, A., Hauck, S. and Banerjee, P. [2000]. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit, *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada, pp. 225–35.

Yoon, J. W., Shrivastava, A., Park, S., Ahn, M., Jeyapaul, R. and Paek, Y. [2008]. SPKM: A novel graph-drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures, *Proceedings of the Asia and South Pacific Design Automation Conference*, Seoul, South Korea.

Yu, P. and Mitra, T. [2004]. Scalable custom instructions identification for instruction set extensible processors, *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, Washington, DC, pp. 69–78.