Modeling, Analysis and Exploration of *Layers*: A 3D Computing Architecture

Von der Fakultät für Elektrotechnik und Informationstechnik
der Rheinisch–Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften
genehmigte Dissertation

vorgelegt von
Diplom–Ingenieur Zoltán Endre Rákossy, M.Sc.
aus Kronstadt, Rumänien

Berichter: Universitätsprofessor Dr.-Ing. Anupam Chattopadhyay
Universitätsprofessor Dr.-Ing. Tobias G. Noll

Tag der mündlichen Prüfung: 02.12.2015

Abstract (en)

Current trends in architectural design require high-performance, low-power, flexible architectures that can adapt quickly onto the ever shifting and evolving application landscape. Finding the best architecture matching these stringent constraints is further limited by a short time-to-market window, which severely limits design exploration options. This work tackles these problems by proposing a different view on architectural flexibility, which can be exploited to achieve high energy-efficiency and performance instead of being traded off, by exploiting the advantages of reconfigurable architectures. Starting from a theoretical view, a methodology is produced for exploration of two different approaches in achieving high energy efficiency with two different architectural concepts: an architecture perfectly tuned to the application; and a new reconfigurable layered architecture, which can adapt its structure to match the application.

The design space of reconfigurable architectures spans a wide range, which allows different number of processing elements with different options on granularity, control structure, degree of specialization, scalability, regularity and programmability. For the theoretical point of view, these features can be captured by defining *architectural flexibility*, which quantifies how well a given architectural design point from the design space is matching a given application. If there is a good match, the application is efficiently executed and high performance and low power consumption is gained. In the view proposed in this work, architectures can be separated into small pieces of elementary hardware functions. These functions can be designed and rearranged such that the required function of the application is closely matched. The rearrangement of these small functions into larger functions is called *functional reconfiguration*. A categorization is also proposed into four functional domains: memory access, computation, communication and control flow. Via this concept, exploration, configuration and control of reconfigurable architectures becomes easier and allows design of a wide range of efficient architectures.

To efficiently explore which configuration of elementary hardware components produces a design point that respects necessary constraints, a methodology is described based on High-Level Synthesis tools. Using this methodology, tens of architectural variants could be explored and evaluated. The guidelines presented in the methodology part of this work show how different types of architectures can be described and proposes two exploration directions: 1) weakly flexible application-specific architectures featuring elementary components specifically tailored for the architecture – targeted architectural flexibility –; and 2) architectures with a variable degree of flexibility, featuring a richer set of elementary functional components by which adaptation to changes in the application is possible – tunable architectural flexibility –.

For the first direction, two WCDMA channel estimation algorithms, significantly different in performance and complexity, are targeted with a barely flexible architecture. The algorithms are analyzed carefully to expose common operations, parallelism and data movement patterns. Then, elementary hardware functions are created and an architecture is assembled which supports these two applications efficiently. High energy-efficiency gains are achieved with the resulting architecture supporting both algorithms, showing similar performance to architectural counterparts specialized to a single algorithm. The study is extended by fine-tuning the elementary functions with the addition of a reconfigurable fabric, yielding a closer application match and higher energy savings.

For the second direction, a novel reconfigurable architecture called *Layers* is proposed, featuring a layered design with elementary hardware components tailored for different functional classes of an application: control flow, data movement, processing and memory access. By providing a pool of elementary functions for each class, a structure can be configured in each layer, that allows a close match to different application requirements. To demonstrate the degree of tunable flexibility that this solution achieves, an entire application domain is targeted. Multiple different applications from numerical linear algebra domain are mapped and evaluated on the architecture, achieving excellent scalability, performance and energy efficiency results. Scaling parallelism and resources of *Layers*, a clean trade-off of area vs. performance could be achieved for all tested applications while keeping energy constant, a result achieved by the high flexibility that the proposed structure provides.

The work concludes by proposing enhancements to the *Layers* architecture: a force-directed scheduler and mapper for the computation layer of the architecture, which focuses on automating the application mapping process; and a new approach on automatically deriving and generating the architectural components for the control flow layer using a graph-theoretical approach contrasted by two manual designs.

Keywords:

Coarse-Grained Reconfigurable Architecture (CGRA); Weakly Tunable ASIC; High-Level Design and Synthesis; High-Level Design Methodology; Functional Reconfiguration; Numerical Linear Algebra (NLA); WCDMA Wireless Channel Estimation; Energy-Efficient Mapping and Architectural Design; Scalable Architectures; Application Mapping and Optimization; Application-architecture Mapping Tools; 3D Architecture;

Zusammenfassung (de)

Die gegenwärtige Entwicklungstendenz von Rechnerschaltkreisen und -architekturen braucht anpassungsfähige Architekturen, mit hoher Rechenleistung und niedriger Leistungsaufnahme, die flexibel auf variable und sich ständig entwickelnde Anwendungsanforderungen angepasst werden können. Die architekturelle Erkundung der Designoptionen um eine passende Architektur zu finden, wird nicht nur durch solchen strengen Anforderungen erschwert, sondern auch durch kurze Entwicklungszyklen in der Industrie limitiert. Diese Dissertation spricht diese zwiespältige Anforderungen durch eine unterschiedliche Auffasung von architektureller Flexibilität an. Anstatt die Flexibilität gegen hoher Rechenleistung und -effizienz einzutauschen, sollte man diese mit Hilfe von rekonfigurierbaren Architekturen verwerten. Nach einer theoretischen Analyse der Flexibilität, eine effiziente Entwurfsmethodologie von solchen rekonfigurierbaren Architekturen wird vorgeschlagen, wobei hohe Energieeffizienz durch zwei verschiedene Konzepte abgezielt wird: direktes Entwurf von einer Architektur die perfekt zu bestimmten Anwendungen angepasst ist; und das Entwurf von einer mehrschichtigen Architektur, die sich an mehreren Anwendungen anpassen kann.

Verschiedene Optionen beim Entwurf von rekonfigurierbaren Architekturen ergeben sich vom hohen Freiheitsgrad was die Komponenten anbelangt: Anzahl und Körnung der Prozessorkerne, deren Kontrollmechanismus, Grad von Anwendungsspezialisierung, Skalierbarkeit, Regularität und Programmabilität. Die daraus resultierende Flexibilität kann man aus theoretischer Sicht als die *arhitekturelle Flexibilität* definieren. Die architekturelle Flexibilität einer bestimmten Architektur wiederspiegelt den Anpassungsgrad der Architektur zu einer bestimmten Anwendung. Hohe Anpassungsgrade bedeuten dass die Anwendung hocheffizient auf der Architektur ausführbar ist, weshalb man dann hohe Rechenleistung und niedrige Leistungsaufnahme gewinnen kann. In Rahmen dieser Theorie, Architekturen werden auf eine Sammlung von Elementarfunktionen in Hardware aufgeteilt, womit man dann höhere Anwendungsfunktionen flexibel zusammensetzen kann, um einen hohen Anpassungsgrad an die Anwendung zu erzielen. Die Zusammensetzung dieser elementaren Hardwarefunktionen wird als *funktionelle Rekonfiguration* genannt. Vier Kategorien solcher Funktionen werden auch abgeleitet: Speicherfunktionen, Rechenfunktionen, Datenkommunikationsfunktionen und Kontrollfunktionen. Mit diesem Konzept wird die Erkundung, Konfiguration und Lenkung von rekonfigurierbaren Architekturen erheblich erleichtert um eine vollständige Abdeckung der Designmöglichkeiten zu erzielen.

Eine Entwurfsmethodologie mit Tools auf hoher Abstraktionsebene wird beschrieben, um eine effiziente Ableitung von Elementarfunktionen und -kombinationen für bestimmte Anforderungen zu ermöglichen, womit man meherere Architekturvarianten ausgewertet werden können. Die Leitsätze der beschriebenen Methodologie sind in zwei Entwurfsrichtungen zusammengefasst: 1) anwendungsspezifische Architekturen mit geringer Flexibilität, die nur soche Elementarkomponenten aufweisen die für die Anwendung relevant sind – gezielte architekturelle Flexibilität –; und 2) Architekturen mit einen variablen Grad von architektureller Flexibilität, die eine Sammlung von elementaren Hardwarefunktionen aufweisen, womit man eine perfekte Anpassung an verschiedene Anwendungsanforderungen möglich ist – abstimmbare architekturelle Flexibilität –.

Die erste Richtung wird durch die Entwicklung einer Architektur ausgewertet, bei der zwei verschiedene WCDMA Algorithmen für Kanalschätzung flexibel unterstützt werden. Die Komplexität und Leisung dieser Algorithmen sind wesentlich verschieden und die Architektur wird mit einer Sammlung spezifischer Elementarfunktionen entworfen, die genau auf diese Algorithmen abgestimmt sind. Dadurch wird ein hohes Maß an Energie-effizienz und Rechenleistung sichergestellt, das vergleichbar mit den anwendugsspezifischen Architekturvarianten der einzelnen Algorithmen ist. Diese Richtung wird durch eine Feinabstimmung durch rekonfigurierbare Strukturen noch erweitert um zusätzliche Leistung zu entfesseln.

Die zweite Entwurfsrichtung wird durch eine rekonfigurierbare Architektur – *Layers* – erforscht. *Layers* wird aus mehreren Schichten von Elementarfunktionen zusammengesetzt. Jede Schicht wird einer funktionellen Kategorie aus der Theorie zugewiesen. Mit einer reichen Sammlung von Elementarfunktionen wird die Architektur auf verschiedene Anwendungen einer Domäne angepasst. Die Skalierbarkeit, Effizienz und Leistung der Architektur wird mit verschiedenen Linearalgebraalgorithmen untersucht und eine saubere Abstimmung zwischen Fläche und Leistung unter konstanter Energieanforderungen wird erzielt, was nur durch den hohen Grad an architektureller Flexibilität möglich ist.

Zuletzt werden noch zwei Erweiterungen zur *Layers*-architektur vorgestellt: eine Heuristik und Werkzeug für eine automatisierte Anwendungsabbildung und ein paar neue Ansätze um die Kontrollflusskomponenten von *Layers* automatisch zu generieren.

Schlüsselwörter:

Grob-Körnige rekonfigurierbare Architektur; abstimmbares ASIC; Design und Synthese von hoher Abstraktionsebene, Methodologie für Design aus hoher Abstraktionsebene, Numerische Linearalgebra, WCDMA Kanalschätzung; Energie-effiziente Anwendugsabbildung; Energie-effizienter architektureller Entwurf; Skalierbare Architektur; Anwendungsabbildung und -optimierung; Tools für automatische Anwendungsabbildung; 3D Architektur;

Acknowledgements

This thesis is the result of my work as research assistant at the Institute for Communication Technologies and Embedded Systems (ICE), Multiprocessor System-on-Chip Architectures (MPSoC) group, at the RWTH Aachen University. During this time I have been accompanied and supported by many people. It is my great pleasure to take this opportunity to thank them.

Foremost I am extremely grateful to my supervisor, Professor Dr-Ing. Anupam Chattopadhyay, for offering me the opportunity of conducting such interesting research, guiding me and supporting me over all these years. Enriched by his valuable insights, I could conduct a deep and broad exploration of processor design, summing up to an unforgettable experience.

Invaluable support was offered by my secondary supervisors within ICE, Professor Dr.-Ing. Gerd Ascheid and Professor Dr. rer. nat. Rainer Leupers, providing my professional home for the past years, for which I'd like to express my warmest heartfelt thanks. My respect and gratitude also goes to all Professors part of the UMIC cluster within the German Federal and State Government Excellence Initiative, who created the opportunity for me to freely explore my ideas.

Special thanks go to my secondary supervisor, Professor Dr.-Ing. Tobias G. Noll, for his valuable feedback and improvement suggestions during the conception of this disseration. Furthermore I would like to thank Professor Dr. Yusuf Leblebici for his ideas, feedback and interesting discussions and for hosting and guiding me during my 6-months research visit at EPFL Lausanne. I have learned many new and exciting things during my visit from him and his team at LSM.

Transforming bits of inspiration and challenging ideas into results could not have been completed without the invaluable help of my students and collaborators. I am very grateful to Axel Acosta, Dominik Stegele and Tejas Naphade for their invaluable help in kernel optimization, Dr. Alexander Fell for exploring mapping strategies together, Farhad Merchant for providing me with parallel algorithms and Saumitra Chafekar.

Great ideas can be born only when the surrounding environment is inspiring. I would also like to thank all my colleagues at MPSoC and ICE for interesting times and challenging discussions. Dissecting wild ideas and listening to technological rants was truly fun.

Highly appreciated non-technical support was also received all over these years. I owe deep gratitude to Anne Breuer, Susanne Kleinle, Sonja Müßigbrodt, Marion Zimmer, Elisabeth Böttcher, Tanja Palmen, Dr. Ute Müller, Christoph Vogt, Gabriele Reimann and Melinda Mischler. Also my thanks go to the entire ERS team and the TI and ComSys research groups – especially Dr. Grit Claßen, Dr. Simon Görzen, Dr. Christian Dombrowski, Dr. Oscar Puñal for making coffee breaks extremely enjoyable every day.

Finally, my deepmost gratitude goes to my parents and family, who encouraged and supported me, always, unconditionally, no matter which road I chose to take. I would not have become what I am now without your help. Thank you so much.

Zoltán Endre Rákossy, December 2015

Contents

Chapter 1

Introduction

In recent times there has been a stunning trend of integration of computers and automation in daily lives. People are still fascinated how their smart-phones are faster than their huge multi-hundred-Watt desktop PCs from a few years ago. Computation has become part of our lives evermore. Now it started invading our environment, the so-called Internet-of-Things, where people dream to peak into their home refrigerator while out shopping, that the house comes to life, warms up and greets them when they return in their fancy self-driving electric car. The revolution to include computation in everything we touch already started and soon, the final frontier – our selves – will be next on the list to be conquered by microchips. People find this trend fascinating and excited to see what research can enable in the future.

1.1 Current Technology Background

For more than five decades, the number of transistors per chip in CMOS technology is still increasing almost exponentially, according to Gordon Moore's law [119]. *More Moore* and *More-than-Moore* [21] has been on the lips of system designers, high performance computing specialists and wireless experts. However, the scaling process formulated by Dennard's law [51] is starting to reach its upper bound [57]. The International Technology Roadmap for Semiconductors [1] posted that scaling is slowing down, from 2× per technology node to 1.6× per node. The frequency wall was already hit more than a decade ago: up to 41% increase before year 2001 (device speed), 17% in 2001 (platform power limit), 8% in 2007 (device scaling limit), finally hitting only 4% per year in 2011. Power envelope limitations and foremost fabricability limitations are stopping the trend. New transistor designs are compensating the ever increasing difficulties, such as the tri-gate FinFET transistor [37], which are already in the mainstream market. Radical new ideas, such as 3D stacking technology are constantly explored to try compensate for this slowdown and prolong the life of Moore's law a little bit longer, until scientists find an alternative to CMOS technology, such as quantum computing [80], memristors [167], carbon nanotubes [106], etc.

1.2 Need for Design and Energy Efficiency

One major problem rose in importance in the last few years: design complexity. Even with the slowdown in process scaling, already in 2003, this was a major problem [78]. The massive availability of transistors caused not only a design gap – not being able to make use of all those transistors in a design –, but also a power gap – not being able to power all the transistors on the chip at the same time, *dark silicon* [57].

Designers are looking at ways to be able to design complex architectures, while respecting the power constraints, with fast time-to-market. Electronic System Level (ESL) design offers the refuge of high abstraction levels and automated code generation, simulation, testing and virtualization to designers, lowering complexity at the expense of design flexibility and efficiency. This is contrasting with the requirements of low-power design, for which highly optimized circuits are designed in long development cycles manually.

1.3 What the Industry Wants – Directions

Given the fact that the market for small integrated circuits is rampantly increasing, industry is hard-pressed to tackle complexity and constraints. On one hand, time-to-market is so important in the design cycles, that products receive only incremental optimizations and changes. For instance, Apple releases a new iPhone every two years, with one version released in between having only minimal changes. Release cycles are highly tuned to capitalize on major consumer events, like Christmas sales, tracing hard deadlines in the production process. High design complexity forbids designing systems from scratch. Virtualization and high abstraction level design help evaluating the effects of design changes quickly, before too many resources and time are invested into a design direction, that may not be feasible. High-level synthesis and design of systems and architectures are playing a key role in quick evaluation.

On the other hand, for energy-efficiency, the industry is looking at an increasing library of already designed components [45]. These off-the-shelf highly optimized and tested products targeted at certain applications, often offer bleeding edge results in terms of performance and energy efficiency, ready to be integrated into larger System-on-Chips. Intellectual Property (IP)-based design fills in the gap of components missing in a vendors portfolio allowing quick integration.

1.4 Outline

This work looks closely at both sides of these needs. I pursue design of energy-efficient domain-specific accelerators which can serve as IP blocks for higher order designs. Finding the right design point in an overwhelmingly huge design space can be very complex and requires a lot of time and effort. Considering this, I look also into underlying theory and methodology on designing energy-efficient architectures, exploiting high-level synthesis and high-level abstraction exploration tools. Broadly scanning the design space and based on the theoretical concepts described here, I present two directions of design at core of which flexibility plays a pivotal role. The thesis is organized as illustrated in Fig. 1.1. After a brief landscape presentation, functional reconfiguration theory is introduced. Next, a methodology to explore the designs, that the theory suggests, is formulated. Two architectural directions are explored: taking an approach on tightly controlling architectural flexibility towards optimizing for some applications; increasing flexibility to be able to support an entire application domain while still keeping energy-efficiency with a layered design. Finally, some enhancements are proposed for the layered design, tackling two difficult parts: automatic mapping and reconfigurable control flow.

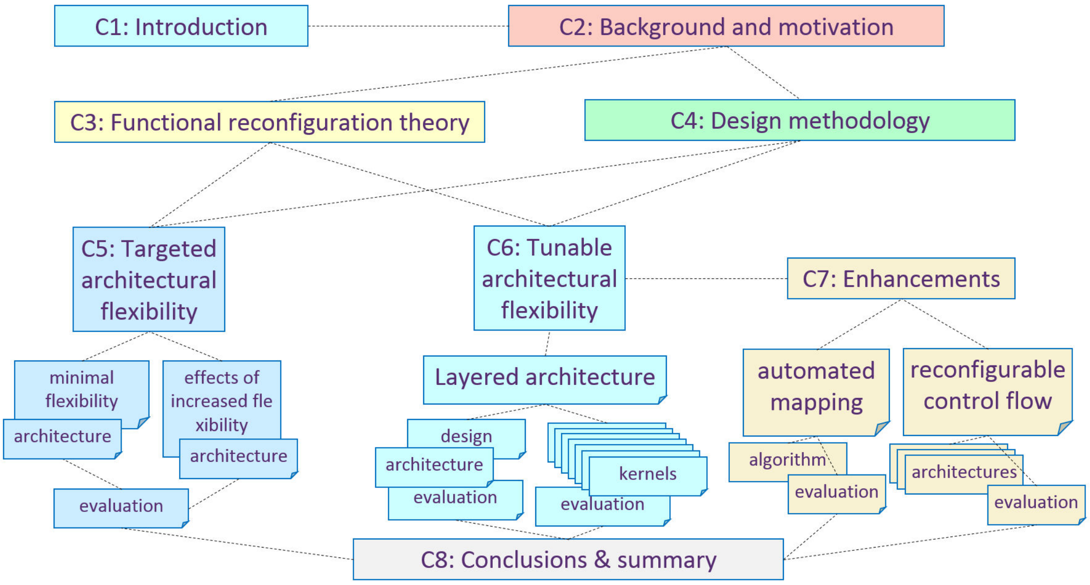


Figure 1.1: Dissertation outline

Chapter 2

Technological Landscape, Problem Definition and Objectives

Four motivational vectors are presented in the following, to properly place this work in the computing landscape. Each vector has been considered in the conception of this work, and solutions are proposed towards solving the problems.

2.1 A View From 4 Motivational Vectors

2.1.1 Technology Scaling, 3D Integration

Aside from the slowdown in technology scaling mentioned in the introduction, there are two problems that gain importance with scaling. An increased number of transistor means also that more, larger designs can fit on a chip. The larger the designs, the more data hungry they are, especially for data-centric applications. The problem is that the input-output (I/O) bandwidth of these chips do not scale as well, since the I/O pads are much larger and require much more power in addition to the physical limitation on how many of them can be placed on a chip package. Moreover, as transistors are scaling, relative distance between a signal's source and sink is increasing. Also, the wires that link them have a limit on how thin they can be due to parasitic effects causing power loss for long wires and reliability issues such as electro-migration, etc.

To alleviate this problem, 3D stacking and 3D integration is proposed, featuring thinned die-to-die bonding and Through-Silicon-Via (TSV) interconnections. Here, two silicon dies are placed one above another, and wires are connected vertically *through* the top silicon die, to reach the metal layers. Not only this reduces wiring in microprocessors [30, 31] but also enhanced performance due to shorter distances. However, there is a trade-off: area is sacrificed for the TSVs, which are relatively large, to gain inter-wafer bandwidth [65].

The challenge addressed in this work, is that, until now, there are no architectures that exploit such kind of structures. Architecturally, either memory is stacked onto a processing plane [30], or several identical chips are stacked to reduce the area footprint (e.g. memory stacking [117]). So far, to the best of my knowledge, no architecture is reported so far, that inherently exploits 3D stacking in its internal structure.

2.1.2 Applications: Energy Efficiency, Complexity

From the application point of view, today's applications are gaining complexity, backed by more powerful computation solutions. A detailed analysis on the constituents of applications, revealed that all applications share one or more of 13 kernels, called *dwarfs* [22]. Entire domains of applications can be efficiently executed, by just executing efficiently and optimizing for the constituent *dwarf*.

Application specific integrated circuits (ASICs) are the best solution when high performance and low power is required. These have, however, the downside of having little flexibility (not being able to adjust to application changes) and requiring long development time and costs. ASICs can not be used to accelerate *dwarf*s, since dwarfs alone do not make the application. A high degree of flexibility is required to adjust to various applications within a domain, even if all members are based on the same *dwarf*.

The architecture proposed in this work tackles domain-specific acceleration by having enough flexibility to adjust to application member changes. The dense linear algebra *dwarf* is chosen for this case study.

Furthermore, applications require high energy efficiency: in mobile devices battery life while in high-performance computing, the advent of big data demands energy efficient data centers. A well-tuned architecture for energy efficiency would play a big role here.

2.1.3 Architectures

Architectures can be easily categorized into a few large categories, in terms of their performance:

- General Purpose Processors - this class has a high degree of flexibility in executing applications at the expense of performance and power dissipation. Different sub-categories can change the trade-off point in favor of one point. Highly parallel and top performance processors consume upwards of 100W even in the latest technology, while low power processors have measly performance. An emerging sub-category are General-purpose Graphics Processing Units, which are highly parallel processors initially designed for video output processing in desktop computers, which turned out to be great parallel platforms for scientific computation and other parallel applications. They have a more limited flexibility, very high power consumption, but excellent performance.

- Digital Signal Processors - this class emphasizes on custom instructions to accelerate certain applications. While this boosts performance in comparison with the GPPs, some flexibility is sacrificed – non-target applications still can be executed, but performance is poor.

- Application Specific Instruction-Set Processors [19, 43, 160] are taking the DSP paradigm further and make use of hardware support for custom instructions, giving a great deal of performance boost while keeping processor-like flexibility. Reconfigurable versions of such ASIPs, called rASIPs [40], allow these hardware-supported custom instructions to be reconfigured in case the application changes.

- Field Programmable Gate Arrays - this category makes use of highly regular, fine-grained (bit-level) programmable cells connected by a network of bit-level wires, to construct the circuit necessary for the application by only configuring. This kind of programmable logic device started a new category, called reconfigurable computing [46, 77], several decades ago. Initially designed for prototyping integrating circuits, it quickly raised to a wide-spread platform for applications benefiting for varying degrees of parallelism and hardware acceleration [46].

- Coarse Grained Reconfigurable Architecture - this category employs word-size interconnect and ALU-sized processing elements as an underlying platform to implement applications. While this category combines some advantages of reconfigurable flexibility with the high performance of specialized processing elements, it has some disadvantages such as lack of design methodology and tool flow.

- Application Specific Integrated Circuits - this category provides still the highest performance for a given application and requires lowest energy. This category is complex to design and it is completely inflexible to application changes. If further effort is invested by going full-custom designing transistors and not relying on standard cell libraries, the performance is unrivaled.

In [50] it is argued that from the point of view of computational density reconfigurable computing is superior to programmable platforms such as DSPs. This meant that per feature size, an order of magnitude increase in computation could be executed in the reconfigurable device. Although inferior to ASICs, reconfigurable computing has the advantage of flexibility to adapt to another application.

CGRAs are especially interesting, striking a perfect balance between performance and flexibility. The challenge is, however, to find a way to design and program them easily, exploiting adaptability towards efficiency and performance, a subject currently still under research [39]. Furthermore, scalability is also an important factor to consider, providing an additional design dimension.

2.1.4 Design Methodology

Current design methodologies are heavily driven by two factors: quality-of-results and time-to-market. These two requirements clash.

If quality is sought, ASIC design flow is the answer for reaching optimal efficiency and performance. The cost is long development cycles and large initial investment. Furthermore ever increasing complexity of the design is forcing a block-based approach. Readily designed ASICs and other off-the-shelf components are slowly becoming the transistors of the new complex system-on-chips.

When time-to-market is important for a design that cannot be build readily from existing components, only high-level abstraction can provide the necessary leverage to reach the target. By raising the abstraction level via high-level design and synthesis tools, quick evaluation of architectural decisions on performance and constraints can be conducted, without committing to a lengthy ASIC design process in the exploration phase.

The challenge of exploring the huge design space offered by reconfigurable computing via high-level abstraction tools has not yet been solved. A solution would open the door to highly flexible new design which could hold the key in balancing out flexibility and performance.

2.2 Problem Definition and Contribution Summary

To summarize, this work is focused on answering the following points:

- Architectural side: a low-power, efficient, high-performance flexible architecture is required.
- Application side: flexible application support is necessary, domain-specific support is desired.
- Methodology side: a quick exploration flow is required for covering the design space in search of the required architectural design point
- Technology side: if possible, the architecture should align to current technology trends

To tackle these points, this work analyzes ways to eliminate inefficiencies in design, architecture and application mapping, by formulating a theory enriched by a methodology. Two distinct exploration directions are researched to validate the proposed theory and methodology, one of which exploits coarse-grained reconfigurable platforms in a way that is easily implementable in 3D silicon capable technologies. Furthermore, an automatic mapping tool and flexible control flow are proposed as enhancements to these architectures. High energy-efficiency, scalability and flexibility is kept as guiding constraints all over this work.

Chapter 3

Functional Reconfiguration Theory: A New View on Programming Reconfigurable Architectures

As the current landscape of domain-specific accelerators is struggling with conflicting constraints of high performance, low energy and fast time to market, flexible solutions based on reconfigurable architectures are an interesting alternative to IP-based MPSoCs. Chapter 2 highlighted some design trends and gave a detailed view on why architectural flexibility could be advantageous. The big gap between the GPPs (too flexible, performance and energy is lost) and ASICs (not flexible, high performance and low energy) can be effectively bridged by tuning the amount of flexibility and thus adapting the architecture to the application. This has been done before – from the GPP side, the DSPs and ASIPs removed flexibility by introducing custom instructions and reducing the size of the instruction set, while from the ASICs side, FPGAs added flexibility by allowing different fixed application implementations on the same physical device. rASIPs and CGRAs added more specific flexibility to the architecture, allowing a great degree of adaptability [39] to changing applications, physical degradation effects, tunable performance, thermal- and power-aware execution, etc.

Although CGRAs and rASIPs achieve a perfect balance between performance and flexibility and have the advantage of better computational density compared to other architectures [50], major drawbacks stopped wide-spread adoption: difficult programmability, high design effort and lack of proper development tools. Additional complexity derives from the tremendous design space that can be covered with reconfigurable structures, exploration of which is not straightforward:

- *granularity* of the reconfigurable fabric: fine grained solutions provide great amounts of flexibility, which can be better suited to the application at the expense of programming and configuration overhead; coarse-grained solutions remove some structural flexibility and configuration overhead, however tool programming tool complexity increases dramatically

- *control*: the reconfigurable architecture can be a loosely or tightly coupled accelerator with a host processor, or can work stand-alone for the given application domain

- *specialization* of the processing elements: e.g. fixed vs. floating point, application specific PEs, custom bit-width data paths

- *scalability* and *regularity*: how many PEs to use, how easily PEs can be added or removed, what kind of interconnect topology, heterogeneous or homogeneous structures
- *programmability* or *configurability*: control via either high-level (programming at a high level language with compiler-support) or low-level (configuration bitstream)

Once an architectural instance from this design space is chosen, the question of how to (efficiently) program or configure it, remains. A stored program GPP-like architecture would require a custom compiler which can target the specific architectural structures, while an architecture relying on multiple contexts of configuration bits to control the data path switches needs a tool to derive these bits. None of these solutions are trivial, especially for an architectural instance with novel features, picked from the design space.

It is even more challenging to design and program a scalable architecture from this design space that has *tunable flexibility*, i.e. the ability to change its internal structure, programming interface and reconfigurability options to better suit an application. In the following, a solution is proposed where this complexity can be abstracted to a tractable level, unlocking a new view on programming such highly flexible and scalable architectures.

3.1 Flexibility and the *von Neumann* Bottleneck

3.1.1 The *von Neumann* Bottleneck

From the earliest stored-program computers to modern day processors, the problem called the ***von Neumann* bottleneck** is dominating flexible architectures. The computing model proposed by and named after John von Neumann [162] described an architecture consisting of a CPU with registers and an ALU, a memory for both instructions and data linked by a bus to the CPU and I/O interfaces. This model was improved over the years and as technology advanced, it exposed the von-Neumann bottleneck, identified by John Backus [24]. When a program *instructs* the processor to modify the contents of the memory, data needs to travel back and forth from memory to the processing unit via a bus or interconnect of limited bandwidth, thus contention occurs and execution efficiency is lost. Quoting John Backus, the problem from the architectural structure has also deeper ramifications:

„Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann *bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the* von Neumann *bottleneck, and much of that traffic concerns not significant data itself, but where to find it."* [24]

Due to ever-increasing application complexity and with it the massive increase of data, this bottleneck will continue to dominate every architecture where computation can not be done in the place where the data is located. As a solution to the intellectual bottleneck, Backus proposed functional programming, in the sense of compositionality, in contrast to traditional functional programming based on lambda calculus. It is postulated, that a desired function can be composed by smaller functions, by respecting a certain composition algebra, thus avoiding the imperative character of programming, which partially causes the *von Neumann* bottleneck.

Several architectures were proposed to support such a paradigm [66,82,159]. Since all these architectures tried to interpret and execute a functional language directly, hardware resource availability and recursion problems impeded progress. When trying to apply the theory directly to make a functional architecture, key elements of functional programming theory like functional reduction ($f(f(f(...)))$) clash with the physical bounds of the hardware (e.g. stack size), even though there are successful attempts to create large architectures which could handle this to some degree, like the functional neurons of SyNAPSE architecture by IBM [2]. FPGA-based functional architectures are also attempted, such as the Reduceron [120] and functional design of reconfigurable architectures has been thoroughly explored [29]. Recombination of complete kernels to compose more complex applications by chaining inputs and outputs in a configurable fashion within a processor pipeline (Function Level Processor) has been also attempted with great results, but it does not apply functional programming theory [155].

Another approach to exploit the ideas from [24] is to create functional programming languages to exploit the theory, which are gaining increasing popularity with languages like Haskell [3] and the Wolfram Language [4]. Powerful compilers and abstractions help translate the power of the theory into machine-executable code, which in the end works in a non-functional (imperative) way, but gains are limited by loss of abstraction and by the unsolved problem of the *von Neumann* bottleneck of the underlying architecture.

A loss of abstraction happens also in traditional programming languages, when mapping high level languages or functions (e.g. C) back to instructions. These are then executed on data, which are stored inefficiently away from the processing units. The mapping of the intended high-level function written by the programmer (e.g. matrix multiplication) to a large set of instructions of the underlying machine causes also a loss of *meaning* – it is not easily understandable from the resulting assembly code of the compiler (or the binary code) *what* the machine is actually doing, or *how* it is actually executing it.

There are different ways of representing an algorithm (e.g. flowchart, control-data-flow graph (CDFG), pseudo-code, etc.) as there are many ways to represent their function in terms of architectural functions via the mapping process. The ultimate goal is to represent the algorithm/application function efficiently in terms of hardware functions ($f_a \equiv f_{hw}$), to achieve high performance and low energy consumption. The degree of composability and flexibility of the available hardware functions for a given architecture determines the complexity of the representation of f_a in terms of f_{hw}. Moreover, the complexity of the application itself (length, data/control flow peculiarity, parallelism, etc.) can have a positive or negative effect on this representation. A highly parallel application on a highly parallel architecture with plenty of bandwidth will be efficiently represented, while the same application on a sequential processor will yield poor results. Furthermore, a well tailored set of f_{hw} of the architecture could allow matching of f_a on a high level of abstraction (by *meaning*), for a more direct and efficient programming.

3.1.2 Definition of Flexibility

In the literature, the term *flexibility* is broadly used for a number of concepts regarding an architecture: configurability of ASICs and FPGAs and programmability of CPUs, GPUs, ASIPs and DSPs. One definition of the flexibility of an integrated circuit $\mathcal{F}_{\text{IC}}$ is regarded as the inverse of (re-)implementation time of a given/new application in software and/or hardware [32, 33, 166], including design, test, fabrication and verification.

In this sense, GPPs have the highest amount of flexibility, whereas full-custom ASICs have the lowest. Once designed and fabricated, a new application can be implemented on a GPP by just writing and compiling its high-level code, whence ASICs require an entire design and fabrication process from algorithm analysis, design and RTL description of the architecture, verification, layout, manufacturing and testing, every time the application changes. These processes incur different amounts of non-recurring engineering costs bounded by strict time-to-market or cost constraints.

A measure for such flexibility can be thus expressed in time, or just qualitatively as a relative comparison. The definition of flexibility in [32, 33, 166] also captures indirectly the idea that if an architecture is flexible enough, it can support multiple applications, by having a reduced implementation time of the new application. In [166] it is argued that flexibility is a resource that must be traded off to achieve efficiency based on an analysis for soft-input soft-output sphere-decoding architectures for MIMO wireless. When viewing a highly flexible CPU compared to an ASIC for a given application, this is true, as CPUs are more flexible but less efficient than ASICs.

Throughout this work however, flexibility is exploited to *gain* efficiency. The term flexibility is broken down into a more fine-grained view, to reveal how flexibility can influence efficiency. Henceforth, the term flexibility is used in the sense of *architectural flexibility* $\mathcal{F}$, defined as follows:

Definition 3.1.1. The **architectural flexibility** $\mathcal{F}$ is the degree of adaptability to (a change in) application requirements by reconnecting, rearranging, or reconfiguring internal architectural structures such that mapping, execution efficiency and/or performance constraints of the applications can be met. It reflects how well a given architecture with given processing capability is suited to execute a given application, measured in mapping or execution efficiency relative to the theoretical optimum of the application. □

In the sense of the intellectual bottleneck mentioned by Backus [24], an architecture must have enough flexibility $\mathcal{F}$, such that an efficient and easy translation of the application functions f_a to the pool of physical hardware functions f_{hw} can be made. $\mathcal{F}$ has a direct effect on how efficient the representation of the application function is and thus a direct effect on overall mapping/execution efficiency, power/energy consumption and performance. A deeper analysis follows.

3.2 Functional Reconfiguration Theory

3.2.1 Concept

Conceptually, when a target algorithm has to run on a given architecture, the meaning or function f_a of the algorithm itself has to be translated into physical hardware functions (wires, processing elements) in space and time. The characteristics of the architecture define what kind of spatial or temporal mapping is possible, while the application characteristics define the requirements. This kind of mapping procedure requires addressing the functional capabilities of the hardware f_{hw}, for every unit of time, such that the meaning (function) intended by f_a is reflected in the meaning (function) realized by using available hardware functions f_{hw}. The set of physical functions provided by the architecture constitutes the *hardware function pool* p of the architecture.

Definition 3.2.1. An *elementary hardware function* f_{hw} is an addressable (controllable) physical hardware function of an architecture. The *elementary function pool* p is the set of all possible hardware functions available in the architecture. □

Available hardware functions are addressed (controlled) by the *language* that the architecture provides, defined as follows:

Definition 3.2.2. *Architectural Language* $\mathcal{L}$ of an architecture represents the set of all addressable operations or functions that can be created, combined and executed from the pool of its physical resources. $\mathcal{L} = \mathcal{C}(f_{hw}) \mid f_{hw} \in p$. □

The language $\mathcal{L}$ directly controls the existing resource pool p of available hardware functions f_{hw}. The recreation of the target algorithm using the language of this pool is the *representation* $r(f_a)$, also known as the application *mapping* in hardware.

Now of course, the target application function may or may not perfectly match the language of the architecture. When the target application function can not be represented in a direct way using the given language elements, efficiency is lost when trying to construct $r(f_a)$. The more flexible or abundant a language $\mathcal{L}$ is, the larger the number of applications which can be represented is, translating into higher architectural flexibility $\mathcal{F}$. Since $\mathcal{L}$ is constructed from the available functions f_{hw} of the pool p, the abundance of available hardware functions f_{hw} directly controls flexibility $\mathcal{F}$, as it allows construction of more varied language elements. This, in turn, allows the combination of available physical resources in various ways, providing adaptability to a larger set of applications with different characteristics. Fig. 3.1 reflects this concept: a given architecture features a language derived based on its physical properties, such as number of processing elements and their operations, interconnect richness, local storage options and memory access, parallelism, etc. An application can be optimized (compiled) to match the language, or the language can be configured to match an optimized application. The area where the two pieces are interfacing is extremely prone to the effects of the *von Neumann* bottleneck. A mismatching application/architecture interface would trigger, for instance, data congestion in buses, sequential execution due to insufficient memory or processing bandwidth, etc. A perfect match alleviates this via a more direct data flow, parallelism, and less interfacing overhead.

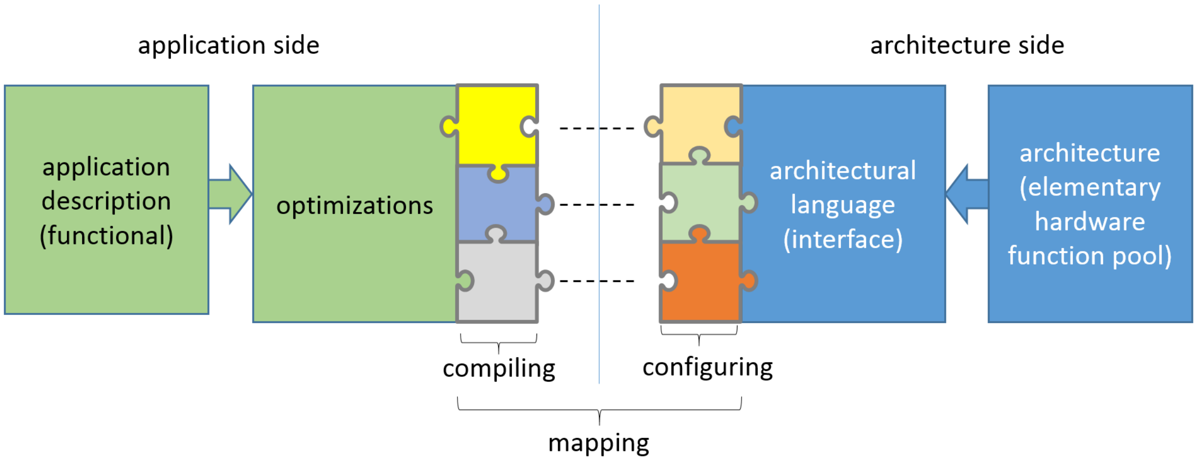
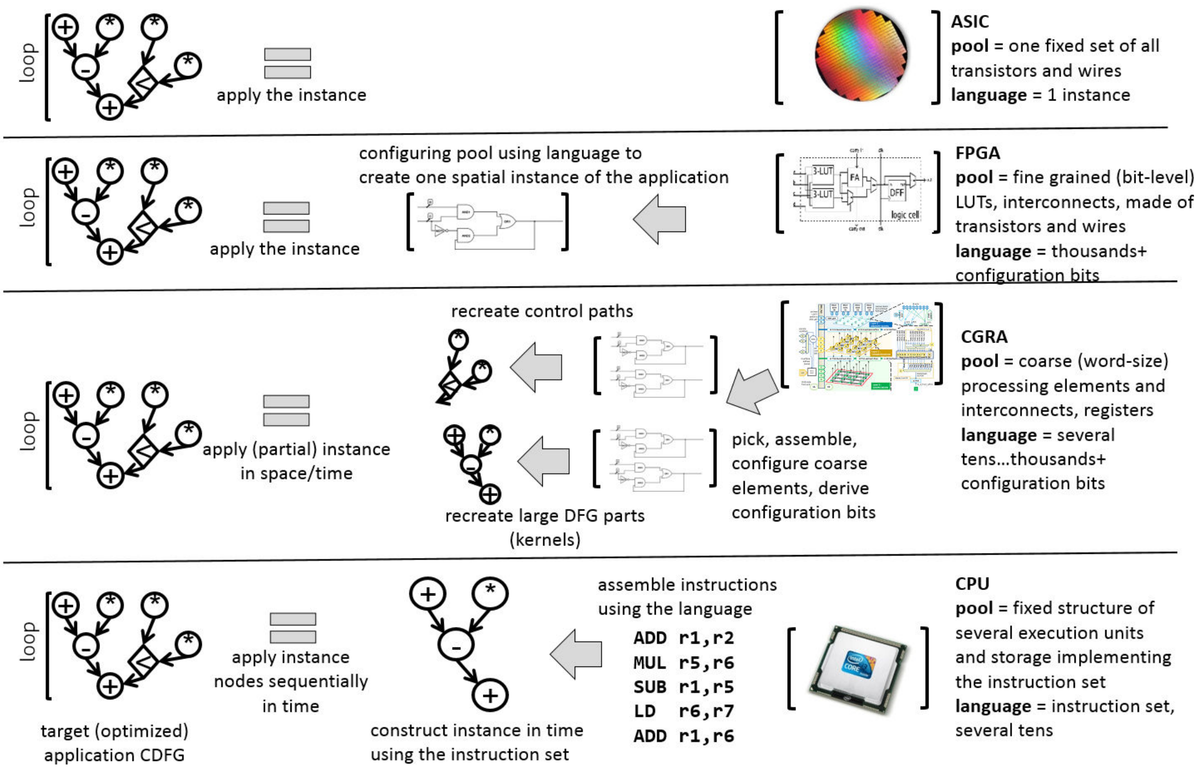


Figure 3.1: The link between application and architecture via the architectural language. An application can run on a given architecture only if the application can be expressed with the architecture's language.

In practice, besides the algorithmic data/control/loop dependencies of the application which limit mapping options (application-specific properties), two factors make the matching process complex:

- 1) a fine granularity of elementary hardware functions f_{hw} and a large pool p produce a complex language. This in turn makes it difficult to create the representation and complex helper tools are needed for this translation (e.g. synthesis of RTL code using a transistor library – the possibilities to construct the application using the language of the architecture are rich, having multiple options to implement the same RTL function).
- 2) a small pool of elementary hardware functions produce a well defined language, but this small size produces contention on the available resources, forces a sequential call of language elements, creating a large representation which is inefficient. Due to the sheer size of the representation that would be required to recreate f_a, again, helper tools are needed (e.g. compiling a large application to a RISC CPU core – every function has to be represented by the limited language elements of the instruction set architecture).

It is also to be noted, that from the abundance or scarcity of $\mathcal{L}$, the physical characteristics of the underlying physical architectures such as size, number of PEs, memories,



```
ADD r1,r2
MUL r5,r6
SUB r1,r5
LD  r6,r7
ADD r1,r6
```

Figure 3.2: Recreating the *function* of an application data flow graph snippet from the pool of hardware functions for each architectural class.

complexity can not be implied. The language, if poorly defined, can obscure elements of the functional hardware pool, by not having enough flexibility to expose them directly in the language. For instance, an FPGA exposes bit-level interconnect hardware functions whereas a CGRA only uses word-level interconnect functions, but both of them need to have the physical wires of the interconnect.

3.2.1.1 Languages of different architectures

Diving deeper into this concept, for every architecture the mapping procedure is different, leading to different design, compilation and configuration flows. Basically, mapping of an arbitrary application to an (existing) architecture can be done conceptually as *recreating* the Control/Data Flow Graph (CDFG) of the application in computational resources. This creates a time-space mapping problem of target graph to the available computational resources, known to be NP hard. This mapping can be viewed as an *instantiation* of existing hardware in time or space to recreate the desired application in hardware. Taking the Control/Data Flow Graph (CDFG) of an application as an example input f_a on the left side of Fig.3.2, a different representation $r(f_a)$ can be done in different architectures by (partial) instantiation of the language $\mathcal{L}$ reserving (partially) the hardware pool p.

In case of Application-Specific Integrated Circuits (ASIC), the pool p is constituted by one fixed hardware function made of wires and logic gates, which is the result of the direct synthesis process of the hardware for the application function f_a. The language $\mathcal{L}$ for this case is the pool itself, instancing its only f_{hw} member, the physical hardware itself. Every new application needs recreation of the instance and the pool from scratch.

For FPGAs, a large number of small bit-level look-up tables (LUT) with bit-level configurable interconnect make up the pool, hiding (abstracting away) the actual gates and wires, which are physically fixed. The language $\mathcal{L}$ is composed of the configuration inputs of the pool, and is used to construct elementary logic cells, which in turn reconstruct the ASIC instance in space-time, using a subset of the pool. Every new application requires a re-synthesis of the instance to derive the configuration bits (=subset of the language), the instantiation of which recreates f_a in terms of LUTs configurations (f_{hw}).

General Purpose Processors (GPP) rely on the instruction set as the language, which is bound to a pool of small hardware operations, such as reading or writing a register, instructing the ALU to do an addition or subtraction. The application instance f_a is constructed from calling repeatedly language elements in order to represent small f_a slices sequentially in time. Any new application can be represented by just rearranging in time the language calls, no hardware modifications are required. For more complex processors, the language $\mathcal{L}$ allows calling multiple pool elements simultaneously, instancing larger parts of the f_a control/data flow at a time cycle (parallelism). For DSPs or ASIPs, several language elements can be grouped together in custom instructions, to accelerate certain parts of the application, otherwise sequential language calls are executed, similarly to the GPP.

Coarse-Grained Reconfigurable Architectures (CGRA) take a middle ground between these architectures. The pool p is made of word-size processing elements and configurable interconnect to represent large parts of the f_a in space, reconfiguring as required for the next CDFG time slice (choosing a different combination/subset $\mathcal{C}(f_{hw})$). The language $\mathcal{L}$ is a mix between instructions and configuration bits, both of which can be static or dynamically changed with every cycle. Due to the coarse-grained nature of the pool elements, it is more difficult to find a good match, which makes the recreation of the complete application instance more difficult, especially since the language is also more limited. Conversely, the inefficiencies of recreating functions such as multiplication or division from small elements are avoided by using optimized ASIC-like function instances implementing these functions, in contrast to FPGAs.

Ideally, the instance of f_{hw} elements from the pool p should perfectly match the required function of the application f_a, like in ASICs, to achieve optimal efficiency, and additionally retain the option to change input application f_a without efficiency loss and without changing the f_{hw} pool - the physical hardware. Flexibility can have an important role in adapting f_{hw} such that an efficient match to f_a can be realized, reducing the effects of the *von Neumann* bottleneck.

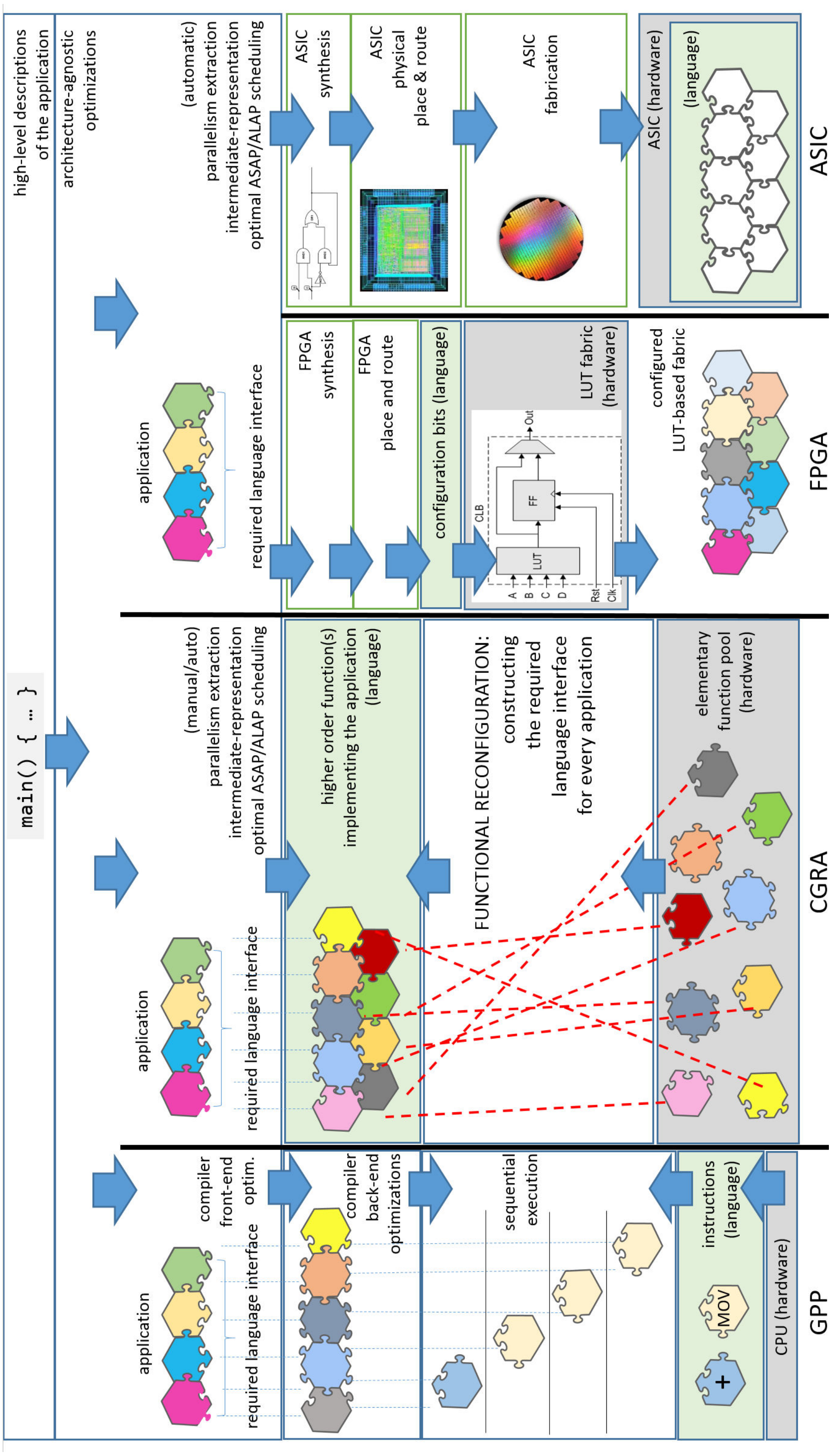


Figure 3.3: Application-side language interface matching the hardware-side language: each architecture type constructs the required language via different design flows. In case of CGRA, functional reconfiguration defines a flow where a direct adaptation of the architectural language to the application requirements is possible.

3.2.1.2 Where functional reconfiguration fits in

The concept of functional reconfiguration is exploited to increase architectural flexibility such that the architectural language can be customized, to gain a better and a more direct match to the application requirements. Fig. 3.3 shows different mapping flows for each architectural class, and highlights where functional reconfiguration fits into the architectural landscape. A more direct representation of intended functions is sought, using a *(reconfigurable) function set* (as opposed to instruction set) such that a better application matching can be realized, at a high abstraction level.

For CPUs, ASICs and FPGAs, well defined design flows already make the translation of the application to the language relatively easy, albeit with loss of efficiency for some. In CPUs, sequential execution limits efficiency and also the loss of abstraction when translating the functional meaning of the application to a small set of instructions for the CPU is a contributing factor for inefficient mapping. ASICs design flows make sure that the hardware perfectly matches the target application function, yielding optimal efficiency, but this flow has other penalties, such as complexity and long development cycles. FPGAs are emulating the ASIC flow, but fine granularity makes FPGAs orders of magnitude inferior in terms of power and execution efficiency when compared with ASICs.

As for CGRAs, there is no universally accepted design flow definition, CGRA mapping of applications still being a hot research topic. The inherent coarse-grained elementary functions allow reconfiguration and construction of language elements with a graspable complexity, making CGRAs the ideal candidate to fully exploit the concept of functional reconfiguration. Using functional reconfiguration, the ideal architectural language for the target application could be constructed, such that a close-to-optimal match can be realized.

Thesis 1. *Every architecture has its architectural language $\mathcal{L}$ by which its physical hardware functions f_{hw} can be accessed, addressed and controlled. Implementation of an application function f_a to a target architecture is possible, only when the application function can be expressed in terms of hardware functions f_{hw} via the architectural language $\mathcal{L}$. The richness of the architectural language $\mathcal{L}$ influences directly the amount of architectural flexibility $\mathcal{F}$, which can and should be exploited to fine tune, simplify and enhance functional translation of f_a to f_{hw}.* ■

3.2.2 Elementary Functions, Mapping and Representation Complexity

3.2.2.1 Elementary hardware functions

In order to construct efficiently the meaning of the target application function f_a using elements from the hardware pool, the architectural language $\mathcal{L}$ has to be formed. As each architecture features a different pool p of elementary functions f_{hw}, classification and composition rules of f_{hw} have to be defined. Ultimately, the functional *meaning* of the application has to be reflected in the hardware physical structures.

For every architecture, different hardware functions are available, e.g. arithmetic operations, storing a bit, selection of an input. In the following, the concepts defined in the previous sections are detailed by using examples.

For instance in ASICs, the arranged and complete (static) set of logic gates, wires with multiplexers and flip-flops create its single elementary function. As hardware granularity of the architecture changes, the size, number and visibility of these functions changes, creating elementary function classes which form the pool. E.g.: in the FPGA, many look-up tables and configurable logic blocks are the elementary hardware functions. How these logic blocks are formed physically is not architecturally visible as hardware functions, since they are not addressable or controllable – the wires and gates that construct the look-up tables.

$$\begin{aligned} p_{ASIC} &= \{\alpha\} \\ p_{FPGA} &= \{\beta_1, \beta_2, ..., \gamma_1, \gamma_2, ...\} \end{aligned}$$

To be able to use the elementary hardware functions, they must be rearranged, connected and called such that the desired target function f_a can be realized. This functional composition can be done in many ways based on hardware flexibility. ASICs allow no ways of reassembling their elementary hardware function, as it made of one element: the placed and wired set of transistors. FPGAs allow many degrees of freedom in rearranging elementary functions due to the flexible nature of the elementary functions themselves (architectural flexibility). CPUs on the other hand have a reduced, fixed set of elementary functions, but with a high degree of interoperability. To make use of possible rearrangement options, elementary functions are combined into a language. Thus, by instantiation of the language in space and time, the functional meaning of the application can be captured and reproduced. This is independent of the inputs of application function f_a, as only the transformation (meaning) of the function is captured. Of course, the language set has to provide the hardware elements necessary for guaranteeing Turing completeness.

Example: A processor features elementary hardware functions such as select a register (α_1), execute an addition (α_2) or forward an immediate value from the instruction word ($\alpha 3$). These and many more such elements form the pool p. Taking specific combinations of pool members, architectural language elements (A_1, A_2) can be formed, i.e. the *way* how these elementary functions can be addressed and controlled, defining the language $\mathcal{L}$ of the processor.

$$\begin{aligned} \alpha_1 &:= sel\_reg(\text{reg}); \quad \alpha_2 := +; \quad \alpha_3 := sel\_imm(\text{imm}); \\ p &= \{\alpha_1, \alpha_2, \alpha_3, ...\} \\ A_1 &= f(\alpha_1(\text{reg}) \circ \alpha_2 \circ \alpha_3(\text{imm})) && \text{add register to immediate} \\ A_2 &= f(\alpha_1(\text{reg}) \circ \alpha_2 \circ \alpha_1(\text{reg})) && \text{add register to register} \\ \mathcal{L}_{CPU} &= \{A_1, A_2, ...\} && \text{instruction set} \end{aligned}$$

For FPGAs, this would be the set of configuration bits to configure the look-up tables and interconnect to implement an addition, or the complete set of configuration bits which recreate the ASIC equivalent of the input f_a. △

3.2.2.2 Representation (mapping)

Different architectures allow various ways to instantiate language elements sequentially, in parallel or nested. The mapping of the target application is constructed by instantiation of these language elements, step by step, until the functional meaning of f_a is completely recreated. Once this procedure is complete, input data can be fed into the mapped function and the application is executed.

Definition 3.2.3. The hardware *representation* $r(f_a)$, or mapping of a desired input algorithmic function f_a is the ordered set of all instantiations λ of the architectural language elements of the target architecture in space and time.

$$r(f_a) = \{\lambda_1(A_*), \lambda_2(A_*), \ldots, \lambda_n(A_*)\} \quad | \quad A_* \in \mathcal{L} \tag{3.1}$$

□

Example: The representation of an application in CPUs, is the sequential readout of the binary instruction words (i.e. the program), which executes the target function, language element by language element until it is halted. For FPGAs, the representation is the complete set of bits that configure the look-up tables to perform the desired function. In CPUs, the representation is layed out temporally, while in the FPGA it is a spatial layout. △

3.2.2.3 Representation complexity

A representation of a target function for a target architecture features a complexity, which reflects how well the application matches the given architectural language.

Describing the complexity of a representation has its roots in the algorithmic information theory proposed in [92, 93]. Here, of special interest is the the combinatorial approach of describing representation complexity in [93], where the notion of language entropy is linked with an estimate of its flexibility, which is an index of the *diversity* of possibilities for developing a language with a given dictionary and given rules for the construction of sentences.

The similarities are striking, when the concept is viewed from functional reconfiguration point of view: a low complexity of the representation ($r(f_a)$) is sought, based on a language ($\mathcal{L}$), which has an entropy/flexibility ($\mathcal{F}$), stemming from the diversity and composition rules of its dictionary (elementary hardware functions f_{hw} of the hardware pool p).

A fundamental concept of theoretical computer science can be captured by a a simple definition of the algorithmic information, also known as the *Kolmogorov complexity* as follows [147]:

Definition 3.2.4. The algorithmic information or Kolmogorov complexity of a bit-string x is the length of the shortest program that computes x and halts. □

Similarly, the complexity of the representation of the target function $r(f_a)$ can be defined via this complexity, in view of functional reconfiguration:

Definition 3.2.5. The complexity of the representation $\mathcal{K}$ of the desired target application function f_a can be formally captured by the Kolmogorov complexity, which is the smallest length of the representation $r(f_a)$, given the set of language elements of a given architectural language $\mathcal{L}$.

$$\mathcal{K}(r(f_a)) = |\lambda_1, \lambda_2, ..., \lambda_n|, \quad \lambda_{1..n} \in \mathcal{L} \quad \text{[bits]} \tag{3.2}$$

□

Thesis 2. *The complexity $\mathcal{K}$ is controlled by the number of required instances of architectural language elements from $\mathcal{L}$. Therefore, an optimally matching language to the application, allows a short and exact representation, which yields least complexity. Complexity $\mathcal{K}$ is dependent on the combination possibilities and composition rules of the language elements of language $\mathcal{L}$, directly affecting the required length to describe a target function by needing a varied number of language element instances. This means, that if a language is tuned such that it yields the least complexity $\mathcal{K}$, it can be said that the language has a good matching degree w.r.t. the required target function f_a. However, $\mathcal{L}$ is, in turn, dependent on the features of the elementary hardware functions physically present in the architecture and the composition rules, which define the language and the resulting architectural flexibility $\mathcal{F}$. Thus, to achieve an optimal match between architecture and application, either 1) design the language to exactly match the application requirements; or 2) exhibit variable architectural flexibility to allow rich combination possibilities of elementary functions to form a custom language perfectly tuned to the application requirements. Both reduce mapping complexity $\mathcal{K}$, thus increase efficiency.* ■

Moreover, even if the representation complexity is reduced and an efficient application implementation is created, it is important to consider also the difficulty of the process by which the representation was derived.

3.2.3 Elementary Function Classification

In order to enable a variable and flexible composition of language elements from elementary functions, a classification is needed. This classification captures physical properties of the hardware and creates the hooks by which higher order functions and language elements can be created. Elementary functions are bound to the underlying physical hardware and are clearly defined in function, space and time and the nature of these properties is the base ingredient of architectural flexibility.

Thesis 3. *To efficiently capture hardware properties and enable flexible elementary function composition, it is necessary to characterize elementary functions not only by classical properties, such as latency, input/output, location and hardware operation, but also by functional properties (the* ***meaning****). Four large classes are proposed, which attribute a main meaning to* *such functions:* ***control****,* ***memory****,* ***communication*** *and* ***computation*** *class. This classification (with refined sub-classes) exposes hardware and software parallelism, data-flow features and scheduling possibilities and enables functional composition on a high abstraction level for the programmer and also allows class-specific hardware optimizations for increased architectural efficiency.* ■

Control class

The control class of functions implements those hardware functions that are responsible for the control flow part of the application. These can contain structures that implement conditional and unconditional jumps, enable/disable signals, full or partial predication and loops. Elementary functions of this class contain counters, registers and comparators. Small arithmetic units are sometimes needed for execution of more complex predication conditions.

Memory class

This class handles all the functions required to interact with physical memory and mass storage modules. Members of this class execute address generation, memory protocol handling, collision detection and multi-bank distribution functions.

Communication class

Elementary functions belonging to the communication class are specialized on copying, broadcasting, moving, delaying and local storing of data. These capture internal data movement dynamic of the application and feature structures dedicated for such tasks, like buses, register banks, rich multiplexing, etc. Functions in this class do not compute any data, their main purpose is to prepare, shuffle and move data to the correct space/time coordinates, as the application requires.

Computation class

This class is responsible for the elementary function dedicated to execution and calculus on application data. Members contain different arithmetic/logic units and specific interconnect that enables composition of complex processing functions. Structures in this class usually contain highly optimized ASIC pieces dedicated for a certain task.

Each of these classes are functionally independent and thus help expose parallelism in the application, followed by an energy efficient execution due to the class-optimized hardware implementation. Additional properties, such as number and size of arguments, time delay and interface define the possibilities in constructing higher order functions in the language.

Example: In Table 3.1 some elementary functions are shown for a 2×2 mesh-connected array of processing elements featuring 8 local registers. For each elementary function, the definition is written for the hook (name), parameters are derived based on the physical properties and hardware-bound location. The transfer function and its timing is characterized and a class assignment and a *meaning* description further classifies the function. Although the structure is regular and processing elements are identical, several elementary functions can be created using the same hardware bind, for instance a function that selects from the west input and another function that selects from the north input still bind to the same physical input multiplexer of the processing element. Architectural features, like exceptions to regularity (e.g. `PE0` being at the edge of the array and not having a `west` or a `north` I/O connection), are captured within the elementary function definitions easily.

This information is required when composing rules are defined and language instances are called. Calling one member that binds one hardware resource excludes the possibility of calling another member which binds to the same resource, unless the time-index differs.

Table 3.1: Example classification of the properties of 3 elementary functions in a 2×2 PE mesh with 8 local registers

Function def:	$add_{locus}(a, b)$	$src^{a\|b}West_{locus}()$	$del_2(r)$
Interface:	in:2 out:1	in:1 out:1	in:1 out:1
Class:	comp	comm	comm
Function:	$f(a, b) = a + b$	$f(x) = x$	$f(x) = x$
Delay [cycles]:	$\tau = 1$	$\tau = 0$ (instant)	$\tau = 2$
Location:	$locus \in \{0, 1, 2, 3\}$	$l \in \{1, 3\}$	$r \in \{0..8\}$
Hardware bind:	PE(locus)	MUXa(locus)|MUXb(locus)	reg(r)
Meaning:	add	select source	delay/store

△

Meta-functions

Meta-functions are functions that only serve functional composition, ease of description and representation complexity reduction and are an important enabler of architectural flexibility. These functions provide extra flexibility in describing complex compositions of functions, especially when large amounts of function arguments need to be specified. Compared to elementary functions, meta-functions are not hardware bound and take no functional part in actual application execution. Although they do not bind computational resources themselves (which is done via their arguments), they require hardware structures which do the physical interconnects needed for argument forwarding. These resources are however completely transparent to the application. Meta-functions enhance architectural flexibility by providing higher-order hooks to a group of elementary functions without changing their actual function.

Example: Forwarding the same elementary call to all processing elements in an array can be done conveniently with the `forall(f(x))` meta-function. The arguments are forwarded to all PE members of the array and represents a spatial meta-function. Similarly, `repeat(f(x),times)` repeats a function call in time, avoiding extra calls from assembly and reducing complexity $\mathcal{K}$. This represents a temporal meta-function. Clearly, combinations can also be created. △

Meta-functions are freely definable depending on what kind of support is required for a given architecture and are physically implemented during the architecture design phase.

3.2.4 Composing the Language

Once all elementary functions and meta-functions are defined and characterized, the pool is complete. When composing the language constructs, elementary functions are selected and interconnected. Only a subset of the elementary functions can be instantiated at a time due to exclusivity, since each are bound to physically existing hardware and occupy time-slots when called.

In view of realistic physical hardware resources, also the amount of arguments which functions can take is well defined and limited. For physical implementation clear coding fields have to be assigned to arguments such that physical decoder generation is possible. This implies, that not every possible elementary function composition can be instanced ind the language, however, limitations are designer-controlled at the time of architectural implementation.

When defining language elements, the significance and the location of every available argument-bit is defined. In the architectural implementation phase, the designer can choose whether language elements are hard-coded – customizing the language to the application with little flexibility, or implement configurable decoders which allow the redefinition of language elements – creating language adaptability with tunable flexibility. This is essential so that correct arguments are forwarded to elementary functions.

Example: Consider a 2×2 mesh architecture and 8 shared registers. The elementary functions available are defined in a similar way to Table 3.1:

- memory: load to reg `ldr`, store to reg `str`;
- communication: `forall` (meta), select reg `R`, select north `N`, south `S`, east `E`, west `W`;
- computation: addition `add`, multiplication `mul`.

Table 3.2: Example of creating *broadcast* and *multiply-accumulate* language elements from pre-defined elementary functions

Func def:	$mac_{locus}(a, b, c)$	$bcst(r)$
Interface:	in:3 out:1	in:1 out:4
Class:	comp	comm
Function:	$f(a, b, c) = a * b + c$	$\bigvee f(x) = x$
Delay:	$\tau = 2$	$\tau = 0(instant)$
Location:	$locus \Subset \{\{0_{\tau_0}, 1_{\tau_1}\} \mid \{2_{\tau_0}, 3_{\tau_1}\}\}$	all
Lang slot:	4'b0000	4'b0001
Elem calls:	$add_{\{1\vert 3\}}(W_{\{1\vert 3\}}(mul_{\{0\vert 2\}}(R(r), R(r))), R(r))$	$forall(R(r))$
Hardw bind:	PE(locus), MUXa,b(locus),	MUXa(locus)
Arg_len (32'b):	1+2+3+2+1+2+3+3+3+3+3+3=29'b	3+3+3=9'b
Meaning:	(pipelined, horizontal) multi-accumulate	bcast a reg to all loci

To uniquely identify composed language elements, a number of encoding bits are assigned to them. Similarly, argument bits are needed for encoding the number of argument combination possibilities allowed physically by the architecture and the elementary functions. This is a tunable parameter for the hardware designer, restricting or enriching possible language constructs.

In this example, communication class needs 3 bits to encode all argument options, while computation and memory need 1 bit each. Encoding physical location information of the processing elements requires 2 bits (4 PEs of the 2×2 mesh), just like in Table 3.1. Load from register (`R`) argument requires 3 bits to specify which of the 8 local registers is accessed, while other source functions need 2 bits for location (north, south, east, west), but do not require arguments. The meta-function (`forall`) can have an arbitrary argument length, depending on called elementary functions, but it may not exceed physically available argument length.

Let the target application require a *broadcast* function, and a *multiply-accumulate* function. These two functions are defined by *meaning*, which needs to be reconstructed in hardware in terms of language elements. Using the proposed functional reconfiguration concept, these two new functions can be composed from existing elementary functions. Assuming designer-imposed architectural limitations of a) maximum 16 language constructs, and b) maximum argument length of 32bits, functional words of 36 bits are possible.

In Table 3.2 to the semantics, properties and the actual elementary function calls which compose these functions are summarized. The new functions occupy 2 of the 16 possible language construct codes and bind to certain hardware structures (inherited from the elementary function calls). Latency of the two new language elements is based on the components and physical location of the elementary functions. The sum of all argument bits of the new functions respect the limitation (here 32bits) of the function word.

If the `forall` meta-function is used, one argument is forwarded to all locations (simple bit-copy). Alternatively, repeated `R` source calls can be done at each location, the `bcst` language construct would require then $locus \times (3+3)$ bits. Here it can be immediately observed that $\mathcal{K}(bcst_{forall}) < locus \times K(bcast_R)$; fewer bits are required to represent the functionality and the representation when calling the language construct `0001` is easier and clearer. Syntax can be constructed arbitrarily and custom to each language construct if desired, which enables great flexibility in defining the assembler rules and programmability. △

All available language constructs define the **function set** of the architecture. These can be called directly from assembly via a defined syntax, such custom assemblers can be easily generated on the fly by high-level synthesis tools or manually designed. By composing functions and simplifying the calls, a side-effect of encoding bit compression can also be observed. The assembly code is completely data-independent, as it just calls the hardware functions and enables even larger (software) compositions. The application can be coded by means of the function set directly.

3.2.4.1 Composing rules

It is to be noted that language constructs are space-time mapped. The programmer has to follow two simple rules:

- input/output interfaces must match with the previous/next function in time (concatenation) and space (nesting)
- hardware binds reserved for the function at the respective time index should be free.

Basically a time-unrolled grid of the architecture is created, then filled up with the puzzle-pieces of the constructs according to (time) size and location. Assembling the application function is straightforward using the meaning-based high level of abstraction. Composition correctness is guaranteed by the above two rules.

For instance, the `mac` function needs two cycles to complete ($\tau = 2$) but not all resources are bound during the duration of the function. In the second time-cycle, mul_0 is done, leaving `PE0` available for something else while PE3 executes the addition part of the function. Language constructs can be arbitrarily nested with the limitation that it has to fit the available argument size. Functions can be arguments to other functions. Different classes of constructs are using different pools of elements and also combine in a different way. This enables a great degree of architectural flexibility.

3.2.4.2 Hyperfunctions

From the functional programming point of view, it is essential to be able to redefine functions, i.e. redefine the language constructs to select elementary functions (already existing in hardware) in different compositions and configurations. Due to physical hardware limitations, only a subset of the elementary functions are used at a time, limited by the number of the language constructs and their composition rules. This enhances architectural flexibility, by allowing a larger set of language constructs, albeit not all of them can be called simultaneously.

Hyperfunctions enable a tunable degree of architectural flexibility, even during execution, by internally modifying the hardware structure and composition rules via reconfiguration. By means of a hyperfunction, dynamic reassignment of elementary function selection rules and its coding rules (argument forwarding rules) is possible. Hyperfunctions make use of special hardware structures that are above the functional classification plane, and specialize on *redefining* language elements and executing these modifications physically in the hardware. It adds an extra dimension of flexibility to the architecture, by adding the necessary structures to remodel the internal structure itself.

Example: Redefinition of the previous `bcst` example for language slot `4’b0001` to a simple `add(r,r)` on `PE0` would imply forwarding argument bits in different patterns, activating different elementary functions and linking different wires. Three hyperfunctions execute this physically in hardware, as follows:

```
hf_delete(0001);
hf_redefine(0001:= {[31..29]:add, [28..27]:0,
                [26..24]:R, [23..21]:R, [20..0]:nop} );
hf_activate(0001);
```

After hyperfunction execution, normal language calls of position `0001` will have different *meaning* and will activate different elementary functions. Whenever the handle is executed, the respective pattern is set and arguments are redirected accordingly. △

Physical implementation of hyperfunctions is complex, but a solution is proposed in Chapter 6.

3.3 Conclusions

A new theory on how to exploit architectural flexibility via functional reconfiguration has been proposed, describing ways of constructing application-specific *function set architectures* directly in hardware via functional selection and composition of basic hardware function-pools, creating the necessary ingredients for energy-efficient designs. Moreover, the proposed framework allows full exposure of available hardware resources and a direct translation of the programmer’s intention into these, and proposes a different view on programming architectures with increased inherent flexibility.

Two major objectives are presented to achieve:

- a direct exposure of hardware resources and an organized way of combining these to an efficient *architectural language*. An architecture can be adapted using such a language to closely match the application requirements (by matching optimal application scheduling, available parallelism, data flow patterns, etc.) such that mapping and execution efficiency is increased.

- an increase of *architectural flexibility* such that the architectural language can be redefined and modified whenever the application (or its requirements) change. A tunable flexibility can additionally allow intra-application optimization, pushing further the efficiency envelope. This being done at a medium granularity, the advantages of fine-grained flexibility (FPGAs) and fast execution (ASICs) can be combined.

However, finding an architecture in the design space that allows a direct application of such a theory is not trivial. The insights of the theory presented in this chapter are focused on three theses, which will be explored and backed in the following chapters. In the next chapter, a quick way of exploring the design space and evaluating a selected design is presented first.

Chapter 4

Methodology for Exploring Functional Reconfigurability

In order to explore which architectural paradigm provides the best flavor of architectural flexibility, new ways of modeling and designing need to be explored, to cope with the huge number of options from the design space. In addition, shortened time-to-market and stringent quality constraints should also be respected by this methodology, if it is to be applied in industrial contexts. Clearly, one can not implement one particular design point from the design space just to check whether the chosen architecture meets the requirements or not. Even with flexible architectures, it is not clear whether a certain degree of flexibility is optimal to accommodate the application or not, unless some kind of performance or efficiency tests can be conducted.

In the industry and academia, two main design directions are prevalent for designing complex architectures, to compensate for the overwhelming complexity:

1. ***IP design and reuse*** – The system is constructed from off-shelf application-specific accelerators from a great variety of specialized vendors [5–7]. The initial development costs of the IP components are high and their reuse is only advantageous until applications remain unchanged. Evolving standards and better algorithms limit this advantage for cutting-edge applications because new components would be required to adapt to change.

2. ***High-level modeling*** – Designers move to a higher abstraction level for designing and evaluating SoC components [48]. High-level descriptions or libraries of high-level modules provide the building blocks to generate lower-level RTL or gate-level code automatically, short-circuiting large portions of implementation and verification effort.

As the first direction requires ready-made components that employ the traditional design chain of ASICs, it has the major disadvantage of long development cycles and lack of system design space exploration, because such tightly optimized components offer little flexibility, even when combined. Especially if multiple different applications need to be supported, it becomes less efficient to use dedicated components for each. Hence, the language of the application is not necessarily matched perfectly unless specific designs are created.

The second direction, looks more promising for exploring a large design space. In this chapter, based on my work [41, 135, 142][1], a methodology is presented to trace

---

[1] Parts of this chapter appear in these publications, reprinted with permission. ©2013, IEEE, ©2014, CRC Press, ©2012, Hindawi Publishing Corp.

the effects of flexibility and scalability in the sense of functional reconfiguration in a large design space. High-Level Design (HLD) and High-Level Synthesis (HLS) are two concepts that allow such an exploration at a high abstraction level. Several tool suites exist that expose the effect of every design decision directly at high abstraction, without having to actually fabricate or even simulate the architecture at gate-level. Of course, HLD and HLS can be used to design the IPs with custom flexibility that can be reused later, to align with the first design direction. Over 30 top semiconductor companies were already adopting HLS tools in 2010 [63], relying heavily on automated tool-chains to generate sythesizable hardware description code, in order to quickly and efficiently conduct design space exploration. In the remainder of this chapter, the options are discussed and a methodology, is proposed which fully exploits the benefits of functional reconfiguration theory.

4.1 Why Flexibility Is Key

According to my theory presented in the previous chapter, performance and efficiency is governed by how well the architecture suits the application. This can be controlled in terms of how well the language of the application (application requirements and characteristics) matches the language of the architecture (hardware-bound resources and how they can be used). While applications can be optimized by an optimizing compiler and other transformations to exhibit a certain language interface towards the architecture (albeit sub-optimally), the architecture language is fixed by the physical features present, or can be modified by a degree proportional to the architectural flexibility $\mathcal{F}$ of the architecture. For instance, arbitrary C code can be compiled for a target instruction set architecture (e.g. Intel X86), forcing the application to be described in the architectural language of the instruction set (e.g. ADD, MOV, etc.), thus changing its optimal application language interface (e.g. an ASAP-scheduled data flow graph with a high degree of parallelism) into a binary program which encodes instructions for the processor.

Therefore, to find an architecture that can match an application or a family of applications, careful selection of features and components and their interplay must be evaluated, along with the effects of inherent flexibility that the choice exposes. Moreover, a well-designed architecture also must allow some degree of scalability, such that the designer can leverage trade-offs in area, performance or efficiency, e.g. the number of parallel processors, cache memory size, size of the register file, etc.

To tune an architecture's flexibility to the needs of the application, one needs to experiment with the architectural types, language constructs, parallelism and pipelining options to see the effects on performance and energy efficiency. Figure 4.1 shows the architectural design space, an how it is spanned by flexibility, performance, area and power metrics [32, 166] and how functional reconfiguration can help a reconfigurable architecture to span features of multiple architectural types, reaping performance, power or flexibility benefits. Reconfigurable architectures have the ability to

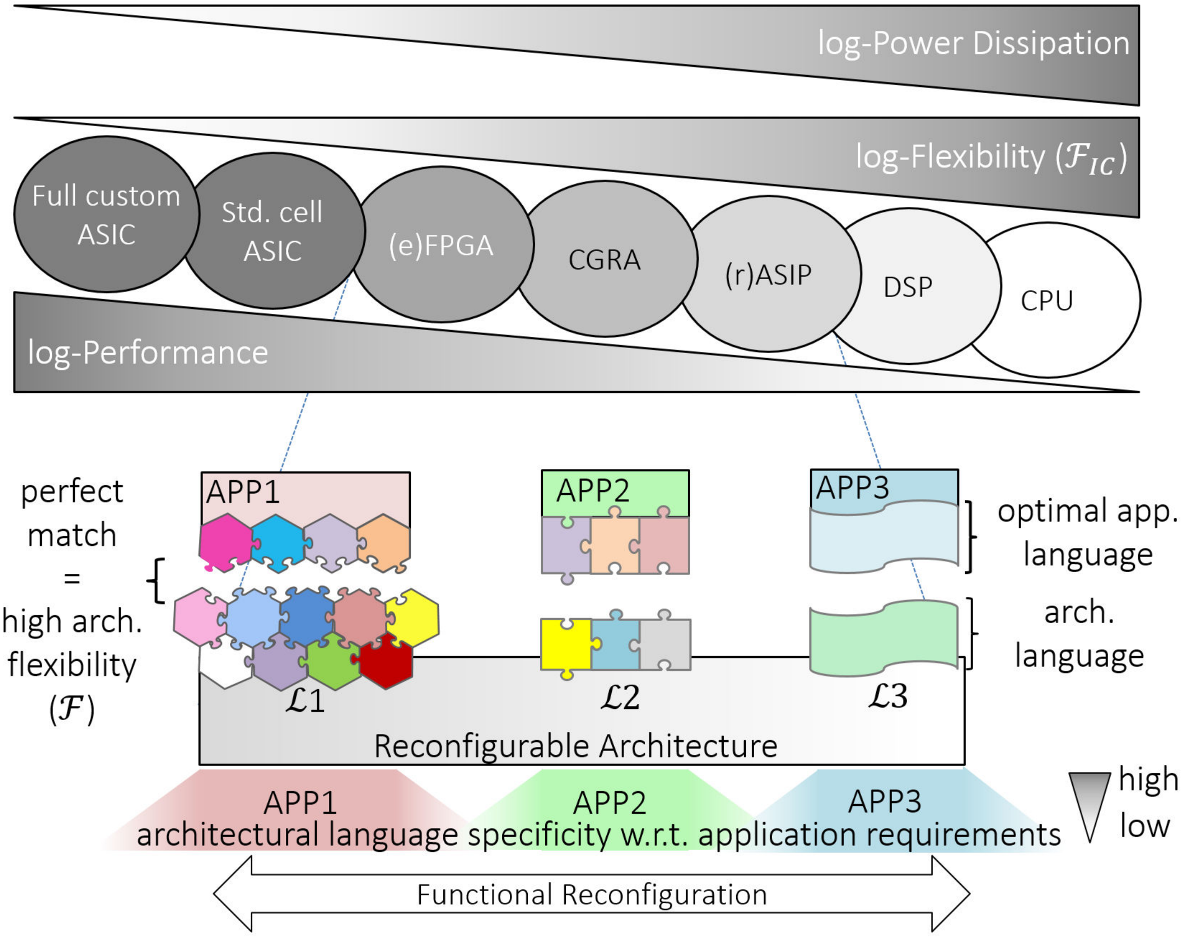


Figure 4.1: Perspective of the design space from power, flexibility and performance [32,166] point of view, enhanced with the concept of functional reconfiguration: a reconfigurable architecture with high architectural flexibility ($\mathcal{F}$) can tune its language ($\mathcal{L}$) to match different application-side requirements.

change their architectural language ($\mathcal{L}$) such that a perfect match to the application requirements can be realized.

For applications with very well defined needs in parallelism and specific data paths (APP1 in Fig. 4.1), only a very specific language set of the architecture resembling ASIC-like structures is efficient. Other applications that contain complex sequential code with little parallelism and occasional hot-spots (APP3 in Fig. 4.1), a DSP-like processor with a few custom instructions in its language set can yield great results.

A reconfigurable architecture, can perform such adaptation, via functional reconfiguration, if it has enough architectural flexibility to reconfigure its language. Matters become complex when different applications or entire domains have to be supported:

- Derivation of the right mix of architectural features extracted from the design space spanning FPGAs, CGRAs, DSPs, ASICs, CPUs, etc

- Quick construction and modeling of such an architecture
- Quick (coarse) evaluation and selection of different design points, refining
- Producing a synthesizable RTL or gate-level description of the resulting architecture

In the light of the points mentioned above, a methodology to easily support these steps is required. Moreover, it should support easy derivation and exploration of different architectures, architectural language elements, and structures based on functional reconfiguration.

An important side-effect of optimizing the architectural language is a significant decrease of Kolmogorov complexity $\mathcal{K}$ for describing the application in terms of the optimized language, formulated in Thesis 2 of Chapter 3. This enhances ease of programming and reduces mapping complexity.

4.2 High-Level Abstraction and High-Level Design Exploration

High Level Synthesis (HLS) and design is gaining traction in commercial and academic circles, as an answer to increasing design complexity and short time-to-market. It provides the perfect vehicle to allow exploration of the sizable design space and provides the power to quickly model complex architectures. In this section, a short survey on the HLS landscape is presented and modeling concepts are proposed to extract and exploit inherent flexibility for a commercially available high-level design tool. Structural descriptions, representation, tool set flexibility and limitations of HLS are discussed.

The HLS landscape is very fragmented, some targeting only a specific type of components and limitations. Since functional reconfiguration is a new concept that spans several architectural types, including CGRA, there is no off-shelf solution that delivers high-level exploration, simulation, RTL generation including tools support.

Table 4.1: High Level Synthesis Tools [135]

Tool	Input	Output	Target	Optimizations	Observations
Catapult-C [8,34] (now part of Calypto)	ANSI C++	RTL, SysC	ASIC	profiling, area, timing, power loop, memory, auto-verif.	highly tunable arch. gen. fixedp/floatp support, synth-scripts
Synphony-C [9]	C subset	RTL, sim	ASIC	algo, timing, throughput, auto-testbench	template-based several variants co-generated
Cynthesizer [113]	SysC subset	RTL	ASIC	constraint-based timing, auto-testbench	directive-based, tech-lib used for timing closure
Vivado HLS [10] (former AutoESL)	C, C++ SysC	RTL	ASIC	area, timing, power, loop interconnect, memory	good optimization options, tunable gen. data locality opt, fixed/float supp.
Bluespec [122]	BSV – Bluespec SystemVerilog	RTL, sim	ASIC	atomic transactions manual	designer-controlled synthesis quality high quality RTL gen.
Mimola [104]	Mimola ADL	RTL, sim	ASIP	manual	code quality suffers for complex designs
nML [58] now part of Synopsys [7]	nML ADL	RTL, sim	ASIP	manual	code quality suffers for multi-cyc. units limited multi-word instruction support
EXPRESSION [72]	EXPR. ADL	sim compiler	ASIP	manual, compiler optimizations	template-based, memory, constraint-based
Tensilica [5] (now part of Cadence)	TIE ADL	RTL, sim compiler	ASIP	manual, tunable auto	template-based, arbitrary extensions, IPs available
ReCore [11]	config	RTL	ASIP	modular design, optimized IP	customizable and reconfigurable DSPs and IPs
Processor Designer (now part of Synopsys [7])	LISA ADL	RTL, sim compiler assembler, linker	ASIP	manual, tunable auto, auto-testbench SoC interfacing, RTL co-sim	complete tool-set gen. complex arch. support, micro-step debugging, high quality RTL gen. and external RTL supp.
Menta [12]	eFPGA template	soft/hard RTL macro	e-FPGA	fine grained, tunable technology-independent IP	domain-specific, customizable template-based, embedded FPGA-like
PD extension [86]	LISA variant	RTL	CGRA	hierarchical, arbitrary interconnects	LISA language extension for CGRA RTL gen., interface gen., only RTL simulation
CGADL [105]	CGADL	SysC, (RTL)	CGRA	manual	heterogeneous, multi-context rule-based interconn., no RTL gen. yet
RaCAMS [123]	Java	sim	CGRA	high-level sim.	work-in-progress, no RTL gen. yet

4.2.1 High-Level Synthesis Overview

The landscape of HLS provides a large collection of tools both commercially and in academia, trying to tackle the design exploration problem and time-to-market constraints from different angles. Given the complexity of modern designs, various high level synthesis methodologies for quick architectural exploration are employed in industry and academia. These can be categorized into:

- *methodologies of direct translation* of high-level C language to hardware description like Mentor Catapult-C [8,34], GAUT [47] and Bluespec [122], yielding custom ASICs;
- *customizable processor design* such as Tensilica [5] and ARC [13], using highly optimized blocks as components;
- *Architecture Description Language (ADL)-based processor design*, creating fully flexible and custom processors, such as nML [58] and LISA [42] [7].

Generally, the problem is attacked from one of two sides: a) synthesizing high-quality architectures from application description at the expense of flexibility [8–10], b) providing the means to freely design, evaluate and create architectures using high-level description languages and generate tools for application support [5, 7, 58, 135]. A summarized non-exhaustive list of HLS tools is shown in Table 4.1.

Application-specific hardware generation usually allows high-level description of the application in popular code (e.g. C++), and after profiling, several optimizations are performed to generate circuit descriptions mirroring the application functionality. This limits architectural flexibility by synthesizing very specific circuits. In many cases, the designer does not have control over the generation process or choice of the generated architecture, limiting design space exploration. C-based HLS techniques offer no easy way to specify flexibility and custom processor designs often bring a lot of additional overhead in terms of fine-grained instruction execution. Quality of results are driven directly by the required constraints and internal optimization quality. Supported high-level input language set also plays an important role. Some tools support only subsets of a high-level language. [135]

On the other hand, tools that allow Architecture Description Languages (ADL) provide the designer with full flexibility over the design process, leaving it to the designer to explore the design space. The usual tool-set provides some form of high-level simulation environment, where early design decisions can be guided by iterative design and simulation loops. Quality and ease of exploration directly depends on ADL flexibility, generated simulator, helper tools and optimizations during RTL generation. Ideally, the environment should provide post-RTL-generation tools to allow easy application input (e.g. compiler support). [135]

For supporting a large variety of architectures, application-specific synthesis tools fall short of flexibility requirements. Languages like Bluespec [122], not targeted at a particular architectural type, could provide enough flexibility to describe arbitrary structures, however the tools support is limited. Solutions can be found for ASIP and

ASIC synthesis support and good ASIP simulators exist [5,7]. For CGRAs however, most tools focus on either RTL generation [86] or simulation [123], breaking high-level iterative exploration loop. [135]

In the proposed methodology, Processor Designer was chosen for the experiments, because it provides a large tool set and easy simulation, coupled with ease of exploration of different architectural variants in a short time. Another major advantage is that high-level LISA models can be instantiated at system level, as System-C interface wrappers of the high-level simulator can be generated with ease. Additionally as of late, custom RTL blocks can be co-simulated and taken into account at RTL generation, such as floating point units and in-house RTL code. Tool maturity, ease of exploration, auto-generated tools, but especially the flexibility of LISA ADL itself helped to experiment and push the boundaries beyond its intended purpose, revealing interesting results, which are detailed below. Experiments exist to extend LISA with reconfigurable architecture support [86], generating RTL for CGRA structures, however creation of mapping tools for these remains a big challenge. This tool is extremely versatile for spanning several architectural types with the proposed methods. [135]

4.2.2 LISA Language Overview

The Language for Instruction-Set Architectures (LISA) is an Architecture Description Language (ADL) which is used for modeling processors [7,42]. This language is a high-level language with C-like constructs and was part of the commercially available Processor Designer tool-set from Synopsys. As shown in Fig. 4.2, the design flow with LISA ADL allows generation of a synthesizable RTL description coupled with automatic generation of a set of tools such as C/C++ compiler, architectural simulator, assembler and linker. LISA has been developed to facilitate exploration, simulation and RTL code generation of processors.

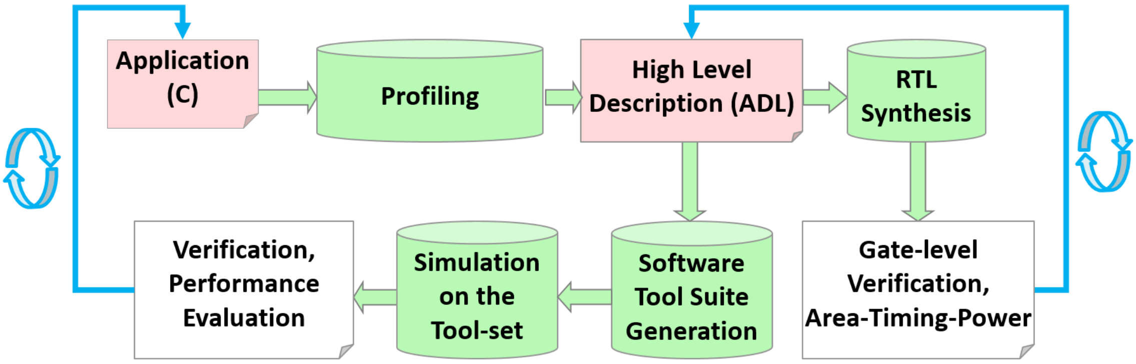
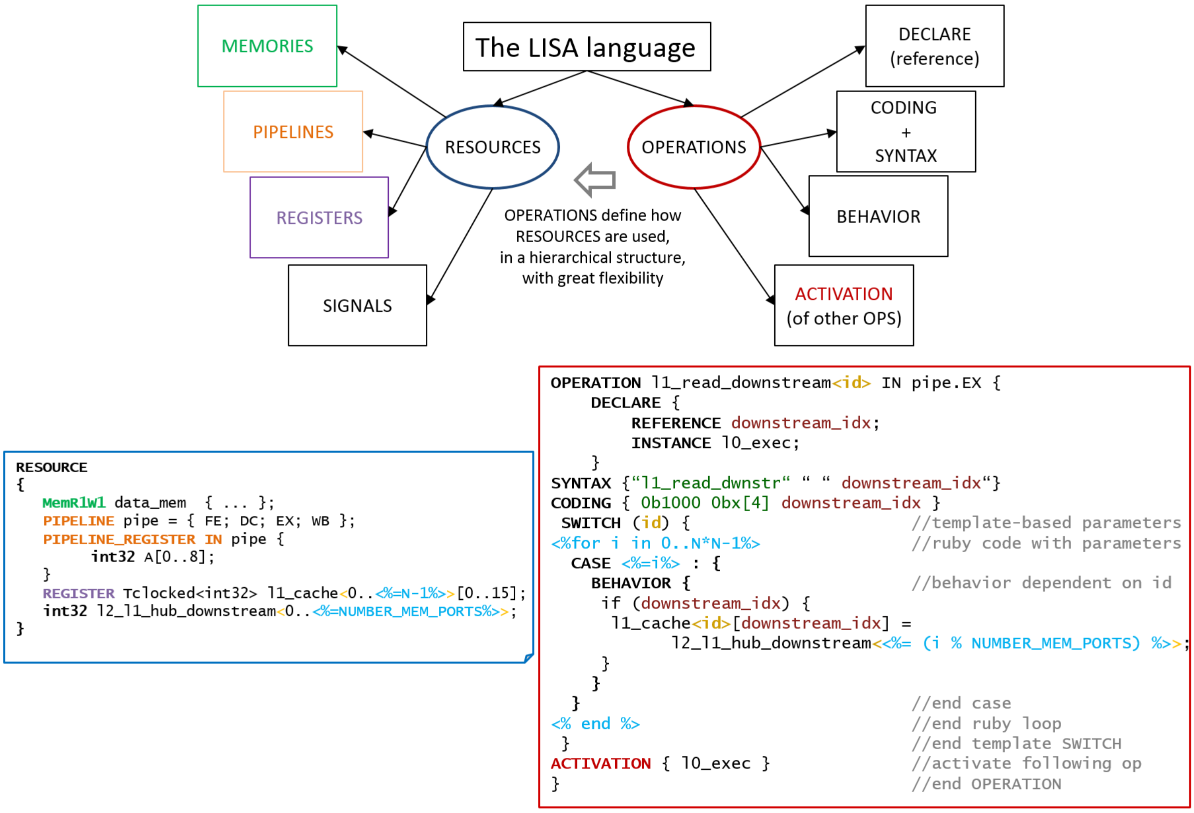


Figure 4.2: The standard LISA design flow [142]

The flow starts with an application described in C, which is profiled to expose computational hot-spots and give insight about what kind of structures would be needed in the architecture. Usually, starting from a skeleton template processor, the architecture is described using the LISA ADL, which represents the main input to Processor Designer. This generates the tool-suite specially tailored to the architecture, like the simulator (step-by-step debugger), the compiler, assembler and linker to run the application on the simulator. This first exploration loop allows major design changes easily, coupled with a quick performance evaluation in the generated instruction set simulator. Iterative design based on the performance evaluation allows incremental improvement on the LISA description. The second exploration loop finishes with the synthesizable RTL generation once constraints are satisfied also with the generated RTL code. If gate-level results are not satisfactory, the design exploration iterations can continue in either of the two loops.



```
RESOURCE
{
   MemR1W1 data_mem  { ... };
   PIPELINE pipe = { FE; DC; EX; WB };
   PIPELINE_REGISTER IN pipe {
        int32 A[0..8];
   }
   REGISTER Tclocked<int32> l1_cache<0..<%=N-1%>>[0..15];
   int32 l2_l1_hub_downstream<0..<%=NUMBER_MEM_PORTS%>>;
}
```

```
OPERATION l1_read_downstream<id> IN pipe.EX {
    DECLARE {
        REFERENCE downstream_idx;
        INSTANCE l0_exec;
    }
SYNTAX {“l1_read_dwnstr“ “ “ downstream_idx“}
CODING { 0b1000 0bx[4] downstream_idx }
 SWITCH (id) {                          //template-based parameters
<%for i in 0..N*N-1%>                   //ruby code with parameters
  CASE <%=i%> : {
    BEHAVIOR {                          //behavior dependent on id
     if (downstream_idx) {
      l1_cache<id>[downstream_idx] =
            l2_l1_hub_downstream<<%= (i % NUMBER_MEM_PORTS) %>>;
     }
    }
  }                                     //end case
<% end %>                               //end ruby loop
 }                                      //end template SWITCH
ACTIVATION { l0_exec }                  //activate following op
}                                       //end OPERATION
```

Figure 4.3: The mainstay of the LISA language: operations define how resources are used. The language is segregated in well-defined code sections, which can be easily parametrized and scripted with embedded scripting languages like Ruby.

The LISA language is built upon a C-like syntax, with special structures to model instruction set, timing, op-code and behavior of the processor. It has two main roots, as described in Fig. 4.3: `RESOURCE`s and `OPERATION`s. The `OPERATION` is the key hierarchical construct to describe such structures. Several `OPERATION`s can describe one or part of an instruction, or create a tree of mutually exclusive instructions, called `GROUP`s (e.g. ALU instructions). For example, `OPERATION alu` can contain child operations like `add` or `sub`, which in turn can be parents to special cases like adding an immediate

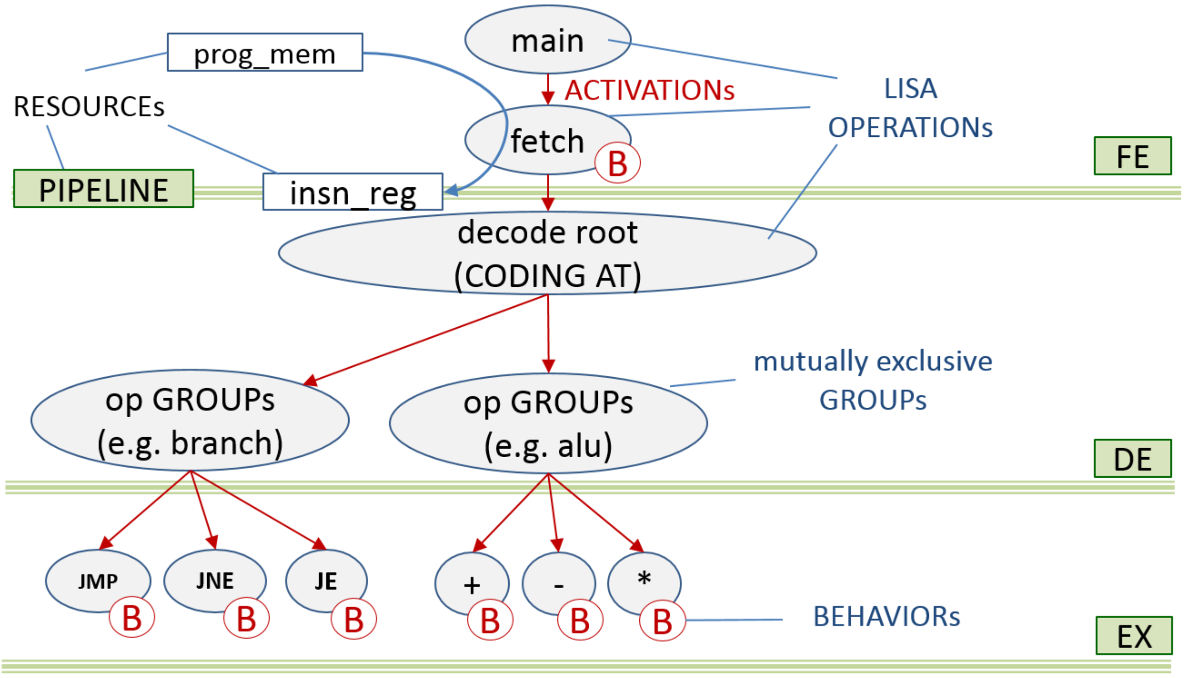


Figure 4.4: LISA language hierarchical tree: RESOURCEs, OPERATIONs, ACTIVATIONs and BEHAVIORs. In each time cycle, **main** is called and an instruction is fetched, decoded and executed.

or a register. Parent `OPERATIONs` can activate their children via the `ACTIVATION` section, which assures correct timing across pipeline stages. Each operation can be a member of a pipeline stage. Within this construct, arbitrary assembler syntax can be defined with `SYNTAX`, instruction encoding with `CODING` and instruction behavior with `BEHAVIOR`. In the `BEHAVIOR` section, plain C code specifies the arbitrary functionality of the instruction, and supports special data types such as `bit[width]` to allow close to hardware specification. The `RESOURCE` section is where global processor resources are defined such as memories, registers and signals, along with pipelines, and pipeline registers. With these constructs, a processor can be fully described. [135]

Once described, the code is simulated according to the described hierarchical tree, simulating each clock cycle by executing the `main` functions, as shown in Fig. 4.4. Custom instruction sets can be easily described, by adjusting mutually exclusive `GROUPs`, creating `ACTIVATION` chains and adding different resources, for instance multiple arithmetic units, custom sub-pipelines more memories. Complex architectures such as VLIW with deep pipelines can be described. The language exposes tremendous flexibility in defining how the resources can be used via high-level C code of the `BEHAVIOR` sections.

For more details about LISA language and tools, a comprehensive description is conducted in [38,42].

4.3 Proposed Methodology to Exploit LISA HLS Tools

4.3.1 Inherent Flexibility of the LISA Description and Design Space Coverage

LISA is a powerful and flexible tool for modeling processors. If a broader design space exploration is to be conducted for evaluating the coverage span of functional reconfiguration, LISA needs to be exploited beyond its original purpose.

Processors are limited from the architectural language point of view (the application must be described in terms of the instruction set). Thus a large number of applications can be supported, because the language of the architecture is static, and an optimizing compiler can be easily targeted to the instruction set. Due to the fine-grained instruction-based execution there is, in the broad sense, far more flexibility available in the design than required. This results in a performance decrease and energy inefficiency, because the transformed application is no longer executed efficiently.

Applying the theory from Chapter 3, the language of the architecture (in this case the instruction set) should be customized towards the application. Modifying architectural language to be more application specific, removes the flexibility ($\mathcal{F}_{\mathrm{IC}}$) to support a broader range of applications, while keeping architectural flexibility ($\mathcal{F}$) constant: basically the application specificity of the language slides from the CPU-side towards the ASIC side:

- *coarse-grained reconfigurable* structures which employs an array of word-size granularity, parallel, reconfigurable execution units with configurable interconnects for a greater amount of architectural language options
- *weakly programmable* structures, where processor data path is replaced by individual custom paths representing the data flow of the application, mimicking ASIC-like structures with an architectural language that tries to match optimal application language as close as possible.

Although LISA was designed for describing processors, the two architectural flavors described above require no modification of the traditional LISA-based design flow. This is accomplished by a shift in how the data path is viewed, structured and modeled in LISA only, enhanced by the inherent modeling power of the LISA language itself.

Control flow operations can be modeled by use of *State Machine* charts and *Single Qualifier Double Address (SQDA)* assembly. These are similar to finite state machines, where state transitions can be encoded as unconditional jumps. SQDA handles calling of states directly from assembly code. Data flow operations can be modeled by entering very specific behavioral code into the `BEHAVIOR` sections of LISA operations. These can be then attached to specific states of the state machine via `ACTIVATION`s. Modeling details follow in the next sub-sections.

LISA exhibits a few key features, which allows construction of these structures [135]:

1. **Activation** - automatic control of when to activate operations, in a specified order, even when crossing pipeline boundaries. Any operation with an `ACTIVATION` section can activate one or many other operations in the hierarchical tree. Complex chains can be designed, and sub-chains can be shared.
2. **Template operations** - originally added to support VLIW processors, template operations allow definition of several identical or quasi-identical operations. These constructs are similar to C++ templates, with the restriction that template variables have to be constant at compile time. For instance, when describing 8 identical ALUs, only one ALU needs to be described in a template form (e.g. `OPERATION alu<id>` generates separate instances `alu<0>` to `alu<7>`). The identifier `id` is static for an instance and can be used to differentiate it from another one in certain cases, or used to encode topological information in complex designs.
3. **Automatic tools generation** - besides the simulator, which helps evaluate the architecture, assembler and linker generation helps to immediately test the architecture partially or fully with the target application. This provides also means for passing configuration data of configurable architectures.

In the remainder of this section, I propose modeling abstractions using simple examples, from which it is straightforward to generalize for complex designs.

4.3.2 Proposed Exploration Flow: Towards Application-Specific Architectural Language

In order to target architectural language to a specific application, the architecture must be stripped of unnecessary structures. Control and data flow execution must be transformed as closely as possible to ASIC style of execution. The fetch and decode logic that supports tens of instructions in a normal processor (i.e. architectural language elements) needs to be reduced to only those that would be relevant to the target application. Generic ALUs should be customized to accommodate frequent data operations within the application. A deep analysis of the target application is necessary, as such algorithm-level analysis reveals the required operations, their parallelism, their frequency and potential hot-spots which would benefit from custom-made data paths. This creates an architectural language tailored for that specific application, with the advantage of a better match with the optimal application language, i.e. best way to execute the application. Of course, the downside of this that other applications can not be executed anymore [41].

4.3.2.1 Control Flow: Modeling a State Machine in LISA

Fig. 4.5 illustrates how to represent a **state machine, i.e. control flow** of an application using a minimal representation with a processor′s fetch/decode and (conditional) jump structures. Such structures can be modeled very easily in LISA. The state

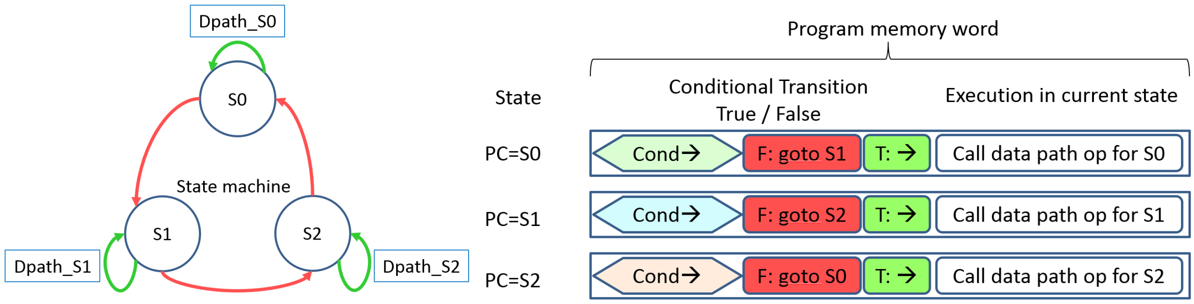


Figure 4.5: A simple state machine with state transitions and executed data path operations represented in program memory of a weakly programmable processor with an application-specific architectural language [41].

machine is encoded into the processor's *program* while the state transitions are the *instruction words* within the program memory. The program counter (PC) is the indicator of the current state, kept in a register of minimal size ($log_2(nr\_states)$). The conditions represent the state transition conditions, which can be considered as guarded (predicated) execution, based on input and/or current execution state. Depending on the outcome of the condition, a *branch* is issued to the next state transition (PC is updated) or a call to the computation linked to the current state is executed. The processor fetch logic will fetch the instructions for the new state. Unconditional jump or execution is just as easily possible, by forcing a condition to always evaluate to true or false. To further customize these architectural language hooks, application-specific conditional evaluation is created in hardware, as LISA `OPERATION`s with specific `BEHAVIOR`, and encoded as an instruction in the decoder [41].

4.3.2.2 Data flow: Modeling a Custom Data Path with LISA

For the **data path** part of the application, a custom structure can be created for every state. This encompasses custom memory reads, writes and execution grouped into one complex operation, then encoded into a specific instruction. Basically, it creates a collection of custom architectural language elements which encode relevant data path processing parts of the application. If control flow conditions evaluate as true, these elements are called at the right time, thus executed. To allow resource sharing of large structures such as multipliers, data paths can be pipelined in order to separate common execution from state-specific reads and writes from/to registers and memory [41]. A data-path can be linked via `ACTIVATION` to a state, which is activated every cycle as long its corresponding state is active (`pc` has the required value), or the transition condition evaluates to true (conditional activation). Storage of partial results or local data is distributed across the data path by introducing local variables and `REGISTER` resources within the `BEHAVIOR` description. This avoids using large register files with complex multiplexing access logic.

Using high-level design tools, these major architectural changes are easily supported by automatic generation of new assemblers, linkers and simulators for every tested change [41]. Architectural flexibility is achieved with in-depth customization of each resource, yielding a better match to application language.

Separation between data path operations and the state machine allows clean partitioning into pipeline stages and resource sharing. This concept has been explored in a collaboration paper [165], with automatic generation of a state machine from application code, then using an architectural template to generate LISA code.

4.3.3 Proposed Exploration Flow: Variable Architectural Language for *Tunable* Architectural Flexibility

As mentioned at the beginning of this chapter, tunable architectural flexibility is key to be able to modify and adapt architectural language elements to application requirements. This is possible with a reconfigurable architecture by combining its elementary hardware functions in various ways. The level of granularity of these elementary hardware functions defines the amount of flexibility, i.e. how many new language elements can be formed by combining them. In LISA, reconfigurability can be described, controlled and simulated, using the following considerations [135]:

4.3.3.1 Structural considerations

LISA is efficient when describing linear data paths which are easy to pipeline. However, in 2D structures such as CGRAs, data flow is difficult to cleanly partition into LISA pipeline stages. For CGRA structures with regular, shared resources, modeling granularity in LISA must be reduced, to expose interconnect, topology, configuration and data flow. Data can flow in any direction within the mesh, forcing to model the complete data path into one LISA pipeline stage and creating an explicit 2D pipeline between the units, by forcing all processing element outputs through a register. Interconnect and PEs must configurable and topologically well defined. For managing and relaying configuration data, regular LISA pipelines can be used, which also allows to add control flow processing to the CGRA, similar with those described in the previous sub-section.

4.3.3.2 Modeling similar resources and topology

A powerful feature of LISA is the template `OPERATION<id>` used to topologically define one instance. Interconnect can also be modeled with templates, since instance `id` must be constant at compile time. This excellent feature is less error-prone, because only one template description is required, no matter how many times the operation is instanced. Moreover, it permits great scalability by just parametrizing the template `id` bounds. Addition of more interconnect lines or extending array size of a CGRA can be done buy just increasing `id`.

For the 3×3 mesh from Fig. 4.6, for each PE an `id` is assigned and all resources which are incident to or used by that PE receive the same `id`. For uniquely linking a resource to a PE automatically, template signals or (template) register arrays are defined. Thus, `pe<4>`, will have input signals `a<4>`, `b<4>` and output register `out[4]`.

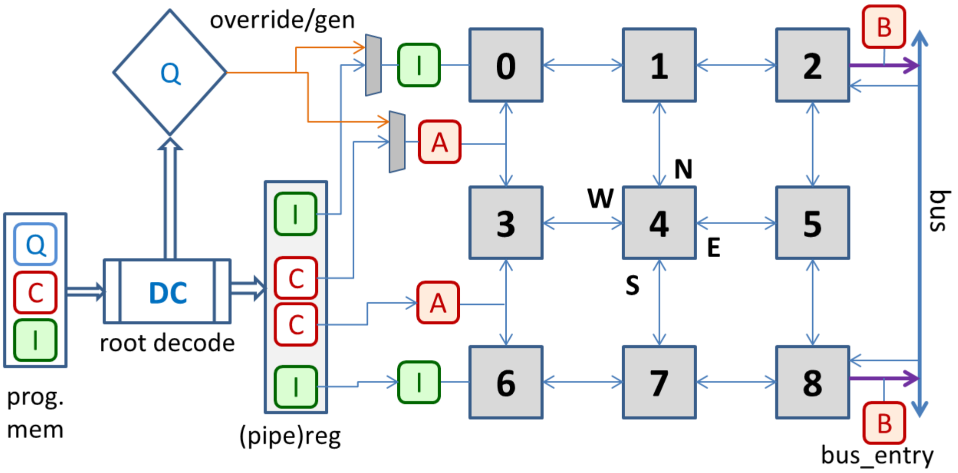


Figure 4.6: A basic CGRA structure to show architectural elements. Q is conditional call, C is configuration, I is processing element instruction. A and B are activation signals. [135]

Resources must be declared globally, since multiple `OPERATION`s need access to them, without a hierarchy (e.g. `a<4>` will be written by the resulting signal from either N,S,E,W sources, described also as `OPERATION`s). Based on topology, certain `id` values will be invalid, controllable with either `SWITCH()/CASE` or `IF()/ELSE` LISA keywords, e.g. `pe<0>` can not have a north input, so `OPERATION north_src<0>` has no `BEHAVIOR`. The same constructs can be used to describe a heterogeneous array adding conditional behaviors and local resources for certain `id` values, e.g. `pe<4>` can also activate a divider. [135]

4.3.3.3 Configurable interconnect

A configurable wire from *source* to *sink* is modeled in LISA as an `OPERATION` with a simple assignment in `BEHAVIOR{sink=source;}`. Conditional activation of this operation acts like a configuration bit and *sink* is written (Fig. 4.6, between PE0 and 3). For multi-entry wires, *sources* are modeled as additional *entry* operations, activated by mutually exclusive activation signals (*B* in Fig.4.6), multiplexing them. In turn, the bus *sink* wire acts as PE *source* for the east link of PE2 and 8. Broadcast is possible, if other PEs are also reading the bus wire. Regular topological structures are modeled by template operations using derived topological rules (e.g. in Fig. 4.6, all north source links can be defined as `n_src<id>=out<id-N>`, excepting the first row with `SWITCH/CASE`, where `N=3` is the $N \times N$ array size). For correctness of simulation only, proper activation order is required, so that *sink* is not read before being written, using `WRITES_BEFORE/_AFTER` keywords, when defining the operations. [135]

4.3.3.4 Configuration data can tune the language

LISA modeling offers several ways to pass configuration data to a reconfigurable structure. Most straightforward is relaying the configuration via assembly, directly specifying configuration bits which are then distributed without any decode logic to *configuration registers*. A simple configuration root operation conditionally activates those operations which have a `1` in their respective registers, called unconditionally at every cycle. These registers are globally declared, or part of a pipeline register, as shown in Fig. 4.6. For dynamic reconfiguration, a very large instruction word can be fed every cycle, quasi-static configurations can be loaded once and read every time from a register. An interesting feature is to conditionally generate or override configurations, if control flow operations Q are added using concepts from ASIC-like design (Fig. 4.6), effectively creating self-reconfigurable circuits depending on execution state. `ACTIVATION` signals can be used as operand or clock gating enable. This also can save power, if a resource (tree) is unused, it is not activated, hence generates no switching activity.

Language constructs can be combined or destroyed via configuration patterns. If the elementary hardware functions are coded as LISA operations at a small granularity, configuration data can provide the necessary information to select and combine them.

4.3.3.5 Exploiting assembler and scripting

The automatically generated assembler and linker has enough flexibility to allow definition of an arbitrary syntax. A well designed syntax avoids cumbersome derivation of configuration bits, but also allows direct control and access to the architectural language. For instance, one can encode the activation of the north source link to fixed symbol `"N"` instead of the multiplexer selection bit value. Coupled with a good PE syntax, one can program a PE with simple directives, e.g. `PE4: NS(+)`, for taking north/south inputs and add them on PE4 or `PE8: WE(-)` for west/east inputs with subtract. The assembler then replaces this with the respective binary configuration bits. The instruction word can be further simplified and partitioned into regions, just like a VLIW processor and lightly scripted with easy scripting languages such as embedded Ruby or Perl, empowering the assembler to behave like a meta-compiler. A very detailed long instruction word syntax can even encompass open/closing braces, can be indented arbitrarily, exploiting the automatically generated parser of the assembler to help easily write and configure such code. [135]

Since LISA code can be tedious to write for a large number of similar operations, `SWITCH/CASE` exceptions, `DECLARE` sections, scripting can be employed to parametrize and generate such code. Especially for regular interconnects, which can be generated by a rule, it is easier to code the rule in the embedded script part, further increasing description flexibility: e.g. a scalable $N \times N$ reconfigurable structure, exploring and generating RTL for all N values of interest by just replacing a constant in script headers. This works also for structures that are normally not parametrized, such as large number of pipelines (e.g. each PE has its own pipe). For the architectures

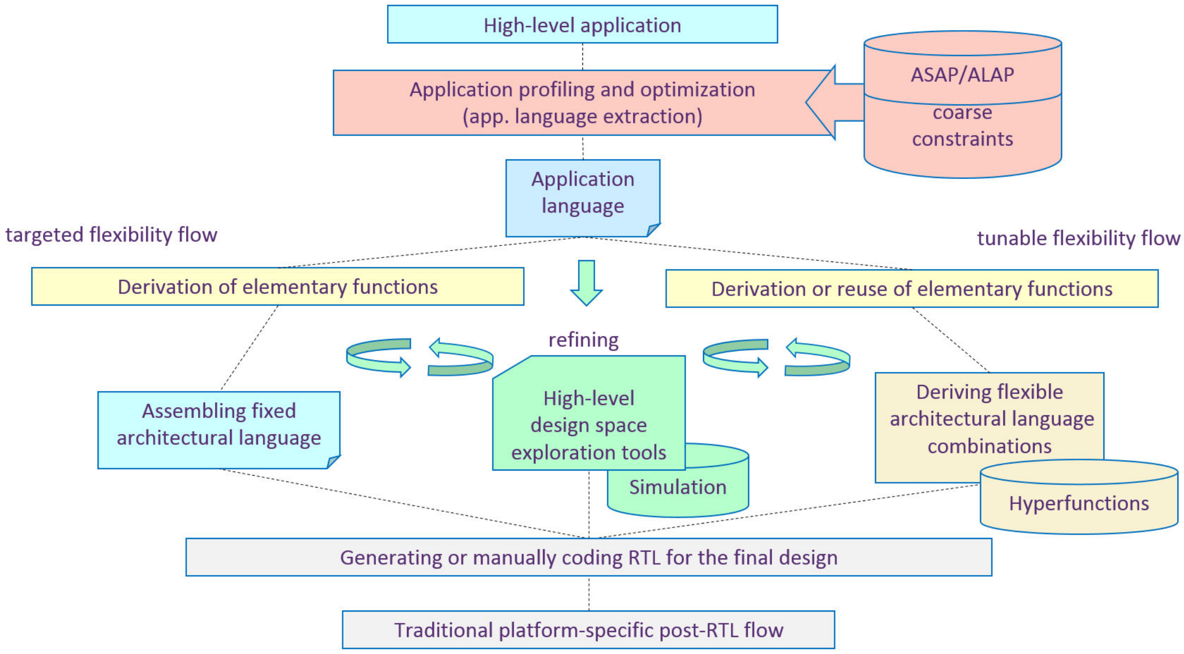


Figure 4.7: The proposed methodology flow.

implemented in this work, an average LISA code size reduction of 50-60% can be achieved by scripted LISA. Moreover, scripting eliminates coding mistakes such as copy-paste errors, avoiding tedious debugging (bad rule in the script is immediately visible). [135]

4.3.3.6 Limitations

Due to the fact that LISA ADL was designed for sequential, pipelined data flow, there are some limitations when describing CGRAs. Special care must be taken for the `ACTIVATION` tree design, as LISA does not support circular dependencies. For instance, the data source of one of the row and column broadcast pair must be limited, to break a circular activation dependency (read_row→write_column→read_column↛write_row). Another limitation is that LISA does not support multiple coding roots (`CODING AT`) statements, limiting decoding tree options. This is also present in the assembler, not allowing multiple program memories (e.g. separate instruction and configuration memories). Furthermore, for very large CGRA designs, the 32-bit binary of the LISA tool-chain can hit memory limits. [135]

4.3.4 Resulting Methodology

To summarize, the top level view of the complete flow is illustrated in Fig. 4.7. In the case of targeted flexibility, the elementary functions and the language elements can be tailored to match the application as closely as possible. For the tunable flow, existing architecture with existing elementary functions can be reused, only the language elements which map well with the application have to be derived.

High level exploration is not limited to LISA only, other tool-chains can be employed as well, as long as quick evaluation of the options, language suitability and constraints fulfillment can be done. Finally, after generating or manually coding the RTL code, traditional post-RTL tool-flow can be used.

4.4 Summary

In this part, modeling concepts for architectures using functional reconfiguration are proposed. Functional reconfiguration is applicable to a variety of architectural classes. A quick and wide exploration and evaluation of design points of the large design space is required, which is why the inherent power of a high level design language and tools is integrated into a flow. Enhancements for going beyond the original design purpose of such tools are also discussed for a commercial HLS tool:

- LISA ADL and the generated tool-suite can also be used to design *ASIC-like* or
- *coarse-grained reconfigurable* architectures additionally to ASIPs.

This enables *tunable architectural flexibility* to adapt exactly to the application's requirements. Using this methodology it is possible to explore the effect of scalability and flexibility from a high level abstraction view and opens new possibilities in adapting and fine tuning architectural features to match the target application. In the following chapters, such an exploration is conducted.

Chapter 5

Targeted Flexibility: ASIC-like Structures for Low Energy Consumption

In the previous chapter, exploration methodologies for functional reconfiguration were presented. This chapter delves into the first proposed direction: reducing architectural flexibility to a bare minimum, such that the application(s) are supported in a very efficient way. Reduction of architectural flexibility means a reduction in the flexibility of supporting changes in the application or replacing support for new applications, however, it also focuses the architecture towards the target applications for increased efficiency. The philosophy behind it is to get as close as possible to ASIC-like structures and reduce unused elements and overhead, optimizing for the given application and thus achieve higher performance and lower energy when compared with generic solutions. The gains in performance and power are proportional with how well the architecture matches the application. A fine trade-off between removed flexibility and usability versus gained efficiency must be conducted to achieve the optimal balance. In the following, this chapter treats a scenario in the wireless communication domain, based on the results of my work published in [41,141,142][1] and a collaboration work in [165].

5.1 Towards ASIC-like Architectures

As per the definition from Chapter 3, architectural flexibility $\mathcal{F}$ reflects the degree of how well the language of the architecture $\mathcal{L}$ matches that of the application. A perfect match translates into high performance and efficiency due to the architecture fulfilling the application's requirements in term of parallelism, data bandwidth and high execution speed. Of course, a perfect match can be achieved also by an architecture that is *over-designed* – an effective overkill of hardware resource availability with respect to application requirements. While this also gives a high degree of architectural flexibility to potentially support other applications, the overhead associated with unused resources chiefly causes high area and high (leakage) power usage, among other disadvantages. Minimizing the excess of architectural flexibility and focusing it only to match the target application leads to an ASIC-like design philosophy. If a single application is targeted, pure ASIC design methodology yields the best results,

---

[1] Parts of this chapter appear in these publications, reprinted with permission. ©2011-2012, IEEE, ©2014, CRC Press, ©2012, Hindawi Publishing Corp.

however when several applications (and only those) need to be supported, using an ASIC for each is very inefficient from area, power, resource sharing and design-effort perspective. Important questions answered in this chapter are:

- how to reduce over-design such that architectural language can closely match requirements
- how to share resources between target applications
- what are the potential benefits versus the ASIC approach
- how architectural language granularity affects architectural flexibility (and thus efficiency)

5.1.1 Target Scenario: Minimal Flexibility to Support Two Applications

For the case study, representatives of two complexity classes of Wide-band Code-Division Multiple Access (WCDMA) channel estimation algorithms are selected. Aiming to reduce excess architectural flexibility to perfectly match these two target applications, the effect of different design factors that influence final energy efficiency is explored.

The background scenario was chosen from the wireless domain, because flexibility at architectural level would have major impact. Due fast changing standards and process technologies, mobile devices increasingly rely on the Software Defined Radio (SDR) and Cognitive Radio [115] [116] concepts to achieve adaptability, flexibility, spectral and energy efficiency. SDR is envisioned to possess enough flexibility to enable seamless upgrades and support for multiple standards clearing the way towards cognitive radio, at the software level.

However, SDR implementation presents an interesting challenge for the architecture designers, namely, to develop an underlying hardware platform for SDR with fine balance of performance and flexibility. This demanding problem led to major research activity in recent years [19,35,43,52,67,99,144,157,160,166]. One of the key ingredients in the SDR architecture design is to determine the algorithmic kernels across various standards. While the kernel can be implemented in the most efficient manner, it can be re-targeted to different standards by means of tunable parameters or weak programmability. To that effect, the final architecture can be an ASIC, a reconfigurable platform or an application-specific processor. The complete system is often built by combining such accelerators, targeted for different blocks of a wireless standard [67]. In a scenario where the architecture has to switch from one application to another, with SDR this can be done easily by just running another program. Application changes are managed by software, therefore hardware acceleration is best-case limited to isolated application-specific blocks.

In this part, flexibility is moved to the architecture, switching of applications is done at hardware level via a software interface. Architectural flexibility is maximized to match both applications, to gain maximum efficiency in switch scenario detailed next.

5.1.1.1 Case-study Scenario

A critical part of wireless communication is maintaining a good level of signal-to-noise ratio (SNR) on the link. This is influenced negatively by multi-path fading, mobile terminal speed relative to the base transmitter, scattering, shadowing, etc. To counter this, channel estimation (CE) is performed so that corrections can be done by considering dynamically altering channel conditions. CE constitutes an important building block for SDR, as this is used across multiple wireless standards.

There are 3 large classes in which one can categorize CE algorithms:

- *1)* low-complexity, low-performance algorithms;
- *2)* high-complexity, good performance algorithms; and
- *3)* extremely complex, iterative algorithms with near-optimal performance.

While *1)* deals with simple (linear) interpolation algorithms and improvements on those (typically $O(n)$ complexity), *2)* is the class where still tractable $O(n^2)$, $O(n^3)$ complexity yields high gains in performance, typically in orders of magnitude. Class *3)* employs iterative (data-aided) expectation-maximization algorithms with $\geq O(n^3)$ complexity, are typically unfeasible for implementation in software when considering the performance improvements that they yield. Hardware acceleration of this class is currently under heavy research. [141].

The selected target applications are two multi-user WCDMA pilot-aided CE algorithms: **polynomial interpolation (PI)** (class *2)*) proposed by Yue et al. in [168], and **weighted multi-slot averaging (WMSA)** (class *1)*) proposed by Abeta et al. in [15]. The rationale for this selection is the following: in the context of cognitive radio and low-power mobile devices, wireless link state not always requires a class *2)* algorithm performance, hence selecting a lower complexity class *1)* algorithm could yield significant efficiency increase. This is due the fact that more complex algorithms require more hardware resources and more processing to finish. Exposing and exploiting the structural similarity, and adapting the architectural language accordingly, it is possible to design an architecture which can adaptively switch among the two, without having the area overhead of two separate dedicated circuits [141]. The architecture should be able to adapt to weak input signals by using the high-performance, high-complexity algorithm while switching to a low-performance and low-complexity algorithm when the input signals are strong enough to maintain quality of service, saving energy. [41]

5.1.1.2 Architectural Background for the Target Algorithms

Over the years there have been several approaches to SDR architectures based on different architectural approaches. However, with the ever increasing complexity of new wireless standards a migration from flexible solutions towards clusters of inflexible ASICs can be observed.

Processors are flexible enough to implement complete standards. Architectures like SODA [99], EVP [157] and Imagine [84] tackle performance and power demands by employing high-speed vector processing or stream processing. In these architectures data parallelism is explicitly exploited. In SODA, a Single Instruction Multiple Data (SIMD) architecture is employed, where one ARM processor is coupled with 4 parallel processing elements. Tight control of bit-width and bandwidth fulfilled power and performance requirements of two wireless standards. EVP takes a Very Long Instruction Word (VLIW) front-end to control several optimized SIMD units for specific SDR tasks, capable of handling multiple standards. Imagine is a media stream processor which has been re-targeted for base-band processing in works like [133], exploiting clusters of parallel processing elements controlled by a host processor.

Application Specific Instruction-set Processors (ASIPs) sacrifice flexibility in order to gain enough performance to tackle the more demanding applications from more recent algorithms. The FlexiChaP architecture [19, 160] customizes the pipeline, execution units and data flow of a processor to accommodate convolutional, turbo and LDPC decoding families, yielding an order of magnitude of speed-up compared with fully flexible processors like SODA.

Coarse-grained Reconfigurable Architectures like ADRES [109], RaPID [56], MorphoSys [151], RAW [156], Montium [144], IMEC coarse-grained accelerator [35] employ arrays of data word level reconfigurable processing elements linked by a reconfigurable network, which can be tailored to a wider family of applications. Such coarse grained cores can cover a wide flexibility/performance range between ASIC and ASIP, like the application-specific FlexDet [43] or an ASIP-coupled rASIP [86]. Although this class promises SDR implementation capability due to excellent computational density [50], the difficulty in programming and exploring the design space of such architectures discourages wide-spread adoption.

System-on-Chip solutions like Sandbridge [67] are increasingly popular, especially when high-performance scalable ASIC cores [166] are employed to construct SDR components. Even hybrid approaches using accelerators and reconfigurable units are advocated [52, 62].

Field-programmable Gate Array (FPGA)-based designs for SDR, like the WARP board, are extensively used for prototyping and research of new wireless standards and optimizations [71] [153], but power requirements make it prohibitive for end-products.

All these solutions except the ASIP/rASIP approach need “manual design” on either the hardware or the programming side or both. In this work, adapting HLS methodologies described in Chapter 4, architectural flexibility is adapted to support these two algorithms. In this regard, the approach, the architecture and the concept differentiate this work from existing solutions.

5.1.1.3 WCDMA Implementations

Extensive work has been conducted by Rajagopal et al. during the early stages of WCDMA research to implement channel estimation and detection on stream processors and compare it with traditional DSP implementations [131–133] and also a VLSI implementation is conducted in [130], where area- and time-driven implementations are explored.

In case of the implementation on DSP, it is shown that the time required for computing channel estimation is 600ms in case of 32 users [131], which is far too slow for real-time requirements. A dual-DSP and FPGA hybrid is shown to reach the real-time requirements for up to 7 users, however the implementation also contains detection [132].

The implementation on the Imagine stream processor simulator shows major improvement over the DSP, but only the number of cycles could be extracted [133] which are at least an order of magnitude higher than the number of cycles reported in this work. Also, it is worth noting, that the stream processor architecture uses 8 clusters of 3 adders and 3 multipliers which not only implies large area but also great power consumption. The computational hot-spots of matrix-matrix multiplication are implemented as a series of matrix-vector iterations which requires a large number of cycles in the stream processor also due to data load/stores and movement.

For the VLSI implementation [130], the algorithm was analyzed and redesigned for efficiency, considering fixed/floating point representation trade-offs (up to 16 bits) and their effect on bit error rate, however no direct comparison with this work could be made for several reasons: the design has not been synthesized, operating frequency is assumed and area is expressed in terms of full adders, with no mention of storage. Additionally, no power consumption data has been reported.

Nowadays, such a small block of a WCDMA receiver is too deeply integrated into high-performance SoC solutions, making an individual data extraction and direct comparison impossible, especially since this standard has been obsoleted by newer, even more complex MIMO OFDM algorithms. However, this case study is sufficient for the purpose of demonstrating the proposed theory and methodology from the previous chapters.

5.1.2 Identifying Options: Target Algorithm Analysis

5.1.2.1 Differences in Algorithmic Performance

A comparison between the two target algorithms is presented in [168], showing that these two algorithms differ significantly in terms of performance, under multi-path fading conditions. For single user single antenna systems, the Bit Error Rate (BER) of PI is lower than that of WMSA over the whole bit energy to noise energy ratio $\frac{E_b}{N_0}$ range and more than an order of magnitude less when $\frac{E_b}{N_0}$ is greater than 6dB. In case of multiple antenna, WMSA is outperformed by more than 2 orders of magnitude. The algorithm performance of PI stays superior for normalized Doppler frequencies in the range of $0.005 < f_d T < 0.013$ for both single and multiple antenna cases at an SNR of 8dB. For multi-user systems, the performance of PI in rake receivers stays superior over that of WMSA over the whole range with the difference reaching 2 orders of magnitude when iterative interference cancellation is employed in medium to high

SNR (>7dB). [141] On the other hand, as it is analytically shown later, WMSA involves significantly less computation compared to the polynomial interpolation method.

5.1.2.2 Channel Estimation with WMSA

WMSA [15], is based on linear interpolation of known pilot symbols, and has low computational complexity. For every k-th user's l-th path several N_p pilot symbols of slot m of the slot window are averaged first from received signal r_{kl} and initial estimate b_k:

$$\hat{\eta}_{kl} = \frac{1}{N_p} \sum_{n=1}^{N_p} r_{kl}(mN_s + n) b_k(mN_s + n); \tag{5.1}$$

where $l = \{1, ..., L\}; k = \{1, ..., K\}; n = \{1, ..., N_s - N_p\}$. The averaged values of several pilot symbols in a slot are weighted with pre-computed coefficients α according to (5.2), to generate the estimates $\hat{g}_{kl}$ for each data symbol N_d, as Yue et al. summarized it in [168]. The values of the coefficients α are thoroughly deduced and analyzed in [15].

$$\hat{g}_{kl}(mN_s + N_p + n) = \sum_{j=-J+1}^{J} \alpha_j(n) \hat{\eta}_{kl}(m + j) \tag{5.2}$$

Table 5.1: WMSA complexity: number of divisions, addition/subtractions and multiplications [41]

Tasks	Complexity	Storage	Execution
calculate α	$div : 2N_s$, $add/sub : 6J$	$2J \times N_p \times N_s$	once
average pilots	$div : 1$, $add/sub : N_p$	N_p	every N_s
calculate estimate	$mul : J$, $add : J$	1	every N_d

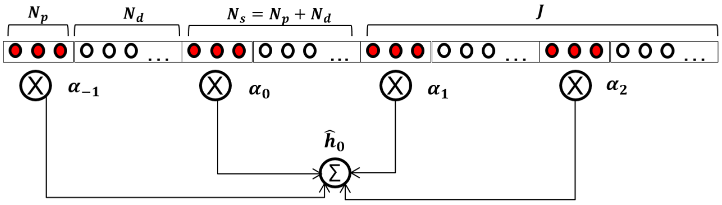


Figure 5.1: Weighted Multi-Slot Averaging: Pilot symbols N_p are averaged, weighted ($\times\alpha$) and summed for the channel estimate. [141]

5.1.2.3 WMSA Analysis

In Fig. 5.1 WMSA concept and in Table. 5.1 the complexity of each of the sub-tasks of the algorithm is shown. The task for computing the α coefficients is executed once. The coefficient set can be changed based on WMSA algorithm parametrization and partially depends on the estimated symbol position. *Averaging* has to be done once per slot, in case there are several pilots in a slot. Then, for each data symbol, the estimate is calculated by summing the products between the averaged pilot value and the corresponding α coefficients of the symbol and slot. The dominant parameters of this algorithm from the complexity point of view is the size of the slot window $2J$ and the coefficients α. The larger the analyzed window, the greater the amount of needed storage. For the same J, storage needed for WMSA does not exceed that of polynomial estimation. [142]

From the application language point of view, WMSA needs following support from the architecture:

- parallel add/sub and multiplier elements, according to slot size (execution class)
- loads/stores or streaming i/o for the new pilot symbols and the channel estimate and the α-coefficients (memory class)
- one optional divider (execution class) – as window sizes are powers of 2, shifters can be also employed

There are no special communication class requirements, as the data can be immediately calculated in a pipeline. An architecture that provides these elements, would execute this algorithm efficiently.

5.1.2.4 Channel Estimation with Polynomial Interpolation

The second algorithm is based on polynomial interpolation (PI) (Fig. 5.2) of the pilot symbols' channel values to calculate an approximation on channel fading.

As described in [168], the channel values are fit with a polynomial model of order q over $2J$ slots (5.3). Approximation is done by minimizing the mean-square error α based on the pilot symbols N_p in (5.4); N_s being the sum of N_p pilot and N_d data symbols. This translates to a Lagrangian interpolation problem, solved with (5.5) in (5.6), where $\boldsymbol{\eta}_{kl}$ represents the transpose of the pilot symbol vector constructed from $2J$ slots.

$$\begin{aligned} \hat{g}_{kl}((m+j)N_s) &= \sum_{i=0}^{q} \alpha_i \psi_i(jN_s) \\ j &= -J+1, ..., 0, ..., J \\ \psi_i(n) &\triangleq n_i; \qquad i = 0, 1, ...q \end{aligned} \tag{5.3}$$

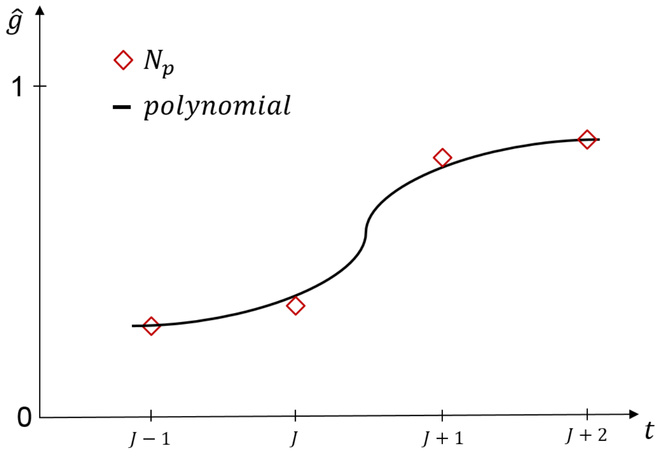


Figure 5.2: Polynomial Interpolation concept: several data points are interpolated via a polynomial of order q.

$$\begin{aligned} \boldsymbol{\alpha} &= \arg\min_{\alpha} \sum_{j=-J+1}^{J} [\hat{\eta}_{kl}(m+j) - \hat{g}_{kl}((m+j)N_s + \frac{N_p}{2})]^2 \\ \text{where} \quad & \boldsymbol{\alpha} = [\alpha_0, \alpha_1, ..., \alpha_q]^T \end{aligned} \tag{5.4}$$

$$\boldsymbol{\Psi} \triangleq \begin{bmatrix} 1 & (-J+1)N_s & \cdots & ((-J+1)N_s)^q \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & JN_s & \cdots & (JN_s)^q \end{bmatrix}_{2J\times(q+1)} \tag{5.5}$$

$$\boldsymbol{\alpha} = (\boldsymbol{\Psi}^T\boldsymbol{\Psi})^{-1}\boldsymbol{\Psi}^T\hat{\boldsymbol{\eta}}_{kl}(m) \tag{5.6}$$

Finally, the channel coefficients for data symbol n are calculated by using (5.7), the part of $\boldsymbol{\alpha}$ not depending on slot index m staying constant over the slot.

$$\begin{aligned} \hat{g}_{kl}(mN_s+n) &= \boldsymbol{\psi}(n)^T\boldsymbol{\alpha} \\ n &= N_p+1, ..., N_s \\ \boldsymbol{\psi}(n) &\triangleq [\psi_0(n), ..., \psi_{2J-1}(n)]^T \end{aligned} \tag{5.7}$$

5.1.2.5 PI Analysis

The complexity and performance of this algorithm completely eclipses the one of WMSA. It is highly tunable with the polynomial order q and the size of analyzed slots $2J$. Table 5.2 illustrates the sub-tasks of this algorithm, their complexity and storage requirements. The computational hot-spot contains matrix inversion and multiplication, therefore the complexity rises steeply with the two main tunable parameters of the algorithm, q and J. Other parameters such as number of pilots in a slot N_p, number of total symbols N_s add additional flexibility to the algorithm. Most of the sub-tasks need to be recalculated for each slot and some simpler tasks (e.g. multiply-accumulates) are recalculated for every symbol. Computationally it is dominated by matrix operations, especially matrix inversion of $(\mathbf{\Psi}^T\mathbf{\Psi})^{-1}$, which grows in complexity with higher q and J, due to its iterative calculus. Additional complexity can come from tuning N_s and N_p. Several divisions, multiplications combined with additions and subtractions in different ways make this a very demanding application, summarized in Table 5.2. The trade-off range of parameters for the polynomial order is between 1 to 3, while the analyzed slot window $2J$ ranges from J equal to 1 to 4. Data dependency within the algorithm allows some of the storage to be reused, thus decreasing the demand on memory. [41,142]

From the application language point of view, PI needs following support from the architecture:

- parallel add/sub and multiplier elements, according to slot size (execution class)
- one divider (execution class) – for inversion of the Ψ-matrix
- as the Ψ-matrix has to be constructed and used, special data forwarding channels (e.g. transpose) have to be supported (communication class)
- local storage of the Ψ-matrix is desirable, but minimally the results of the once/slot calculus should be stored locally to calculate the estimates for every data symbol (communication class)
- beside the standard loads/stores or streaming i/o for the new pilot symbols and the channel estimate, special matrix load-stores would be optimal when working on rows/columns of the Ψ-matrix (memory class)

Clearly, the complexity of PI is reflected in the desirable application language. Detailed requirements can be deduced from the control/data-flow graphs, when algorithmic parameters are fixed. Additional flexibility would be required to support a wider range of parameters, mainly in the matrix operations part. An architecture that optimizes matrix multiplication and provides the language items listed above would be efficient for the PI case.

5.2 Application Specific Architectural Language

This section explores the steps towards achieving best architectural flexibility to support the case-study scenario from Section 5.1.1.1 by closely customizing architectural language to the application.

Table 5.2: PI complexity: partitioning of the algorithm, number of divisions, addition/subtractions and multiplications [41]

Tasks	Complexity	Storage	Execution
Ψ	$mul : q \times 2J$, $add : 2J$	$2J \times (q+1)$	once/slot
$\Psi^T\Psi$	$matmul : (q+1) \times 2J$	$2J \times (q+1)$	once/slot
inversion part1	$div : 2J$, $muladd : (q+1) \times 2J$	$(q+1) \times 2J$	once/slot
inversion part2	$div : 1$, $divsub : (q+1)$	$+2J$	$2J\times$/slot
inversion part3	$matmul : (q+1) \times 2J$	$+2J$	$2J\times$/slot
$(\mathbf{\Psi}^T\mathbf{\Psi})^{-1}\mathbf{\Psi}^T$	$matmul : (q+1) \times 2J$	$(q+1) \times 2J$	once/slot
calculate $\Psi(n)$	$mul : (q+1) \times 2J$	$q+1$	every N_d
calculate α	$mat/vecmul : (q+1) \times 2J$	$q+1$	every N_d
calculate estimate	$vecmul : q+1$	1	every N_d

5.2.1 Structuring and Partitioning Architectural Language Elements

In order to create an architecture that features a language that matches the requirements of both applications, fine-grained elementary functions have to be defined, for each function class: memory, communication, execution and control. For instance, add/sub operations in the DFG imply creation of an add/sub elementary function. To create a higher order language element, surrounding nodes in the DFG of each elementary functions are analyzed. Frequently occurring clusters of nodes suggest a good language element candidate, frequently occurring stand-alone elementary functions suggest creation of a language element on its own. The key idea is to match the application language/structure as closely as possible with architectural language elements.

Once the complete algorithms are processed individually using the DFGs, identification of common elements and paths can be done. The aim is to reuse as many resources as possible across the two algorithms. An example partitioning is shown in Fig.5.3, on a piece of the flow charts of both algorithms. Nodes are partitioned and common points are identified. [142] This step is essential for sharing resources and creating parametrized custom language elements which can support both algorithms.

Finer analysis points follow, such as whether to use fixed point or floating point arithmetic. Looking at the algorithm and profiling details, coupled with the fact that for battery-powered devices floating point implementation is usually not necessary nor feasible, implementation using fixed point arithmetic is the obvious choice. In this case-study the Q-format fixed-point representation is employed. A maximum bit-width Q is divided into two fields of width M and N, such that $Q = 1 + M + N$. The most significant bit is the sign bit, bit field M represents the integer part while the bit field N represents the fractional part. In these implementations Q33.30 for Q = 64-bit precision and Q16.15 for Q = 32-bit precision is used, respectively. All calculus has to respect Q format arithmetic, detailed in Fig. 5.4.

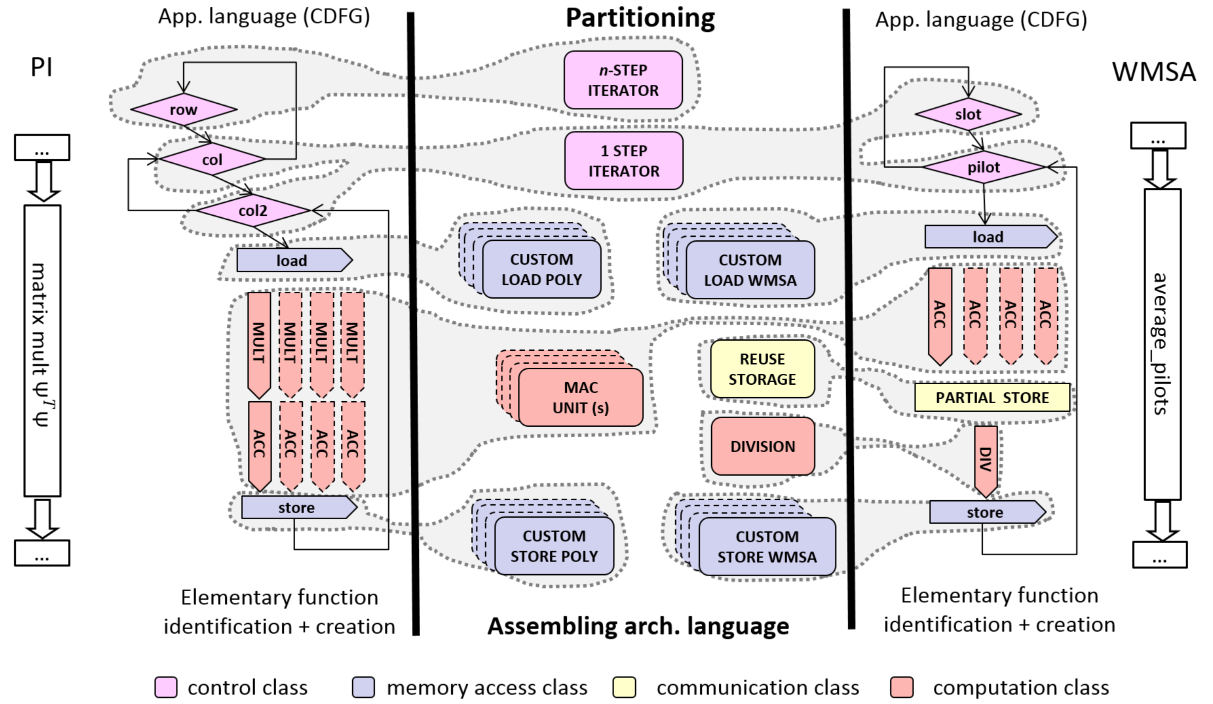


Figure 5.3: Construction of the application-specific language for two applications: identification of elementary hardware functions for each task class from application CDFGs, partitioning to common language elements for resource sharing and construction of the final architectural language. (Adapted after [141])

MUL

```
#define K   (1 << (N-1))
signed bit[Q] a, b, result;
signed bit[2Q]  temp;
temp = (bit[2Q]) a * (bit[2Q]) b;
temp += K;
result= temp >> N;
```

DIV

```
signed bit[Q] a,b,result;
signed bit[2Q] temp;
temp = (bit[2Q]) a << N;
temp = temp+b/2;
result = temp/b;
```

Figure 5.4: Q-format MUL and DIV arithmetic for Q(M.N) case. [142]

From the architectural point of view, implementing the target algorithms is challenging because of the mixed internal computational components: matrix inversion is a sequential process where control flow dominates (partial pivoting and backward substitution), while matrix multiplication is a task where much parallelism is available. Architectural language elements that execute both very efficiently must be selected and constructed. Here, the granularity of the chosen language elements plays an important role. It is the trade-off point of how close the language has to match the requirements of one application, without sacrificing support or induce inefficiencies in the other application.

5.2.1.1 Control Flow Structures

The control flow essentially requires support for if-else statement and loop statements. From the CDFG chart description, the control statements are identified which require independent tuning. Otherwise, the if-else statements are merged within a larger data path. Various step sizes for the loop iterators are supported, allowing shared structures across different loops. In the control-flow implementation via SQDA, the program counter always jumps to the new instruction address for both true and false outcomes of the conditional checks as illustrated in the methodology chapter, Fig. 4.5. Program counter jumps should not incur delay penalties, therefore the condition check and instruction fetch are part of the same pipeline stage. [142] When the architectural language is highly customized, the complexity $\mathcal{K}$ of the application representation (mapping) is significantly reduced. Hence a limited number of assembly instructions required to program the architecture, the program memory is quite small in size (640 bits). Therefore, the instructions can be conveniently stored in register files constructed out of standard cell memories offering fast asynchronous access. This was also one of the advantages of using the proposed methodology.

5.2.1.2 Data Flow: Specific Language Patterns

Analyzing the amounts of data needed by the sub-tasks and usage patterns, optimal load operations become coarse-grained operations composed of multiple memory accesses, shuffling and selection of data. This is problematic for an efficient implementation with SRAMs with limited number of ports. Standard cell memory-based implementation was considered, which offers asynchronous read and synchronous writes. This takes a heavy toll on area but reveals the maximum possible run-time performance. Therefore, data access addresses can be hard-coded for each load/store pattern of an execution node in to a language call, bundling the complete load/store processes in sets of patterns tuned for the respective data path. [142] This not only renders data fetch address computation unnecessary, but also avoids memory operations like matrix transpose, which are hard-coded into that specific language element.

Looking closer at the flexibility requirements, i.e. what kind of parameters the applications have and how they influence the amount of processing, the complexity can be linked to the required execution resources. For PI, the width of the matrix depends on the polynomial order + 1, while the height depends on $2J$ slots considered as observation window. A typical value for the polynomial order is 3 and for the slot window J is 2 and can change by factors of 2, yielding matrix operations of matrix size of 4×2, 4×4, 4×8. A variable number of Multiply-Accumulate (MAC) elementary functions, allows trading off area for parallelism, so the design is easily adaptable to energy, area and timing needs. Thus, when using 4 MACs, a 4×4 matrix multiplication can be done in 16 cycles. Parameter changes result in a different number of iterations, which translates in different counter increments in the control path. These elementary functions can be shared among multiple language elements specific to tasks which use matrix multiplication, and can accommodate other multiplications and additions as needed. In matrix inversion, division is also needed, so

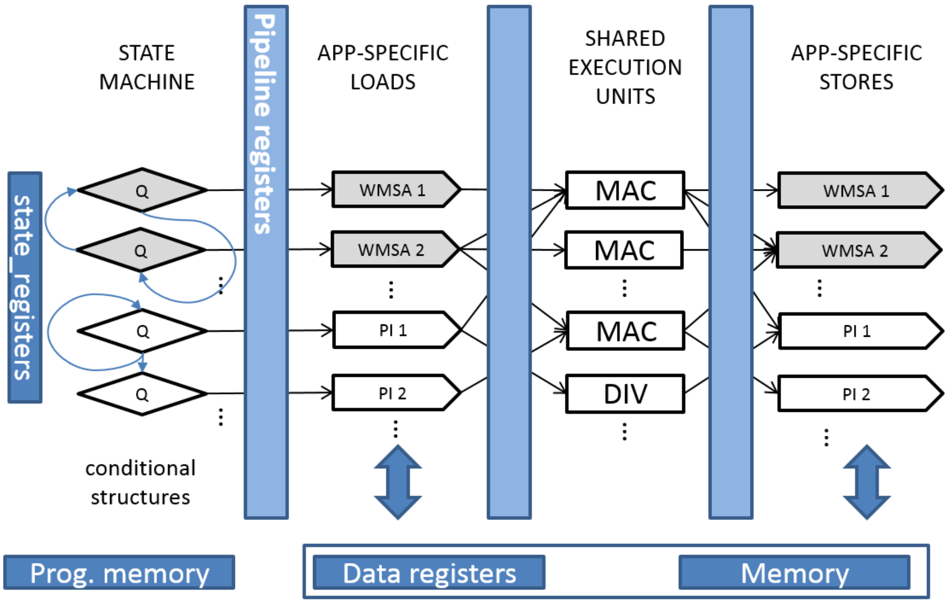


Figure 5.5: Architecture A1 for the two channel estimation algorithms with custom-made language elements and shared resources to fit both target algorithms. [41]

one divider completes the minimum elementary function set of execution units. It must be noted that these MAC units and the divider are executing all operations in Q-format calculus, meaning that each multiplication or division is a concatenation of operations packed into one unit (coarser-grained elementary functions). While simplifying architectural description and programming, this causes a very long critical path in the design. [142]

5.2.1.3 The Resulting Architecture (A1)

After application partitioning and language construction, the physical implementation became straightforward: control (state machine), load, execute and store parts suggested a pipeline of 4 stages, shown in Fig. 5.5. Pipelines are inherently supported by LISA, generating control and pipeline registers automatically. This architecture is denoted with **A1** in later sections during evaluation.

The state-machine stage takes care of instruction fetch, qualifier evaluation and activates the respective data-path in the next cycle. The start of the data path associated with this state is activated in the next pipeline stage (next cycle). The load/store stages contain memory access pattern language sets, activated by the state-machine stage and properly timed, thus loading/storing relevant operands to/from execution unit input/output registers. Some load/store patterns are parametrized, yielding different data for qualifiers in different states. These language elements receive extra parameters from the assembly call function. Compared with the control logic and LD/ST pattern sets, execution units are physically much larger, even for fixed point arithmetic. To reduce the area impact, these units are shared across the application, statically linking the units to the input and output pipeline registers. By that, area consumption is increased in form of multiplexing the data from register files to the input/output points of MAC/DIV units (communication class specific links, aiding data movement). Data dependency is avoided by performing data forwarding between the pipeline stages. With most language elements shared among the algorithms, few algorithm-specific elements remain in the form of specific load/store operations, which translate in little area overhead.

5.3 Increasing Architectural Flexibility: What Are the Gains

The architectural language elements of A1 closely follow the requirements of the applications, except for the execution units, which are large, shared MAC units. This creates coarse-grained language elements in the architecture, since the underlying elementary functions are also coarse-grained. However, for a closer match and resource share, these large, easy to share units have to be reduced in granularity. More varied language elements can be created to accommodate application needs if the elementary functions are more fine-grained. The reason for exploring such a solution is the fact that the Q-format MAC units and DIV unit are extremely large and have long critical path. Some operations, especially in WMSA, use the large MAC unit although only plain addition or subtraction is needed. Also, in the load/store patterns, some fine-grained addition and shifting is required. A reconfigurable core with an array of elementary functions extended with a regular mesh interconnect for the communication class, would provide a sufficiently large pool of elementary hardware functions to construct more complex, well-matched language elements. Thus, the critical path of the MAC units is reduced, while making pre- and post-processing tasks more efficient by using structures more suited for the tasks of each application. Also, the language elements in the execution class are tailored to execute Q-format arithmetic more efficiently. This increases the design effort, leading to a trade-off of granularity versus architectural flexibility.

5.3.1 CGRA Block Extension, Partitioning Effects and Remodeling the Language

Except for the execution part, the considerations for partitioning, control-flow and load/store processing from the previous subsection apply also for such an architecture, denoted in the following with **A2**. The configurable core is defined and implemented in LISA using the key enablers for reconfigurable structures mentioned in Chapter 4.

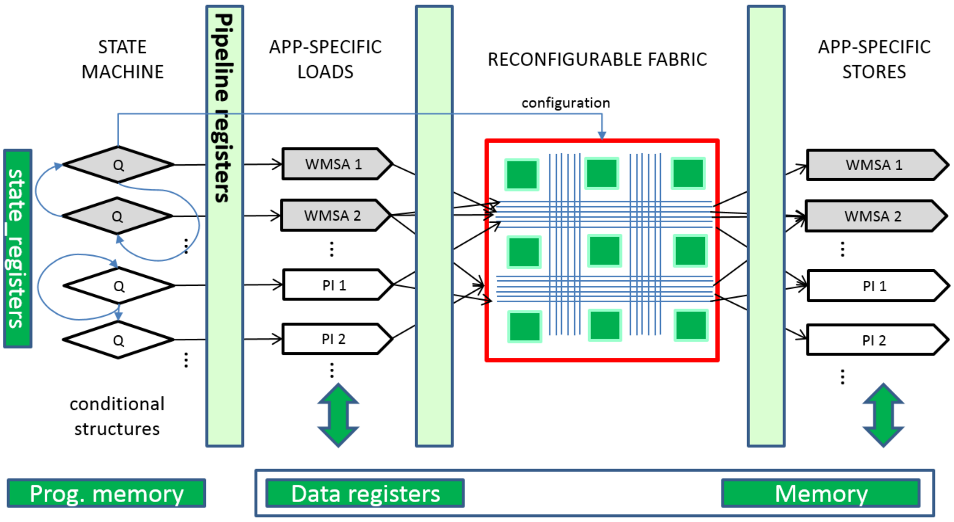


Figure 5.6: Architecture A2 with enhanced architectural flexibility due to the reconfigurable fabric: finer language elements can be created for each application. [41]

5.3.1.1 The Resulting Architecture (A2)

The resulting architecture A2 is shown in Fig. 5.6. The pipeline structure is heavily modified, now only 2 pipeline stages remain, one for control and one for execution. The structure of the core, an mesh-array of 4×4 processing elements, is based on expanding the large Q-format MAC units from the previous architecture giving a heterogeneous structure (Fig. 5.6): the first column (PE0,4,8,12) contains elementary hardware functions for multiplication, addition and shifting, the second and third columns contain only adders and shifters of double bit-width (2×Q, Q=32,64) while the last column contains Q bit-width adders and shifters. This customization of the pool of elementary functions reduces granularity and opens new language combination options by combining them. A divider is added in the middle of the array, with a direct connection to the outputs of element 5 and 9, to be physically close to the elements used when computing the shifting and adding operation within Q-format division. A mesh structure is sufficient to link these structures together and link to the load/store patterns from/to memory, providing elementary functions for data forwarding within the array. It must be noted that, due to the way Q-format arithmetic works, intermediate results within a multiplication or division are of double bit-width (2×Q), hence when forwarding results from columns 0 to 1, 1 to 2, the communication functions must accommodate double bit-width 2×Q. [142]

One processing element can take input data from 12 sources, 6 for each input. Besides the 4 neighboring PEs (north, south, east, west), one PE can also take its own output register as the source, or connect to the load/store patterns. By combining the elementary communication pieces, the wires can be concatenated to directly connect to the output registers of the respective neighboring nodes (essentially creating a 2D pipeline within the array), or a certain load pattern in case of the memory links. PEs on the border of the array take the load pattern as a source, when a neighbor in that direction doesn't exist (e.g., PE0 north and west links are the same with the memory link). The output registers are also directly connected to certain store patterns. Depending on the application, the load/store patterns may or may not connect to each PE, as the data can be processed by a PE chain before being ready for storage. [142]

5.3.1.2 Configuration and Control

To control language element construction, there are 3 configuration bits for each source multiplexer, and 3 bit for the op-code selector. In LISA this is described as a template `OPERATION<>` for each element, which takes the data on input wires `in_A` and `in_B` and outputs to the output register, considering the op-code for the execution. The 6 interconnect links for each source are modeled also with template `OPERATION<>`s which are activated based on the configuration bits of the respective language element. Thus, to completely configure one element, 9 configuration bits are needed, resulting in 144 configuration bits for the entire reconfigurable core of 4×4 elements. [142] These bits, defined by the pattern for the respective language element, are stored directly in the pipeline register after the state-machine stage, right after reading and activating the language element from program memory.

The instruction word contains also the decoding bits of the load/store language elements, 6 bits for each, enabling a maximum of 128 load and store language elements (communication class). The control flow elements use five *timers* as elementary functions, essentially configurable counters which decide for how many cycles one SQDA instruction word holds true. This allows creation of sub-states, in which language elements can be configured from assembly. The complete SQDA instruction word holds thus control-class language element encoding (4 bits) immediate true and false addresses (2×8 bits) ad parameters, the memory access class language elements (2×6 bits) and the communication and execution language elements (144 bits) resulting in a 176 bit function word. [142]

Creating the representation of the application (mapping) using these architectural language elements is straightforward. Necessary parameters of language calls are derived by meaning, i.e. the path traced by the application DFG is recreated in hardware via the correct parameter. Larger DFG portions may require several subsequent language function calls, spanning several time cycles. This will create SQDA instruction words with only some language calls of a specific task class, e.g. some function words not having control-flow calls, or some control flow calls (like nested loops) not having any of the 3 data-class calls. Some can control state machine status (e.g. reset/set the *timers*, initialize registers, etc).

5.4 Evaluation and Comparison

In order to highlight the range of flexibility and the advantages of the proposed design, the generated RTL descriptions of 18 different design points for Architecture 1 (A1), 9 design points for Architecture 2 (A2) were synthesized using Synopsys DC

D-2010-SP3. For all designs, **Faraday 90nm** standard cell technology library was targeted. Extra clock-gating and operand isolation power optimizations were enabled and RTL switching activity was annotated from simulations to give a more exact power evaluation in Synopsys Power Compiler. Since synthesizable RTL code is generated by LISA tool flow, FPGA-based implementations may also be targeted, however standard cell library was chosen to provide a clearer comparison for area and power.

For the evaluation, design points across an architectural class is compared, then cross-class comparisons are presented with the following convention for the graphs [142]:

```
arch. class ":" algorithm "_" design point specialty
```

For instance, `A1:wmsa_1mac` means first architecture class (“just-enough” flexibility), supporting only WMSA and having only one MAC unit, while `A2:both_25` means second architecture class (coarse-grained core) supporting both algorithms, running at 25MHz.

All results are for a complete algorithm execution for a slot of 10 symbols for the respective application. Slot structure was comprised of 2 pilot symbols and 8 data symbols, while $J = 2$ for both algorithms, $q = 3$ for polynomial.

5.4.1 Complexity

A1 template required 2k lines of code in LISA for 61 operations, and has 40k lines of generated Verilog code. A2 has a larger LISA description of 4.4k lines for a total of 98 operation instances (expanding the templates), which generates 51k lines of Verilog code.

In A1, partitioning of PI with $J = 2$ and $q = 3$ yielded 12 control-flow language elements and 34 data-flow language elements. PI had a representation complexity ($\mathcal{K}$) of 39 function words. WMSA partitioning with $N_p = 2$ and $J = 2$ resulted in 1 extra control-flow element and 17 data language elements, requiring $\mathcal{K} = 13$ words for a complete representation.

In the case of A2, with the same algorithmic parameters, polynomial mapping yielded 36 data, 7 control language elements. Complexity of the representation ($\mathcal{K}$) required 102 words. WMSA yielded 16 data language elements with the same control flow elements (reconfigured) requiring $\mathcal{K} = 50$ words.

In terms of complexity $\mathcal{K}$, A2 with a fine-grained elementary function pool and more architectural language options requires a higher representation complexity for PI and WMSA by 2.61× and 3.84×, respectively, although language elements fit the applications better. This is due to the fact that A2 has no physical structures for combining fine-grained elementary functions, this functionality had to be compensated in software via splitting larger states into sub-states in time. The smallest representation length could not be achieved due to technical reasons. In the next chapters, this flaw is addressed, significantly reducing complexity.

5.4.2 Intra-Architectural Comparison Across Design Points

5.4.2.1 Architecture 1

A1 design points resulted from combining 1, 2 or 4 MAC units, and 32/64-bit versions for dedicated structures for one algorithm, then for the hybrid architecture to explore the design for low power or low energy, while maintaining minimum flexibility. Comparing the dedicated design points for one algorithm with the ones supporting both, it can be noted that for A1, when running PI, the combined flexible architecture supporting both algorithms comes close ($\leq 5\%$) to the energy per symbol of the respective dedicated architecture as shown in Fig. 5.7, 5.8. When the combined architecture is running WMSA, it uses comparable or less energy ($-5 \sim 8\%$) compared the dedicated WMSA architecture, explained by the fact that with some partitioning, WMSA may be executed more efficiently on the structures for PI, leading to energy saving without much overhead. For all design points, the Q-format divider was the timing bottleneck, limiting frequency to 25MHz. It must be noted that in the implementation, one can easily switch between these design points in order to seamlessly trade off performance against energy, power or run-time, supported by the high level design methodology. [142]

The architectural flexibility can be utilized within one algorithm or across algorithms depending on performance constraints. Varying execution unit count yields up to 20% energy and up to 50% power savings for PI, while for WMSA up to 14% energy and up to 38% power can be saved. Fig. 5.7, 5.8 show the relative energy per symbol difference across design points for one application. For WMSA the values are within 12%, however the dedicated 2 MAC architecture is most efficient. For polynomial interpolation, both the dedicated and the combined one have similar values. [142]

The execution time per slot ranges between 31-78μsec for PI and 7-11μsec for WMSA, while WCDMA hard deadline is 670μsec, allowing extra savings by frequency scaling, etc. Fig. 5.9 illustrates how much energy is saved when adapting to better signal conditions by switching between the two algorithms: **10-41% power** and **81-88% energy**. The savings stay consistent across design points. For switching between the algorithms, one only needs to load the respective instructions from the program memory. On top of this, the algorithms can be adapted further internally by fine-tuning the points typical of WMSA and PI. [142]

For area critical situations, the architecture template can be easily re-targeted, e.g. the 1 MAC unit design executes in double number of execution cycles of the 4 MAC design, but saves 36% area. The area difference between the dedicated PI architecture and the combined one over the design points is between 5.69% and 12.8%, which is negligible when compared with the joint area overhead of two dedicated structures (205%-212%), even more so when considering the energy savings. [142] Unfortunately, the results could not be compared with the existing implementation in [130] due to the reasons stated in Section 5.1.1.3. Detailed data can be found in the Appendix A into two tables, one for 32-bit architectures (Table A.1) and one for 64-bit architectures (Table A.2).

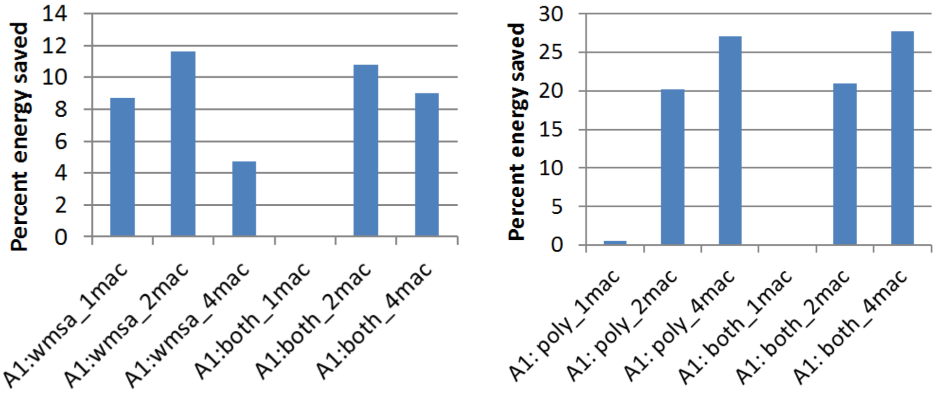


Figure 5.7: 32bit: Dedicated A1 vs. hybrid A1 energy consumption normalized to worst case. WMSA (left) and PI (right) [142]

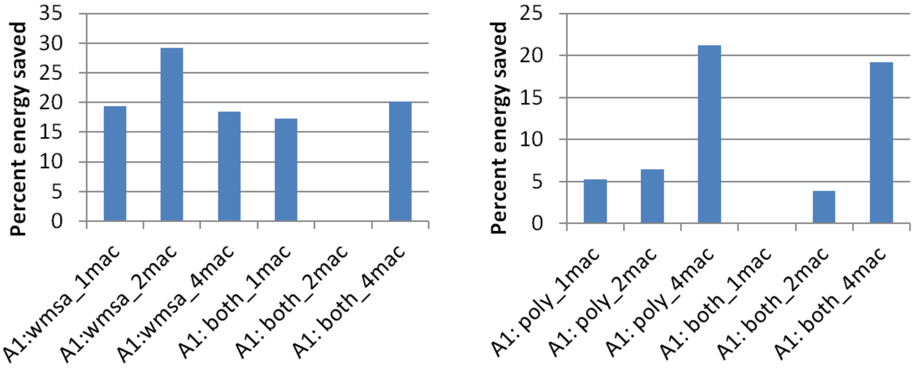


Figure 5.8: 64bit: Dedicated A1 vs. hybrid A1 energy consumption normalized to worst case. WMSA (left) and PI (right) [142]

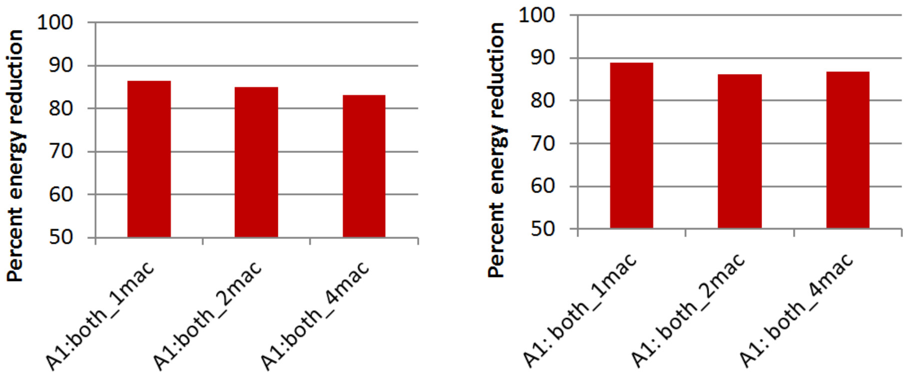


Figure 5.9: Energy savings in percent for hybrid A1, during adaptive switching. (32-bit left, 64-bit right) [142]

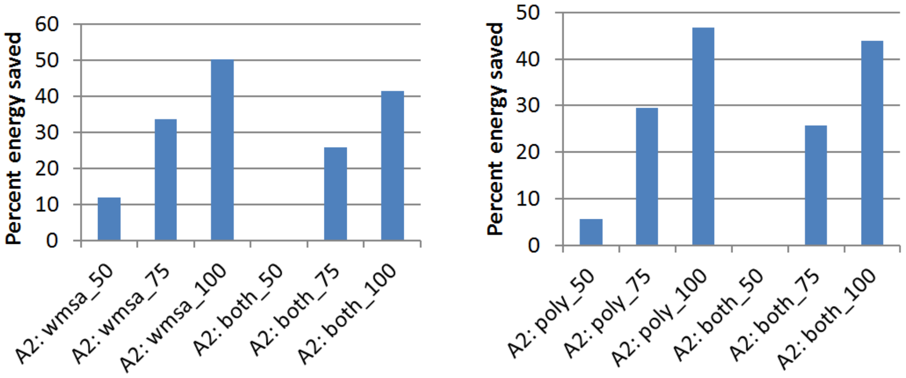


Figure 5.10: 32bit: Dedicated A2 vs. hybrid A2 energy consumption normalized to worst case. WMSA (left) and PI (right) [142]

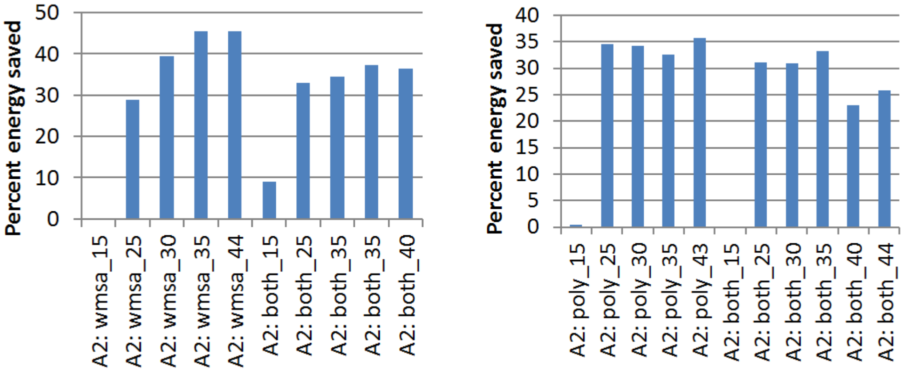


Figure 5.11: 64bit: Dedicated A2 vs. hybrid A2 energy consumption normalized to worst case. WMSA (left) and PI (right) [142]

5.4.2.2 Architecture 2

A2 design points have been constructed from 32/64-bit versions of the architecture tailored for each algorithm and the hybrid version. To further analyze how the coarse-grained core effects energy, different frequencies were targeted, to reveal how frequency affects energy per symbol value. When the frequency is low, the algorithms take longer time to finish, even if they have lower total power, consequently, the resulting energy value is high. Near critical-path operation severely impacts power consumption and area. This is especially the case in the 64-bit design (Fig. 5.11) [142].

Different mapping choices did not affect the power values, since the employed mapping strategy was to use the closest language element which matches the needed operation. During mapping, no congestion was observed, for two reasons: 1) good partitioning of the application requirements which separates the execution in independent, parallel threads. Not many threads are competing for the same processing element (except the divider in WMSA for the filter coefficient calculus); And 2), the matrix inversion processing is sequential, the data dependencies of the inner loops limits parallelization in a natural way, while matrix multiplication exploits 100% ar-

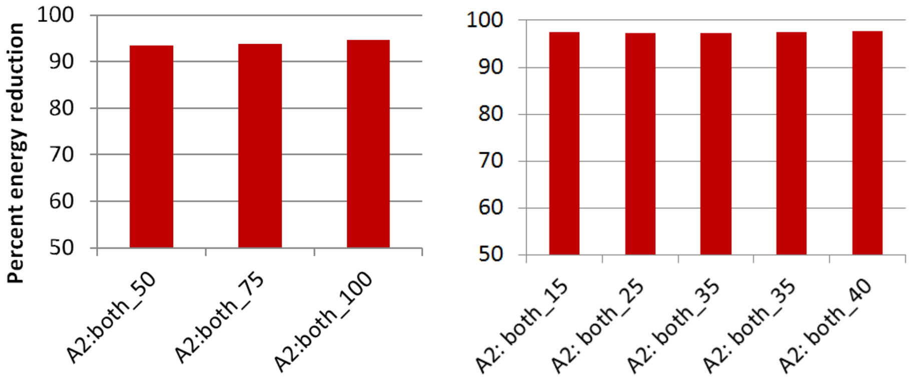


Figure 5.12: Energy save in percent for A2, during adaptive switching at different operating frequencies. (32-bit left, 64-bit right) [142]

ray usage in a well defined manner due to the perfect fit of the architectural language with the application requirements. [142]

The results were gathered in a similar way as for A1, shown in Fig. 5.10, 5.11, as follows [142]:

- 32-bit WMSA (Fig. 5.10 left): as frequency increases, the energy reductions scale linearly with throughput, however the combined architecture has around 10% lower energy savings compared with the dedicated structure

- 32-bit PI (Fig. 5.10 right): energy reductions are similar, the difference is only around 5%

- 64-bit WMSA (Fig. 5.11 left): due to the high power consumption as frequency increases, the increased throughput cannot compensate enough, and the energy saving hits a limit as maximum frequency is reached. Also the combined architecture fares 10% worse when compared with the dedicated architecture when it comes to energy savings

- 64-bit PI (Fig. 5.11 right): when scaling frequency there is an inflexion point, where best energy savings are attained and for which similar savings of the dedicated structure can be reached (within 5%). Similarly to WMSA, near maximum operating frequency the savings diminish (10% difference).

For the scenario that A2 adapts to better signal conditions, **20-44% power reduction** and more than **93-97%energy reduction** can be attained, as shown in Fig. 5.12. Detailed results data is summarized in the Appendix A Table A.1 and Table A.2. Adaptation is similar to A1: only the respective assembly program needs to be executed for a switch from PI to WMSA. However, a higher flexibility allows for a finer control and usage of the structures, improving energy save. [142]

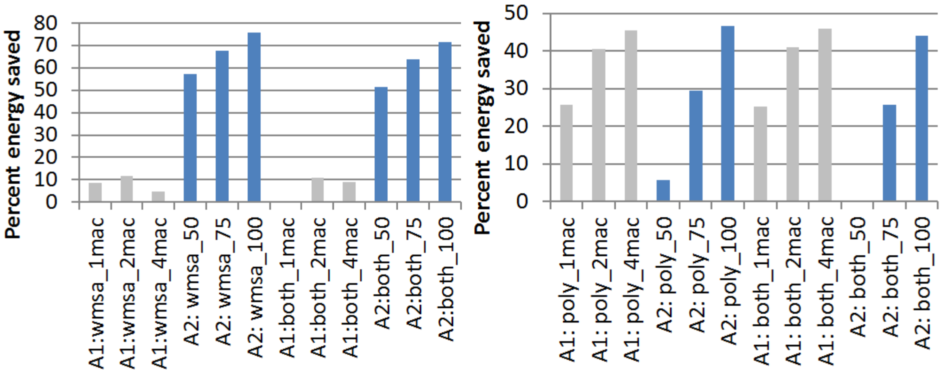


Figure 5.13: 32bit: A1 vs. A2 percent energy saved normalized to worst case. WMSA(left), PI(right) [142]

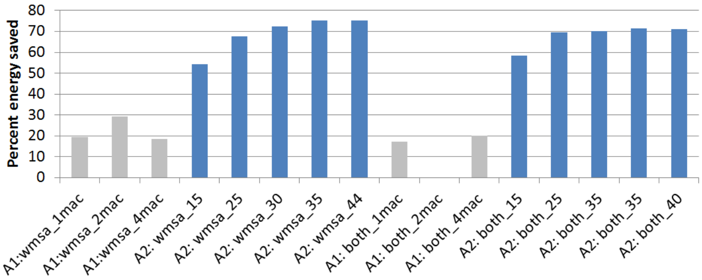


Figure 5.14: 64bit WMSA: A1 vs. A2 percent energy saved normalized to worst case. [142]

5.4.3 Inter-Architectural Comparison

When comparing across architectures the first point to note is the significant difference in area (Fig. 5.16, 5.17). Due to the use of the coarse-grained core, A2 has almost always a larger area than A1 at comparable design points. This can be explained by the fact that due to the Q-format arithmetic, the coarse-grained core is forced to use interconnect structures of double bit-width, incurring not only a greater area use, but

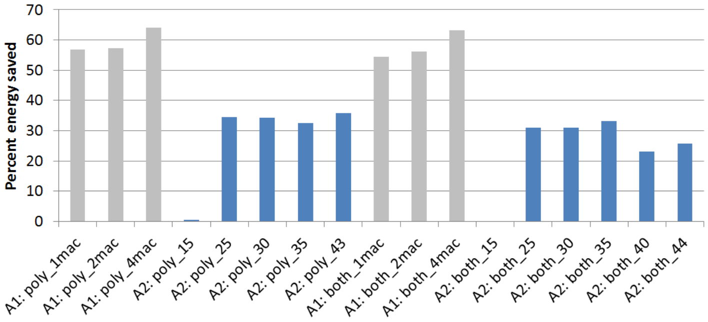


Figure 5.15: 64bit PI: A1 vs. A2 percent energy saved normalized to worst case. [142]

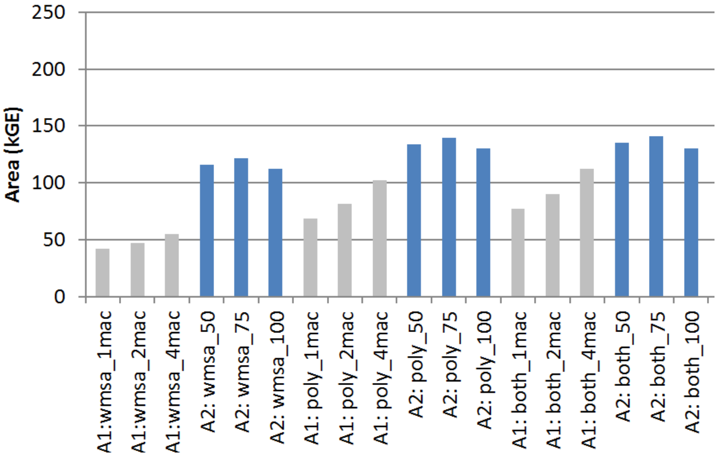


Figure 5.16: Area across all A1 and A2 design points for 32-bit designs [142]

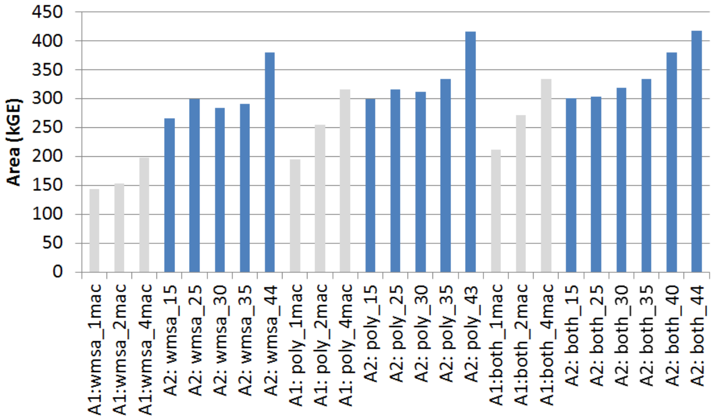


Figure 5.17: Area across all A1 and A2 design points for 64-bit designs [142]

also limiting operating frequency. Additionally, the bloated interconnect incurs extra power consumption, which can not be neglected at higher bit-widths. This is the price of the additional elementary functions to enable fine-grained architectural language elements. Architectural flexibility increases by trading off area. [142]

The second observation to note is the large difference between the energy values for WMSA, A2 having a far better efficiency (Fig. 5.13(left),Fig. 5.14). This is due to having WMSA processing done on the smallest processing elements in A2, and not using the big MAC unit for plain additions, as is the case in A1. Again, since now the architectural language better matches WMSA application requirements, significant efficiency can be gained. [142]

5.4.3.1 Energy per Symbol

Comparison of the two classes is as follows [142]:

- for 32-bit WMSA (Fig. 5.13 left): A2 running at 100MHz has much better energy values ($\geq$**3**× **less**) than all design points of A1
- for 32-bit PI (Fig. 5.13 right): A2 performance is comparable with that of A1 ($\leq$ 3% difference).
- for 64-bit WMSA (Fig. 5.14): A2 has much better energy (**up to 4**× **less**) than A1.
- for 64-bit PI (Fig. 5.15): A2 has much worse energy values (80% increase over A1). In this case, the power overhead of the large bit-widths of the interconnect structure and the elements itself cannot be covered by an increase in throughput any more.

For small bit-widths, A2 would be best for implementation, as the energy saves for the adaptive switching are much greater than in the case of A1. A1 must be chosen when predominantly bad signal conditions are expected and high precision hardware is needed. [142]

5.4.3.2 Area

Evaluation is shown in Fig. 5.16 and 5.17, discussed as follows [142]:

- due to area values spiking at higher frequencies, considering the energy values, the best frequency for A2 32-bit is 100MHz, and A2 64-bit is 35MHz.
- 32-bit, 64-bit WMSA, PI: A1 has much less area than A2 for dedicated structures.
- 32-bit, 64-bit combined structures: area becomes comparable, with a difference of only 18.07kGE for 32-bit and a smaller area (by 0.02kGE) for 64-bit. At very low operating frequencies A2 area values are getting below the ones for A1 in case of PI, but energy efficiency is much worse than that of A1 at those points.

5.4.3.3 Architectural Flexibility

Both A1 and A2 offer the necessary adaptiveness to accommodate the useful range of parameters of both algorithms. Both architectures can be directly tuned from assembly program level (configuring control flow, size of the sliding window, polynomial order, number of pilots, etc), by calling the high-level language elements.

A2 however, due to its coarse grained core, has more architectural flexibility and resources than strictly needed for the two algorithms, hence, it can allow a more efficient execution by better tailoring the hardware language to that of the application. Also, A2 has enough flexibility to enable programming of other external computations, on the idle elements during channel estimation processing. Given the fact that only a fraction of the hard deadline imposed by the WCDMA standard is needed to complete processing, it may be even possible to use the same structure for processing other blocks in the WCDMA receiver chain with some extensions to the load/store language patterns from a new CDFG chart of the new block. This would just result in a new assembly program for the new block with the individual language calls, which can be loaded after channel estimation processing is done, or a new hybrid architecture can be created.

Such flexibility can be a great advantage when aiming more complex blocks for software defined radio

5.5 Conclusions and Summary

5.5.1 Summary: Energy Optimization via High Architectural Flexibility

In the context of software defined radio, an analysis on how flexibility can influence area and power consumption has been conducted, using two WCDMA channel estimation algorithms with sufficient performance and complexity difference. A scenario of saving energy when switching from a complex high-performance algorithm to an inferior less complex algorithm during operation in favorable wireless channel conditions is thoroughly evaluated. Two designs have been explored with different approaches for maximizing architectural flexibility. It is shown that a higher degree of flexibility can yield significant energy savings (up to 97%). Considerable savings (of up to 88%) can still be attained when carefully designing for the *exact* amount of flexibility required to support both target algorithms. The evaluation across 25 design points and two architectural classes of different flexibility experimentally supports the findings.

5.5.2 *Tunable* Architectural Flexibility?

A very interesting fact can be discovered from the case study in this chapter. Architectural flexibility increases when the granularity of the language is decreased. A fine-grained architectural language can better adapt to the application's needs, but requires extending the pool of elementary functions, incurring extra representation complexity and extra area and power. For two target applications, many common language elements could be found, however, adding language elements specific to a certain application improved energy results. Now, if the architecture could support a *tunable* set of language elements, it could have the capability to adapt easily and efficiently to more than one application. The question is, what kind of underlying elementary function pool would be required to support this? How many applications can be supported and how different can they be?

Furthermore, the question of smooth scalability remains: the architecture should be able to adjust to different demands easily at design time. It is easy to scale an architecture from high level, if HLS tools are used to generate new RTL with new architectural parameters. Another approach would be to have a modular architecture, which can adapt the size of the elementary function pool easily, additionally to tunable language support. Thus scalability and architectural flexibility could be gained.

Chapter 6

Tunable Flexibility: The *Layers* Approach and Architecture

In this chapter, the second exploration flow of the methodology proposed in Chapter 4 is discussed. The idea of designing an architecture that can maintain a high level of architectural flexibility ($\mathcal{F}$), even when the target application changes, is explored in detail. The previous chapter demonstrated that a well-defined architectural language produces an excellent match with the application, thus good architectural flexibility. Since with the first flow the architectural language is highly specialized towards a given application, a new application would force a complete re-design of the architecture (and its language) in order to be efficient for the new target.

This chapter proposes a new coarse-grained reconfigurable architecture, based on the theoretical concepts from Chapter 3, where functional separation is exploited to better form, fine-tune and control the architectural language such that high efficiency can be attained. Case studies are conducted from the application domain of Numerical Linear Algebra (NLA) and an exploration of a 3D-silicon physical implementation is attempted. Focus is kept on efficiency and scalability during the case studies, covering a wide range of possible requirements, highlighting the flexibility of this approach. The chapter is based on my work from several publications [135–140][1], treating different aspects of this exploration.

6.1 Adaptability Via Tunable Flexibility

While previously, in Chapter 5, architectural flexibility was maximized by designing the language to fit the target application, in this approach abstraction is decreased by a level. The question is, what kind of elementary hardware functions should be present in an architecture, such that it provides various options when forming a language? Can application requirements fulfilled by reconfiguration, without requiring an architectural redesign?

Application requirements cover a wide range, even within an application domain. Some require fast sequential processing with good control-flow predication, some require a high degree of parallelism or high memory access bandwidth. For the exploration presented here, the application domain is restricted to numerical linear algebra (NLA) kernels, to explore formation of a basic pool of elementary hardware

---

[1] Parts of this chapter appear in these publications, reprinted with permission. ©2012-2015 IEEE, ©2014, CRC Press, ©2012, Hindawi Publishing Corp., ©2015, Springer

functions and to verify the degree of architectural flexibility when forming different architectural language elements for each kernel.

6.1.1 Domain-Specific Approach

6.1.1.1 Domain-Specific Patterns and Domain Classes

In the *Berkeley View on Parallel Computing Research*, computationally intensive applications can be classified into large categories, called *computational dwarfs*, based on which modern parallel applications are constructed [22]. Initially there were 7 *dwarfs* defined, which each of them representing a class applications with commonalities in their data structure, processing requirements and execution patterns. Later, these were extended to 13 to include other computation classes. According to Berkeley researchers, the 13 most important classes of applications viewed from a high abstraction level are: dense linear algebra, sparse linear algebra, spectral methods, n-body methods, structured grids, unstructured grids, MapReduce, combinational logic, graph traversal, dynamic programming, backtrack and branch-and-bound, graphical models, and finally finite state machines.

This classification allows a reasoning about most commonly occurring operations at a high abstraction level, giving insight on what kind of architectures each class would require for efficient execution. In the view of my work, this classification gives insight on how an application language might look like such that the required architectural language can be deduced for maximum efficiency. The high-level Berkeley view has some resonance with the theory proposed in Chapter 3, as it also tried find answers to important questions, such as:

- what are the applications and what are their common kernels?
- what are the hardware building blocks and how to connect them?
- how to describe applications and their kernels and how to program the hardware?
- how to evaluate success

Since each of these *dwarfs* are quite different, the exploration conducted in this chapter is limited to the first *dwarf*. The implementation of multiple applications from a *dwarf* category, using the theoretical concepts proposed, would give insight on application and hardware design problems and solutions, at least for one domain, a procedure that can then be repeated for the other dwarfs.

6.1.1.2 Target Domain Analysis - Dense Linear Algebra

The choice of the first *dwarf*, dense linear algebra, has been made due to the complex balance between memory access patterns, complex computation requirements ($\approx \mathcal{O}(n^3)$ for some routines) and a high demand on data movement patterns. The kernels of this category play an important role in several application domains, such as support vector machines/machine learning, principal component analysis, quantum computer simulation, image and video processing, latest wireless communications standards, linear programming, etc. Another reason for choosing this category is that it exhibits quite regular (but various) patterns across kernels, with a high degree of parallelism. This enables also an exploration on scalability as well, limited only by the cost one invests in the underlying architectural resources.

Although the kernels are part of the same class and domain, each of them have peculiarities in memory access patterns, available parallelism and internal execution dependencies. Therefore, this is a perfect case study domain to evaluate the degree of architectural flexibility $\mathcal{F}$ that can be attained by adapting to the various requirements within a domain, without having too large differences which would require DSP/CPU level of flexibility. Supporting various kernels from the same domain via reconfiguration aligns perfectly with the concept of tunable architectural flexibility.

6.1.2 Domain-Specific Acceleration, Accelerating NLA Kernels

In the literature there is a multitude of soft- and hardware approaches to tackle the complexity of NLA algorithms.

6.1.2.1 Software/CPU

Software optimizations are delivered in the form of a highly optimized, architecture-targeted library of functions, by which higher order applications can be constructed and optimized. Software and a general purpose architectural platform are perfect for supporting entire domains, every application within a domain and every flavor and configuration of the application.

For CPUs, perhaps the most well known collection is BLAS – Basic Linear Algebra Subroutines [94], a general collection of kernels which have optimization hooks for certain architectural features, e.g. SIMD (Single Instruction Multiple Data) instructions. BLAS is one of the first collections, with bindings in C and FORTRAN languages and has defined the standard interface of such acceleration libraries for linear algebra.

Other libraries build upon BLAS, such as Intel's Math Kernel Library (targeted at Intel CPUs), AMD Core Math Library (targeted at AMD processors), ATLAS (Automatically Tuned Linear Algebra Software) pack of re-targetable optimizations, cuBLAS (targeted at GPGPUs), and many other derivatives of these. Since these software libraries imply an architectural flexibility close to CPUs, they provide limited gains, usually bounded by hardware and/or memory bandwidth. Targeted optimizations, such as the FLAME library [27, 158] are one of more recent efforts into bringing efficiency to NLA execution on processors by dissecting the algorithms for parallel execution and efficient memory access.

GPGPUs also are increasingly popular as an architectural platform for accelerating linear algebra, especially with cuBLAS [25] and CULA [81] software libraries yielding excellent acceleration performance at the cost of high energy consumption [17,161] A comparison shows the superiority of GPGPU-based approach in [146].

DSP-based implementations are also popular, especially in the embedded domain. Such implementations exploit software optimization and special instructions to gain performance, e.g. the Gauss elimination-based Squared Givens rotation [60], which is a variant of squared Givens rotation [54] targeted at a TMS320C6670 DSP for high-speed MIMO applications, or Level-3 BLAS optimizations on multi-core DSPs [16].

Although this approach provides flexible domain-specific support, decreased performance (in some cases) and high power consumption limit their applicability, especially in the embedded space.

6.1.2.2 FPGA

FPGAs are also common in attempting to extract maximum performance with little energy, with many possible examples, this paragraph being limited to a few approaches. A comparison between FPGA, CPU and GPU is conducted in [88] concluding that the FPGA implementation is more energy-efficient. The study in [170] highlights that FPGA-based execution units specifically designed for kernels, can be more efficient than GPU implementations, although GPUs outperform. High-performance designs for scalability are explored in [171], where kernels are partitioned across several Cray XD1 blades. LAPACKrc [69] is another approach on FPGA-based solvers yielding 150× the performance of Intel Woodcrest processors.

Example of kernel-specific solutions, such as a 2*D* systolic array implementation of Givens rotation [164] on an FPGA platform targeting Virtex-5 XC5VLX220 outperforms one-dimensional systolic implementations and commercially available QRD implementations like QinetiQ and Altera's QRD prior to that.

Other examples of FPGA-based solution are the FPGA-based architecture implementing matrix inversion [85], exploiting a systolic array design well-suited for FPGA platforms; the variable-precision floating point arithmetic implementation described in [97]; or the portable and scalable direct linear system solver presented in [169].

The disadvantage of using FPGAs for domain-specific acceleration lies withing the fact that for each application, a suitable low-level (RTL) implementation has to be conducted, requiring high design costs. However, these platforms are perfect for single application prototyping and system integration, towards a later ASIC implementation of the tested RTL source code.

6.1.2.3 ASIC

ASIC implementations usually are incorporated into higher-order designs, few targeting specific kernel variants due to a lack of flexibility in configuring them. Such a lack of configurability limits the use-case of such ASICs which need to be closely targeted at the application. Supporting entire domains with all the variance for each application member not only poses a very difficult design problem but also would come with serious resource overheads, minimizing the advantages of the ASIC paradigm (as the name says).

Example single-application ASICs, such as for the Givens rotation kernel, algorithmic optimizations are conducted and implemented directly: the Tournament-based Complex GR targeting MIMO receivers presented in [96] is similar to the scheme presented in [79] where multiple elements of a column of a matrix are annihilated simultaneously by operating on the pairs of rows simultaneously. In Modified Squared GR [101], a conventional mathematical operator based approach is adopted over CORDIC-based approach due to area advantage. Although both implementations handle Givens rotation, none covers all the options and configurations possible for this particular application, complete application domain support would be out of the question.

6.1.2.4 CGRAs

REDEFINE CGRA implementation of QR factorization is presented in [28] where the performance is achieved by Custom Functional Unit (CFU) inside Compute Elements (CEs). The focus of [28] is on emulation of systolic schedule for GR on REDEFINE and hence synthesis of systolic array on REDEFINE. Other optimizations on REDEFINE CGRAs are also attempted in [111, 112], firstly targeting algorithmic optimization which is then exploited for CGRA-based acceleration.

LAC [125–129] CGRA is another prominent example of CGRAs attempting acceleration of NLA. Here too, specific transformations and optimizations are exposed and custom CGRA structures are adapted and extended to efficiently execute complex kernels. Here a wider range of algorithms is tackled, proving that CGRAs are suitable for domain-specific acceleration.

The disadvantages of the CGRA approach are two-fold: 1) architecturally CGRAs are severely limited in memory bandwidth, forcing many available processing elements to idle due to data starvation; and 2) programming of CGRAs is very complex, due to lack of tools support, methodology and due to the complexity of the application mapping problem.

In the following, these two problems are tackled via a new architectural concept based on functional reconfiguration theory, since CGRAs are the most promising platform to support domain-specific acceleration.

6.2 The *Layers* Concept and Architecture

6.2.1 Variable Flexibility via Task-Class Specific Structure and Language

The *Layers* architectural concept is centered around the philosophy formulated in Chapter 3, Thesis 3: functional separation into the four classes of control flow, memory access, communication and computation allows a finer control over architectural design points, scalability and the application mapping process. The fundamental difference in how the architecture executes the application lies in the fact that instead of *instructing* the hardware to execute the application directly, *functional reconfiguration* is employed to instruct the hardware to modify its functional structure in the data and the control path, such that the application function is reflected in the architecture. By specializing parts of the architecture for these tasks, higher efficiency and lower energy can be achieved, since the architecture is a reconstruction of an optimized application image. Moreover, programmability is greatly enhanced by the programming view and functional separation, allowing to exploit data locality, memory access optimization and control flow predication.

This is achieved chiefly by providing a generous pool of elementary functions, that can combine into higher order language elements targeted at an application domain. Here, customization of the underlying elementary function pool to the respective functional class is key. For instance, incrementing a loop counter should not be done by a floating point adder unit, neither should ALU units execute data forwarding and movement operations, as is the case in many CGRA-style architectures. Executing the required function with specialized hardware for that function boosts efficiency and cuts the energy and time losses when compared to traditional approaches. Elementary hardware functions are custom-made for basic tasks of the functional class and allow formation of a wider range of higher order architectural language elements, which are used to reconstruct the application language. Thus, a close match to the application requirements can be created. As it was demonstrated in the previous chapter, a close match yields high efficiency, however with this architecture a finer-grained design and architecture is proposed to enable matching multiple applications to cover an entire domain.

The architecture proposed in the following, called *Layers*, fully exploits the advantages of exploration methodology from Chapter 4. The goal is to provide a modular and scalable architecture, which covers the features of a large portion of the architectural design space. The proposed architecture does not make assumptions about memory bandwidth nor is it restricted by a fixed number of processing elements, yielding full configurable scalability by design. It has 4 layers for each function class, 3 layers handling the data path while the control layer executes application control flow.

In a top level view, data is streamed to the layers in a cascade-like manner, in two orthogonal pipelines, as Fig.6.1 illustrates. In each layer, the flow of data is directed by reconfiguration of the path, *guiding* the data in ways that resemble the application DFG and transforming it towards the final result. The layers are connected via register interfaces, for a clean hand-off between them. During application execution, the processing elements in the lowest layer need a mix of new input data and partial results at well-determined time points for an efficient execution. To provide this data, the communication layer above it shuffles, delays, stores and moves data to the correct processing element at the right time. As input data and resulting output data is stored in memories, the memory access layer provides the necessary functions for loading and storing data, and links the memory banks with the communication layer and execution layer. The control layer loads the application control flow, predicates tasks in the 3 layers and forwards the language calls from program memory. Thus a complete flow can be created from memory to memory, streaming the data through a series of transformations according to the application.

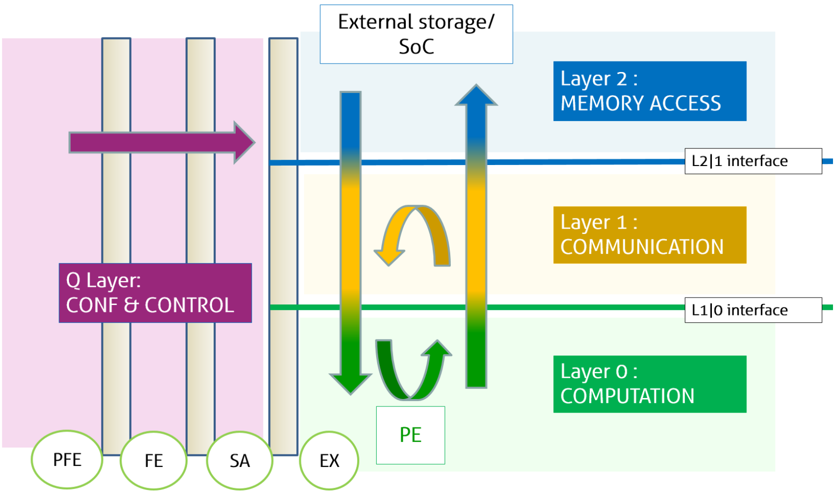


Figure 6.1: The Layers concept separates control and data flow of the application into 4 parts: control, memory access, communication and computation, each with dedicated hardware elementary functions to realize a cascaded flow.

An additional advantage of the proposed layered approach is that it is naturally mappable to 3D silicon technologies, providing a clean cut between these logical layers to be transformed into separate physical layers for Through Silicon Via (TSV) 3D process integration.

In the following, after an architectural overview, each component is discussed in detail, pointing out flexibility and scalability elements.

6.2.2 Architectural Overview of *Layers*

An overview of the architecture is shown in Fig. 6.2, divided into 4 pipeline stages: pre-fetch (PFE), fetch (FE), state machine or state automation (SA) and layers core (EX). Data flows from left→right (control and configuration, main pipeline) and top↔bottom (layered data flow). The *pre-fetch* and *fetch* stage serve only to forward the instruction word to the *state machine* stage, where the reconfigurable control path is implemented. In *state machine*, the current execution state is stored and updated, also data path configurations are decoded and forwarded.

The *layers core* stage implements a reconfigurable data path in a waterfall-like manner and is divided into three layers dedicated to each operation class: memory, communication and execution. Each layer can be configured to work at different speed ratios $r_{(L0:L1:L2)} = r_0 : r_1 : r_2 | r_0, r_1, r_2 \in 2^n$, to maximize efficiency for each layer for a given application: e.g. $r_{(L0:L1:L2)} = 1 : 8 : 4$ is tuned for slow execution, fast communication and medium memory access speed. The control layer speed is always $max(r_0, r_1, r_2)$. In the architecture targeting NLA kernels, $r_{(L0:L1:L2)} = 1 : 8 : 8$ is used, a ratio that matched application requirements well.

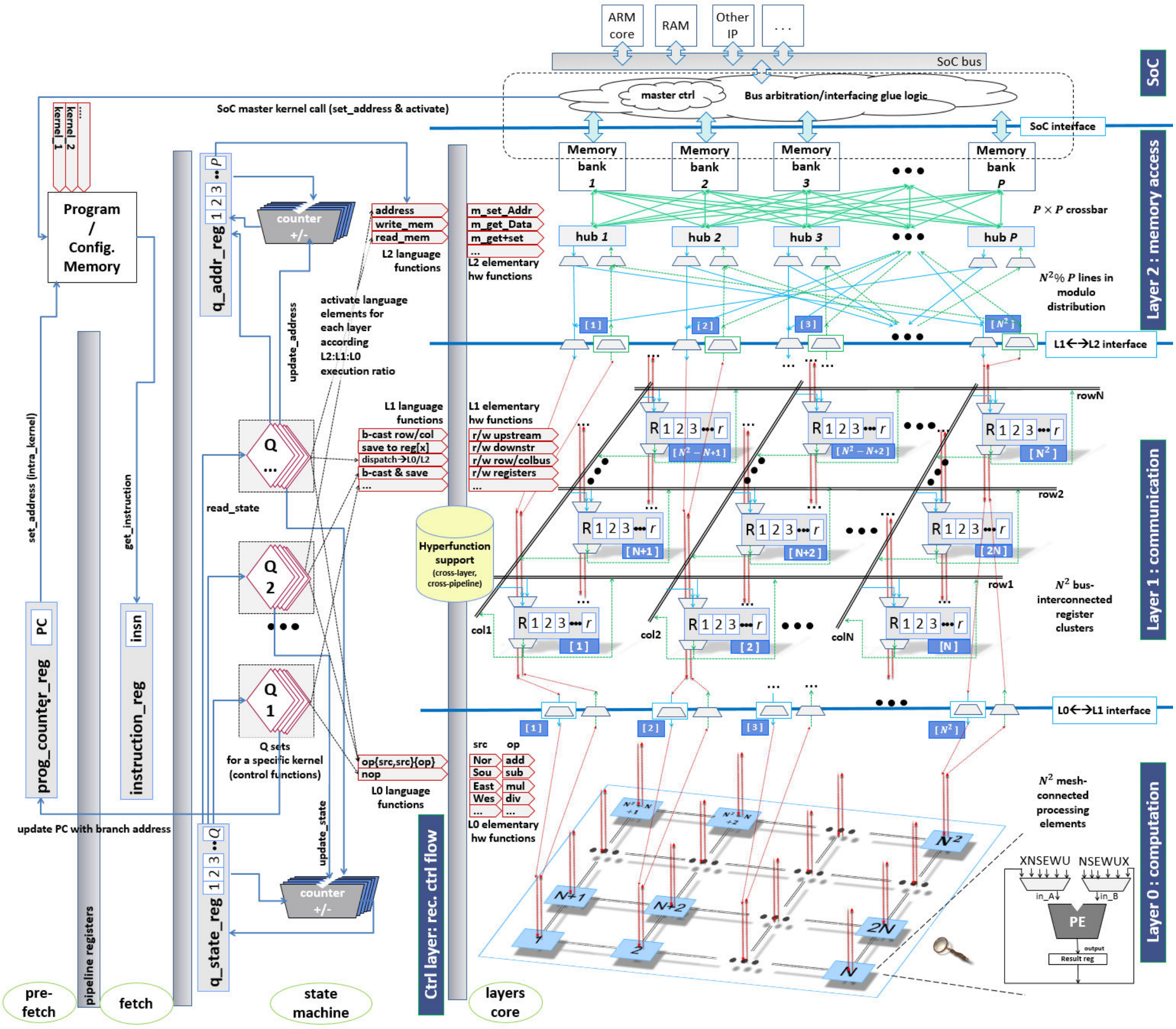


Figure 6.2: The *Layers* architecture: scalable and modular layers dedicated for memory access, communication and computation are managed by a reconfigurable control flow stage. Control stage calls language elements at the right timing, which in turn activate elementary hardware functions in each layer. Hyperfunctions allow reconfiguration of language elements (adapted after [137]

The topmost layer (*SoC*) implements an interface which allows system level integration and control.

Timing in the architecture is driven by the 4 main pipeline stages (horizontal), which control simultaneously the explicit vertical pipeline stages created by the data layers and their intra-layer interface registers, shown in in Fig. 6.3. The complex timing model is efficient because it splits up the long data path operations in small parts, executed by the elementary functions on each layer, forcefully exposing parallelism and concurrency. Thus memory loading, data movement and data processing operations are fully parallelized. Moreover, due to the internal register structure, this timing model allows a recirculation of data within the architecture without creating combinational timing loops. A high rate of data reuse and sharing via the communication layer avoids contention on the memory interface.

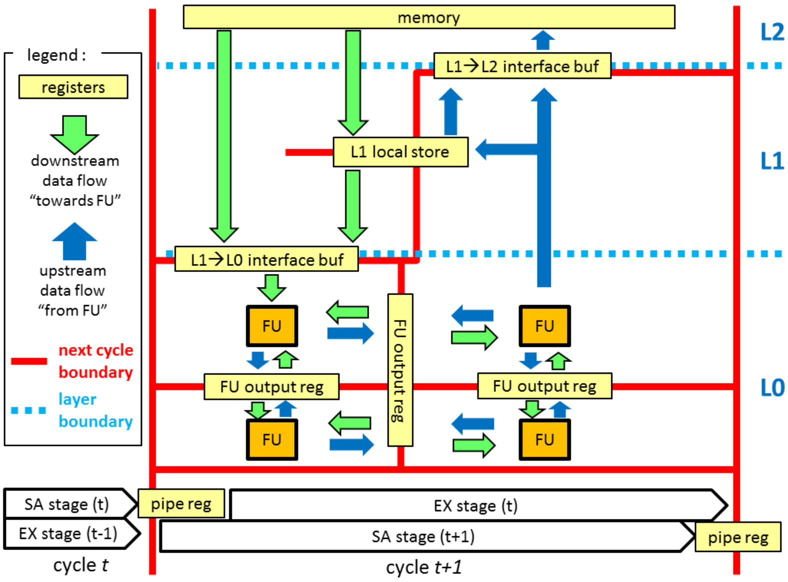


Figure 6.3: The timing model of the architecture: two orthogonal pipelines create a complex timing schedule for streamlined parallelism. [138]

6.2.3 Computation Layer, Structural Details

L0 is comprised of a scalable and customizable square array of size $N \times N$ of processing elements (PE) interconnected with a mesh network of configurable bit-width. Other interconnect topologies can be adopted, but in this case nearest neighbor mesh provided the best trade-off between cost and performance. Each PE has its own pipeline, is replaceable and modular in design, allowing the designer to plug in custom RTL components. In this implementation, 32-bit floating point add/sub units and multiplication units for all units and one multi-cycle configurable floating point divider for PE0 is used. PE capability is captured in elementary functions for arithmetic and operations. The mesh interconnect with the interface towards the communication layer provide the elementary functions for source selection and result output forwarding. Each array member PE defines the *locus* coordinates in the array. For maximum efficiency, the upper layers act as slave data source and sink for this layer to keep all units in this layer busy.

$$\begin{aligned} op(PE_n) &= \{+,-,*,\ldots\}; \\ src(PE_n^{ports}) &= \{North, South, East, West, Up, Self, \ldots\}; \\ n &\in \overline{1..N^2}; \end{aligned}$$

DesignWare floating point (fp) modules provided by Synopsys are employed, $op(PE_n) = \{+, -, *\}$. One pipelined multi-cycle fp divider is added to `PE0` for the architecture targeted at the CSF algorithm, providing one 32-bit fp division result in 4 cycles. Each PE reads input data from 6 sources for each input port $src(PE_n^{a,b}) = \{N, S, E, W, U, X\}$ and outputs results into a register.

Thus, the elementary function pool for each location in the N^2 L0 mesh is:

- input select (a | b):

  `North, South, East, West, Xself, Up;`

- execute for locus 0:

  `add, sub, mul, div, rcp;`

- execute for locus $1..N^2 - 1$:

  `add, sub, mul;`

L0 language constructs for each location can be composed based on these, but for the case study domain, two language functions sufficed:

- **nop**
- **do**`(op(),src(),src())`

6.2.4 Communication Layer, Structural Details

The main role of L1 is to serve as a staging area and preparation network for the input data of L0 coupled with transporting results from the L0 result registers upstream. It is organized in register clusters of parameterizable size for each of the N^2 elements, which are interconnected by two bus-like structures topologically on row and column. Additionally, the upstream interface to L2 and the downstream interface to L0 are buffered and act like a pipeline register between the layers. Formally, in every cycle, each L1 cluster can perform a combination of core elementary functions, with the condition that it does not violate architectural laws (e.g. creating loops, double write to same target, etc).

For the communication layer, 21 elementary functions have been identified, categorized by target, supported in each N^2 location:

- column bus:

  `wCbus(),`
  `rDownstreamToCbus(), rUpstreamToCbus(), rRbusToCbus(), rRegToCbus(r)`

- row bus:

  `wRbus(),`
  `rDownstreamToRbus(), rUpstreamToRbus(), rCbusToRbus(), rRegToRbus(r)`

- downstream a, b and upstream interface lines:

  `wRegToDownstreamA(), wRegToDownstreamB(), wRegToUpstream(),`
  `rRegToDownstreamA(r), rRegToDownstreamB(r), rRegToUpstream(r)`

- registers:

  `wDownstreamToReg(r), wUpstreamToReg(r),`
  `wCbusToReg(r), wRbusToReg(r)`

- special:

  `nop`

Using these core elementary functions, useful language constructs can be grouped and made available at assembly level, for instance:

- parallel save to registers from multiple sources:
  save(`wDownstreamToReg(r),wUpstreamToReg(r),wRbusToReg(r),wCbusToReg(r)`)
  where each `reg` can/must be a different register from the cluster allowing several simultaneous reg-writes from downstream, upstream and buses;

- row broadcast and save a value into local register:
  $\Phi()$=`[wDownstreamToRbus(r) | wUpstreamToRbus(r) | wCbusToRbus(r)]`
  rowbcsave(`w[`$\Phi()$`]ToRbus(), w[`$\Phi()$`]ToReg(r)))`
  where one of downstream, upstream or column bus are selected exclusively as a source by getting a non-zero `reg` parameter, then broadcast onto the row bus, while simultaneously taking the parameter as the register index where the source is to be saved at the call locus. The exclusivity arbitration and the result of the winner elementary source function Φ are automatic and embedded in the hardware implementation of the each elementary function, sufficing to call the top language construct with arguments from assembly.

Such compound operations can be application-tailored at run-time via hyperfunctions, or one can use a default static set, hooks of which are provided at assembly front-end. The buses are access-guarded wires with variable cluster span, and a priority and conflict resolution occurs in hardware to solve any violating function calls. In case of large arrays, long-distance and short-distance bus structures can be added as necessary (additional elementary functions).

Any pattern that is useful for the application can be constructed, and if necessary, the underlying elementary pool can be extended. Currently up to 32 language constructs are supported, fully sufficient for the target application. Alternatively, 8 hyperfunctions can be used instead with the flexibility of redefining the elementary composition of each at run-time. The more language elements are supported, the higher the cost in hardware resources.

6.2.5 Memory-Access Layer, Structural Details

This layer provides access structures to a variable number of memory ports P and is used to distribute these ports to N^2 L1 structures downstream. Distributing data across a number of memory ports allows for higher load/store bandwidth. To avoid the necessity of a full crossbar, scalability and for access conflict resolution from N^2 elements to P ports, P *hubs* are introduced in-between. Each hub has access to each port, needing only a $P \times P$ crossbar, which is reasonable since usually memory ports are scarce ($N^2 \geq P$). From the hubs, a static modulo $N^2 \% P$ distribution is employed, uniformly distributing hubs across downstream elements, e.g. if $PE(n) \% P = 0$, the n-th PE is connected to hub 0, each hub having $\lceil \frac{N^2}{P} \rceil$ connections. Another role of the hub is to use the memory's protocol to forward access requests and select the correct port based on the desired data address.

The elementary set for each P location is:

- memory access (memory protocol):

  `setAddr(addr), setData(d), getData(locus), mask(m)`

- hub selection:

  `sel_rHub(locus), sel_wHub(addr),`

- memory port selection:

  `sel_rPort_{locus}, sel_wPort(addr)`

- interface/reg:

  `wDownstream(locus), rUpstream(locus), rAddrreg(r)`

The implementation of the elementary functions embeds conflict resolution and priorities, such that memory and hub access conflicts are resolved in hardware.

The assembly language constructs for every P port are formed, such as

- **LSET**`(setAddr(sel_wPort(rAddreg(r)), sel_wHub(rAddreg(r))))`
  sets the address contained in a given address register to a certain memory hub which is to access a given memory port

- **LGET**`(sel_rHub(loc), getData(loc), wDownstream(loc))`
  gets the return data from a **LSET** request as synchronous memory modules have at least one cycle of latency, on a given hub and forwards to the downstream interface

- **LGET+SET**`(LSET, LGET)` - pipelined load
  executes simultaneous getting of previous data with a new set address request

- **STOR**`(LSET, mask(m), setData(sel_rHub(rUpstream(loc))))`
  stores data incoming on a given hub from a given upstream line, with a given mask to a given address.

It is worth mentioning that all `locus` specifications can be avoided by fixing a predetermined order of the argument calls. For instance, second function call always goes to the second port, without specifying `loc=2`, nested functions inherit location information, easing syntax and selection complexity. This is true for all other layers as well. With hyperfunction support, the ordering of arguments becomes reconfigurable, which is the required ingredient for redefinition of language elements at run-time.

6.2.6 Control Flow Layer, Structural Details

The *state machine (SA)* pipeline stage incorporates program control, language calls, execution state supervision and predication. *Pre-fetch* and *fetch* stages aid in loading the functional word from program memory and together with the *state machine* stage they form the control layer of the architecture. The elements of the control class assemble a finite state machine coded via several control functions, called *qualifiers (Q)* for next state decision. These elementary functions of the qualifiers are constructed from registers, counters (up/down), comparators and enable/disable signal collections. A combination of these can encode control states, such as incrementing and checking matrix height index, comparison (and branch decision) with the limit, or generate address seeds and increments according to required access strides, partially disabling, modifying and assembling language calls of any layer, etc. The control elements can, for instance, override one routing segment or one PE function by just overwriting or modifying arguments of the respective function calls from the program memory during forwarding, based on execution state. Also, qualifiers implement control flow components such as `for` loops, `if-else` branches and update the current execution state stored q_registers.

Non-exhaustive elementary pool for the control layer:

- update state:

  `writeAddreg(r), writeStatereg(r)`

- read state:

  `readAddreg(r), readStatereg(r)`

- predication for each layer, at element level granularity

  `override(structure_id), forward_if(Statereg(r))`

- calculate/update state or address registers:

  `cntUp(step), cntDwn(step), cmp(a,b), geq(a,b), etc.`

- branching – conditional or unconditional:

  `jmp(Statereg(r)), jmp()`

The flexibility of assembling control language elements at run time is more limited than in the case of data flow layers, because control flow is highly application specific – it requires complex evaluations of state variables under *certain* application-specific conditions. Creating an elementary function pool with flexibility of handling all combinations is a very daunting task, if not extremely resource costly. For example, `q_checkrow_jmp_if_neg` is the final construct which is composed by many smaller constructs chaining functions from the elementary pool. To be able to create this construct at run time, the physical possibility of interconnecting elementary functions in this specific combination has to be available in hardware. Therefore, in the following case studies for the NLA domain, such language elements have been hard-coded for each kernel (i.e. elementary function relationships are static for each language element). Deep exploration, including a graph-theoretic approach on derivation of more flexible structures in explored in Section 7.1.

6.2.7 Hyperfunction Support

Although with static architectural language assemblies it is possible to implement a multitude of applications from the same domain, *hyperfunctions* provide the ability to reconfigure language elements at run-time. Hyperfunctions (hf) are functions that are outside the architectural control/data path, and provide the means for higher functional reconfiguration flexibility, as it is defined in Chapter 3.

Ideally, from the functional programming view, arguments of function calls can contain other functions, each with its own arguments, which in turn can be functions again (nesting). In functional reconfiguration, the architectural language is constructed from elementary functions (ef), each needing arguments. Technically, however, it is not straightforward to realize a circuit that can forward parts of the functional argument space to other functions. For instance, given an argument space of x bits, an elementary function of argument size 3, can be physically located anywhere between $[(x-1)..(x-3)] \rightarrow [(x-2)..(0)]$ range. Allowing such flexibility requires the presence of physical wires and multiplexers to realize the connection physically from the incoming function word argument space to each elementary function argument space. Furthermore, an efficient way of controlling such structures is needed.

Fig. 6.4 illustrates how the *function word* of a layer is constructed. Besides `NOP` and the `RECONF` reserved functions, reconfigurable hyperfunction calls can be used to form arbitrary language elements. The free argument space of each hyperfunction can be filled with function calls to different elementary functions, in any order. Each elementary function can have an arbitrary argument size, depending on its function.

Defining a hf is implemented by using two sets of configuration registers: one for indicating how full the argument space of the respective hf is, the other to save to physical location within the hf argument space of each added ef. Therefore,

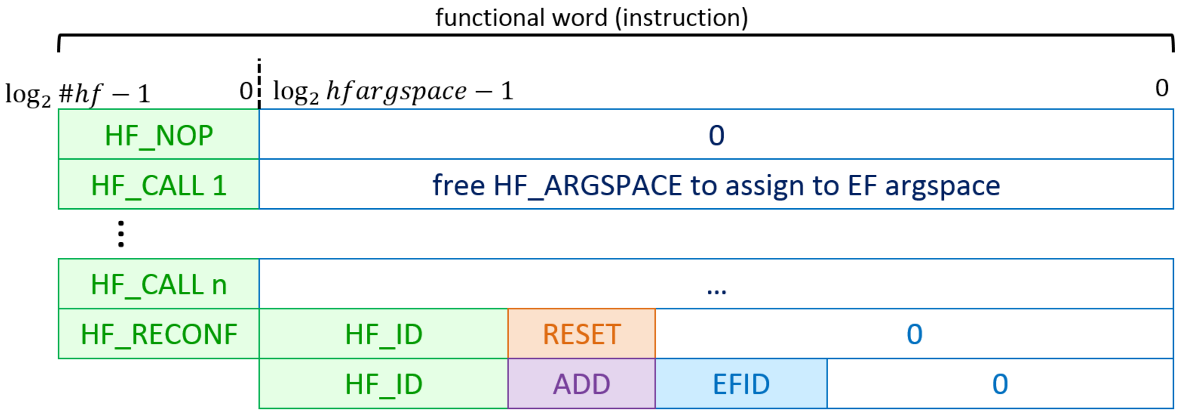


Figure 6.4: How the function word is structured when using hyperfunctions. The architectural language hooks are replaced by hyperfunction call IDs and the arguments are forwarded to the elementary functions.

each hf has one `usemask` register of size $log_2(hf\_argspace\_bits)$, and each ef has $\#hfs \times log_2(hf\_argspace\_bits)$ sized registers, pointing to the currently assigned location within the hf argspace. Fig. 6.5 shows how ef-s are added to a hf, filling up the hf argspace. Each position is saved in the `ef_conn` registers of the respective ef, reserving the hf argspace portion from which it will receive its arguments.

Of course, different hf_id-s can have a different mix of ef-s, in a different ordering, which allows construction of various architectural language elements. In the current implementation of *Layers*, the hf definitions are valid globally for each element in the respective layer, however, support can be added for heterogeneous sets of hf for each element (heterogeneous array). This requires adding a dimension to the `usemask` register, saving the hf argspace usage separately for each element.

Execution of a hf call is shown in Fig. 6.6. A function word with a `HF_CALL` on a hf_id, activates the `ef_connectors` for those elementary functions that are configured for the respective call. The `ef_connectors` are a reconfigurable set of physical wires, connecting the respective ef argspace to the hf argspace. The location, which was previously saved at hf definition in `ef_conn` registers, is read out, and the respective bits from the hf argspace are connected. Forwarding of the arguments happens only to those ef-s which have a valid connector and a non-zero argument value. All elementary functions in *Layers* are inactive by default, if their arguments are all zero.

Thus, architectural language elements can be constructed from ef-s in arbitrary ways. Scalability is available in modifying the number of hf slots, size of hf argspace, number of available ef-s or the choice of homogeneous or heterogeneous array elements, additionally to the standard scalability due to array size and memory ports presented previously. Supporting hyperfunctions comes with an overhead in physical structures, which are orthogonal to the application data path. Initial experiments show however, that the cost in area and power compared with a *Layers* version with static language elements is negligible. Advantages of hf support are manifested in an increased degree of tunability of architectural flexibility $\mathcal{F}$, by allowing more varied

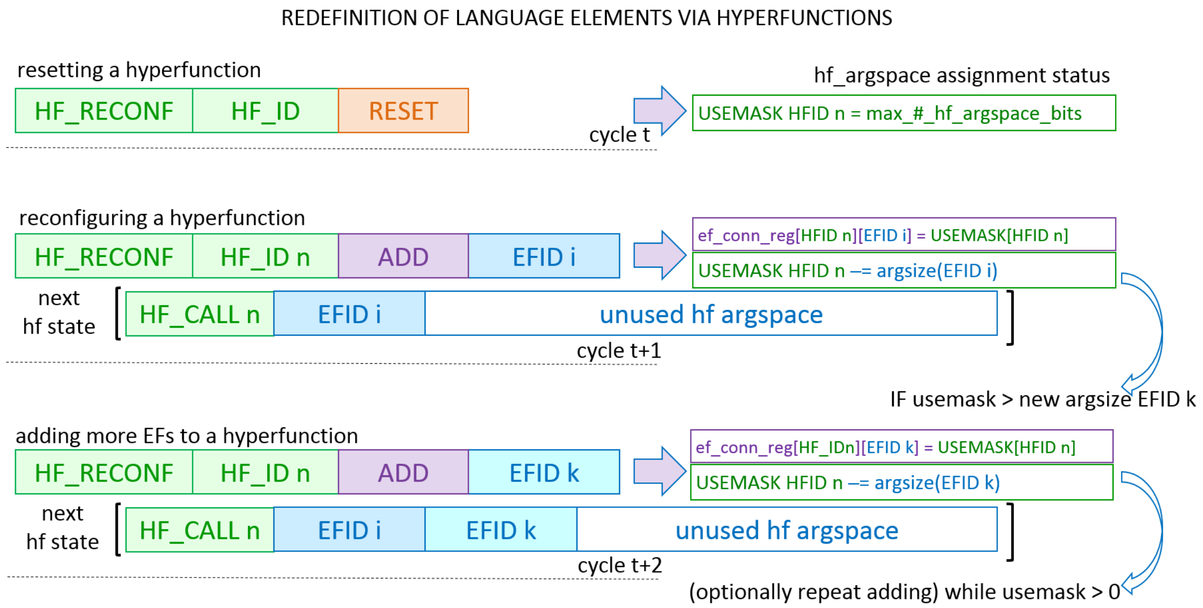


Figure 6.5: Hyperfunction assignments can be reset and reconfigured with new elementary function combinations to form the language that is efficient for application execution.

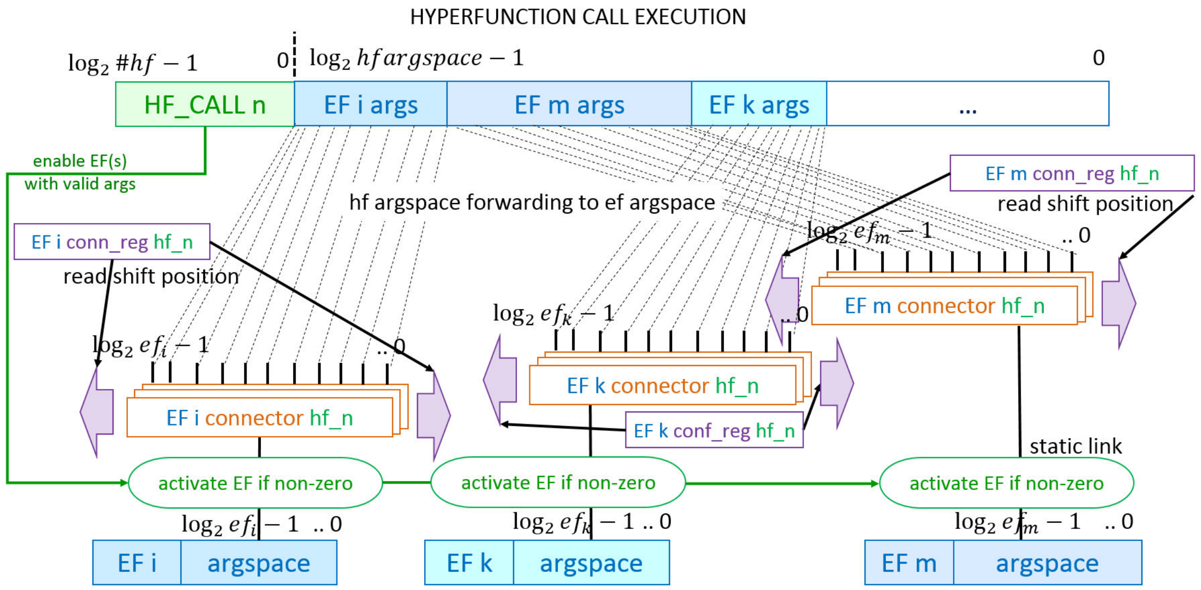


Figure 6.6: Hyperfunction execution forwards parts of its argument space to the configured elementary functions. Variable connectors for each elementary function lock on to the hyperfunction argument space and forwards the arguments. Configured EFs with non-zero arguments get activated and executed.

and finely tuned language elements. Also, *Layers* with hf support has a great domain retargetability, due to the ability of adapting its architectural language with hfs.

In the following, applications are mapped on *Layers* without hf support, using only static language functions, tailored for the linear algebra domain, however an implementation with hf support has also been produced.

6.3 Mapping Linear Algebra on Layers

6.3.1 Importance of Efficient Mapping, Adapting the Language

6.3.1.1 Bird's Eye View of the Optimization Flow

The key idea is to derive an *efficient and scalable* scheduling with coarse mapping elements, based on available algorithmic features (parallelism, dependencies) and then *reconstruct* this mapping in the architecture by reconfiguring the data path. The problem is attacked from two sides: optimizing the application scheduling to expose a language that coarsely respects cost bounds and tuning the architectural language to match application language, while exploiting architectural scalability to also respect the cost bounds.

First, this can be done by optimizing on the application side for the best possible scheduling, such as the optimal ASAP/ALAP. Lax architectural resource constraints can be considered here, such as projected number of parallel execution units and maximum amount of memory bandwidth. Algorithmic parameters, such as required bandwidth per cycle, dependencies, commonly used data and partial results and progress tracking during execution are analyzed. The ASAP/ALAP optimal scheduling is modified then to fit into the coarse architectural bounds (e.g. memory bandwidth per cycle). The modifications are done in a way that minimizes any forced extensions of scheduling time, by using modulo scheduling, heavy pipelining and data flow optimizations (e.g. caching a partial result instead of storing, duplicating data, broadcasting, etc.). The resulting CDFG represents the *application language*, basically the interface that the architecture should match. This procedure needs to be repeated for every target application.

It should be noted, that the derivation of the application language is valid for any mesh-connected architecture of processing elements, as this optimization of the application is largely architecture-agnostic. It is only constrained by dominant cost requirements, such as how many processing elements on architecture may have. Here, it is also important to note, that during these optimizations, *scalability* is also identified. For instance, if the application optimally can support 10 parallel execution kernels, it also provides the scalability range of 1-10 physical execution units. If the cost is limited to 5 execution units, it will mean no special change in the application language elements, as it will only force a sequential execution of available parallelism.

Next, the architecture is scaled to respect cost bounds, followed by deriving the optimal architectural language for the derived scheduling. Elementary hardware functions are selected from the pool on each of the functional layers and new language elements are formed to match the most common structures of the application language. These language elements will form the function calls (e.g. broadcast, copy, add, store, etc.) which can be called during programming the architecture, like in assembly code. There are two approaches here:

- optimize the architectural language to be a super-set of all target applications, such that the architecture can adapt well to each
- add *hyperfunction* support – allowing to reconfigure language elements at runtime, such that the architecture exposes the perfect language w.r.t. each application change.

6.3.1.2 Architectural Factors

Generally, when mapping in a scalable way on a scalable architecture, the execution window size has to match the size of the array for efficiency and respect available memory bandwidth. Especially for architectures with many processing elements, such as CGRAs, memory access contention is one of the main hurdles for adopting optimal application mapping. Additionally for CGRAs, internal resource (interconnect) contention can also occur, limiting efficiency. For CGRAs with banked memories, as is the case with *Layers*, all mappings must additionally respect the constraint of not executing more than one operation on a bank per cycle. Especially for matrix-matrix operations, accessing column and rows of values from either matrix makes scalable mapping difficult. While, internal contention is a question of forming a richer pool of elementary functions (a question of extra area and power cost), major constraints such as number of memory ports and execution units need to be respected. Block-based approaches which work best for CPUs, are not scalable when modifying CGRA size N or memory port amount P and produce complex addressing problems. These constraints guide all optimization approaches discussed in this chapter.

6.3.1.3 Evaluation Flow

Evaluation is conducted in two steps:

- algorithmic mapping optimization targeting efficiency (deriving optimal application language), for each kernel (Sections 6.3.3-6.3.7)
- architectural performance, area, power evaluation and comparisons with related work (Section 6.4).

The case study[2] focuses on the following kernels k from numerical linear algebra domain:

- DOT product
- GEMV – General Matrix-Vector Multiplication

---

[2] It is my pleasure to acknowledge the contributions of D. Stengele and A. Acosta-Aponte towards the mapping effort for some of these kernels, during their M.Sc. degree preparation time under my supervision. Some thesis parts are reprinted here or belong to the publications we made based on these contributions, mentioned in the introduction of this chapter.

- GEMM – General Matrix-Matrix Multiplication
- TRSV – Triangular Solve Vector
- TRSM – Triangular Solve Matrix
- LU - lower/upper factorization
- Givens – QR factorization via Givens rotation, especially column-based version, without square-root operations and with/without division variants

Efficiency evaluation is performed by taking the ratio between the theoretically required number of execution cycles $c_{k_{min}}$ and the number of actual cycles executed by the architecture c_k for each design point. Kernel-specific input parameters such as input data size is denoted with $\{\cdot\}$. $c_{k_{min}}$ represents the optimal execution time of the derived CDFG after application language optimization. c_k represents the actual execution cycles on *Layers*, and it perfectly reflects not only execution efficiency of the architecture, but also gives a graspable value of architectural flexibility – shows how well the architectural language is matching that of the application. Evaluation is scaled with main architectural parameter N, which is the square array size of processing elements in Layer 0.

$$\eta_k(\{\cdot\}, N) := \frac{c_{k_{min}}(\{\cdot\}, N)}{c_k(\{\cdot\}, N)}. \tag{6.1}$$

The calculus of c_k comprises of the actual cycle times, location and latency of the operations required and is more complex to derive. It contains also necessary tasks like register initialization, memory access times, data distribution, predication and other overheads. $c_{k_{min}}$, on the other hand, contains only the coarse constraints set during the derivation of the application language and the mapping, accessible via the scalability parameters array size N, memory ports P or speed ratio r between the layers. Since the processing elements in Layer 0 can have different latencies and certain capabilities limited to certain units, the evaluation takes these facts into consideration when exploring parallelism and scalability: $op_k(\{\cdot\})$ for the arithmetic operations `ADD`, `SUB` and `MUL` and $op_{k_{div}}(\{\cdot\})$ for the arithmetic operations `RCP (reciprocal)` and `DIV (division)`. `RCP` and `DIV` need 4 *L0-cycles* each to finish execution, while all others finish in 1 cycle, which has to be considered when calculating minimum cycle complexity.

$$c_{k_{min}}(\{\cdot\}, N) := 8\left(\frac{1}{N^2}\, op_k(\{\cdot\}) + 4\, op_{k_{div}}(\{\cdot\})\right). \tag{6.2}$$

op_k is scaled by N^2 processing elements, which can work in parallel and $op_{div,k}$ is multiplied by 4, since in this architectural configuration, only one processing element (`PE0`) is augmented with a divider (4 *L0-cycles* latency time). Finally the result is multiplied by the speed ratio inverse $\frac{1}{r_{L0}}$, here $r_{L0} = \frac{1}{8} r_{L1,L2,SM}$ to get *cycles* instead of *L0-cycles*. Due to implementation overheads, the actual executed cycles are higher than the theoretical ones and $c_k(\{\cdot\}, N) \geq c_{k_{min}}(\{\cdot\}, N)$ holds. From $c_k(\{\cdot\}, N) \geq c_{k_{min}}(\{\cdot\}, N)$

follows $\eta_k(\{\cdot\}, N) \in [0,1]$, an efficiency of 100% would also reflect a perfect degree of architectural flexibility $\mathcal{F}$ for the given constraints.

In the following, each kernel is analyzed in detail and the efficiency is calculated. In Section 6.4 the evaluation of architectural performance is conducted.

6.3.2 Scalable Accumulation Folding in a Mesh

6.3.2.1 Mapping

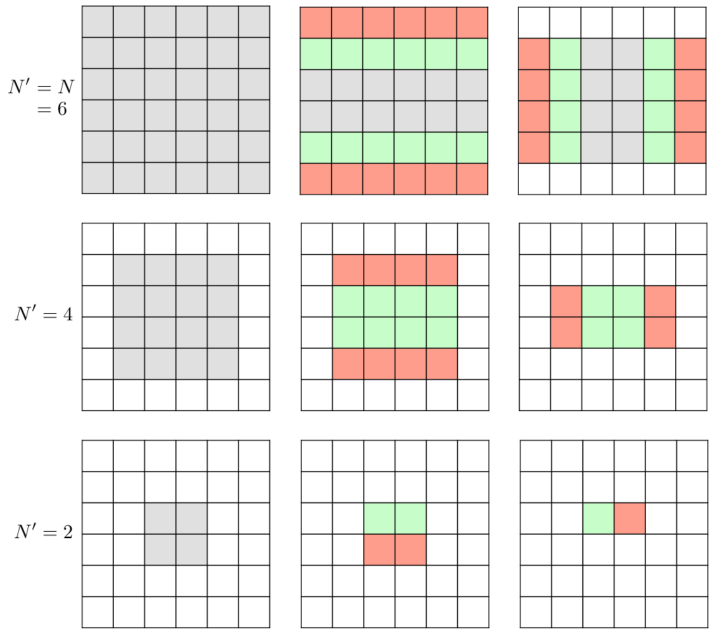
This is not a kernel *per se*, but most kernels make use of this, especially in the epilog portion of hot-spot loops. It concerns the situation when partial results are distributed across a mesh array of processing elements, all of which need to be added to a single end result value. Theoretically, be fastest way is to create an *adder-tree*. The maximum parallelism comes from adding pairs of elements, number of which decrease with every iteration, by adding one pair of new partial results until there is only one pair left, generating the result.

When the number of processing elements is limited, only as many pairs can be added as many elements there are in one time cycle, generating the same number of partial results every cycle. At the end however, these must be added as well. From the scalability perspective, scheduling regularity is broken when the tree width becomes smaller than the number of processing elements. Therefore, a folding of such partial results algorithm is proposed in Fig. 6.7, tailored for processing element arrays, the underlying structure of the execution layer L0. The case of folding partial sums into the final value for a 6×6 architecture is illustrated, following a generic algorithm valid for any square array size.

To preserve simplicity and scalability, the algorithm starts from the edge of the array, with all elements containing data that needs to be summed. Two addition fronts, horizontally and vertically are created, which alternate. Since every iteration there is less input data, in the example from Fig.6.7, only a 66% efficiency is reached in the first wave, diminishing with each iteration. Although it is theoretically possible to use the two available middle rows to perform also one row of additions, it breaks symmetry for a scalable progress towards the final merge.

Symmetry needs to be preserved, especially for data load/store operations and partial result forwarding. A data gap in row 3, in the 6×6 example, would be complex to compensate and capture in a simple iterative algorithm (requiring 2-hop data forwarding in one cycle, unsupported by a mesh network. Moreover, iteration control, data load and movement becomes too complex to be compensated by an eventual execution speed-up. The larger the underlying physical array, the less efficient the folding procedure is, as the middle units remain unused until later iterations. A maximum of only 4 rows or columns of PEs being active in one cycle, regardless of array size, diminishing until the final merge.

It is important to note however, that for large input data sizes, in most kernels, this folding procedure is a small, but important part of the entire operation backlog.



(a) (b) (c)

```
$N' \leftarrow N$
while $N' > 3$ do
    // $1^{st}$ cycle
    for each element in the second row of the current square do
        ADD(self, north)
    end for
    for each element in the second last row of the current square do
        ADD(self, south)
    end for
    // $2^{nd}$ cycle
    for each element in the second column of the current square do
        ADD(self, west)
    end for
    for each element in the second last column of the current square do
        ADD(self, east)
    end for
    // Update $N'$
    $N' \leftarrow N' - 2$
end while
```

Figure 6.7: The accumulation procedure (folding) used in many NLA kernels. (a) shows initial state, (b) and (c) show the iteration transition, which reduces to the same problem of smaller size. Repeating such iterations will reduce to a 2×2 or 3×3 special case, which finally yield the final result in one element. [139]

The procedure is repeated until the folding front reaches a 2×2 or a 3×3 data square, shown in Fig. 6.7. $N' = 2$ and $N' = 3$ have to be treated differently, because it is not possible anymore to reduce the square from top/bottom or left/right simultaneously. For these cases, only `ADD(self, south)` and `ADD(self, east)` are inserted, reducing $N' = 2$ to a single element, or, if N is odd, $N' = 3$ to $N' = 2$ and to a single element after-wards.

There are no special requirements from the language side of the architecture, as addition and mesh source specification are layer 0 elementary functions, allowing easy construction of language elements such as `ADD(self,south)`.

6.3.2.2 Complexity

Operation complexity is $op_{acc}(N) = N^2 - 1$ because always $i - 1$ additions are required to add up i values, only depending on the architecture size N. When N is even, the square will be reduced to a single element in $N/2$ iterations of the accumulation procedure. Every iteration consists of 2 *L0-cycles*, hence it will take $8N$ cycles. When N is odd, a square with $N' = 3$ can not be reduced to a single element in just one iteration. Hence, $(N+1)/2$ iterations are needed, which results in $8(N+1)$ *cycles*. Thus cycle complexity is

$$c_{acc}(N) = \begin{cases} 8N & \text{if } N \text{ is even} \\ 8(N+1) & \text{otherwise.} \end{cases} \tag{6.3}$$

6.3.3 The DOT Product

6.3.3.1 Algorithm

The input to the *DOT-product* kernel consists of two vectors $a, b \in \mathbb{R}^n$ and the output is $c \in \mathbb{R}$.

$$\begin{array}{cc} & \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ \vdots \\ b_n \end{bmatrix} \\ \begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & \dots & a_n \end{bmatrix} & c \end{array}$$

$$c := a_1 \cdot b_1 + a_2 \cdot b_2 + \dots + a_n \cdot b_n = \sum_{i=1}^{n} a_i \cdot b_i$$

Directly from the algorithmic structure, it is obvious that all the multiplication operations are parallel. The accumulation of the sum can be folded also in parallel, in an addition tree. Since the accumulation procedure does not yield an constant parallelism rate, it needs special optimization, discussed in the following.

6.3.3.2 DOT Mapping

To create an efficient application interface for a scalable number of processing elements, the language must expose scalable parallelism and regularity. This helps create a matching scheduling and mapping window compliant with the architecture size, with operations (language elements) ready to be adopted by the architecture.

Fig. 6.8 shows how each element of a $N^2 = 4$ element array is assigned an operation within the execution window, which partitions the two vertical data columns. If N^2 changes, the execution window scales accordingly. After an initial multiplication on all elements, the execution window slides downwards through the data, multiplying and accumulating the results for each processing element. Partial accumulation results are sent to L1 registers for the duration of one L0-cycle and used again in subsequent accumulation cycles, avoiding storing back to memory. When the $N^2 \% n$ does not cleanly match the array at the end of the data, predication signals (`overrides`) are set by the control core, disabling the extra execution units which have no data. This is a source of efficiency loss, however for very large matrices, the amount of full execution windows is dominating. After moving the execution window in N^2 steps, every element holds one partial result and the final result c is the sum of all of those, hence by calling the accumulation folding procedure the kernel is completed.

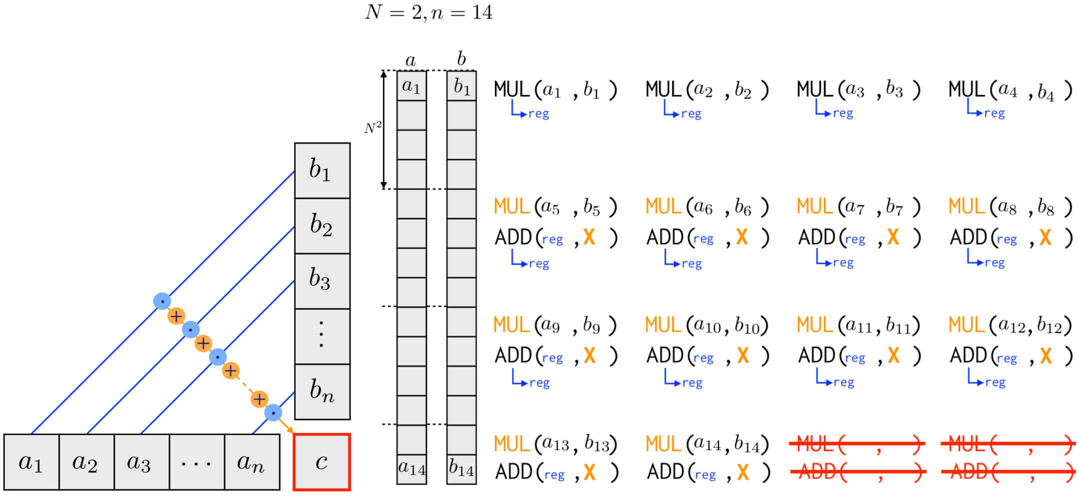


Figure 6.8: DOT mapping with N=2, yielding an execution window of 4 elements, which slides downwards on the two data columns a, b, executing multiply and accumulate instructions alternately, until end of data is reached. If data size does not match N^2, `predication` deactivates extra instructions. X denotes taking previous output (self). [139]

From the language perspective, there are no special requirements: multiplication and addition are alternating, which is basic elementary function of the layer 0. Combined with the source functions, data can be received and sent to layer 1 easily.

6.3.3.3 Complexity

For vectors of length n, n multiplications and $n-1$ additions are required, hence the operation complexity is $op_{dot}(\{n\}) = 2n - 1$. Minimum cycle complexity is

$$c_{\texttt{dot}\,min}(\{n\}, N) = 8\,\frac{1}{N^2}\,op_{dot}(\{n\}) = \frac{8}{N^2}\,(2n-1). \tag{6.4}$$

6.3.3.4 Efficiency

Mapping efficiency is shown in Fig. 6.9 for different architecture sizes and input vector lengths, where inefficiencies of the accumulation procedure are dominating large architectures on small data sets. The expected speedup from this mapping when scaling N, is shown on the right side of Fig. 6.9, closing within $< 4\%$ to the expected theoretical speed-up value for large data sets on large arrays.

When there is enough data to feed the array, efficiency is very close to 100%, yielding a great degree of architectural flexibility and efficiency. Detailed data are added to Appendix B, Fig. B.1-B.2.

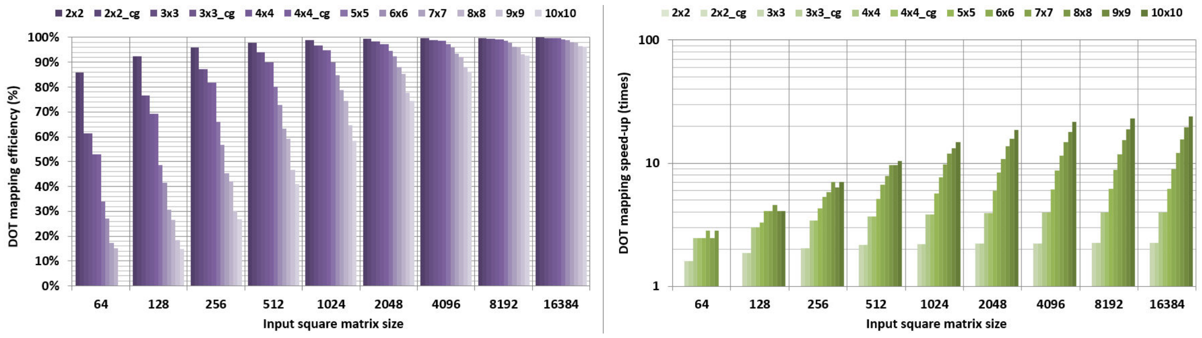


Figure 6.9: DOT mapping efficiency and mapping-based speed-up for various architectures (N=2..8) and data sizes (64..16384). Due to the accumulation procedure, efficiency receives a heavy penalty, especially for large arrays with small input data sizes. When enough data is used, the penalty is much smaller. Worst penalty for large data sets on large arrays is < 4%. [139]

6.3.4 Matrix-Vector Multiplication (GEMV)

6.3.4.1 Algorithm

For every $1 \leq i \leq n$, GEMV is defined as follows:

$$c_i = \sum_{j=1}^{m} a_{ij} \cdot b_j \tag{6.5}$$

This algorithm has also a high degree of parallelism, not requiring any special language elements.

6.3.4.2 Mapping

It is immediately visible from the formula that every multiplication can be done in parallel, but in the end every product has to be added together for every row i. This breaks the symmetry of the algorithmic progress through the data. It would be beneficial if the computation of $c_1, c_2, \ldots, c_{N^2}$ can be assigned to L0-elements $e_0, e_1, \ldots, e_{N^2-1}$. This would yield N^2 elements of c after m multiplications and $m-1$ additions. However, while any L0-element e_{i-1} would operate exclusively on row i of A, the input data a_{ij} in every step would make loading values of a column of A necessary, breaking scalability and efficiency for large N.

To operate on rows only due to the memory access conflict limitations, at any given point in time, a less optimal but scalable scheduling was implemented shown in Fig. 6.10, thus N^2 L0-elements can work in parallel to calculate every c_i. The execution window moves in N^2 steps horizontally in the matrix and vertically on the vector, but forces an accumulation procedure at the row boundary of each row.

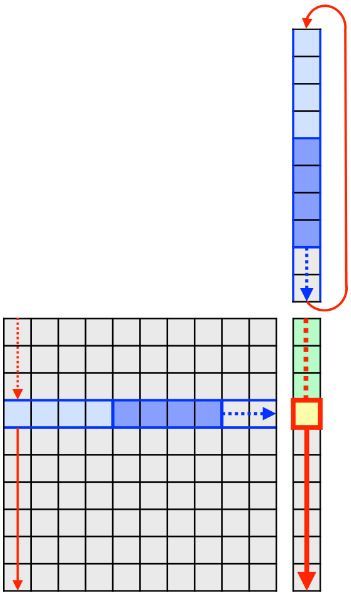


Figure 6.10: GEMV scalable execution window progress (here, $N^2 = 4$ elements), denoted in dark blue, yielding the result in yellow. The progression of this window through the data ensures no more than one load per memory port per cycle. [139]

6.3.4.3 Complexity

Operation complexity of GEMV, with input size n and m, is equivalent to n times DOT complexity for input size m, hence $op_{gemv}(\{n,m\}) = n \cdot op_{\texttt{dot}}(\{m\}) = n(2m-1)$. Minimum cycle complexity is

$$c_{\texttt{gemv}_{min}}(\{n,m\},N) = 8\,\frac{1}{N^2}\,op_{\texttt{gemv}}(\{n,m\}) = \frac{8}{N^2}\,n(2m-1). \tag{6.6}$$

6.3.4.4 Efficiency

Mapping efficiency is shown in Fig. 6.11, similar to the DOT product described previously, due to the extra accumulation procedures for every row. Expected speedup from mapping when scaling N, is shown on the right side of Fig. 6.11, which for large arrays and large data sets is $< 3\%$ close to the expected theoretical maximum. Detailed data are added to Appendix B, Fig. B.3-B.4.

6.3.5 General Matrix-Matrix Multiplication (GEMM)

6.3.5.1 Algorithm

Matrix Multiplication is defined as follows: For every c_{ij} $(1 \le i \le n, 1 \le j \le m)$:

$$c_{ij} = \sum_{l=1}^{k} a_{il} \cdot b_{lj} \tag{6.7}$$

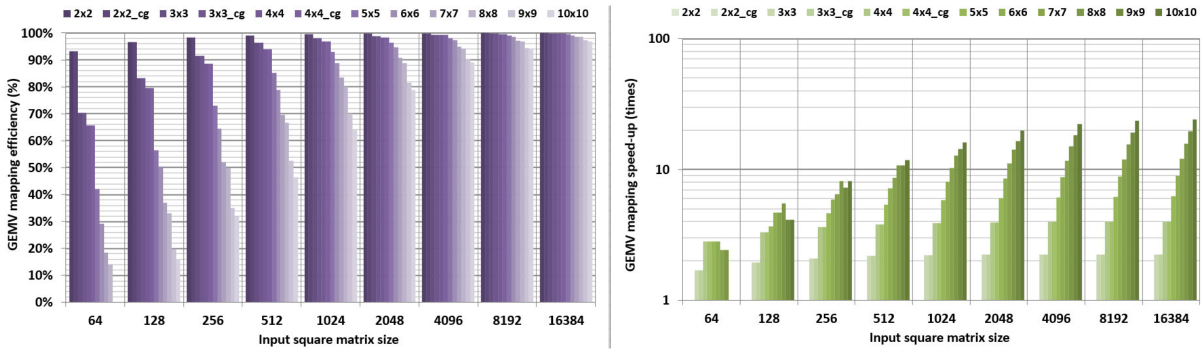


Figure 6.11: GEMV mapping efficiency and mapping-based speed-up for various architectures and data sizes. The penalty for performing accumulation every row can be observed, but stays negligible for large data sizes.

It is well known that an efficient mapping of this highly parallel kernel can be of paramount importance for many applications. It is not trivial however to exploit available parallelism due to memory access, dependency and computation complexity, which is why it represents one of the most revealing benchmarks in the high-performance community.

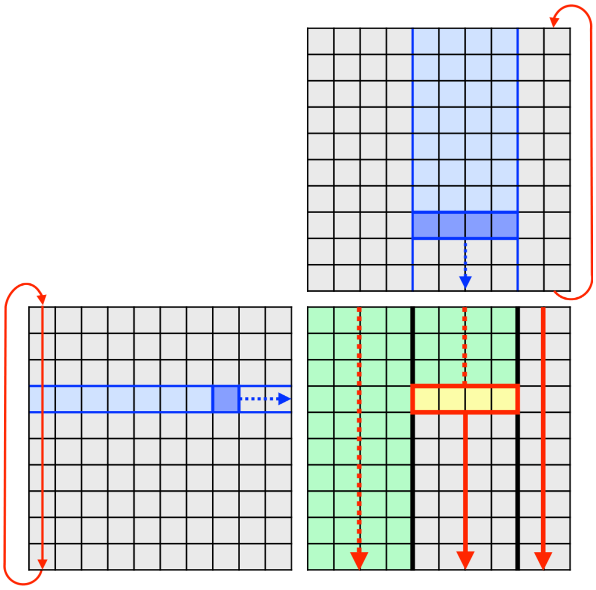


Figure 6.12: GEMM scalable execution window progress (here, $N^2 = 4$ elements), denoted in dark blue, yielding the result in yellow. The progression of this window through the data ensures no more than one load per memory port per cycle. [139]

6.3.5.2 Mapping

Respecting the same constraint of not loading data on the column (keeping one access per memory port), for *GEMM* the data dependencies turn out to be problematic. Most obvious mapping solutions would require to load a column of values from either A or B and multiply it with a row from the other. Rectangular or square windows with a height of more than 1 were not possible either since this would require to work on columns in either matrix. Block-based approaches are not scalable when modifying N or memory port amount P and produce complex addressing problems.

To bypass this, the following mapping is proposed, allowing a scalable execution window without column loads, as shown in Fig. 6.12.

1. Load a_{11}
2. Multiply a_{11} with $b_{11}, \ldots, b_{1N^2}$
3. Continue with a_{12} and $b_{21}, \ldots, b_{2N^2}$ and accumulate the partial results
4. When finishing with the last row, store the resulting $c_{11}, \ldots, c_{1N^2}$ and continue with the next window.

This implies more memory loads due to reiterating through the data several times, but scales perfectly since a window of height 1 and width N^2 can be used, without causing memory port conflict or exceeding available bandwidth. The window iterates matrix B and C column-wise and only in the last column of windows there may be overrides necessary, making the override logic efficient. There are no special needs for the language point of view for this kernel, except a clean distribution of loaded data to the L0 elements.

6.3.5.3 Complexity

GEMM, with input size n, m and k, is equivalent to m times *GEMV* for input size n and k, hence $op_{\texttt{gemm}}(\{n,m,k\}) = m \cdot op_{\texttt{gemv}}(\{n,k\}) = mn(2k-1)$. Minimum cycle complexity is

$$c_{\texttt{gemm}_{min}}(\{n,m,k\},N) = 8\,\frac{1}{N^2}\,op_{\texttt{gemm}}(\{n,m,k\}) = \frac{8}{N^2}\,mn(2k-1). \tag{6.8}$$

6.3.5.4 Efficiency

Even with the constraints considered, the implementation of *GEMM* is very efficient because the actual core of the implementation consists of 2 *L0-cycles* only, in which all processing elements are always occupied – a multiplication/accumulation procedure. Fig. 6.13 shows that even for corner cases the efficiency penalties are minimal. The speedup from mapping perspective when scaling N, is very close to the expected theoretical value, shown on the right side of Fig. 6.13. For scaling by x amount of elements a speedup of close to x is achieved. Please note how the efficiency reaches optimality when the data set is a multiple of N^2 for large arrays. Detailed data are added to Appendix B, Fig. B.5-B.6.

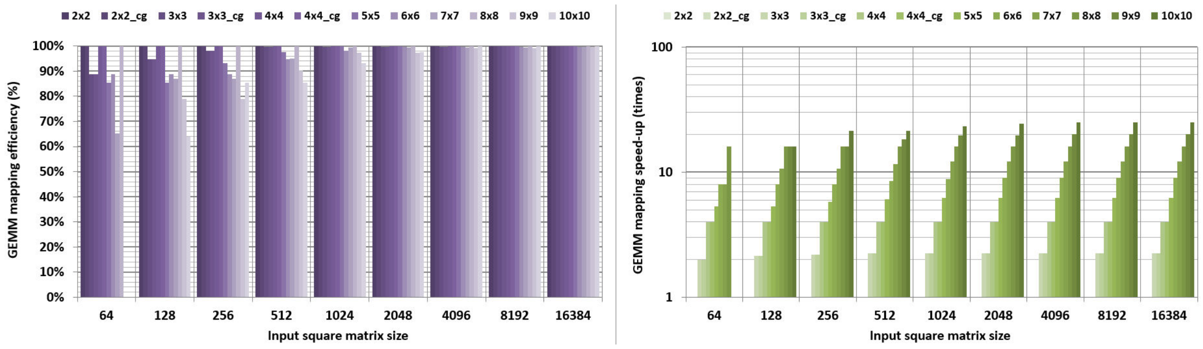


Figure 6.13: GEMM mapping efficiency and mapping-based speed-up for various architectures and data sizes. When the data size matches a 2^n multiple of the array size, optimal efficiency can be achieved. Otherwise, at the final execution window which does not perfectly match the PE array causes overrides, which is a source of minimal efficiency penalty.

6.3.6 Triangular Solve Vector (TRSV)

6.3.6.1 Algorithm

The input to *TRSV* consists of a lower triangular matrix $A \in \mathbb{R}^{n \times n}$ and a vector $b \in \mathbb{R}^n$. The output is a vector $x \in \mathbb{R}^n$, such that

$$\begin{bmatrix} a_{11} & 0 & \dots & \dots & 0 \\ a_{21} & a_{22} & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & 0 \\ a_{n1} & a_{n2} & \dots & \dots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}$$

Solving for x results in

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \cdot {a_{11}}^{-1} \\ (b_2 - a_{21} \cdot x_1) \cdot {a_{22}}^{-1} \\ (b_3 - a_{31} \cdot x_1 - a_{32} \cdot x_2) \cdot {a_{33}}^{-1} \\ \vdots \\ (b_n - a_{n1} \cdot x_1 - a_{n2} \cdot x_2 - \dots - a_{n,n-1} \cdot x_{n-1}) \cdot {a_{nn}}^{-1} \end{bmatrix}$$

or generally

$x_i = \left(b_i - \sum_{j=1}^{i-1} a_{ij} \cdot x_j \right) \cdot {a_{ii}}^{-1}$. This algorithm is much more complex, although parallel: while the dot product part is parallel, the subtraction is depending on the result, which needs to be scaled by the reciprocal. This implies that a divider (or a reciprocal) unit has to be present in the array, which is bound to a processing element. Furthermore dependencies in calculating terms $x_1..x_i$ make scaling of this algorithm complex.

6.3.6.2 Mapping

Since all x_i $(1 \leq i < k)$ are needed to compute x_k, the computation of all x_i must finish before x_k can be computed.

Theoretically, computing x_i and then computing the partial results $a_{ji} \cdot x_i$ for all rows $i < j \leq n$ using vertical windows of width 1 and height N^2 is possible, however, this would not scale due to the limit on column access on banked memory modules. Therefore, all computations necessary for x_i have to be completed before starting to compute x_{i+1}. To do that, while maintaining scalability, a similar approach as with the DOT product is employed for calculating the sum $s_i := \sum_{j=1}^{i-1} a_{ij} \cdot x_j$ using all N^2 L0-elements in parallel, partial sums are kept in each processing element, finally followed by the accumulation procedure to sum up the result in one single L0-element. Finally, $b_i - s_i$ is performed on that single element and scaling with ${a_{ii}}^{-1}$ to compute x_i occurs.

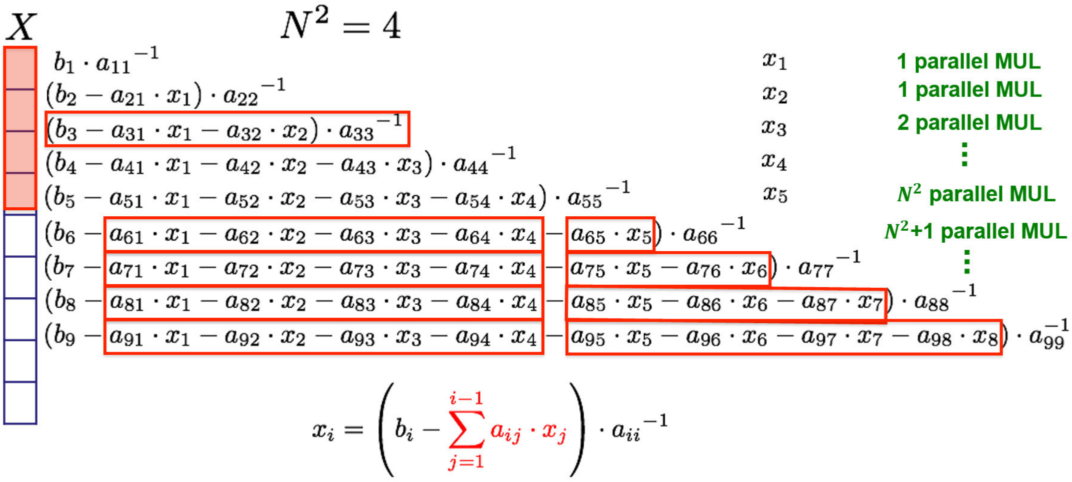


Figure 6.14: TRSV scalable execution window progress (here, $N^2 = 4$ elements), the progression of this window through the data ensures no more than one load per memory port per cycle. Every row there is an accumulation procedure and one incomplete execution window, unless perfectly matching array size (input data size divisible by N^2).

Row 1 — The computation of x_1 is a special case because s_1 does not exist. This case has to be treated differently in the scheduling because instead of $(b_i - s_i) \cdot {a_{ii}}^{-1}$ just $b_1 \cdot {a_{11}}^{-1}$ has to be computed.

Rows $2 \ldots N^2$ — The computation of $x_2, \ldots, x_{N^2}$ theoretically works the same way as for all following x_k $(N^2 < k \leq n)$. However, in practice, accumulation procedure has to be performed, to add up the values of the output registers of every L0-element. For these partial results it does not matter whether they are just the result of one single multiplication or already an accumulated result of many operations. It is also not relevant if, due to a smaller window size in the last iteration of operations on a given row, the partial result saved in e_i is a result of fewer operations than of an element e_j with $j > i$. In the end, all data has to be accumulated.

However, if the first window size is already smaller than N^2, then there are N^2 minus window-width L0-elements with values in their output registers that belong to previous operations or kernels, having nothing to do with the value currently being computed. For the other algorithms using the accumulation procedure, this situation was avoided by requiring an input size $n \geq N^2$. This is no notable limitation, because $n \gg N^2$ is very usual. For TRSV, however, such constraints are not possible due to the triangular form of A, which results in $i-1$ values to be added up for s_i $(1 \leq i \leq n)$.

One possible solution would have been to reset all L0-output-registers before starting computations on a row. While this is not possible from a control flow function, it would be possible to implement an language element, alongside the existing arithmetic functions, to output 0 regardless of the `A`- and `B`-registers, which would require an elementary function extension.

Another solution is to compute $x_2, \ldots, x_{N^2}$ differently from the others in a serial fashion. In fact, only a single L0-element is used, multiplying and accumulating all $a_{ij} \cdot x_j$ $(1 \leq j < i)$, necessary for row i $(2 \leq i \leq N^2)$, one after another to s_i. Then, similar to the parallel computations for rows i with $N^2 < i \leq n$, $b_i - s_i$ is performed on that single element and, finally, multiplication with ${a_{ii}}^{-1}$ to compute x_i. The number of cycles this serial execution takes, can be considered small compared to the total number of cycles for performing the *TRSV* algorithm, because it is used only for $N^2 - 1$ rows with a total number of

$$1 + 2 + \ldots + (N^2 - 1) = \frac{(N^2-1)\cdot N^2}{2}$$

multiplications and

$$0 + 1 + \ldots + (N^2 - 2) = \frac{(N^2-2)\cdot(N^2-1)}{2}$$

additions which can be performed in

$$\frac{(N^2-1)\cdot N^2}{2} + \frac{(N^2-2)\cdot(N^2-1)}{2} = (N^2-1)^2$$

L0-cycles. Also note that the number of rows is usually significantly larger than the number of L0-elements, i.e. $n \gg N^2$.

From the architectural language perspective, data forwarding to the elements and horizontal communication elements for folding and storing partial results are needed. These are however in the initial set of *Layers*, requiring no special elements.

6.3.6.3 Complexity

For each x_i, $i-1$ subtractions and multiplications are needed

$$2 \sum_{i=1}^{n} (i-1) = n(n-1)$$

Additionally, one multiplication and reciprocal per row is required which results in

$$op_{\texttt{trsv}}(\{n\}) = n^2$$
$$op_{\texttt{trsv}_{div}}(\{n\}) = n$$

The minimum cycle complexity is

$$\begin{aligned} c_{\texttt{trsv}min}(\{n\}, N) &= 8 \left(\frac{1}{N^2}\, op_{\texttt{trsv}}(\{n\}) + 4\, op_{\texttt{trsv}_{div}}(\{n\}) \right) \\ &= 8n \left(\frac{n}{N^2} + 4 \right) \end{aligned}$$

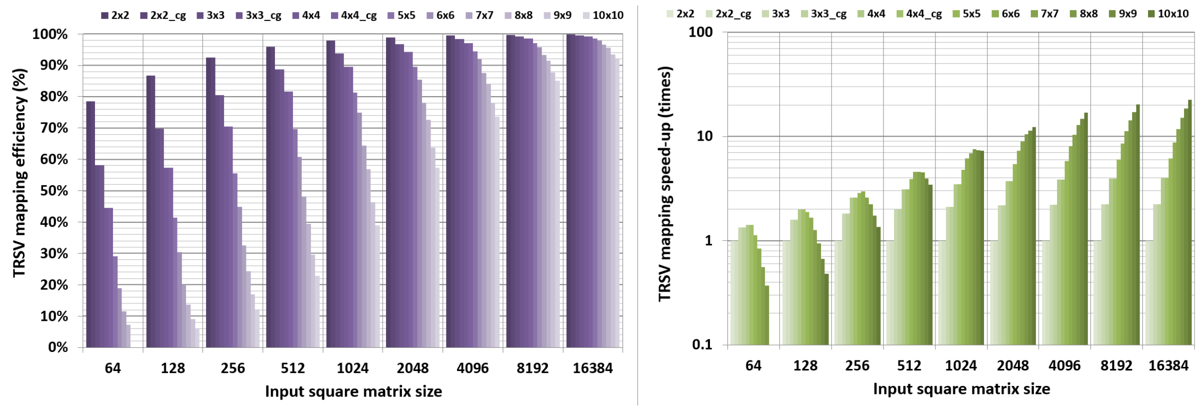


Figure 6.15: TRSV mapping efficiency and mapping-based speed-up for various architectures and data sizes. Inefficiencies occur when the accumulation procedure is executed, signifying a lower degree of architectural flexibility.

6.3.6.4 Efficiency

Due to the special sequential calculus on every row, which needs an accumulation folding procedure at the end, TRSV is one of the most inefficient kernels implemented in the case study, shown in Fig. 6.15. Not only it has sequential dependencies which stop parallel execution, but complex memory load/store patterns force a less parallel execution. Furthermore, calculation of the reciprocal is also required, but due to efficient pipelining the latency could be hidden from the critical path.

Overall, there is a less degree of architectural flexibility due to these limitations, however for large arrays this becomes negligible. A more flexible memory access would enhance this, but not by much, as the main limitation comes from intra-algorithmic dependence. Detailed data are added to Appendix B, Fig. B.7-B.8.

6.3.7 Triangular Solve Matrix (TRSM)

6.3.7.1 Algorithm

The input to *TRSM* consists of a lower triangular matrix $A \in \mathbb{R}^{n\times n}$ and a matrix $B \in \mathbb{R}^{n\times m}$. The output is a matrix $X \in \mathbb{R}^{n\times m}$, such that

$$\begin{bmatrix} a_{11} & 0 & \dots & \dots & 0 \\ a_{21} & a_{22} & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & 0 \\ a_{n1} & a_{n2} & \dots & \dots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_{11} & x_{12} & \dots & \dots & x_{1m} \\ x_{21} & x_{22} & & & x_{2m} \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & \dots & x_{nm} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & \dots & \dots & b_{1m} \\ b_{21} & b_{22} & & & b_{2m} \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & \dots & b_{nm} \end{bmatrix}$$

Solving for x_{ji} results in

$$x_{1i} = b_{1i} \cdot {a_{11}}^{-1}$$

for any element x_{1i} $(1 \leq i \leq m)$ in row 1,

$$x_{2i} = (b_{2i} - a_{21} \cdot x_{1i}) \cdot {a_{22}}^{-1}$$

for row 2,

$$x_{3i} = (b_{3i} - a_{31} \cdot x_{1i} - a_{32} \cdot x_{2i}) \cdot {a_{33}}^{-1}$$

for row 3 and, generally,

$$\begin{aligned} x_{ji} &= (b_{ji} - a_{j1} \cdot x_{1i} - a_{j2} \cdot x_{2i} - \ldots - a_{j,j-1} \cdot x_{j-1,i}) \cdot {a_{jj}}^{-1} \\ &= (b_{ji} - \sum_{k=1}^{j-1} a_{jk} \cdot x_{ki}) \cdot {a_{jj}}^{-1} \end{aligned}$$

for any row j $(1 \leq j \leq n)$.

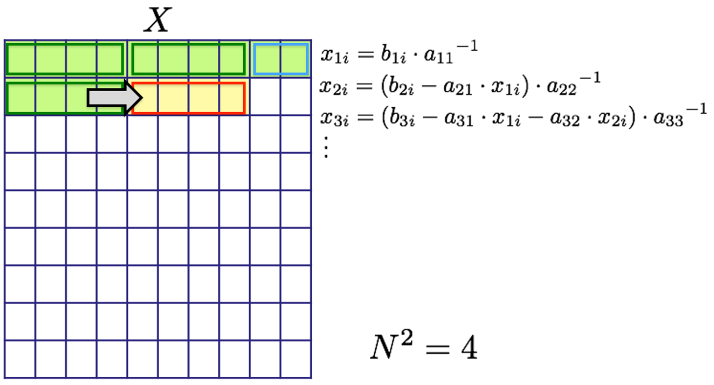


Figure 6.16: TRSM scalable execution window progress (here, $N^2 = 4$ elements), the progression of this window through the data ensures no more than one load per memory port per cycle. Execution runs on full efficiency until the last columns, where a partial array fill is possible every row.

6.3.7.2 Mapping

For TRSM, operation on rows only is possible. A scalable execution window of height 1 and width N^2 is created (Fig. 6.16) which iterates X row-wise. In every window, all necessary computations for $x_{ji}, \ldots, x_{j,i+N^2}$ are completed, before proceeding with the next window. This implies loading a single value a_{jk}, distributing it to all L0-elements.

N^2 values of x_{ki} are loaded and multiplied in parallel. Then, the next set of values from A and X are loaded, accumulating the result. After adding up all partial results to

$$s_{ji} := \sum_{k=1}^{j-1} a_{jk} \cdot x_{ki},$$

subtracting this value from b_{ji} and multiplying it with ${a_{jj}}^{-1}$ for the scaling, this value is broadcast to the local store in the communication layer of every element.

Regarding the override logic, there is one window of width

$$m \mod N^2$$

in every row, which might ideally be 0, which makes predication in the control flow side easy to implement.

Language-wise this kernel requires however 1-to-all broadcast capability, which can be easily assembled from the elementary function pool of the communication layer.

As with TRSV, **Row** 1 has to be considered as a special case because s_1 does not exist. For that purpose, only the multiplication $b_{1i} \cdot {a_{11}}^{-1}$ is executed.

6.3.7.3 Complexity

For each x_{1i}, there are $i-1$ subtractions and multiplications necessary, hence for all x_{mi}:

$$2m \sum_{i=1}^{n} (i-1) = mn(n-1)$$

Additionally, one reciprocal per row and one multiplication per element is computed, which results in

$$\begin{aligned} op_{\texttt{trsm}}(\{n,m\}) &= mn(n-1) + mn \\ &= mn^2 \\ op_{\texttt{trsm}_{div}}(\{n,m\}) &= n \end{aligned}$$

The minimum cycle complexity is

$$\begin{aligned} c_{\texttt{trsm}min}(\{n,m\},N) &= 8\left(\tfrac{1}{N^2}\, op_{\texttt{trsm}}(\{n,m\}) + 4\, op_{\texttt{trsm}_{div}}(\{n,m\})\right) \\ &= 8n\left(\tfrac{mn}{N^2} + 4\right) \end{aligned}$$

6.3.7.4 Efficiency

As TRSM has less execution dependencies and more data-streaming-friendly algorithm, the optimizations from the algorithm side are easily scalable. The special requirement of broadcasting can be accommodated by Layer 1, therefore each iteration can be pipelined efficiently. This kernel exhibits similar efficiency levels to the GEMM kernel, despite the complexity. A high degree of architectural flexibility could be attained, as shown in Fig. 6.17. Detailed data are added to Appendix B, Fig. B.9-B.10.

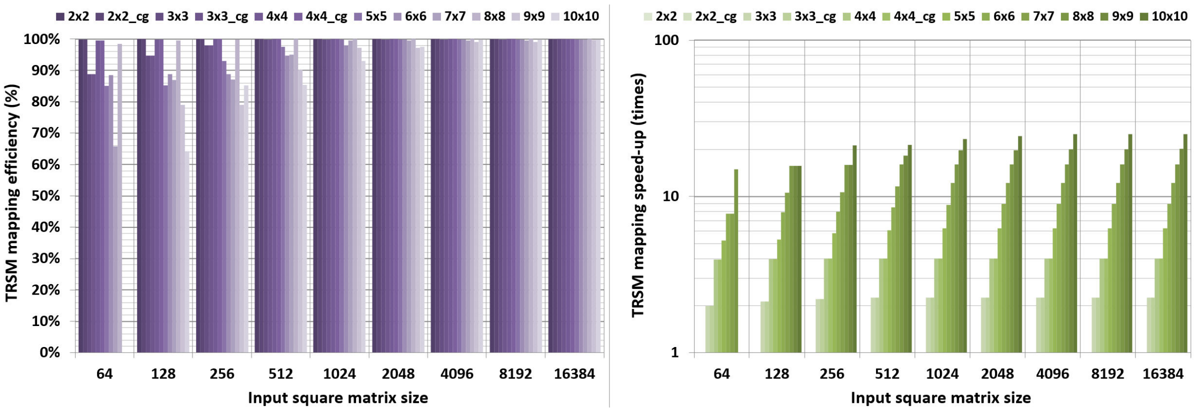


Figure 6.17: TRSM mapping efficiency and mapping-based speed-up for various architectures and data sizes. Since it uses a similar mapping scheme to the efficient GEMM kernel, very high efficiencies can be attained.

6.3.8 Lower-Upper Factorization (LU)

6.3.8.1 Algorithm

The input to the *LU*-factorization is a square matrix $A \in \Re^{n \times n}$ with $|A| \neq 0$. The output for our implementation[3] consists of two matrices, $L \in \Re^{n \times n}$ and $U \in \Re^{n \times n}$, such that values above and below the diagonal are zero, respectively [140]:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & \dots & \dots & a_{1n} \\ a_{21} & a_{22} & & & & a_{2n} \\ \vdots & & \ddots & & & \vdots \\ \vdots & & & \ddots & & \vdots \\ \vdots & & & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & \dots & \dots & a_{nn} \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & \dots & \dots & \dots & 0 \\ l_{21} & 1 & 0 & & & \vdots \\ l_{31} & l_{32} & 1 & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & 0 & \vdots \\ \vdots & & & \ddots & 1 & 0 \\ l_{n1} & \dots & \dots & \dots & l_{n,n-1} & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & \dots & \dots & \dots & u_{1n} \\ 0 & u_{22} & & & & u_{2n} \\ \vdots & 0 & \ddots & & & \vdots \\ \vdots & & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & \ddots & \vdots \\ 0 & \dots & \dots & \dots & 0 & u_{nn} \end{bmatrix}$$

---

[3] It should be mentioned here, that there are other implementations of the *LU*-factorization, which do not require the diagonal elements of L to be 1. Furthermore, our implementation does not deal with the fact that not every input matrix A can be processed using this algorithm. An advanced implementation of LU, possibly using techniques like *Pivoting*, is subject to future work.

6.3.8.2 Mapping

Since LU factorization dependencies and algorithm are complex, additionally to the general mapping considerations from Section 6.3.1.2, a mapping procedure was conceived to respect the constraints, while being scalable and efficient. The most important restriction is that rectangular or square windows with a height of more than 1 are not possible either since this would require to work on columns in either matrix. An efficient block-based scheduling and mapping solution has been discussed in [138] for an earlier version of *Layers*, however only for a fixed 4 × 4 PEs and 8 memory ports configuration, yielding a fixed mapping. This mapping is focused on scalability and shows superior results due to improved design, programmability and architectural flexibility.

For clarity, the proposed mapping is illustrated for the LU-algorithm using an example with $n = 3$, hence $A_3 = L_3 \cdot U_3$, however the mapping holds for any $n > 3$ [140]:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

where L_3 and U_3 can be computed as

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ a_{21}{u_{11}}^{-1} & 1 & 0 \\ a_{31}{u_{11}}^{-1} & (a_{32} - l_{31}u_{12}){u_{22}}^{-1} & 1 \end{bmatrix}$$

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} - l_{21}u_{12} & a_{23} - l_{21}u_{13} \\ 0 & 0 & a_{33} - l_{31}u_{13} - l_{32}u_{23} \end{bmatrix}$$

For more efficient memory usage, the matrices L_3 and U_3 are combined to matrix Q_3, superposing the matrices such that [140]:

$$Q_3 := \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & u_{22} & u_{23} \\ l_{31} & l_{32} & u_{33} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21}{u_{11}}^{-1} & a_{22} - l_{21}u_{12} & a_{23} - l_{21}u_{13} \\ a_{31}{u_{11}}^{-1} & (a_{32} - l_{31}u_{12}){u_{22}}^{-1} & a_{33} - l_{31}u_{13} - l_{32}u_{23} \end{bmatrix}$$

which is computed as follows. First, $l_{11} := 1$ implies $u_{1n} = a_{1n}$, which results in [140]:

$$= \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ a_{21}{u_{11}}^{-1} & a_{22} - l_{21}u_{12} & a_{23} - l_{21}u_{13} \\ a_{31}{u_{11}}^{-1} & (a_{32} - l_{31}u_{12}){u_{22}}^{-1} & a_{33} - l_{31}u_{13} - l_{32}u_{23} \end{bmatrix}$$

Then, $\boxed{l_{21}}$ is computed, which is used to calculate $\boxed{u_{22} \dots u_{2n},}$ where the multiplications and subtractions are parallel. This gives a flexible execution window of size $N^2 \times 1$, sliding to the right on the row until the end, where possible size mismatches are truncated by predication in the Q-layer [140]:

$$= \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ \boxed{a_{21}{u_{11}}^{-1}} & \boxed{a_{22} - \boxed{l_{21}u_{12}}} & \boxed{a_{23} - \boxed{l_{21}u_{13}}} \\ a_{31}{u_{11}}^{-1} & (a_{32} - l_{31}u_{12}){u_{22}}^{-1} & a_{33} - l_{31}u_{13} - l_{32}u_{23} \end{bmatrix}$$

For the next row, $\boxed{l_{31}}$ is computed yielding the partial results $\boxed{{l_{32}}' \dots {u_{3n}}',}$ using again all execution units.

The procedure is repeated until l_{n1} is reached [140]:

$$= \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & u_{22} & u_{23} \\ \boxed{a_{31}{u_{11}}^{-1}} & \boxed{(a_{32} - \boxed{l_{31}u_{12})}}\,{u_{22}}^{-1} & \boxed{a_{33} - \boxed{l_{31}u_{13}}} - l_{32}u_{23} \end{bmatrix}$$

Please note the dependencies on the row on l_{mn} and on the column for u_{mn}. The higher m and n are, the longer the string of dependencies, which are rank-nested within the matrix. In the proposed mapping, the sub-matrices of decreasing rank are iterated repeatedly, as shown in Fig. 6.18, until the complete Q_n matrix is computed, as shown in Fig. 6.18. In the above example, only the last row remains, after which the final form of Q_3 is reached. [140]

In general, while the the first row of Q, i.e. $u_{11}, \dots, u_{1n}$, needs no computation, there are $n-1$ values which can be computed in parallel in any other row within the first iteration, $n-2$ in the second iteration and so on until the computation of u_{nn} is a single value. Because only one single l_{ij}-value has to be computed to perform computations in all the following values in row i in parallel, which might result in final u_{ik} or partial results ${l_{ik}}'^{\cdots}$ respectively ${u_{ik}}'^{\cdots}$, the scaling performs well on large matrix sizes. Having a window of height 1 makes the window fit efficiently regardless of the input matrix size, because there can only be a maximum of one window in any row which needs truncation. [140]

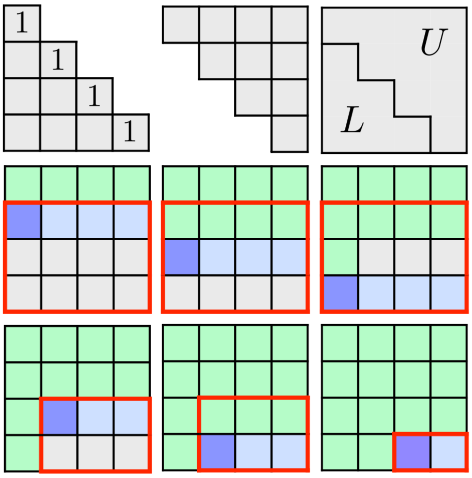


Figure 6.18: LU execution window progress over the input matrix. Rank of the matrix is iteratively decreased (red square) while simultaneously computing l and u values on each row (blue), in window sizes of N^2 (shaded blue). Once a rank is complete (green), next lower rank is executed. [140]

From the architectural language perspective, the algorithm makes heavy use of 1-to-many broadcasts, as the pre-computed l_{ik}-term must be stored and broadcast to all execution windows during the execution of a row. Otherwise, the standard set perfectly satisfies data flow requirements.

6.3.8.3 Complexity

Complexity can be divided in 3 parts: division complexity $op_{\texttt{lu}_{div}}$, complexity for lower $op_{\texttt{lu}_l}$ and upper factorization $op_{\texttt{lu}_u}$, then summed up to $op_{\texttt{lu}}$. In order to compute L, one multiplication per element in the lower diagonal part, excluding the diagonal itself, is needed, i.e. $\frac{n}{2}(n-1)$. Additionally, 0 pairs of subtractions/multiplications are needed in the first column, 1 pair in the second column and so on, hence [140]:

$$op_{\texttt{lu}_l}(\{n\}) = \frac{n}{2}(3n-7)+2. \tag{6.9}$$

Similarly, in order to compute U, 0 pairs of subtractions/multiplications in row 1 are needed, 1 pair in row 2 and so on, resulting in

$$op_{\texttt{lu}_u}(\{n\}) = 2\sum_{i=1}^{n-1}(n-i)i = \frac{n}{3}(n^2-1). \tag{6.10}$$

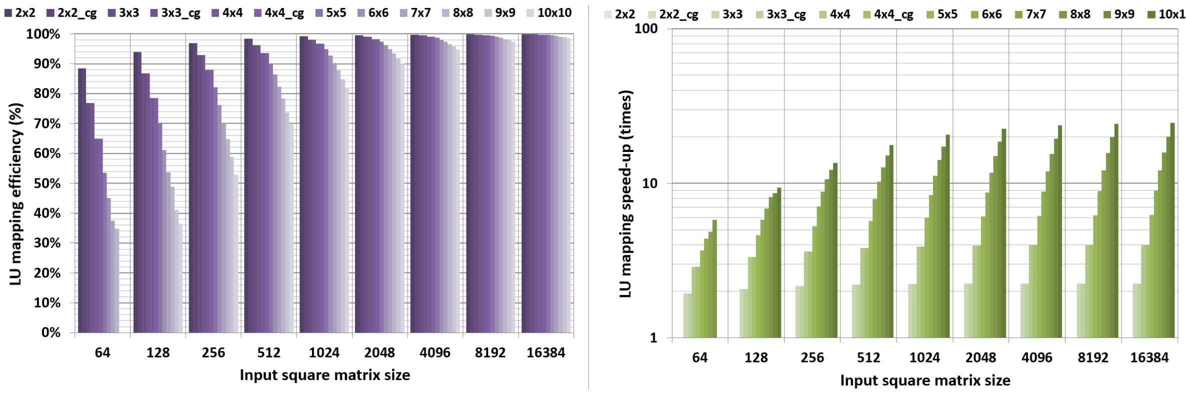


Figure 6.19: LU mapping efficiency and expected mapping speedup for various architectures (N=2..10) and data sizes (64..16384). [140]

Finally, adding these to $op_{\mathtt{lu}_{div}}(\{n\}) = n - 1$ the minimum cycle complexity on Layers with $r_{L0:L1:L2} = 1:8:8$ and 4 cycles for a division:

$$c_{\mathtt{lu}_{min}}(\{n\}, N) = 8\left(\frac{1}{N^2}\left(\frac{n^3}{3} + \frac{3n^2}{2} - \frac{23n}{6} + 2\right) + 4\,(n-1)\right)$$

6.3.8.4 Efficiency

Mapping efficiency is shown in Fig. 6.19. With enough data to fill up the execution layer of array size N, the efficiency is approaching maximum, however, it can be noted that it comparison with simpler kernels, it is less efficient. Main contributor to inefficiency is execution at low matrix ranks, where the ratio of full execution windows w.r.t. partial execution windows decreases. Expected speedup from mapping when scaling N, is shown on the right side of Fig. 6.19, very close to the expected theoretical value i.e. scaling by x amount of elements a speedup of close to x is achieved. Detailed data are added to Appendix B, Fig. B.11-B.12.

6.3.9 Givens Rotation (GR)

6.3.9.1 Derivation of Parallel CSFG and CSDFG Algorithms

In classical GR [68], zeroing out one element is done by applying a *rotation* locally i.e. multiplying with a constructed *Givens matrix* of size 2 × 2, such that the chosen element – part of a local 2 × 2 sub-matrix – becomes zero, then updating the rest of the matrix to compensate for this multiplication and conserve the information of the annihilated element. [137] This is one of the more complex NLA kernels used for QR factorization, employed in many applications.

The construction of this Givens matrix involves square-root and division operations and its execution contains heavy sequential parts. Since especially square root is a complex operation to implement in hardware and even ASIC implementations are resource-intensive, attention was focused on simplified versions of the algorithm. In Square-root Free Givens rotation (SFG) [68] and Square-root and Division-Free Givens rotation (SDFG) [70], these architecturally complex operations are omitted by increasing computational complexity by using more additions, subtractions and multiplications at the expense of numerical precision and stability. Although this simplifies hardware implementation, it still does not allow a parallel implementation, because a new Givens matrix generation for zeroing out a new element requires completion of the updates of the matrix elements of the affected rows. [137]

In [111] it is shown that by merging the effect of several Givens matrices in a large set of operations, several elements can be zeroed out at once, affecting several rows. The column-wise versions of the algorithms (CSF and CSDF) are also proposed in [136], derivation of which is reproduced here for clarity. The significant advantage of this approach is that this larger set of operations for completing a large rotation is highly parallel, especially the updates on several rows. When mapped onto highly parallel architectures, such as *Layers*, significant efficiency is gained although the amount of computation is increased. [137]

In either case (SFG or SDFG), it takes $\frac{n(n-1)}{2}$ sequences to upper triangularize the matrix of size $n \times n$, as shown in Fig. 6.20.

The Givens matrix (for SFG and SDFG) G is defined by:

$$G_{i,j} = diag(I_{i-2}, \tilde{G}_{i,j}, I_{m-i}) \tag{6.11}$$

A repeated application of the Givens matrix for each sub-diagonal element yields:

$$(G_{n,1}G_{n-1,1}.....G_{2,1})(G_{n,1}G_{n-1,1}.....G_{2,1})^T = I_{n\times n} \tag{6.12}$$

Each step creates an execution dependency on the (partial) updates for the affected rows which limits parallelism and scalability.

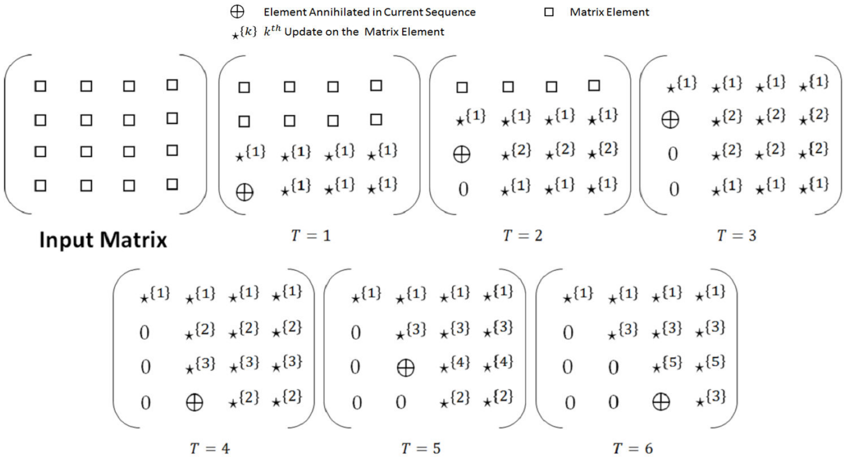


Figure 6.20: Annihilation regime in classical SFG and SDFG algorithms [136]

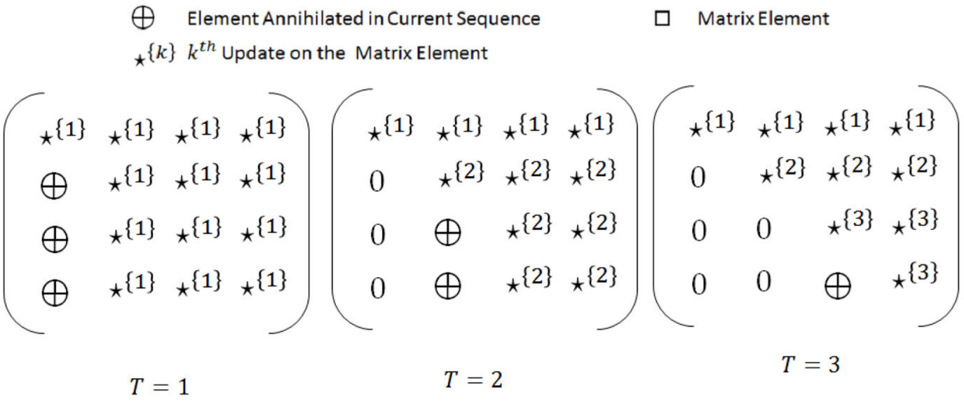


Figure 6.21: Annihilation Regime in Column-wise GR Algorithms (CSF GR and CSDF GR) [136]

The main idea for the CSF and CSDF algorithms is to merge the effect of several Givens matrix applications, such that several elements can be zeroed out at once, affecting several rows. Thus Eq. 6.12 can be extended for multiple columns of the matrix to annihilate $\frac{n(n-1)}{2}$ elements simultaneously. [136] If $Q_1 = G_{n,1}G_{n-1,1}....G_{2,1}$, $Q_2 = G_{n,2}G_{n-1,2}....G_{3,2}$ are defined so and $Q_{n-1} = G_{n,n-1}$ then

$$Q_1Q_2....Q_{n-1}X = QX = \begin{bmatrix} R \\ 0 \end{bmatrix} \tag{6.13}$$

where R is an upper triangular matrix of size $n \times n$. This creates a large set of highly parallel operations for calculating the large Givens matrix and also yields a significantly larger parallel update field encompassing several rows. The number of dependent steps is reduced, as shown in Fig. 6.21, allowing flexible mapping of the computations to available resources. [136]

Without delving into mathematical details, the following example illustrates how the Column-wise versions of the SFG and SDFG work. An interested reader can check the mathematical background in [68] [70] and [111]. The updated matrix Q_1X is shown for SFG and SDFG in (6.14) and (6.15) respectively, after applying cumulative Givens matrix Q_1 zeroing out the sub-diagonal elements of the first column of a 4×4 input matrix and the necessary updates for each affected row. [137]

Example: Taking input matrix $X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix}$ and applying one iteration of CSFG yields

$$Q_1X = \begin{bmatrix} \frac{p_3}{x_{41}} & \frac{x_{11}x_{12}+s_{11}}{x_{41}} & \frac{x_{11}x_{13}+s_{21}}{x_{41}} & \frac{x_{11}x_{14}+s_{31}}{x_{41}} \\ 0 & x_{12} - \frac{x_{11}}{p_2}s_{11} & x_{13} - \frac{x_{11}}{p_2}s_{21} & x_{14} - \frac{x_{11}}{p_2}s_{31} \\ 0 & x_{22} - \frac{x_{21}}{p_1}s_{12} & x_{23} - \frac{x_{21}}{p_1}s_{22} & x_{24} - \frac{x_{21}}{p_1}s_{32} \\ 0 & x_{32} - \frac{x_{31}}{x_{41}}x_{42} & x_{33} - \frac{x_{31}}{x_{41}}x_{43} & x_{34} - \frac{x_{31}}{x_{41}}x_{44} \end{bmatrix} \tag{6.14}$$

Similarly, applying one iteration of CSDFG on X yields

$$Q_1X = \begin{bmatrix} p_3 & x_{11}x_{12} + s_{11} & x_{11}x_{13} + s_{21} & x_{11}x_{14} + s_{31} \\ 0 & x_{11}s_{11} - x_{12}p_2 & x_{11}s_{21} - x_{13}p_2 & x_{11}s_{31} - x_{14}p_2 \\ 0 & x_{21}s_{12} - x_{22}p_1 & x_{21}s_{22} - x_{23}p_1 & x_{21}s_{32} - x_{24}p_1 \\ 0 & x_{42}x_{31} - x_{41}x_{32} & x_{43}x_{31} - x_{41}x_{33} & x_{44}x_{31} - x_{41}x_{34} \end{bmatrix} \tag{6.15}$$

where

$p_1 = x_{41}^2 + x_{31}^2; \quad p_2 = p_1 + x_{21}^2; \quad p_3 = p_2 + x_{11}^2$

$s_{12} = x_{31}x_{32} + x_{41}x_{42}; \quad s_{11} = x_{21}x_{22} + s_{12}$

$s_{22} = x_{31}x_{33} + x_{41}x_{43}; \quad s_{21} = x_{21}x_{23} + s_{22}$

$s_{32} = x_{31}x_{34} + x_{41}x_{44}; \quad s_{31} = x_{21}x_{24} + s_{32}$

It is interesting to note how the p and s terms accumulate over the rows and how these are shared in the rows and columns, creating parallelism and also input data locality. [137] △

6.3.9.2 Mapping the Algorithms

Carefully analyzing CSDFG and CSFG, terms with special properties can be identified: "p", "common", "rest" and "s"-terms. For each zeroed element, its row must be updated with effect of current previous zeroed elements, accumulated in the "p" and "s" terms, which required addition of squared terms for "p" and accumulation of one multiplication for "s". During the calculation of partial "p" and "s" terms, data is broadcast on the row, which represents the "common", and is consumed together with the local term for each column, representing the "rests". [136]

Fig. 6.22 shows how the algorithm progresses through the input matrix and the mapping of the kernels of each algorithm. Modulo-scheduling concepts were used to unroll the bottom→top kernel progress over one column to extract the common data points and pipelined the computation such that all PEs are busy, independently. Each PE is responsible for one column vector, allowing scalability with the window size

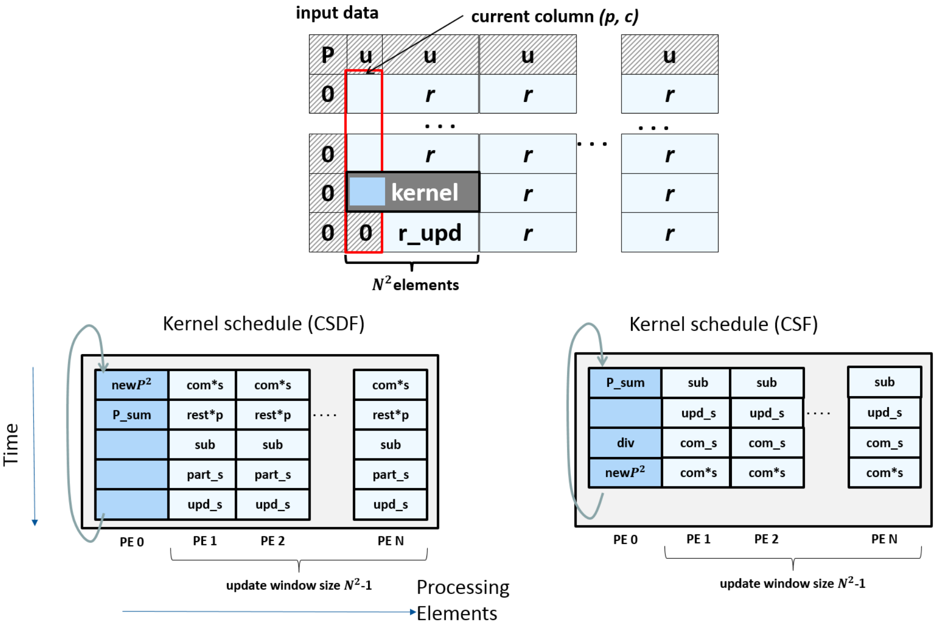


Figure 6.22: Mapping CSDFG and CSFG: algorithm execution over input data, with the two kernel mappings. [137]

$N^2 - 1$, reserving one PE to execute computation for "p" and division in case of CSF. The downside of this kernel scheduling is that this element's efficiency was sacrificed to keep regularity and scalability in the algorithm progression. In the kernel for both algorithms, the "common" and "rests" terms are used twice, once for "s" calculation and once for the updates, in subsequent rows, allowing us to temporarily save these terms in L1 and distributing them as needed. This saves on one hand memory load bandwidth, but also the memory read latency on the other hand. Both algorithms use no more than 4 registers per PE in L1, the current architecture being configured for up to 7 L1 registers. [137]

From the architectural language perspective, to exploit data locality, local storage functions were required, coupled with broadcast functions for the "p" coefficients.

6.3.9.3 Complexity

For an $n \times n$ input matrix, the complexity of the column-wise version in terms of additions and multiplications is as follows, including divisions for CSFG.

$$M_{CSFG} = \frac{2n^3 + 3n^2 + n}{3}; \qquad A_{CSFG} = \frac{4n^3 - 3n^2 - n}{6}$$

$$D_{CSFG} = \frac{n(n-1)}{2}$$

$$M_{CSDFG} = \frac{2n^3 + n}{3}; \qquad A_{CSDFG} = \frac{2n^3 + 3n^3}{6}$$

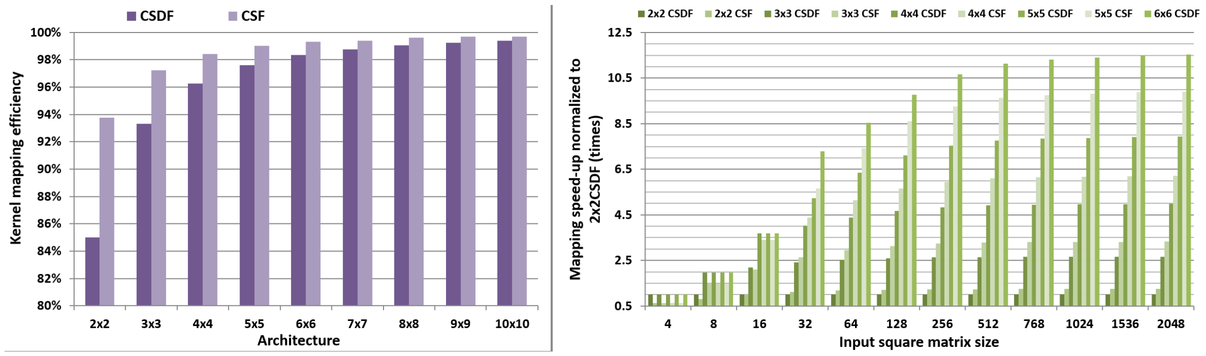


Figure 6.23: Mapping efficiency and speed-up for the two column-wise GR kernels. [137]

where M_{CSFG}, A_{CSFG} and D_{CSFG} represent multiplications, additions and divisions in CSFG while M_{CSDFG} and A_{CSDFG} represent multiplications and additions in CSDFG, which has lesser multiplications than the non-column version in [70], while of CSFG the complexity does not change. [136]

6.3.9.4 Efficiency

The scheduling was optimized for L0 PEs in 5 macro-cycles for CSDFG and 4 macro-cycles for CSFG, taking the array size as a parameter achieving $> 99\%$ mapping efficiency for array sizes $N > 5$, and $> 90\%$ for $N = 2..4$, excepting the 2×2 CSDFG for which only 85% could be achieved, as shown in Fig. 6.23. Optimal usage could not be achieved without breaking regularity and scalability. The first PE recalculates the "p" terms for each update window shift, since storing input matrix column size number of "p" terms would be infeasible for large matrix sizes. In case of CSFG the divider latency (4 cycles) could be pipelined with the calculation of "p" terms. Detailed data are added to Appendix B, Fig. B.13.

6.4 Architectural Performance Evaluation with the NLA Kernels

6.4.1 General Considerations

For this exploration, *Layers* has been coded completely in the LISA ADL of Synopsys Processor Designer, completely parametrized for easy scalability. Simulations have been conducted for random square input matrices of size 4..16384, for different combinations of $P = 2..32$ and $N = 2..10$. The functional assembly program for each of the case study kernels have been coded in a scalable way by means of embedded ruby scripting. Additionally some LISA code parts have also been parametrized with ruby, allowing a parameter-based scalable generation of RTL code. Result values are for single-precision floating point (32-bit), as the execution layer is based on 32-bit floating point library modules from Synopsys DesignWare. For these configurations RTL code has been generated and synthesized with DesignCompiler I-2013-SP5 for Faraday 65nm standard-cell ASIC technology library.

A high-level power estimation is conducted with PowerCompiler with back-annotated switching activity files from RTL simulations. For lower size designs, clock-gating has been enabled at synthesis, marked with `_cg` in the results. For larger designs the additional clock-gating circuitry used more power than it actually saved, hence those results are removed for clarity.

Dual port memory banks were employed, however towards the *Layers* L2 only one port of each bank was visible, the second one remaining reserved for System-on-Chip integration. Also, while scaling the kernels and *Layers* array, using more memory banks (ports) than the minimal amount for sustaining data transfer for the optimized kernel blocks yielded greater power and area usage with no advantages, as the power gained from more relaxed L1/L2 data handling did not compensate for the power and area used for additional structures.

6.4.1.1 Programmability

These assembly programs are completely parametrized using embedded Ruby code, allowing generation of required variants for the architectures considered (different P and N). This occurs by modifying a `defines.h` header file, and generating the required assembly code via Ruby for each kernel and architecture size. As the language of the architecture does not change when the architecture scales coupled with the scalable mapping of the kernels, scripting the appropriate scaling factors was straightforward. For instance, the assembly program contained

- 5 L0-cycles (– 5 L0 functional words) for GEMM,
- 13 L0-cycles for GEMV,
- 11 L0-cycles for DOT including addition folding,
- 12 L0-cycles for LU,

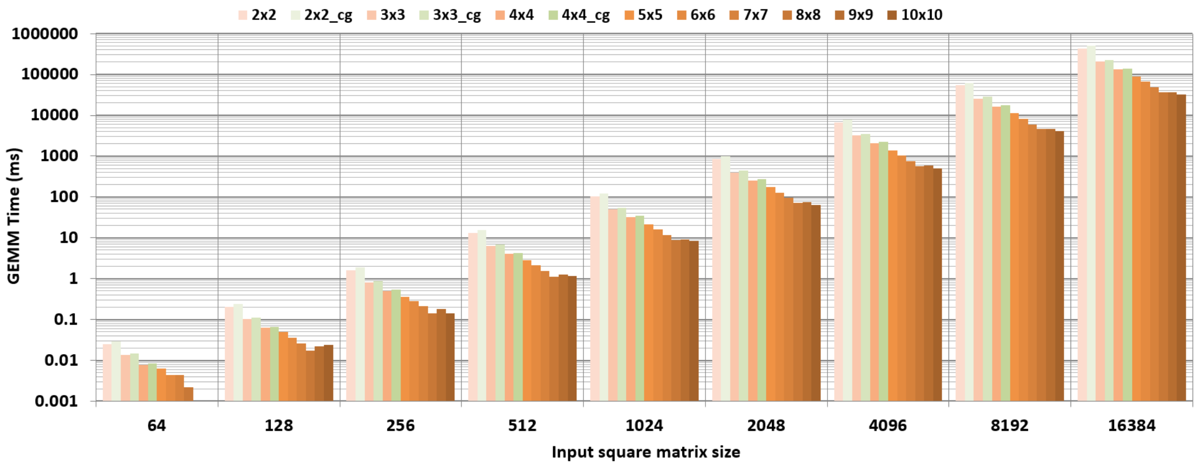


Figure 6.24: Timing results for GEMM. More than an order of magnitude speedup can be observed. Other kernels show similar results. [139]

- 17 L0-cycles for CSDFG and 28 macro-cycles for CSFG, etc.

, including prolog and epilog of the kernels. As the array was scaled, the number of required assembly functional words did not change, but it contained more functional calls (automatically scaled) according to architecture size. The ease of such scaling is one of the main strengths of functional reconfiguration-based programming and scalable mapping optimizations presented in previous sections. Moreover, such short programs can be produced manually without significant effort.

6.4.2 Time, Energy and Scalability

After performing synthesis, the timing of the architecture and thus that of kernel's execution could be extracted, coupled with power estimation data. Detailed timing results for the GEMM kernel are provided in Fig. 6.24 for each configuration and input matrix size, highlighting the scalability of our approach. Execution time spreads over several orders of magnitude with varying input data size, while an order of magnitude speed-up can be maintained between the smallest and largest array for large input data sizes. Except for the largest architectures, where the critical path of the L1 structures severely affected frequency and thus energy, the architecture and mapping scale with almost constant energy (<10% variance), translating into a clean trade-off between area and speed, without affecting energy. Other kernels show similar performance, detailed data is summarized in Appendix C.

The trend is slightly broken for the largest designs. The sudden increase in area and critical path is due to the upgrade from 3-bit multiplexers sufficient for $N \leq 8$ interconnects to 4-bit ones and the longer wire length. Reducing the interconnect length in L1 would cancel out the penalty for large arrays by shortening the critical path, if large designs are necessary. Clock-gating optimization has been enabled during synthesis exploiting the switching activity information from simulations, which for smaller designs (marked with `_cg`) improved results by roughly 20-40% compared to non-clock-gated counterparts. (Fig. 6.27). For large designs the overhead in power and area of inserting clock-gating logic was higher than the actual saved power, those results are omitted for clarity. [140]

Fig. 6.25, shows the energy results for the GEMM kernel. Except for the largest architectures (N=9,10), where the critical path of the L1 structures severely affected frequency and thus energy, the architecture and mapping scale with almost constant energy (<10% variance), translating into a clean trade-off between area and speed, without affecting energy. [140]

Thus, designs that need to respect certain requirements in the amount of memory ports or a certain amount of performance or area, can be easily picked from the scalable set, without needing to consider the energy impact of the choice, as energy stays constant for a given workload. In terms of energy, it is very important to note how the overall energy remains comparable for a given input matrix size, letting designers to directly trade off execution speed with area, without worrying about energy. The loss of frequency with increasing size, power scaling and speed-up scaling balance out to a constant energy requirement. [140]

Fig.6.26 reflects this trade-off capability for the Givens rotation kernel. For given architectural parameters, such as the ratio r, ports P and array size N^2, upper bounds on the available memory bandwidth and processing capability can be traced. Similarly, a given kernel mapping, certain requirements on memory, data movement and processing can be deduced from the optimized kernel and the application language. Lower r would require more than $2 \times P$ worth of bandwidth. On the other hand if both PE and port number is constraint, mapping efficiency must be sacrificed by execution parallel code sequentially, to accommodate the bottle-neck. Either of these parameters can be used as the optimization target when mapping: e.g. if the architecture is limited to 4 memory ports, the chart in Fig.6.26 shows up to how many process-

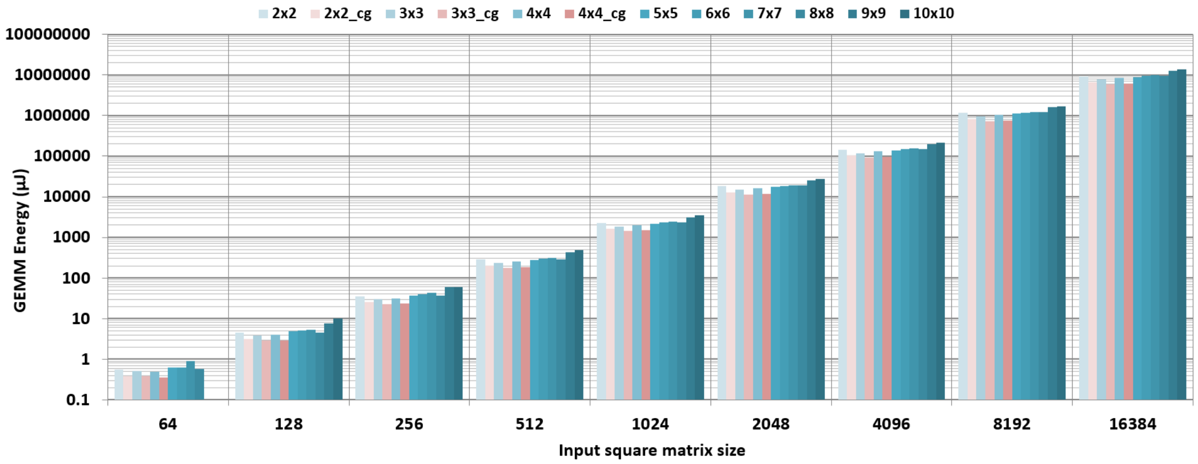


Figure 6.25: Energy results for GEMM. Except the largest arrays energy stays constant when scaling N. Clock-gated designs *_cg perform better. Constant energy is required for the same problem size across variants, giving a clean area:performance trade-off. [139]

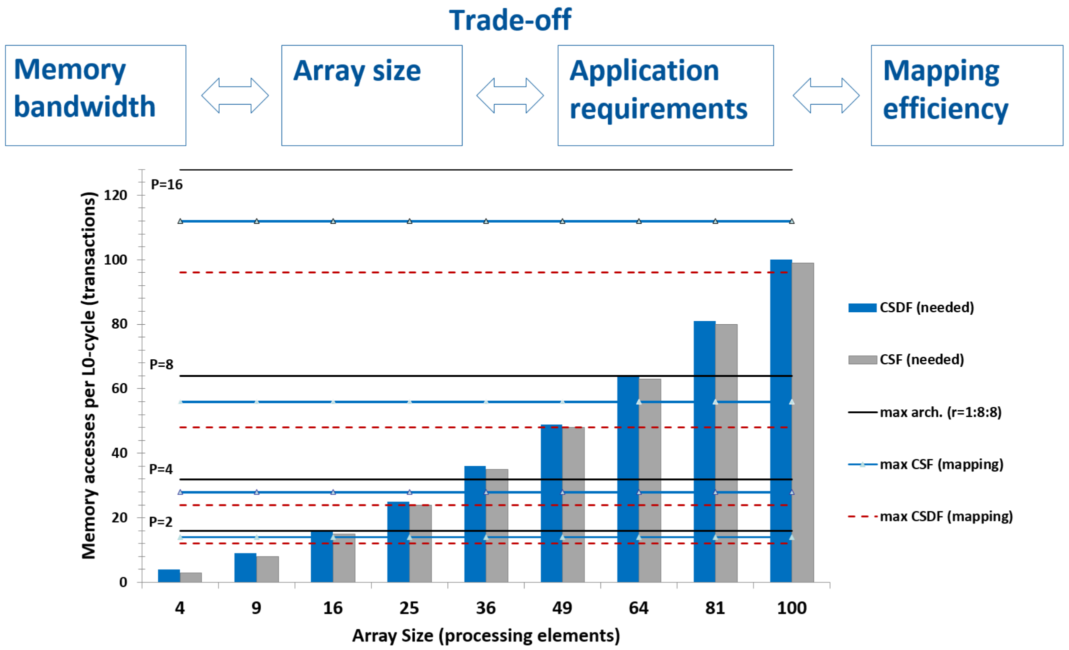


Figure 6.26: Architectural memory bandwidth limits, mapping-based limits vs. actual algorithmic bandwidth requirements with varying N^2 and P. [137]

ing elements these memory ports can accommodate, using which kernel mapping. Vice-versa, if a certain performance of the kernel is desired, the timing performance shows all architecture that respect the performance constraint, therefore necessary memory bandwidth can be determined for a given kernel. Since energy is quasi-constant whichever architectural variant is picked for a given kernel input data size, it can be taken out of the equation. *Layers* allows smooth scaling for exploring the optimal point for a given application, trading off mapping optimality and parallelism for resources.

Thus, any of the variables can be used as a constraint, from which others can be derived, allowing a truly scalable mapping and resource trade-off.

The area and power results are illustrated for the GEMM kernel in Fig. 6.27 and Fig.6.28. Other kernels have very close values, due to small differences in switching activity, for the power. Area increases steeply with an increase of array size, as does power. For energy, this is then compensated however by the architectural flexibility to match the application closely and the resulting speed-up in time.

6.4.3 Comparisons

6.4.3.1 Options with *Layers*

Fig. 6.28 provides some area, frequency and performance density data. The frequency of each architecture is limited by the control flow complexity in the state machine stage for small N, and by L1 critical path for larger N at $r = 1:8:8$. Choosing lower inter-layer speed ratio r, the fp PEs limit overall frequency, while sacrificing

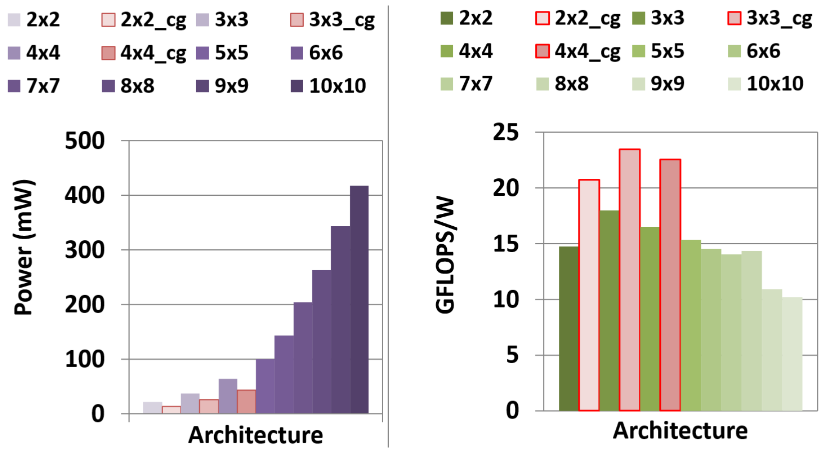


Figure 6.27: Power results for GEMM and GFLOPs/W for *Layers*. Clock gated designs perform much better. [139]

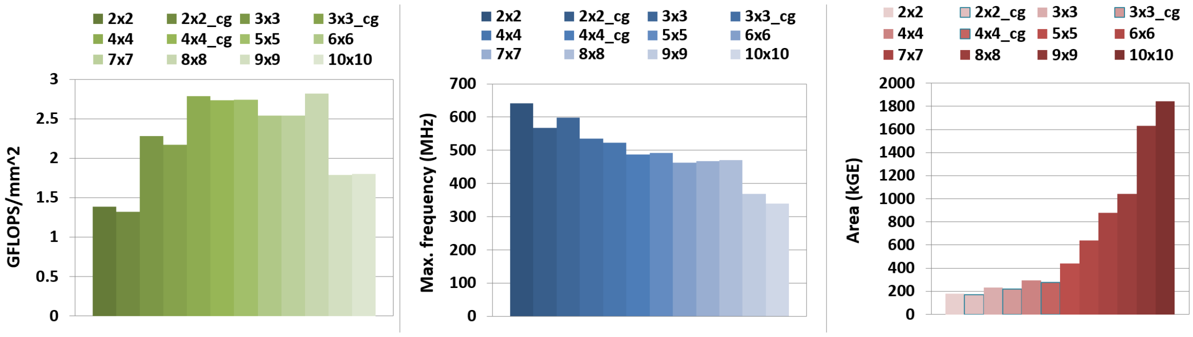


Figure 6.28: Area, frequency and performance density for the Layers architecture. Clock-gated designs (*_cg) have worse performance density and lower frequency but have slightly better area values than their counterparts. Largest designs have significant area increase due to bigger multiplexer and decode logic structures (3-bit → 4-bit) and significant frequency penalty due to long L1 wires. [139]

memory bandwidth, especially when requiring a divider. When comparing to the clean version of *Layers*, it is interesting to note that the reconfigurable control path slows the architecture down, although by not operating near maximum frequency has great advantages in power consumption (Table 6.1).

The clean version does not contain the control flow language elements required for kernels, which are kernel specific and sometimes have a long critical path. An automated derivation of these is attempted and described in Chapter 7.

Comparing with other architectures is difficult for a variety of reasons, mainly because of different algorithms and platforms, making comparisons often unfair.

6.4.3.2 LAC

Most recent results on LAC [126], a CGRA targeting linear algebra, show comparable numbers for LU with partial pivoting. A fixed block-based mapping is used and architectural enhancements for pivoting are employed to reach excellent performance. Although LAC is considering a more complex algorithm, it lacks the seamless scalability provided by our architecture and mapping solution. While performance in terms of aggregate energy efficiency is similar (Table 6.1), the estimated area of LAC is 6.2× larger than a similar-sized *Layers* core. An exact comparison is not possible due to the different process libraries and the estimated nature of the results reported in [126]. It would be interesting to compare actual post-synthesis and post-physical design results in the same technology node. [140]

6.4.3.3 REDEFINE

A recently enhanced REDEFINE CGRA for NLA [112] shows comparable values for 65nm, and shows better performance density if scaled to 45nm with custom DOT product units, however no execution times for large matrices could be found. Based on the reported latencies for 60×60 and 120×120 data size running matrix multiplication in [112], the slowest variant of *Layers* performs 4.2× and 2.8× faster on a 64×64 and 128×128 data set.

It is also interesting to note, how the REDEFINE architecture executes kernels, which conceptually uses a kind of *hyperfunctions* to reconfigure its fabric. While it doesn't rely on functional configuration, the fabric tries to represent patterns from the application DFG, such that an accelerated execution is possible. This approach is however different from the layered approach proposed here. [140]

6.4.3.4 DSP, GPGPU and FPGA aggregated results

Table 6.1 aggregates the results over several architectures that target NLA kernels. As the variance on architectural styles, technology, application kernel widely vary, it is difficult to do a clean comparison. Global values that give a hint on the positioning of *Layers* in the architectural landscape can be provided via power density or computation per Watt numbers. Due to a lack of clear absolute timing or energy details, more fine-grained comparison was not directly possible.

In [60], details about a 64×64 inversion with GSGR on a modern DSP platform are discussed, requiring 2.2ms, which is two orders of magnitude slower than the respective triangularization on the slowest variant of our architecture for GR, however the former handles inversion. Comparing with Tournament-GR [96] for a 8×8 matrix the execution time of its FPGA implementation at 51.9MHz yielded 0.000365ms which is 5.8× slower than the slowest CSFG version. In the 2D-systolic design [164] a 12×12 matrix is processed in 0.0101ms on an FPGA implementation at 139MHz, which is 25× slower than execution of a 16×16 matrix in the slowest of our designs. [137]

In [69], authors presented a novel FPGA-based fine-grained reconfigurable architecture to map several numerical linear algebra kernels and compared with Intel

Xeon Woodcrest processor to report 10-150× speed-up/energy-efficiency improvement. However, no absolute results in time or energy for any particular technology node is reported making it extremely difficult to compare with our proposed approach. A more detailed implementation for our target kernels are reported in [97], where the total performance results include the communication bandwidth with a PC. Considering the overall performance, our implementation is clearly superior by several orders of magnitude, though, the comparison is not accurate as the performance measured for *Layers* is for a stand-alone core without considering complete system integration and communication latency with a host CPU. [139]

6.4.3.5 Scaled aggregate results

To make the comparisons fair, the power and area values have been scaled to 65nm technology node using the following relations from [23,83,100]:

$$NormalizedArea = \frac{Area}{(process/0.065)^2} \tag{6.16}$$

$$NormalizedPower = \frac{Power}{(process/0.065) \times (V_{DD}/1.2)^2} \tag{6.17}$$

The scaled values are noted in Table 6.1 with the † symbol. Even if the scaling is approximate, it gives a good idea of the performance ballpark of *Layers*. *Layers* shows superior values compared to GPU, CPU and other platforms, however, it has 3× worse area and 8× worse power than an ASIC (although, the ASIC can only run one kernel, whereas *Layers* can run multiple).

Table 6.1: Comparison of a few *Layers* design points with similar architectures in the Linear Algebra Domain

Architecture (Application)	Area (kGE or mm^2)	Frequency (MHz)	Power (mW)	GFlops per Watt	GFlops per mm^2	TechLib [References]
2×2 Layers r=1:8:8 (clean)	84kGE (0.12mm^2)	990	17.74	27.95	5.84	65nm
2×2 Layers r=1:4:4 (clean)	84kGE (0.12mm^2)	990	17.74	55.91	11.68	65nm
2×2 Layers r=1:2:2 (clean)	84kGE (0.12mm^2)	990	17.74	111.83	23.37	65nm
3×3 Layers r=1:8:8 (@SGEMM)	219kGE (0.21mm^2)	543	25.42	23.48	2.17	65nm
4×4 Layers_cg r=1:8:8 (@LU)	278kGE (0.35mm^2)	488	44.45	21.94	2.78	65nm
30 LAC cores (@SGEMM)(sim. estim.)	115 (239.93†) mm^2	1400	N/A	30-55	6-11	45nm [129]
4×4 LAC (@LU)(sim. estim.)	2.2 (4.59†) mm^2	1000	N/A	25-30	1-2.6	45nm [126]
REDEFINE with CFU (@SGEMM)	0.16mm^2	416	N/A	N/A	2.54	65nm [112]
REDEFINE with CFU+DOT (@SGEMM)	0.23 (0.12†) mm^2	416	N/A	N/A	12.24	45nm [112]
REDEFINE simple (@C-GR)	N/A	374.1	N/A	N/A	N/A	90nm [111]
Nvidia GTX280 (@SGEMM)	576mm^2	1300	236000 (max)	0.001	0.63	65nm [161]
Nvidia GTX280 (@QR)	576mm^2	1300	236000 (max)	0.001	0.63	65nm [87]
Nvidia 9800GTX (@GEMM)	324mm^2	1670	140000 (max)	0.77	5.2	65nm [161]
Nvidia 8800 Ultra (@CUBLAS)	480 (250†) mm^2	575	175000 (56172†) (max)	1.45	4	90nm [25]
Tesla S1070 (@SGEMM)	610 (851†) mm^2	470	800000 (1.36e7†) (max)	0.73	0.42	55nm [17]
Core2 Quad QX6850 (@SGEMM)	286mm^2	3000	130000 (max)	0.0006	0.28	65nm [161]
TMS320C6678 (@L3BLAS)	N/A	1000	10000 (max)	8	NA	40nm [16]
DSP (@GS-GR)	N/A	1200	9313	8.24	N/A	40nm [60]
ASIC (@QR)	36kGE (0.046mm^2†)	278	48.2 (3.18†)	N/A	N/A	130nm [102]
FPGA with 120PEs (@Lin. solver)	N/A	200	18000	2.61	N/A	65nm [169]
FPGA Systolic array	N/A	139	N/A	N/A	N/A	65nm [164]

Note: values marked with † are scaled to 65nm technology node using the formulae in [23, 83, 100]
V_{DD} assumptions: 3.3V for 130nm, 1.8V for 90nm, 1.0V for 55nm and 0.9V for 45nm

6.4.4 Complexity Evaluation

Layers is a complex architecture, shown to be better than standard CGRA, however this complexity translates also into a lot of options when programming it. Early versions used direct extraction of coding bits for multiplexers and encoding bits for the processing elements to create the bit-stream for the application, which is the established way of configuring CGRAs.

Using the theoretical concepts of functional reconfiguration, the architecture has been transformed to expose elementary functions, and a domain-specific set of language constructs were derived, to target the numerical linear algebra domain. The high-level architectural description and exploration, from which automatic RTL code generation is possible, coupled with generated assembler, linker and simulator helped in fine-tuning the structure. The immediate effect of tuning elementary functions or language elements was visible. Derivation of the final architecture leaves open the option of targeting 3D silicon technologies or FPGAs, as the RTL code can be reproduced at little cost, if the target structure is defined.

After deriving the scheduling of the application kernel, programming of the architecture via functional reconfiguration is easy. Identifying and calling the required functions from assembly is straightforward. Even reconfiguration of language elements can be easily done from assembly.

The scalability of the architecture was greatly enhanced by the functional assembly constructs, since scaling size did not change the core functionality of the elementary functions, their language constructs, not requiring a re-write of the assembly for every modification. Moreover, simple scripting can extend the validity of one target application assembly code, such that it stays valid for any size of the architecture. Just by using the language constructs, several kernels could be seamlessly accommodated in the data path for the targeted domain, e.g. DOT, GEMV, GEMM, LU, GivensRot, etc., however the architecture can be adapted to different application domains by adapting the function set to the respective domain: either change elementary functions to optimize for the target application, or reconfigure the language elements using the existing pool to tune architectural flexibility.

Regarding the Kolmogorov complexity K, for instance, a relative comparison against earlier versions of the architecture from which the program size can be done. Assembly code and programming difficulty have been reduced significantly. The ease of programming and representation of the target algorithm via the language constructs led to reduced programs: e.g. general matrix multiplication (GEMM) requires 5 execution cycles in *L0*, out of which 2 are the hot-spot (mul+acc). The largest of the kernels, triangular vector solve (TRSV) takes 17 execution cycles in *L0*, which includes a lot of exception handling and complex scheduling initiation intervals. For instance, the LU factorization kernel on a 4×4 earlier *Layers* version programmed with configuration bits, took almost 2 months to code, including debugging, with 246 configuration words, giving a complexity $K(LU) = 246\text{words} \times 381\frac{\text{bits}}{\text{word}} = 93726\text{bits}$. Each of these bits had to be derived manually. Programming via functional assembly of 4×4 Layers, took 64 function words (8 *L0* cycles), giving $K(LU) = 64\text{words} \times$

$330\frac{\text{bits}}{\text{word}} = 21120\text{bits}$, which is a 4.43× complexity reduction besides the productivity gain of using high-order functions in a scripted ruby assembly code. Since no direct configuration bits were used, the function calls helped avoid coding errors, reducing debug time. Many potential optimizations such as *meta-functions* (e.g. `forall`, `foreach`, `leftof`, etc.) and *hyperfunctions* were not fully exploited in the LISA implementation due to a limitation of the tools. However, in the RTL version targeted for the 3D silicon implementation, higher-order function support is implemented.

Exploiting functional reconfigurability, the scaled *Layers* architecture from 2×2 to 10×10 is programmed with the same scalable assembly code for each kernel via scripting. Basically once the architectural parameters were entered, the script modified the assembly code to accommodate the architecture. Trying to program and debug this with traditional bit-streams would have been near impossible, especially since few compilers exist, even for CGRAs less complex than *Layers*.

6.5 Conclusions and Summary

In this chapter, a thorough exploration for one application domain has been conducted, via a novel 3D coarse-grained reconfigurable architecture, called *Layers*. This architecture follows the second proposed methodology for exploiting architectural flexibility to match application requirements: tunable flexibility. The key idea of tunable flexibility is for the architecture to have the ability to modify its architectural language according to application language requirements, to produce a very good match, which leads to energy-efficiency and high performance.

The architecture uses a layered approach, separating functionally and physically the four functional classes of memory access, data movement, data processing and control flow processing. By means of a complex pipeline and tunable language via *hyperfunctions*, the architecture can adapt to the application language interface via reconfiguration, yielding a close match.

An deep evaluation is performed using 8 linear algebra kernels, with sufficient differences in scheduling and processing patterns to demonstrate the architecture's adaptability. A performance evaluation and comparison completes this chapter.

Chapter 7

Enhancements for *Layers*

In the previous chapters, the effect of architectural flexibility and high-level exploration and design on energy efficiency was presented and discussed. This chapter mirrors the research achievements of two papers, coupled with a proposal for an integration solution, which enhance the *Layers* architecture:

- Automatic derivation of control flow structures in *LayerQ* – additional flexibility for changing applications, without using static structures, can be achieved [134][1].
- Automatic derivation of an efficient *Layer0* schedule – an important contributor to energy efficiency [59][2].
- Integration into a host System-on-Chip architecture – a proposal on how such an efficient architecture can be used

These enhancements permit re-targeting the architecture to other domains more easily.

Additionally, a study whether CGRAs with application specific processing elements are feasible is also shortly presented, based on a collaboration paper [89]. Here, an entire family of cryptographic algorithms is implemented by using custom processing elements optimized for Addition-Rotation-eXclusiveOR (ARX) operations. The design is based on the high-level design concepts presented in Chapter 4. Similar adaptations are easily applicable also to the *Layers* architecture, which can be advantageous when changing the application domain.

7.1 Flexible Control Flow Via Reconfigurable Structures

In *Layers*, control flow is governed by the algorithmic state-machine within the Q-stage. This is implemented in hardware as static, ASIC-like set of small *q*-operations, specific to each application and split based on the function that is being performed. However, if *Layers* is to be employed in other domains as well, a method for deriving *flexible* control flow structures would be necessary, giving post-silicon flexibility. In the following this research avenue is explored aiming to:

---

[1] Parts of this chapter appear in this publication, reprinted with permission. ©2015, IEEE.
It is my pleasure to acknowledge the contributions of A. Acosta-Aponte to this section, during his M.Sc. Thesis preparation under my supervision.

[2] Parts of this chapter appear in this publication, reprinted with permission. ©2014, IEEE.
It is my pleasure to acknowledge the contributions of A. Fell to this section, during our research collaboration.

- find a methodology by which control flow structures can be generated automatically
- explore some early architectural solutions and compare them with the existing ASIC-like Q-layer implementation, to provide an idea of the potential of reconfigurable control flow structures

7.1.1 Importance of Flexible Control Flow

Generally, efficient manual mapping of the data-centric kernels of applications onto reconfigurable structures yields great results on regular multi-processor structures, such as CGRAs. Due to this regularity, applications that have an irregular execution pattern or include sequential control-flow code, require extra efforts to derive an efficient mapping. Specifically, these irregularities come from introduction of control flow processing into the data flow, such as loop header updates, jumps and address generation. It was clear from Chapter. 3, that a functional separation of application processing tasks could be very efficient with the positive side-effect of clean, controllable hardware implementation. A separation between control flow and data flow could be beneficial. This point is underlined in Chapters 5-6, where such flexibility could be exploited. The control flow structures in the q-stages of these architectures were implemented in a fixed way in hardware, for maximum efficiency. Also, such implementation limits post-silicon flexibility, if the application domain is to be changed.

Therefore, there is a need to explore and analyze design and synthesis of reconfigurable structures for efficient application-specific control-flow processing, aiming to develop a methodology to design reconfigurable control-flow acceleration modules. Such modules can be coupled with any reconfigurable data-flow tailored structure, like CGRAs, off-loading execution of irregular and ill-suited sequential control-flow subroutines, which then enables a clean, regular data-flow centric mapping on the data-side reconfigurable fabric. Such reconfigurable control-flow specific accelerators are a first step towards automating CGRA-based accelerator design and application mapping from high-level descriptions and could perfectly match the *Layers* design philosophy.

In the following exploration, methodology and early experimental results are presented, also published in [134].

7.1.2 Background on Control-Flow Processing in CGRAs

To make CGRAs tackle larger applications, several solutions for the addition of control flow processing have been proposed in the past, discussed in the following [134].

7.1.2.1 Centralized Control

REMARC [118] is composed by a global control unit and a fixed 8×8 array of nano processors. These nano processors have their own RAM and registers. The global controller generates one nano Program Counter (PC) each cycle and is valid for all nano processors. Every nano processor may have different instructions stored at the same nano PC value.

7.1.2.2 Distributed Control

RAW [156,163] uses the definition of recurrent interconnected cells. These cells incorporate a small fully functional RISC processor with its own program counter, program memory, data memory and a configurable switch. It also incorporates distributed SRAM components for memory share between the cells.

7.1.2.3 CPU Coupling

Coupling a processor to the accelerator CGRA is a commonly employed solution, offering flexible options. With a tight coupling of the CGRA to the CPU, the CGRA is introduced inside of the pipeline of the CPU. A loose coupling employs shared registers or a special bus to connect the CPU to the CGRA.

The rASIP [40] is an example of tightly coupling the reconfigurable fabric to an application-specific instruction-set processor model by allowing the CGRA to implement special custom instructions. Not only this exploits extra efficiency of executing complex custom instruction on-demand in the reconfigurable fabric, but also adds post-silicon flexibility to the system: a new application domain can also benefit from efficient execution, since new custom instructions can be accommodated in the reconfigurable side.

Loose coupling is employed in MorphoSys [151], where a Tiny_RISC processor, a fixed 8 × 8 16bit PE array, a data cache and DMA controller are linked together. The RISC processor handles control flow code, while the PE array accelerates data processing. The operation of the PE array follows the Single Instruction Multiple Data (SIMD) principle, which limits the efficiency of acceleration for some applications.

ADRES [110] defines a template which can be molded for application optimization, using two modes of operation. A VLIW mode, where the top row of the PE array is used as a sequential VLIW structure and an array mode, where the entire array is used as a standard CGRA. The VLIW mode allows ADRES to address control flow segments of code, while the data processing is done in the CGRA mode. The modes of ADRES are mutually exclusive, which means it is not possible to run both at the same time. ADRES implements predicated operations for inner loop control flow.

The communication between the RISC processor and the array in FloRA [95] is done through memory centric operations. The rows of PEs in FloRA share the instruction in a pipelined fashion. This means that the first PE of a row fetches an instruction and forwards it through a pipeline register to its neighbor and so on. In an effort to improve control flow performance, Han et al. [76] include Dual Issue Single Execution (DISE) to the PEs, while in [75] it is extended with predication techniques and power efficiency impact is evaluated.

7.1.2.4 Heterogeneous Processing Elements with Control Functionality

The eXtreme Processing Platform [26] has three different PEs. One for memory, one for arithmetic and logic operations and another for functional operations. PE for functional operations implement a full sequential VLIW processor. This integrates more than one VLIW to the CGRA and allows for higher control flexibility.

In [91], different reconfigurable tiles are used, that can be dynamically connected to implement RISC and VLIW instruction set architectures. There is also a mix between fine and coarse grain reconfigurable structures for data path implementation, determined by each application.

Mapping complex ASIC-like finite state machines (FSM) and data path based on a heterogeneous CGRAs is done in [148–150], where special PEs were defined to map FSM functionality with multilevel control hierarchy. The data processing is mapped to PEs supporting data path implementations and memory storage.

A CGRA with capabilities of implementing expression level operators [20] is composed of a control unit and reconfigurable array cell, strictly designed for applications described as software pipelines with modulo scheduling.

7.1.2.5 Dedicated Control Flow Structures

The *Layers* accelerator, as described in Chapter 6, features a design where a functional separation of memory access, data movement, processing and control flow is employed, each class using dedicated hardware structures for execution. This separation enables implementation of dedicated control flow structures, using algorithmic state machines, which are small segregated control-flow pieces, are combined to define complete control-flow parts of application kernels targeting the linear algebra domain. Each piece is implemented as a static ASIC piece, while the data-flow plane is completely reconfigurable.

Due to this clean separation between control and data-flow planes, it is an excellent starting point to conduct a design space exploration for a reconfigurable control-flow accelerator, able to replace the dedicated ASIC structures.

7.1.3 Control Flow Analysis of the Candidate Kernels Running on *Layers*

Taking 9 linear algebra kernels from Chapter 6 as the target application domain, 119 control-flow pieces of the *Q*-stage of *Layers* is analyzed, in order to extract how exactly control flow is processed and what kind of hardware resources, processing element types and interconnects may be needed to construct a reconfigurable control flow structure, presented in the following [134].

7.1.3.1 Control Flow Representation

A graph-based intermediate representation, a control-flow graph (CFG) is employed, starting from the single static assignment forms of the high-level code. To construct this representation, several techniques can be used, as follows. Predication, a technique derived from [18,53] has been used in CGRAs as an efficient way to incorporate control flow capabilities to the PEs [75,110]. Advanced predication using Dual-Issue Single Execution technique [76] executes both outcomes of a control flow branch, at the expense of doubling the instruction code word. Predication transforms the CFG, where branching is needed due to control divergence, into a clean data-flow code allowing a straightforward hardware implementation [103] (Fig. 7.1).

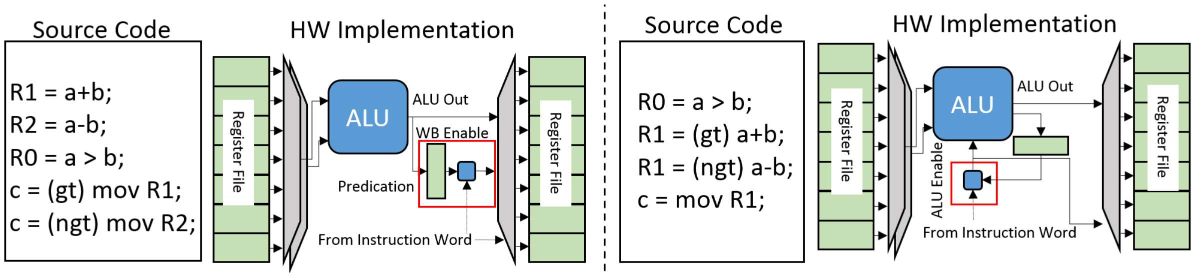


Figure 7.1: Partial (left) and full (right) predication code and corresponding hardware structure. [134]

One essential issue when considering control flow implementation is that the information necessary to decide a control flow divergence must usually happen in *one* execution cycle of the CGRA array block. The most important reason would be efficiency, as the PEs should be always busy with meaningful computation. If the control flow decision processing takes several cycles like in classic CPUs, the entire array needs to wait for the result, as is the case in ADRES [110].

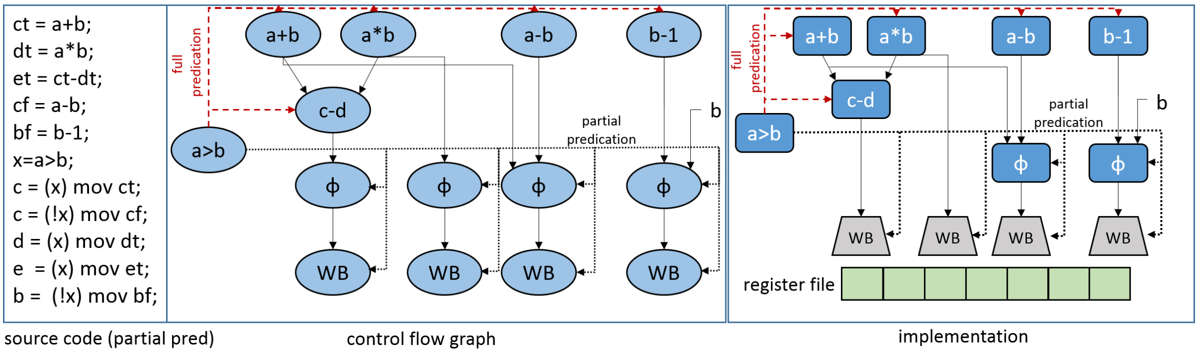


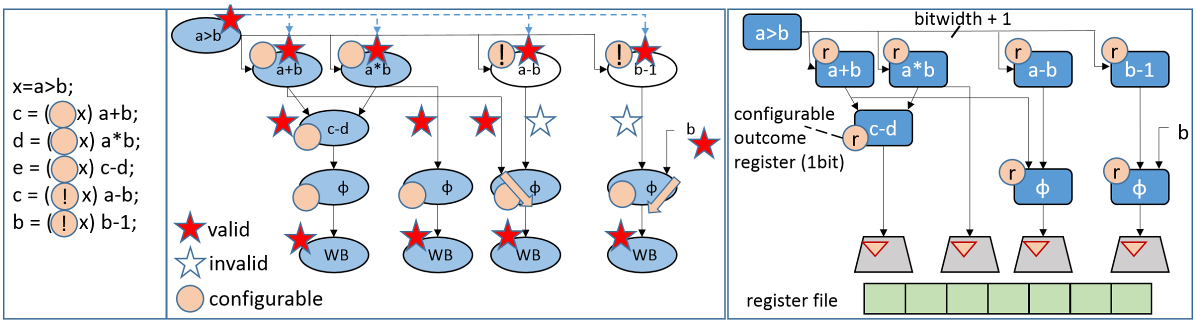
Figure 7.2: Partial (black) and full (red+black) predicated CFG and corresponding hardware structure. [134]

In Fig. 7.2 partial/full predicated source code, CFG and a naïve hardware implementation is presented. Partial predication disables the write-back operation for the invalid branch, while full predication can disable any node if on the invalid branch. For a configurable control flow, however, it is not known when and how a node will be predicated, nor whether or from where a predication line needs to connect, as this is application-dependent information. Moreover, predication results need to be instantly forwarded, if a one-cycle decision is to be made.

7.1.3.2 Full Combinatorial Predication

A new technique is proposed to realize CFGs and its hardware implementation, called *combinatorial full predication*, which allows all the above elements to be configurable. With similarity to Petri nets, this technique incorporates a token flow to distribute predication information, without requiring the high fan-out of full predication, neither the reduced energy efficiency of partial predication. Each conditional statement generates a *token*, which can be compared by child nodes with a pre-defined Boolean value stored in the child node. The token is forwarded towards child nodes, requiring only a parent-child connection. If the local comparison is true, a new *true token* will be generated and forwarded to the new children; a *false token* otherwise.

Configurable flexibility is enabled by the set of configurable Boolean values of each node in the CFG, which may be set to true or false predication expectancy, according to application control flow information. These bits can be reset to other values when the application changes and the CFG nodes are relinked and re-used in the hardware implementation to reflect the application CFG.



source code (full pred) control flow graph (full combinatorial predication) implementation (full combinatorial predication)

Figure 7.3: Full combinatorial predication: a token-flow based predication which allows configurability in hardware. [134]

Fig. 7.3 illustrates this for a full-predicated source code. When a Φ-function needs to be calculated, only the source line which has a true token is forwarded. If both are true, a false token or an error signal is generated (indication of a cyclic graph or other control-flow anomaly unacceptable for a single-cycle combinatorial operation).

A naïve hardware implementation requires an additional token bit-line added to the data lines, a configurable 1-bit register and a 1-bit comparator for every node. The pre-determined values for the registers are application dependent and can be filled at programming time (configuration). Of course, to be able to implement any CFG, the interconnects between the nodes need to be configurable also (details in the next section).

For the case-study domain, all 119 CFGs are generated and analyzed, such that a hint of possible hardware requirements may be deduced. Results show that 76% of the operators were integer arithmetic (`ADD`, `SUB`, `MUL`), 13% comparisons ($>, <, \leqslant, \geqslant, =, \neq$) and 11% logic (`AND`,`NAND`,`NOR`,`OR`,`INV`). Fig. 7.4 shows size distribution, height and width of the considered sub-graphs.

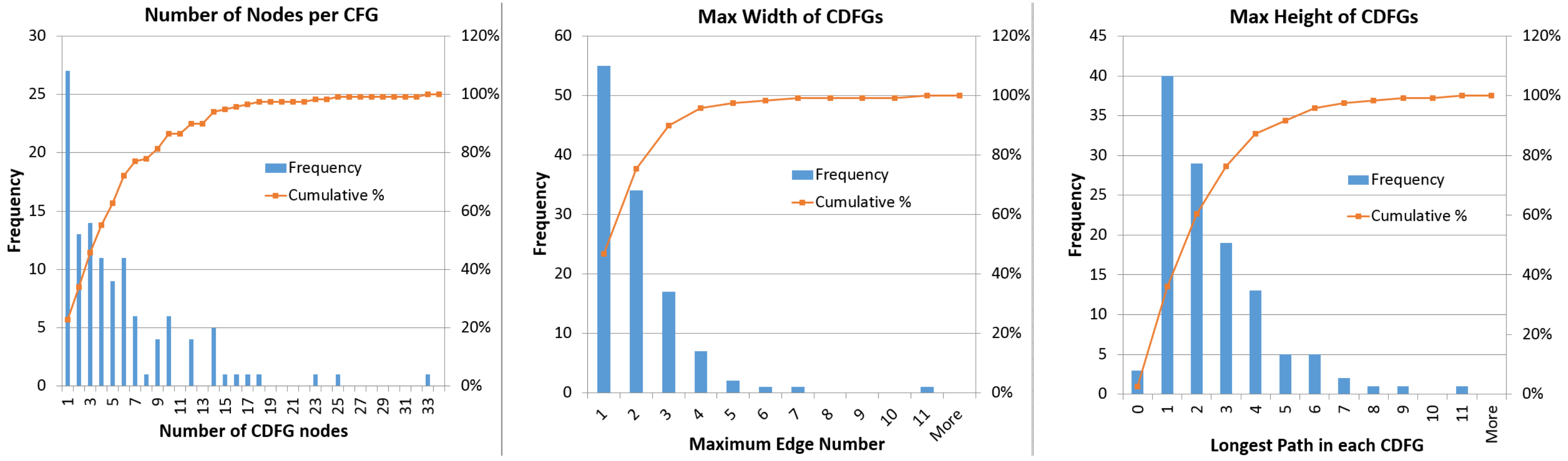


Figure 7.4: 119 CFGs analyzed from 9 linear algebra kernels: GEMM, GEMV, TRSM, TRSV, DOT, GIVENS, LU, etc. [134]

Interesting to note that only a few CFGs are very tall (high critical path), very wide (high parallelism and fan-out) or very large (high number of nodes), hinting that the implementation requires high flexibility while size can be reduced if the largest graphs can be split by trading off the 1-cycle execution constraint. Furthermore, regular node structures with certain operation support suggest a limited control flow architectural language, which can be exploited to generate such structures automatically.

7.1.4 Control Flow with a Homogeneous Array of Functional Units

Based on the CFG analysis, first an architecture based on regularity and modularity is considered, as discussed in the following [134]. Starting from a Functional Units (FU) that can execute all operators encountered and feature a token register, an array is built where via configuration options parent-to-child node connections can be realized between PEs. This allows a high grade of flexibility to accommodate different number of control-flow operations, by adding rows and columns to the array. Every created path needs to start and end in a register file to avoid creating timing loops. The more FUs are chained, the more complex CFGs can be mapped. The interconnect is modeled as Row- and Column Broadcast Lines (RBL, CBL), with multiple word-sized lanes, similar to FPGA bit-level interconnect. It is segmented at every FU access point via a configuration multiplexer, allowing to create shorter or longer point-to-point circuit-switched network, but also easily allows one-to-many broadcasts. To avoid bus congestion, nearest neighbor and nearest diagonal connection on the column is added. Fig. 7.5 illustrates a PE with corresponding input/output lines, which can be tiled to any size.

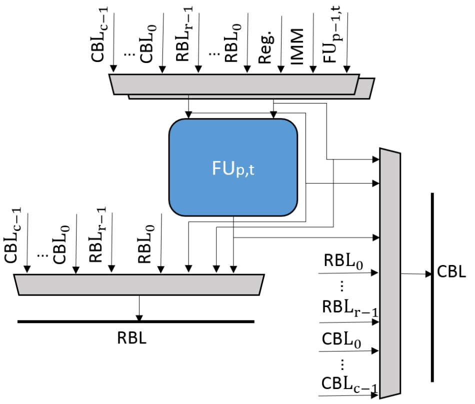
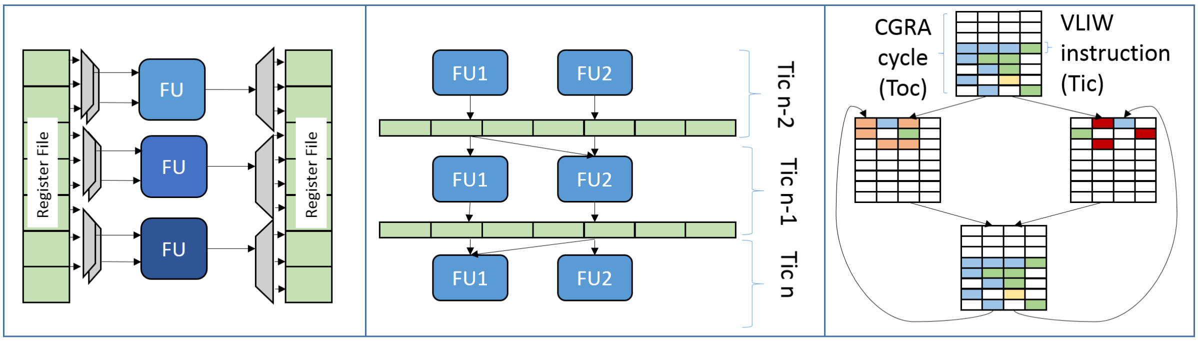


Figure 7.5: A tile-able structure featuring a functional unit with row and column broadcast lines. [134]

To construct the architecture for the 119 CFGs, Alg. 1 is employed. Lines 6 and 7 are not trivial to automate, but a human-guided mapping made this process simple, exploiting the regular structure. The resulting architecture required 11×3 FU array with 2 RBLs and 1 CBL. Top and bottom rows do not have preceding RBL and following CBLs, respectively. The weakness of this approach is 1) large FU replication, 2) long critical path and 3) manual mapping.

Algorithm 1 Homogeneous Array with CBLs and RBLs Architecture Generation [134]

```
Input: CDFG with predication of a Q_Op
Output: Homogeneous $n \times m$ array of FUs, $r$ RBLs and $c$ CBLs Architecture
Select one critical path
Place in a 1-D FU array arrangement of length equal to critical path length
Delete nodes and edges in the selected path of the CDFG
while nodes left without placement do
    Select next longest path
    Place nodes with direct connection to other placed nodes with the current available hardware resources
    Place the rest of nodes with the current available hardware resources
    if previous placement attempts fail then
        if no more FUs available for node placement then
            Add 1xN FUs and corresponding RBLs and CBLs
            continue                                  ▷ jump to start of while loop
        end if
        if not enough CBLs for data movement then
            Add one CBL for each FU column
            continue                                  ▷ jump to start of while loop
        end if
        if not enough RBLs for data movement then
            Add one RBL for each FU row
            continue                                  ▷ jump to start of while loop
        end if
    end if
    Delete nodes and edges in the selected path from CDFG
end while
```



3-FU VLIW structure

splitting the CFG

execution of split CFG in a multiple of *tics* with a 4-FU VLIW

Figure 7.6: A VLIW architecture with different FUs can execute a split CFG efficiently and transparently due to higher execution rate. [134]

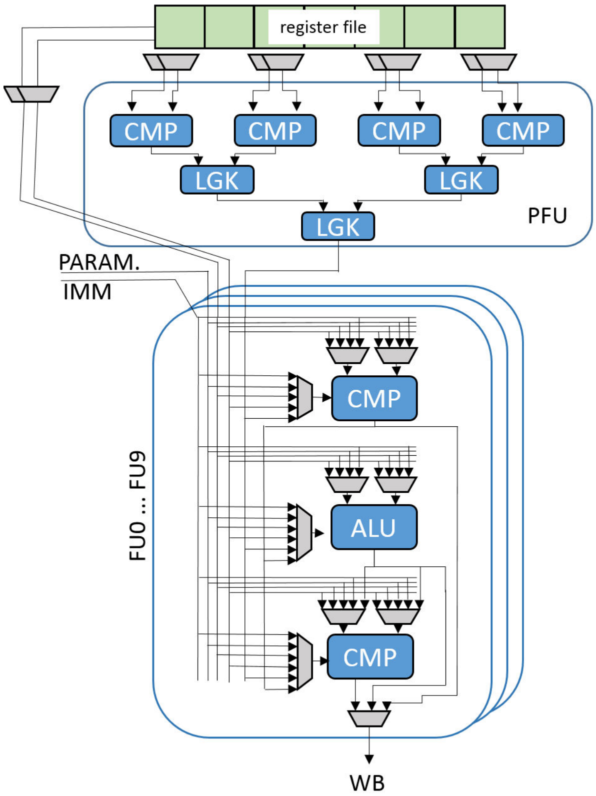


Figure 7.7: The resulting control-flow specific VLIW architecture. Functional units are tailored for configurable cascaded execution. [134]

7.1.5 A VLIW-like Control Flow Processor

If the constraint of single cycle execution of all CFGs is lifted, a VLIW-like architectural solution which takes multiple cycles for processing can be considered. This, of course, implies that this module has to run at higher clock frequency (*tics*) than the Layer 0 array (*tocs*), such that data plane scheduling remains intact.

By splitting a longer CFG into several *tics*, it is mapped to the VLIW architecture and executed transparently w.r.t. the data plane execution of the array, which is bound to *tocs*, shown in Fig. 7.6.

It is important to note that a split CFG may not execute any intra-*toc* jumps, as all jumps have to be synchronized with the CGRA execution. For the mapping, ASAP/ALAP scheduling can be employed such that the CFG nodes can be distributed into a FU×clk_factor grid, taking care to save any intermediate results in the register file, according to token status (values with invalid tokens need not be stored).

The VLIW architecture can be seen in Fig.7.7. The architecture deviates from the standard VLIW (Fig. 7.6) in that there is a Pre-Functional Unit (PFU) and that each FU contains two comparators (COMP) and an ALU in a combinational chain. The PFU can execute a varied range of control divergence processing, which can then use full combinatorial predication for FUs. All FUs have the same input source ability and all can write back to the register file one value. The vertical multiplexers represent the possible input sources for the predicate value.

7.1.6 Graph-Theoretic Approach to Control Flow Architectural Derivation

Having explored some classical avenues for control flow structures, a methodology for automatic optimization and generation of control flow architecture is presented in the following [134].

7.1.6.1 Theoretical Background

Starting from the requirement that all 119 CFGs have to be supported in one architecture, a direct optimization approach is attempted. CFGs are, property-wise, graphs, therefore a graph-theory based solution should be possible. The concepts of Maximum Common Subgraph [98] and Minimum Common Supergraph [36] can be applied to generate the common architecture. This involves two steps (Fig. 7.8):

- determining the graph that encloses most common subgraphs
- adding the remaining parts with a minimum number of additions such that the resulting supergraph contains all members

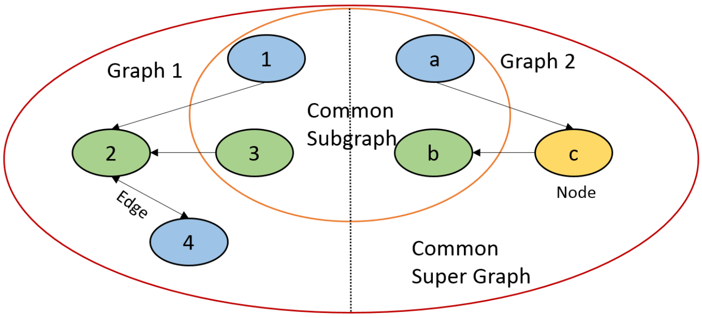


Figure 7.8: Illustration of relevant graph theory definitions. The subgraph isomorphism is $f : 1 \rightarrow a$ and $3 \rightarrow b$. [134]

Some definitions from graph theory will aid in understanding the concept in more detail:

Definition 7.1.1. A labeled graph is a 4-tuple $G = (V, E, \alpha, \beta)$, where V is a set of vertices or nodes, $E \subseteq V \times V$ is a set of edges, $\alpha : V \rightarrow L_V$, $\beta : E \rightarrow L_E$ are functions assigning each node and edge a set of labels, L_V and L_E are finite sets of node and edge labels. □

Definition 7.1.2. Let $G_1 = (V_1, E_1, \alpha_1, \beta_1)$ and $G_2 = (V_2, E_2, \alpha_2, \beta_2)$ be graphs. G_2 is said to be a subgraph of G_1 (notation $G_2 \subseteq G_1$), if $V_2 \subseteq V_1$; $\alpha_2(v) = \alpha_1(v) | \forall v \in V_2$; $E_2 \subseteq E_1 \cap V_2 \times V_2$; $\beta_2(e) = \beta_1(e) | \forall e \in E_2$. □

Definition 7.1.3. Let G_1 and G_2 be graphs, furthermore let $G_2 \subseteq G_1$. Then G_1 is a supergraph of G_2. ☐

Definition 7.1.4. Let G_1 and G_2 be graphs. A graph isomorphism between G_1 and G_2 is a bijection $f : V_1 \to V_2$ (between the vertices of G_1 and G_2), such that: $\alpha_1(v) = \alpha_2(f(v)) | \forall v \in V_1$; any two vertices $u, v \in V_1$ are adjacent in $G_1 \iff f(u)$ and $f(v)$ are adjacent in V_2 and $\beta_1((u,v)) = \beta_2((f(u), f(v)))$ If such a bijection exists, G_1 and G_2 are isomorphic (notation $G_1 \simeq G_2$). ☐

Definition 7.1.5. Let G_1, G_2 and G_3 be graphs, furthermore let $G_3 \simeq G_2$ and $G_2 \subseteq G_1$. In this case, f is called a subgraph isomorphism from G_3 to G_1 and G_3 is said to be subgraph isomorphic to G_1. ☐

Definition 7.1.6. Let G_1 and G_2 and g be graphs. It is said that g is a common subgraph of G_1 and G_2 if there exists a subgraph isomorphism from g to G_1 and from g to G_2. ☐

Definition 7.1.7. Let g fulfill Def. 7.1.6. g is said to be the maximum common subgraph if there exists no other common subgraph with more nodes than g (notation MaxComSub(G_1,G_2)). ☐

Definition 7.1.8. Let G_1 and G_2 and G be graphs. It is said that G is a common supergraph of G_1 and G_2 if there exists a subgraph isomorphism from G_1 to G and from G_2 to G. ☐

Definition 7.1.9. Let G fulfill Def. 7.1.8. G is said to be the minimum common supergraph if there exists no other common supergraph with fewer nodes than G (notation MinComSup(G_1,G_2)). ☐

The first known algorithm for calculating the MaxComSub of two graphs was proposed by Levi [98]. In that work there is a subtle, yet important, difference to Def. 7.1.2. In [98], the definition of the subset of edges is defined as $E_2 = E_1 \cap V_2 \times V_2$. As stated in [107], this is a more restrictive definition of subgraphs. In Raymond et al. [145], the term Maximum Common Induced Subgraph (MCIS) is used when referring to those resulting from the definition of subgraph in [98]. In the same work, the term Maximum Common Edge Subgraph (MCES) is used with the definition of subgraph in [107]. The definition from [107] is adopted here. Replacement of the MCIS by the MCES has no effect on the validity on Def. 7.1.9, as it will become clear in Theorem 7.1.13. The following definitions are taken from [36] and adapted to the MCES definition, which conclude with the relationship between MinComSup and MaxComSub.

Definition 7.1.10. Let $G_1 = (V_1, E_1, \alpha_1, \beta_1)$ and $G_2 = (V_2, E_2, \alpha_2, \beta_2)$ be graphs, with $G_1 \subseteq G_2$. The difference of G_2 and G_1 (notation $G_2 - G_1$), is a graph $G = (V, E, \alpha, \beta)$; $V = V_2 - V_1$ (the set of nodes V_2 without those in common with V_1); $E \subseteq E_2 \cap (V \times V)$; $\alpha(v) = \alpha_2(v)$ for any $v \in V$; $\beta(e) = \beta_2(e)$ for any $e \in E$. ☐

The difference between two graphs can be understood as the resulting graph after all common nodes and edges have been removed.

Algorithm 2 Modified McGregor MaxComSub [134]

```
Input: Two CDFGs $G_1$, $G_2$ with their corresponding node label lists
Output: Optimal MaxComSub of $G_1$ and $G_2$ as node pairing list and edge pairing list
Determine ASAP and ALAP labels of $G_1$ and $G_2$ and insert to the node label list
Adjust the ASAP and ALAP labels of the graph with shortest critical path
Sort topologically the nodes in $G_1$ and $G_2$ based on ASAP
Set MARCS to contain all 1's, $arcsleft = |V_1|$ and $bestarcsleft = 0$
$i = 1$ and mark all nodes of $G_2$ as untried for node 1 of $G_1$
while $i \neq 0$ do
    if there are any untried nodes in $G_2$ to which node $i$ of $G_1$ may correspond then
        $x_i$ = one of these nodes and mark node $x_i$ for node $i$
        Refine MARCS based on the tentative correspondence of node $i$ and compute $arcsleft$
        if $(arcsleft > bestarcsleft)$ OR $MARCS = \emptyset$ then
            if $i > |V_1|$ then
                Store $x_1, x_2, \ldots, x_{|V_1|}$, MARCS, $bestarcsleft = arcsleft$
            else
                Recompute ASAP and ALAP labels based on tentative correspondence
                Store a copy of MARCS, $arcsleft$ and the new node label list in the workspace associated with node $i$
                $i = i + 1$
                Mark all nodes of $G_2$ as untried for node $i$
            end if
        end if
    else
        $i = i - 1$
        restore MARCS, $arcsleft$ and the node label list from the workspace associated with node $i$
    end if
end while
Transform the resulting MARCS into an edge pairing list
```

Definition 7.1.11. Let $G_1 = (V_1, E_1, \alpha_1, \beta_1)$ and $G_2 = (V_2, E_2, \alpha_2, \beta_2)$ be graphs, with $G_1 \subseteq G_2$. The embedding of G_1 in G_2 (notation emb(G_1,G_2)), is defined as $emb(G_1, G_2) = E_2 \cup [V_1 \times (V_2 - V1)) \cap ((V_2 - V1) \times V_1)]$. In other words, the edges which connect G_2 and $G_2 - G_1$. □

Definition 7.1.12. Let $G_1 = (V_1, E_1, \alpha_1, \beta_1)$ and $G_2 = (V_2, E_2, \alpha_2, \beta_2)$ be graphs, with $V_1 \cap V_2 = \emptyset$. Furthermore let $E_3 \subseteq (V_1 \times V_2) \cap (V_2 \times V_1)$ be a set of edges with a labeling function $\beta_3 : E_3 \rightarrow L_E$. The union of G_1 and G_2 including E_3 (notation $G_1 \cup_{E_3} G_2$), is a graph $G = (V, E, \alpha, \beta)$, where $V = V_2 \cup V_1$; $E = E_1 \cup E_2 \cup E_3$ and

$$\alpha(v) = \begin{cases} \alpha_1(v) & \text{if } v \in V_1 \\ \alpha_2(v) & \text{if } v \in V_2 \end{cases} \quad \beta(e) = \begin{cases} \beta_1(e) & \text{if } e \in E_1 \\ \beta_2(e) & \text{if } e \in E_2 \\ \beta_3(e) & \text{if } e \in E_3 \end{cases}$$

□

Using the above definitions, Bunke et al. [36] postulates the following theorem.

Theorem 7.1.13. *Let G_1 and G_2 be graphs. Then*

$$\begin{aligned} MinComSup(G_1, G_2) &= MaxComSub(G_1, G_2) \cup_{E_1} \\ &\quad (G_1 - MaxComSub(G_1, G_2)) \cup_{E_2} \\ &\quad (G_2 - MaxComSub(G_1, G_2)) \end{aligned}$$

where $E_1 = emb(MaxComSub(G_1, G_2), G_1)$ and $E_2 = emb(MaxComSub(G_1, G_2), G_2)$.

The MinComSup is defined as the MaxComSub of two graphs, with the addition of those nodes and edges of each graph which are not part of their MaxComSub.

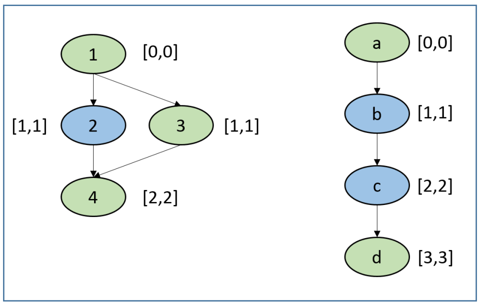
7.1.6.2 Architectural Derivation

If all domain CFGs are merged into one MinComSup, the resulting graph can map any CFG by construction. Some important points are still needed in order to realize this: CFGs are directed graphs, therefore all edges are duplicated into a directed pair; there is no direct algorithm to compute MinComSup; graphs need to by acyclic such that single-cycle requirement is respected; an elongation of the CFG critical path should be avoided. The first problem is trivial. For the second problem, a new algorithm needs to be constructed. The MCES algorithm [107] is used to generate the MaxComSub, but first a topological sort and a depth-first traversal have to be conducted on the graph to eliminate cycles and have a measure to compute critical path length. Using As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) scheduling, the nodes are labeled to be able to differentiate common edges and nodes of the resulting graphs and retain construction information used later in CFG execution in the hardware. Additionally, scheduling mobility is exploited to determine node/edge traversal order in the graph as an additional constraint (Fig. 7.9). In the common use, mobility allows a node to be moved between its ASAP and ALAP time slots without impacting the length of the scheduling.

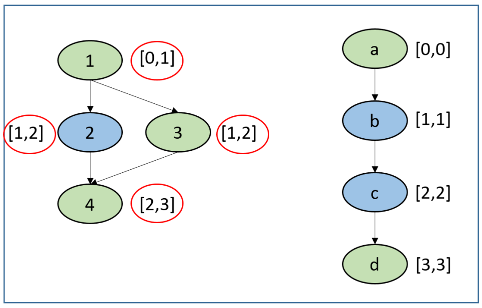
Again, in combinational circuits, this definition requires another interpretation.

Definition 7.1.14. $sched\_mobility(v) = ALAP(v) - ASAP(v), \forall v \in V$ the set of nodes. □

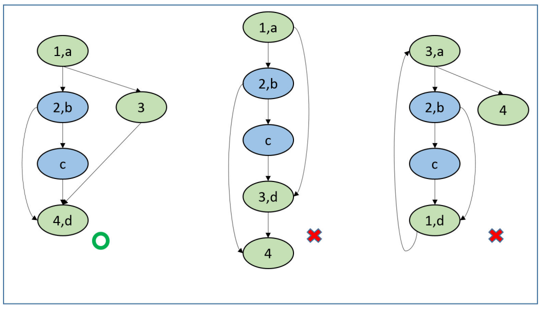
Definition 7.1.15. $comb\_mobility(v) = [ASAP(v), ALAP(v)], \forall v \in V$ the set of nodes. □



(a) Two CDFGs with their ASAP ALAP information



(b) Adjustment of the ASAP ALAP information



(c) Three resulting MinComSups. Only the leftmost respects Def. 7.1.16

Figure 7.9: Illustration showing the validity of Def. 7.1.16 in terms of not elongating the longest path and not creating cycles. [134]

Definition 7.1.16. Two nodes v, w with $v \in V_1$ and $w \in V_2$ and $V_1 \cap V_2 = \emptyset$ are pairable with respect to their mobility, without extending the longest path or creating a cycle in the resulting MinComSup, if $comb\_mobility(v) \cap comb\_mobility(w) \neq \emptyset$. $\square$

Using the considerations above and the optimal McGregor algorithm [107], the MaxComSub is constructed using Alg. 2, with the specification that MARCS is a $E_1 \times E_2$ array data structure which is refined by iteratively zeroing edge correspondences, based on the tentative pairing of nodes (line 9, the backtrack condition – suboptimal solutions *arcsleft* $<$ *bestarcsleft* are discarded). The variable *arcsleft* represents how many of the total edges in G_1 have a nonzero row in MARCS. Line 14 avoids creation of cycles. It is important to add an *empty_node* to the G_2 list, to enable

Algorithm 3 MinComSup Construction [134]

```
Input: Two CDFGs $G_1$, $G_2$ with their corresponding node label lists and the node and
    edge pairing lists of the MaxComSub
Output: CDFG representing the MinComSup
Create an empty CDFG $G_m cs$
for all nodes $v_1$ in $G_1$ do
    Add a node $v_{mcs}$ to $G_{mcs}$ with the same node label list as $v_1$
end for
for all nodes $v_2$ in $G_2$ do
    if node $v_2$ is paired with a node from $G_1$ in the node pairing list then
        Add a node $v_{mcs}$ to $G_{mcs}$ with the same node label list as $v_2$
    end if
end for
for all edges $e_1$ in $G_1$ do
    Add an edge $e_{mcs}$ to $G_{mcs}$ with the same adjacency relation from $G_1$
end for
for all edges $e_2$ in $G_2$ do
    if edge $e_2$ is paired with an edge from $G_1$ in the edge pairing list then
        Add an edge $e_{mcs}$ to $G_{mcs}$ with the same adjacency relation from $G_2$
    end if
end for
```

Algorithm 4 MinComSup Architecture Design Top-Level [134]

```
Input: File containing the CDFG of each graph to be implemented
Output: One CDFG representing the MinComSup of the entire graph space in DOT
    file format
read all CDFGs from the input file and store them in a local list of graphs
sort the local list of graphs in ascending number of nodes
while number of graphs in list $> 1$ do
    pop the two graphs at the front of the list
    find the MaxComSub of the two graphs
    construct the MinComSub given the MaxComSub and both graphs
    add the resulting graph in its sorted place in the list
end while
output the remaining graph
```

creating MaxComSubs excluding certain nodes if better results can be attained; these will be merged later with MinComSup.

The results are then fed into the MinComSup, described in Alg. 3, which basically adds missing nodes and edges, implementing Eq. 7.1.

Finally Alg. 4 is employed to generate the architecture graph, which is straightforward to implement in high-level code or RTL code. Using this methodology, processing nodes capable of executing all operations result in a smaller graph, using

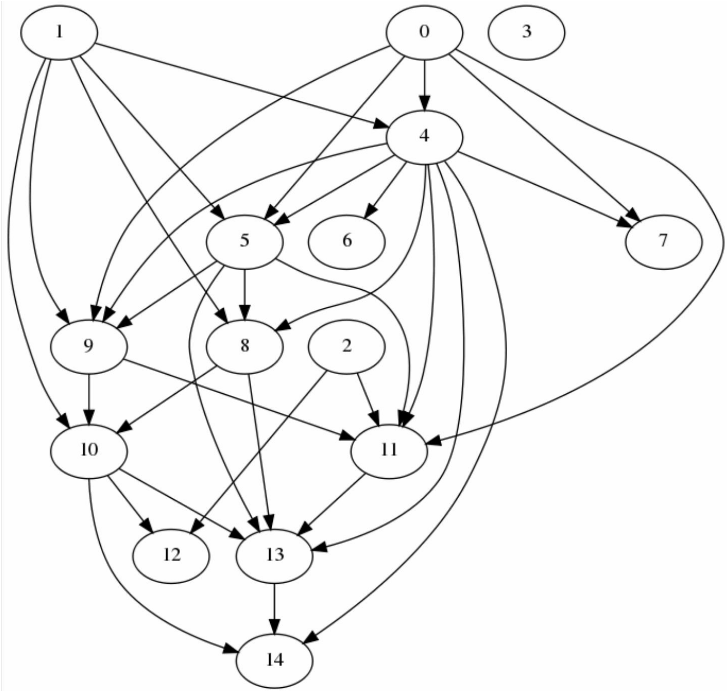


Figure 7.10: A representation of the FUs' interconnect structure of the Unlabeled MinComSup structure. (All nodes are full processing elements capable of processing all operations defined in Section 7.1.3.) [134]

unlabeled nodes (Fig. 7.10), whereas labeling the nodes as having specific processing capability yields more refined results (Fig. 7.11).

7.1.6.3 Limitation

Although theoretically sound, execution of the proposed algorithm over entire CFG space could not be completed. Due to the optimal nature of the MCES algorithm, the larger single CFGs are, the more time it takes to search the design space when including a small CFG, as more solutions and greater mobility increases number of possibilities exponentially. Therefore, the 3 largest CFGs out of the 119 could not be added to the supergraph within a reasonable (several hours) execution time of the C++ implementation, however creation of either an ASIC version of the largest graphs or creating a separate supergraph of the 3 largest ones as a separate module solved the issue.

7.1.7 Evaluation

Also in this part, the high-level architecture description language LISA was used for all architectures, part of the Synopsys Processor Designer tool-chain. HDL generation is done with automatic timing optimizations. For power estimation, RTL simulation is done with VCS_MX version I-2014.03-SP1-1 generating SAIF files which are then fed to the synthesis tool. Synthesis is done with Synopsys DC version J-2014.09-SP1

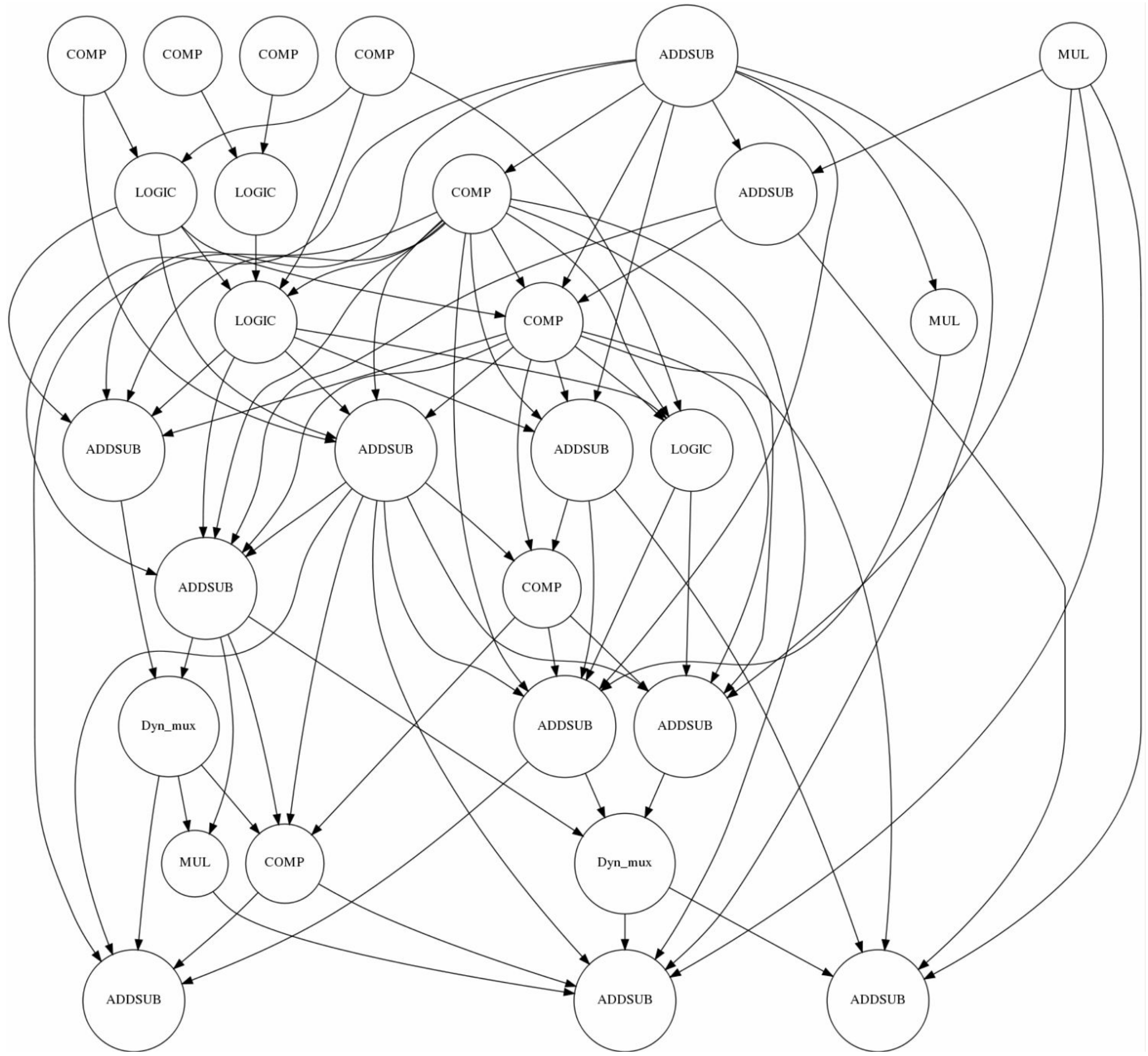


Figure 7.11: A representation of the FUs' interconnect structure of the Labeled MinComSup structure. Fine grained definition of processing element capabilities. [134]

65nm technology library. The proposed solutions are compared versus a cluster of ASIC implementations of each CFG.

7.1.7.1 Area Comparison

As it can be seen from Fig. 7.12, the homogeneous array reaches the largest area value. It is almost three times larger than all other methodologies proposed in this work. This architecture generated 33 FUs with the functionality to implement all operations in the CFG space. Especially, this meant it generated 33 integer multipliers, which were not needed or used in parallel. Added to this, the architecture was also the slowest achieving only 40MHz which is a 73% decrease with respect to the fastest architecture proposed. The Unlabeled MinComSup, if it had been used to process the entire CFG

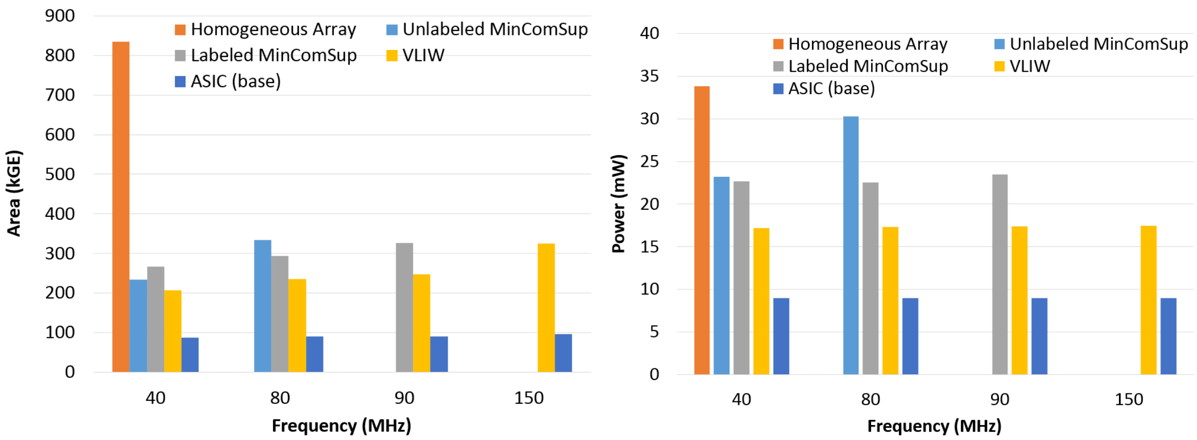


Figure 7.12: Area and power comparison between proposed methodologies and the base. Values are only reported until the highest common frequency. [134]

space, would have generated at least 33 multipliers. This would have made its area requirements comparable to the Homogeneous Array. Because the Labeled MinComSup generates FUs with varied and custom functionality, it derived an architecture with more FUs than the Unlabeled MinComSup version, however the area value is better due to a more fine-grained optimization. The VLIW variant is the most area efficient implementation of the three. Lifting some of the CFG distribution limitations would yield more chances of area reduction and frequency increase, bringing it closer to the ASIC solution, which uses less area.

7.1.7.2 Power Dissipation Comparison

The results for the power dissipation comparison for the GEneral Matrix-Matrix multiplication are shown in Fig. 7.12. Although it was easy to design and program, Fig. 7.12 shows that the homogeneous array is the most power inefficient architecture. The MinComSup variants show small power differences between them at lower frequencies. At higher frequencies, the power dissipation of the Unlabeled version increases considerably compared to its value at lower frequencies. The power dissipation of the Labeled version remains stable throughout the frequencies. The VLIW is the most power efficient variant reaching power reduction of at most 50% with respect to the Homogeneous Array and 26% with respect to the most power efficient MinComSup variant at their highest frequencies. However, this reconfigurable solution is still much slower than the ASIC solution, which also achieves 700MHz with a lower power consumption.

7.1.7.3 Other Possible Avenues

As the three proposed hardware solutions are domain specific, based on pre-defined constraints and requirements, it would be interesting to see what other kind of control-flow processors could be explored. For instance, starting from the full combinatorial predication CFG, a node can be designed that can handle more complex token forwarding (not limited to 1-bit) coupled with the exploration of control-flow specific interconnect topologies. Even for the proposed solutions, optimizations are possible, such as using MinComSup to design the VLIW FUs, etc.

7.1.8 Conclusions

This sub-chapter explores the concept of reconfigurable control flow, based on the published work from [134]. A new token-based predication method is proposed, based on an in-depth analysis of control flow processing from the linear algebra domain. Three avenues of architectural design have been explored and compared, each having the flexibility of supporting control flow in a reconfigurable manner. Although the architectures derived here exhibit opportunities for further optimization, it is shown that reconfigurable control flow could be a viable option to outsource control-flow operations to dedicated structures in CGRA-like accelerators, which require clean and regular mapping of data-plane processing in order to be efficient.

7.2 Automated Mapping and Scheduling with Force-Directed Heuristics

While a very efficient schedule for the execution layer can be deduced manually, it is a slow, error-prone process. In an effort towards automation of this process, automated mapping and scheduling of *Layer0* was attempted. A clean and efficient schedule for the processing elements, not only increases efficiency but also shortens the complexity of mapping the application. Transforming an input algorithm in the form of a Data Flow Graph (DFG) into a CGRA schedule and mapping configuration is, however, very challenging. The necessity to consider architectural details such as memory bandwidth requirements, communication patterns, pipelining and heterogeneity to optimally extract maximum performance is paramount for an efficient mapping.

In the paper on which this section is based on [59], an algorithm is proposed that employs Force-Directed Scheduling concepts to solve such scheduling and resource minimization problems. The heuristic extensions proposed are flexible enough for generic heterogeneous CGRAs as well, allowing to estimate the execution time of an algorithm with different configurations, while maximizing the utilization of available hardware. The experiments, compare also given CGRA configurations introduced by state-of-the-art mapping algorithms such as EPIMap, achieving optimal resource utilization by our schedule with a reduced overall DFG execution time by 39% on average.

7.2.1 Mapping Concept

In order to automate mapping an application to *Layer0* or any generic CGRA, basically the equivalence of the application needs to be mirrored in terms of the hardware execution units, as described and discussed in detail in Chapter 3. This means that matching options between the two language interfaces – the application on one side, and the architecture on the other side – have to be explored and the best-matching solution extracted.

Describing the architectural and application language interface can be done under the form of a Control/Data Flow Graph (CDFG), which is an efficient representation to build automated tools on. For instance, when translating the target application into an intermediate representation of a DFG, vertices represent the operations which need to be mapped onto the PEs, while edges represent the dependencies and data movement among the operations which are translated into communication links of the interconnect. On a CDFG, graph-theoretic optimization algorithms and tools can be employed to find the match. In essence, it is a graph-to-graph matching problem, where the target application DFG is to be converted *automatically* to the graph model of the CGRA fabric, for each time index, as efficiently as possible. Here it is to be noted that this is a known NP-complete problem [64].

An efficient graph conversion (i.e. a good match) in this context means:

- Reduction of execution time of the algorithm as much as possible by placing vertices in such a way that dependencies are preserved, communication overhead is avoided and hardware utilization is maximized.
- Minimizing the initiation interval of the algorithmic kernel (typically loops), such that the next iteration can start as early as possible considering hardware resources and intra-loop dependencies.
- A valid mapping (refer to definition 7.2.2) can be generated in polynomial time.

This represents the optimization of the language interface from the application side to a quasi-fixed language of the architectural side. As described in Chapter 3 a similar optimization can be attempted to optimize the language from the architectural side as well, using a meet-in-the-middle strategy; automation of this attempt is strongly suggested but not yet explored. Naturally, a human component can attempt to optimize both sides, as shown in the previous chapters, but here the possibility of automation of the process is explored, limited to a single side.

To draw the framework of the automation problem, the following points are fixed on the architectural side, to provide an optimization target for the application side:

Definition 7.2.1. A *fabric* is a graph in which each vertex represents one PE and the edges all possible physical connections among the PEs that can be established by the interconnect including time delay (in the form of local storage), i.e. the set of all possible architectural language calls.

In a *time extended fabric* the fabric is replicated for each time index t. One time index is a time unit representing the greatest common divisor of required clock cycles

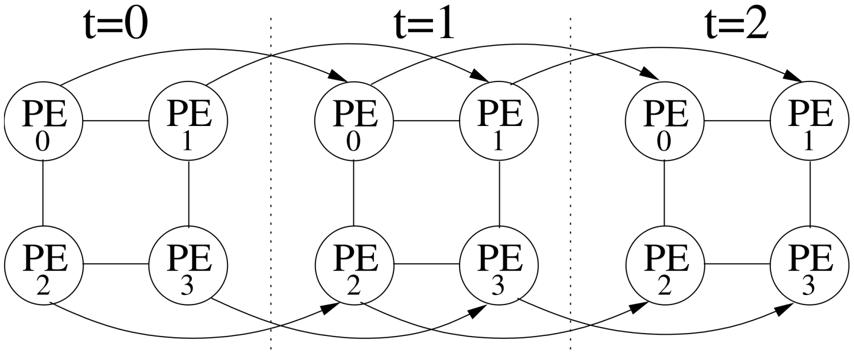


Figure 7.13: An example of a time extended fabric consisting of 4 PEs, connected to an NoC router mounting a mesh topology. This allows capturing the architectural language in a fashion that can be employed by automated graph-based optimization tools. [59]

for all operations supported by the fabric. Further unidirectional edges are introduced connecting the same PE across the time indexes t_i and t_{i+1} (refer to the example given in Fig. 7.13). When edges hop across one or more time indexes, it is a *delayed* interconnect, possibly requiring local storage.

□

Definition 7.2.2. A *valid mapping* is a successful graph conversion from a DFG into an acyclic graph that represents the time extended fabric (refer to definition 7.2.1). Each vertex of the DFG is assigned to one or multiple PEs allowing an instruction to be stretched over several time indexes and the dependencies among the vertices can be established by the interconnect of the fabric. Further vertices which depend on results produced by other vertices scheduled at time index t_1, have to be placed at least at $t_2 > t_1$. □

Basically, the language of the architecture is fixed to a graph representing the time-extended version of the language elements that may be called in a time cycle. Targeting this graph, optimizations of the application-side language are conducted.

Even if the problem is reduced to a single-side matching of the graphs, the complexity of such a graph matching is in NP-complete. Heuristics are applied to derive valid, but sub-optimal mappings. The heuristic proposed method does not only consider a minimum execution time of the algorithm, but also keeps track of architectural features, such as memory constraints and inter-PE communication, using concepts from Force Directed Scheduling, proposed in [124]. The method has been further extended to support arbitrary fabric sizes and heterogeneous fabrics in which differently specialized PEs coexist, e.g. providing hardware support for floating point division, square root or FFT to name a few. In addition, pipelined and non-pipelined execution paradigms and operations requiring multiple cycles, are supported. This covers a rather large set of possible architectural languages.

7.2.2 State of the Art

7.2.2.1 EPIMap

A state-of-the-art scheduling and mapping algorithm is EPIMap [73], in which the instructions of the innermost loop of the algorithm is mapped in such as way that the initiation interval of the loop is minimized allowing to start the next iteration as early as possible on a time extended fabric. In other words, the instructions of the inner loop are spatially and temporally assigned to PEs so that the DFG of an iteration can be interleaved with the DFG of the preceding one. By considering only the instructions of the inner loop, the number of DFG vertices to be mapped, are reduced drastically and independent replicas of the DFG are interleaved.

To generate better mappings, EPIMap employs recomputation and routing. Recomputation is a method in which the same instruction is executed multiple times to increase the set of destination PEs for a result. In the original time extended fabric of EPIMap, the PEs can only communicate to their direct neighbors in a mesh and a result cannot be sent to PEs with a distance of more than one hop. Thus, if the result of an instruction i is consumed by more PEs than the fabric is able to provide, then i is duplicated to increase the number of possible destination PEs.

Routing can be considered as a NOP operand for a PE to delay the consumption of the result. It artificially induces gaps into the time extended fabric, rendering PEs idle to reduce the density of the DFG, so that the next iteration can be scheduled at an earlier time index. Induced NOP operands and recomputation increases the overall execution time of the inner loop which can be partially compensated by the better interleaving of the loop iterations. However, if the inner loop body contains a sufficiently large amount of operations or if the algorithm does not have an inner loop at all, EPIMap generates a schedule, which takes longer to execute due to the under-utilization of available hardware and replication of operations.

7.2.2.2 SPR

Another mapping algorithm is called SPR (Schedule, Place, Route) [61], which can be adapted to the specifications of the architecture. SPR is a collection of tools comprising of a scheduler [143], ordering the operations based on their dependencies, a placer [90] allotting PEs to operations, and a routing mechanism [108] for the data movement. Instead of encapsulating each of these algorithms, the placer influences the scheduler in case the data is transmitted to a distant destination and hence the depending operation cannot be scheduled immediately due to the communication delay.

7.2.2.3 eFPGA Mapping

In [44] an architecture is introduced in which the computation capability is provided by embedded FPGAs (eFPGAs) [121]. A high level modeling language is used to describe the functionality and dependencies among the operations of the DFG. With this information an affinity graph is created allowing to place operations with high bandwidth requirements, in close proximity to each other. Once all required functionality for a particular DFG is known and the affinity graph is calculated, the DFG is placed onto the hardware and the eFPGAs are programmed accordingly. A reprogramming of eFPGAs during the execution of the algorithm is not feasible, since it is a costly and time consuming process. This is in contrast to CGRAs in which one PE can be reused to perform different instructions on other data. Hence not only the PE needs to be identified to perform a particular instruction, but also a time index is required, when to execute this instruction.

7.2.2.4 rASIP Mapping

An interesting approach is introduced in [38] in which the authors not only consider the DFG of the algorithm, but also adapt the CGRA for this particular algorithm using a commercial high level description framework called LISA [42]. While this method ensures the best possible execution time for the algorithm, it also reduces the grade of flexibility that is inherent to CGRAs. Hence this tool might become only interesting, if a design choice between an CGRA and ASIC (Application Specific Integrated Circuit) needs to be taken.

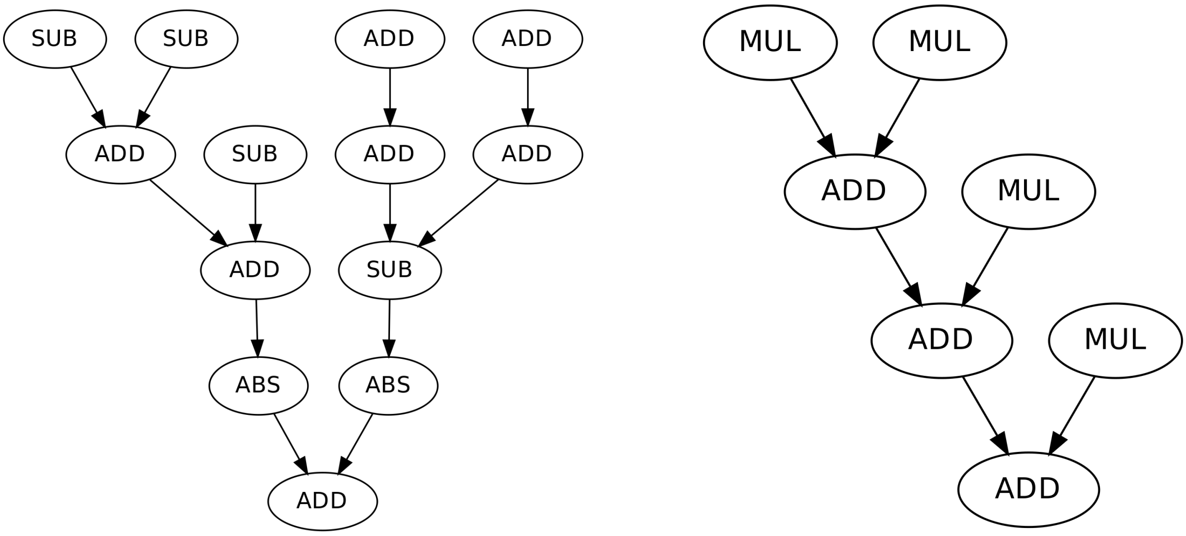
7.2.2.5 Other Considerations

A common problem of current scheduling and mapping algorithms is the lack of support for operations that require multiple clock cycles. For instance in today's scenarios it is unlikely that a 32 bit multiplier returns the result within the same clock cycle [154]. Complex operations consist of pipelines of different depths to achieve reasonable clock frequencies. This implies that language elements of the architecture can span several cycles. These operation need to be handled as atomic operations by the scheduler, but it can allow superposition of different pipeline stages.

In addition, modification of the architectural language, can add to complexity. Heterogeneous fabrics in which the PEs can execute only a subset of all required operations, are not considered in the mapping tools. Nevertheless, specialized hardware accelerators play an important role in algorithm acceleration and the regular structures of CGRAs certainly invite hardware developers to perform experiments by replacing a few or all PEs, thus creating heterogeneous fabrics in which all PEs could differ as it could be the case in Embedded FPGAs [44].

Lastly while most scheduling algorithms are geared towards an optimal usage of the available computing resources, many do not consider memory constraints. Assume an operation produces a result that is an operand to a successive operation. If the result is not consumed timely by the next operation, it needs to be stored intermediately. This important issue needs to be addressed, since if this situation occurs too often, the buffer overflows and the algorithm cannot be executed on the CGRA without modification. In [74], the authors describe an algorithm named REGIMap, which optimizes the mapping depending on the usage of registers.

7.2.2.6 Motivation to Upgrade Existing Approaches



(a) The instructions and their dependency of the inner loop of the Sobel operator for edge detection in an image.

(b) The DFG for a $4 \times m$ matrix multiplication after unrolling the 2 innermost loops.

Figure 7.14: Example DFG for the Sobel operator used in edge detection and a DFG performing a matrix multiplication. [59]

Limited Scope – The available schedule algorithms and the proposed ideas are scattered. If e.g. REGIMap is used, it cannot consider pipelined instructions. If SPR is used, the initiation intervals are not examined. We believe that a schedule algorithm is required, which takes into account recent developments in CGRAs such as heterogeneous fabrics and pipelined instructions and which can generate a schedule that not only considers memory requirements, but also the hardware and interconnect utilization.

Efficient Usage – The input to our algorithm is an acyclic DFG representing the part that is designated to be executed on the fabric. Like in EPIMap, the DFG could be restricted to the inner loop only. This is efficient only, if the DFG of the loop is large enough so that the fabric is fully utilized. An example for such an algorithm is the edge detection using the Sobel operator [152], which has a large inner loop consisting of 13 instructions out of which four can be executed in parallel at maximum (refer to Fig. 7.14a). However as it can be observed in the figure the parallelism is reduced to two and later to only one operation. Hence instead of trying to minimize the initiation interval, the hardware utilization can be increased by allowing a sufficiently large number of iterations of that inner loop to be considered for scheduling, even if it results in a longer execution time for a single iteration.

Scalability – An example is multiplication of two matrices A and B of sizes $n \times m$ and $p \times n$ respectively. The resulting DFG for the inner loop consists of one addition and multiplication only. Unrolling this loop results in a DFG consisting of $n - 1$ addi-

tions and n multiplications and two operations can be executed in parallel (refer to Fig. 7.14b). If the fabric is scaled and more PEs are added, the second nested loop of the matrix multiplication can be unrolled as well, resulting in a DFG consisting of m independent subgraphs with a parallelism of $2m$.
Hardware Awareness – In many algorithms variables need to be initialized, before the inner loop is entered. They require complex operations on special PEs, but further calculus depends on the result. E.g. in the Givens Rotation algorithm, the rotation matrix needs to be calculated before it can be applied, for which a floating point square root and division are required. Hence it is desirable to perform these operations on dedicated processing elements optimized for this operation, limiting execution to a certain PE of the fabric.

7.2.3 Proposed Approach

In this sub-section the proposed algorithm is presented, with emphasis on each module, with examples. In summary, the algorithm proceeds in several stages, as follows:

- Capturing and classification of architectural language elements (hardware resources), such as processing elements and interconnect followed by constructing the architectural language graph model.
- Creating the time-extended DFG by adding language-dependent information to the vertexes such as pipeline depths, latencies for a particular operation.
- Mapping and scheduling the application DFG onto the architectural pattern, applying optimization heuristics to minimize resource requirements.
- Applying architectural resource constraints and finalizing schedule and mapping assignments, applying inter-PE communication minimization.

7.2.3.1 Capturing Architectural Properties

To allow capturing of generic CGRA architectures, the hardware resources are classified as flexibly as possible. Language elements containing interconnect is not modeled at this stage, in order to achieve best possible schedule and operation mapping under ideal interconnect conditions. The tool aims to deliver the temporal, spatial and topological coordinates for each operation in the DFG with minimized inter-PE communication. This allows designers to freely consider the interconnect architecture best suited for the resulting inter-PE communication pattern (e.g. NoC, bus, etc), then applying this result to the interconnect constraints set, yielding the final schedule and making the configuration derivable.

Let $P = \{pe_1, pe_2, \ldots, pe_n\}$ be the set of all PEs in a fabric. We define a pool of operations $Ops = \{op_1, op_2, \ldots, op_m\}$ and $\forall pe \in P : pe \subseteq Ops$. Further it is stated: if two PEs support the exact same set of instructions, they belong to the same set T.

$\forall pe \in P : pe_x = pe_y \ with \ x \neq y \Rightarrow T = \{pe_x, pe_y\}$

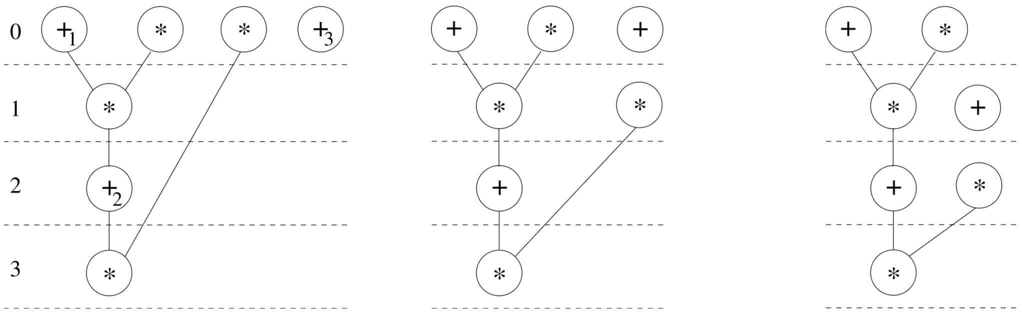
and there are no two different types of PEs which can execute the same instruction:

$\forall pe_x \in T_i,\ pe_y \in T_j \text{ with } i \neq j : pe_x \cap pe_y = \varnothing$.
Let S be a set containing all sets T: $S = \{T_1, T_2, \ldots, T_n\}$.

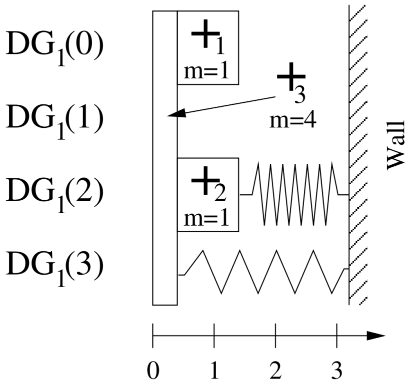
Example: A 2×2 PE array consisting of 3 adders/subtracters and pe_1 having a multiplier, then $P = \{pe_1, pe_2, pe_3, pe_4\}$ with $pe_1 = \{*\}$ and $pe_{2,3,4} = \{+, -\}$. $T_1 = \{pe_2, pe_3, pe_4\}$, $T_2 = \{pe_1\}$, $S = \{T_1, T_2\}$. The sets in S are replicated during mapping, for each time index t (time-extending). △

7.2.3.2 Resource-Aware Scheduling and Optimization

For the input DFG, the As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) hardware-independent operation schedules are created [114]. This gives the initial time bound t_{max} in which the DFG is executable, but also the maximum number of required PEs per time index for every operation. Within the time t_{max}, several scheduling variants may exist formed between the ASAP and ALAP schedules. Consider Fig. 7.15a-c which depict a few scheduling variants. In the table shown in Fig. 7.15e, the maximum required resources for each of the schedules, are listed.



(a) Schedule 1 **(b)** Schedule 2 **(c)** Schedule 3



(d) Distribution Graph (DG) for the addition operation

Schedule	Adders	Multipliers
1	2	2
2	2	2
3	1	1

(e) Maximum required resources (language instances) for the scheduled operations

Figure 7.15: Scheduling variants for a given DFG and the distribution graph for the addition operation. Missing edges are load/store operations. [59]

For a given CGRA, the hardware constraints of the language constructs will determine, if at all, one of the variants can be realized in the fabric and for what values of

$t_{final} \geq t_{ASAP}$. In the ideal case, all available units are used (optimizing parallelism, efficiency) and t_{final} of the final schedule stays at t_{ASAP} (optimizing speed).

The proposed algorithm exploits Force-Directed Scheduling (FDS) [124] heuristics to search for the scheduling variant which respects or gets close to these constraints, details follow:

7.2.3.3 Calculation of Mobility

In the ASAP schedule the time index t_{ASAP} is determined in which an operation can be executed as early as possible, whereas the ALAP schedule calculates the latest possible time index t_{ALAP} for the same operation. The mobility $m = t_{ALAP} - t_{ASAP} + 1$ reflects the ability of a vertex to be scheduled at other time indexes between t_{ASAP} and t_{ALAP}. For vertexes located on the critical path, $m = 1$. This is used for creating the scheduling variants.

7.2.3.4 Distribution Graphs

FDS employs *distribution graphs (DG)*, where $DG_{T_n}(t)$ quantifies the number of occurrences of an operation (scheduling congestion), for a PE in set $T_n \in S$ at time index t. For each of such an operation $DG_{T_n}(t)$ is incremented by $\frac{1}{m}$.

Example: Let $Ops = \{*, +\}$, $P = \{pe_1, pe_2, pe_3, pe_4\}$, $pe_1 = pe_2 = \{+\}$ and $pe_3 = pe_4 = \{*\}$. Thus $T_1 = \{pe_1, pe_2\}$ and $T_2 = \{pe_3, pe_4\}$.

With referral to Fig. 7.15d, the distribution graph equals to $DG_{T_1}(0) = DG_{T_1}(2) = 1\frac{1}{4}$ and $DG_{T_1}(1) = DG_{T_1}(3) = \frac{1}{4}$. The additions marked 1 and 2 are on the critical path and hence cannot be moved, whereas addition 3 can be scheduled to all four available time slots, since it does not have any dependencies.

Similarly for the multiplication: $DG_{T_2}(0) = DG_{T_2}(1) = 1\frac{1}{3}$, $DG_{T_2}(2) = \frac{1}{3}$ and $DG_{T_2}(3) = 1$ respectively. △

7.2.3.5 Scheduling

For each vertex with $m > 1$, the effects on $DG_{T_n}(t)$ of allocations of the operations to all possible time indexes is calculated. The effects are comparable with the expansion or contraction of springs that are attached to a fixed wall on one side and to the vertexes on the other side for each time index and $T_n \in S$ (Fig. 7.15d). The more operations are scheduled in a time index, the more the spring is compressed. Removing a vertex from a time index leads to a spring extension whereas the adding it to a new time index causes the respective spring to be compressed. The sum of elongations and compressions are a force which is equivalent to the overall scheduling congestion of a DFG variant. Each movement of vertexes to a new time index results in an update of $DG_{T_n}(t)$, since the mobility is fixed to $m = 1$ for each placed vertex. Minimizing the overall force is equivalent to the minimized resource requirement (for the given example, Fig. 7.15c) Please note, the vertices are only referred within $0 \leq t \leq t_{max}$.

This schedule however, does not guarantee a valid schedule. If e.g. $|T_n| = 1$ and one time index t has a $DG_{T_n}(t) > 1$, more PEs are required for this particular time index, since the one PE is overbooked. Hence FDS is not constrained by the physical availability of PEs and could acquire nonexistent (imaginary) PEs, as necessary, for each overbooked operation for the respective t. For each time index and assuming infinite hardware resources, imaginary PEs are assigned to each operation in the DFG naïvely. In the next step these imaginary PEs are replaced by physical ones in due consideration of the hardware constraints.

7.2.3.6 Applying Constraints

While the FDS algorithm reduces hardware's requirements at execution time by equally distributing the operations based on PE capabilities, the resulting DFG might still require more resources than the fabric is able to provide. Mapping the FDS schedule to the time-extended graph of the architecture may not be possible without extending t_{max}. In order to provide a *valid mapping*, the proposed algorithm needs to defer operations beyond the current t_{max}. This increases execution time under consideration of inter-PE communication, memory constraints and topological PE information, as described in Alg. 5.

Algorithm 5 Applying hardware constraints to the schedule constructed by FDS. [59]

```
function CONSIDERHARDWARE($DFG$)
    $DG \leftarrow$ GETDG($DFG$)
    $t_{max} \leftarrow$ GETHEIGHT($DFG$)
    for $0 \le t_{curr} < t_{max}$ do
        for all $T_n \in S$ do
            RESCHED($DFG, DG, T_n, t_{curr}, t_{max}$)
        end for
    end for
end function
function RESCHED($DFG$, $DG$, $T_n$, $t_{curr}$, $t_{max}$)
    while $|PE_{free}(t_{curr})| < DG_{T_n}(t_{curr})$ do
        $v \leftarrow$ GETVERTEXSUBGRAPH($DFG, t_{curr}$)
        if $|v| > 1$ then
            INTERLEAVE($v, DFG$)
        else
            DEFER($v, DFG, t_{max}$)
        end if
        $DG \leftarrow$ GETDG($DFG$)
        $t_{max} \leftarrow$ GETHEIGHT($DFG$)
    end while
end function
```

First, the availability of a PE is defined, such that overbooking of the PE is prevented. In case of pipelined PEs, the latency is also taken into account, including result generation conflict avoidance (two operations of different pipeline depth of the same PE should not produce two results at the same time).

Definition 7.2.3. $PE_{free}(t) \subseteq T_n$ is defined as a set in which each element is a PE that is ready to start a particular operation at time index t. In case of pipelined PEs, the time this new operation will take, is equal to the duration of currently executed operations by the same PE (latency). If the pipeline depth of the new operation differs from the depths of currently executed operations, the PE is marked as unavailable (clashing pipeline depths). ☐

Definition 7.2.3 ensures that a PE cannot produce multiple results within the same time index due to different depths of the pipeline. Next, for every set in $T_n \in S$ and every time index $t_{curr} \leq t_{max}$, vertexes are assigned to the actual PEs by comparing $DG_{T_n}(t_{curr})$ with the number of available PEs in $PE_{free}(t_{curr})$ in the fabric. If $|PE_{free}(t_{curr})| \geq DG_{T_n}(t_{curr})$ an assignment has been completed successfully and the next time index is examined by incrementing t_{curr}. Otherwise, the overbooking of PEs needs to be reduced by moving vertexes to later time indexes using two functions called *interleave* and *defer*, by which conflicts between the FDS schedule represented by the distribution graphs for each time index ($DG_{T_n}(t_{curr})$) and the architectural map of available PEs for the same t_{curr} can be solved.

7.2.3.7 Interleave

This function determines, if vertices exist at t_{curr}, which belong to the same subgraph. Assuming undirected edges in the time extended DFG, if a path exists between vertex v_i and v_j with $i \neq j$, then v_i and v_j belong to the same subgraph. As long as the algorithm finds such vertexes belonging to the same subgraph, the size of the subbranches is calculated. The vertex of the smaller branch is then interleaved into the larger branch by deferring the vertices of the larger branch to a later time index.

Example: The DFG depicted in Fig. 7.15, consists of two subgraphs. In Fig. 7.15c the multiplication at $t_{curr} = 2$ belongs to a different subbranch than the addition at the same time index. However both operations belong to the same subgraph.

The multiplication would be interleaved by deferring it to $t = 3$ and assigning the PE of the larger branch to it. Since at $t = 3$ the PE has already assigned a multiplication to it, room must be created so that definition 7.2.3 is not violated and dependencies are preserved (definition 7.2.2). Hence, the vertexes of the longer branch currently scheduled at $t \geq t_{curr} + 1$, need to be deferred sufficiently to a later time index $t \geq t_{curr} + 2$, depending on the required pipeline depth of the interleaved operation. This operation decrements $DG_{T_n}(t_{curr})$ by one, possibly elongating the time extended DFG. Further, since dependent operations are placed as closely as possible by the FDS algorithm described in Subsection 7.2.3.2, an interleaving of operations not belonging to the same branch causes the necessity to intermediately store the result of the preceding operation at $t = t_{curr}$. △

There is no restriction on the amount of available intermediate storage, because the scheduling algorithm is kept generic allowing us to use it on various different

CGRAs. Currently the algorithm only informs the user of how much storage capacity is required at maximum.

Another observation of the interleave is a reduction of inter-PE communication, because even the interleaved operation precedes a depending one in the larger branch. Since inter-PE communication is considered a bottleneck and the latency of interconnects tends to increase exponentially while approaching saturation levels [49], a reduction of the load on the interconnect will have a positive effect on the overall execution time of the algorithm represented by the DFG.

Example: Consider the subgraph given in Fig. 7.16. In the first step the vertexes of the larger branch are deferred. This deferral allows the vertex that was originally scheduled on PE_1 to be executed on PE_0 instead reducing inter-PE communication. However the result that is produced by the addition placed at t_2, needs to be stored intermediately in a buffer represented by the box. △

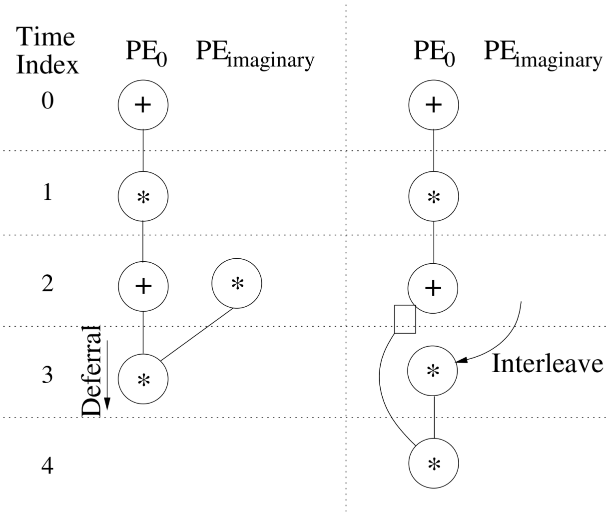


Figure 7.16: The effects of interleaving the multiplication into the larger branch. In this example the pipeline depth all operations is one time index. [59]

7.2.3.8 Defer

If interdependent vertexes cannot be interleaved any further, i.e. no vertex belonging to the same subgraph is found at t_{curr}, and $|PE_{free}(t_{curr})| < DG_{T_n}(t_{curr})$, a more aggressive method is required to create a valid schedule that can be executed on the fabric. The method moves one of the *independent* subgraphs to a time index t with $t > t_{curr}$ in which $DG_{T_n}(t) < |PE_{free}(t_{curr})|$. If no such time index is found, the time extended DFG is prolonged allowing the subgraph to be rescheduled at the end. Subgraph selection is done by the proximity of the root vertex to t_{curr}. If multiple subgraphs are found, whose root is scheduled at the same time index, their heights and orders are considered next. While this deferral seems to be crude, it serves multiple purposes:

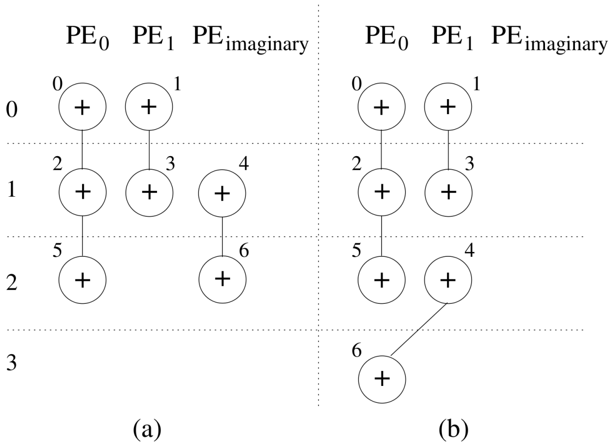


Figure 7.17: A DFG needs to be executed on a fabric consisting of only one PE able to perform an addition. [59]

1. It minimizes the impact to the optimally laid out FDS schedule described in Subsection 7.2.3.2.

2. Compared to deferring the subgraphs to the end of the DFG by default, it potentially prevents a prolongation of the time extended DFG beyond t_{max}, impacting performance.

3. It reduces the complexity of the scheduler, since the subgraph deferral is controlled and situational, preventing a $t_{curr} + 1$ scheduling "avalanche", severely elongating the schedule beyond t_{max} and destroying the FDS schedule.

Example: Consider the time extended DFG given in Fig. 7.17a and a fabric that consists of only **two** PEs able to perform an addition: $T_0 = \{pe_0, pe_1\}$ with $pe_0 = pe_1 = \{+\}$. Despite the mobility $m = 2$ of the vertexes 1, 3, 4 and 6, the FDS algorithm could not generate a valid schedule, because it requires more hardware resources than the fabric is able to provide. Further all vertexes belong to different subgraphs at $t_{curr} = 1$ and hence they cannot be interleaved. However excluding the subgraph consisting of vertexes 4 and 6, at time index $t = 2$, $DG_{T_0}(2) < |PE_{free}(2)|$. Thus the root of the subgraph containing vertex 4 is going to be deferred to time index 2 prolonging the DFG by one time index. △

7.2.3.9 Minimizing Inter-PE Traffic

The repeated application of *interleave* and *defer* leads to a valid schedule, however it lacks topological information of the PEs and vertex dependencies when assigning the vertex to a PE. As a result, a schedule which is shown in Fig. 7.17b, could be the consequence. To reduce inter-PE traffic, it would have been beneficial, if vertex 6 would have been assigned to the same physical PE as vertex 4 without any penalty.

The proposed schedule algorithm also minimizes the load on the interconnect, by considering the dependencies to the predecessors and successors of every vertex and performing the necessary rearrangement such that it does not impact the validity of the optimized schedule, i.e. the operations are temporarily mapped to other possible PEs of the same time index. The composition that causes the least amount of inter-PE traffic, is chosen and the PE is finally assigned to a vertex of the DFG.

7.2.3.10 Mapping

Finally, physical PEs need to be assigned to the vertexes of the DFG, hence the DFG needs to be converted into a graph representing the time extended fabric. Due to the interleaving of vertexes and rescheduling of subgraphs, it can be observed that the time extended DFG enriched with spatial information stating which PE is required to execute which operation at a specific time index, is sparse with respect to the inter-PE communication. This finding reduces the complexity of the mapping algorithm tremendously, since only a few edges need to be considered.

A graph is generated representing the affinities among all PEs, i.e. the number of instances a PE communicates with any other PE. PEs with a highest communication demand, need to be placed closely together in the fabric.

The problem of embedding an affinity graph onto a target graph which represents the CGRA interconnect, is well understood and different methods exist such as the spring method proposed by [55] or a force directed one used in [44].

7.2.4 Evaluation and Results

The evaluation of the proposed scheduler has been done in two stages: 1) scheduling of DFGs of various real-life applications of different sizes and complexities; and 2) comparing with existing state-of-the-art.

Table 7.1 summarizes the results of some of the experiments. Different algorithms on different fabric configurations and different PEs have been tested, while noting inter-PE communication, maximum memory requirements (local storage) in words, overall execution time and fabric utilization.

7.2.4.1 Matrix Multiplication

For executing a matrix multiplication on homogeneous (HM) fabrics of various sizes a full fabric utilization is achieved. Utilization rate decreases significantly, if a more realistic heterogeneous (HT) fabric is used, in which 50% of the hardware resources are capable of performing the addition and multiplication, respectively. Different pipeline depths (e.g. 3) of the multiplication logic were considered versus adder pipeline depth (e.g. 1), yielding lower utilization and longer execution time due to the slow multiplier, but also a significant inter-PE communication overhead.

7.2.4.2 Edge Detection

For edge detection using the Sobel operator, a homogeneous 2×2 fabric is fully utilized. While a larger fabric with 16 PEs in total results in a significantly shorter execution time, it cannot fully utilized. The reason is that at the end of the execution more hardware resources are available to calculate the edges for the last 4 pixels of the image. Hence for these pixels the algorithm uses the parallelism offered by the CGRA, causing inter-PE communication and a reduced hardware utilization to 65%. However compared to the overall execution time, this reduction is negligible.

7.2.4.3 Givens Rotation

To obtain the results for the Givens rotation, the two innermost loops for a matrix of size 5×5 have been unrolled to create the DFG. Since all operations are in floating point arithmetic, the required pipeline depths have been increased to realistic values and one dedicated PE is able to perform the square root and division. As it can be observed, the larger 4×4 fabric does not speed up the execution significantly despite the fact that the number of available PEs which are able to perform the multiplications and additions, have been quintupled (from 3 in the 2×2 fabric to 15 in the 4×4 one). Due to the pipeline in the MUL/ADD PEs, the instructions of the inner most loop can be interleaved efficiently. Increasing the number of MUL/ADD PEs spreads the instructions spatially, leading to a reduced memory footprint, but a denser inter-PE communication pattern, with a negligible impact on the execution time.

The problem in the Givens rotation is that costly operations such as square root and division are on the critical path without the possibility to parallelize those. While the PE performs the square root followed by the division all other PEs are idle leading to a low fabric utilization of 53.49% and 15.47%.

7.2.4.4 LU Decomposition

Similar problems can be observed for a LU decomposition of an exemplary 3×3 matrix. The divisions are executed first and no other operation can proceed till the quotients are available. This nullifies the parallelism which leads to a comparatively low fabric utilization of 65% for realistic ALU configurations.

7.2.4.5 JPEG Downsample

The JPEG Downsample algorithm is a crucial part of the compression algorithm, but it exhibits only a limited level of instruction parallelism. Although several instructions are executed in parallel, the limiting factor is the height of the critical path requiring 14 time indexes. Increasing the number of resources does not lead to a higher parallelism and faster execution time.

7.2.4.6 Smoothing Triangle

The smoothing triangle and interpolating subroutines of [14] show a high level of instruction parallelism and consist of four independent subgraphs. In homogeneous fabrics of 2×2 PEs, a high utilization can be achieved. For larger fabrics this utilization decreases due to dependencies among the vertexes of the DFG. While the overall execution time reduces, a denser communication pattern is observed.

7.2.4.7 FIR Filter

Finite Input Response (FIR) filters are often observed in the signal processing domain. Although several instructions can be executed in parallel, the filter suffers from the same problem as the JPEG down-sample algorithm: along the critical path are not enough instructions available which can be executed in parallel. In fact, only 3 PEs are used at any point in time during the execution resulting in a low fabric utilization.

Table 7.1: Results of scheduled DFGs on various fabric and PE configurations. If not stated differently in the column *PE Configuration*, each instruction requires one time index. Otherwise the column shows the number of PEs available in the fabric supporting the corresponding operations and the time indexes required to execute that operation. [59]

Application	Fabric Configuration	PE Configuration	Inter-PE Traffic [Occurrences]	Max. req. Mem.[words]	Exec. Time [time indexes]	Fabric Utiliz.
Matrix Multiplication (8 × 8)	2 × 2 (HM)		0	6	240	100%
	4 × 4 (HM)		0	6	60	100%
	2 × 2 (HT)	2×MUL: 3 2×ADD: 1	512	5	340	70.59%
	4 × 4 (HT)	8×MUL: 3 8×ADD: 1	512	7	75	80.00%
Sobel Edge Detection (Image size: 640 × 480)	2 × 2 (HM)		0	3	991133	100%
	4 × 4 (HM)		16	3	247785	99.9993%
Givens Rotation (2 inner loops unrolled)	2 × 2 (HT)	1×DIV/SQRT: 26 remaining PEs: ADD/MUL: 7	15	7	86	53.49%
	4 × 4 (HT)		25	2	82	15.47%
LU Decomposition	2 × 2 (HT)	1×DIV: 26 remaining PEs: SUB/MUL: 7	5	4	75	65%
JPEG Smooth Downsample	2 × 2 (HM)		10	4	14	60.71%
Mesa Smoothing Triangle Subroutine	2 × 2 (HM)		15	7	38	99.02%
	4 × 4 (HM)		55	5	14	66.52%
Mesa Interpolate Aux Subroutine	2 × 2 (HM)		2	5	23	100%
	4 × 4 (HM)		33	5	8	71.88%
FIR Filter	2 × 2 (HM)		8	2	9	58.33%
EPIMap Example 1	2 × 2 (HM)		24	4	24	100%
EPIMap Example 2			16	5	24	100%
EPIMap Example 3			2	4	28	100%

7.2.4.8 Comparing with EPIMap

The EPIMap algorithm which is required to reliably compare between the published results and the proposed algorithm, is not available. Hence details about DFG schedules calculated by EPIMap, are unknown. However in [73] the authors show the scheduled vertices of three exemplary DFGs (refer to Fig. 7.18a) by using EPIMap. As EPIMap is considered as an algorithm aiming to minimize the initiation interval of the innermost loop of a kernel, the DFGs are multiplied so that 16 subgraphs need to be scheduled for each example. Further PEs in the time extended fabric, which are used for recomputing or routing only, are considered to be idle. We have replicated the examples, and a homogeneous fabric has been arranged to a mesh of size of 2×2 with each PE executing any instruction within one time index. The results are depicted in Fig. 7.18.

While EPIMap achieves only 75% average fabric utilization (Fig. 7.18b), the proposed algorithm achieves full fabric utilization. In Fig. 7.18c the overall execution time is 39% longer compared to the proposed algorithm due to significant resource utilization for recomputing, recalculation or simply unused PEs. Furthermore, EPIMap does not need to consider inter-PE communication (Fig. 7.18d), since the fabric in [73] can only communicate to its direct neighbors in a mesh topology, a restriction which this algorithm does not consider. This however, could need solving routing congestion (e.g. NoC congestion) later in the mapping phase.

While in EPIMap the initiation interval and the iteration latency is equal to two and four respectively for all presented examples, using the proposed algorithm leads to initiation intervals and iteration latencies of 6.33 time indexes in average. This increase however does not impact the efficiency of the algorithm, since after each iteration all PEs produce a result, i.e. 4 results in the exemplary 2×2 fabric, leading to shorter overall execution times and perfect fabric utilization.

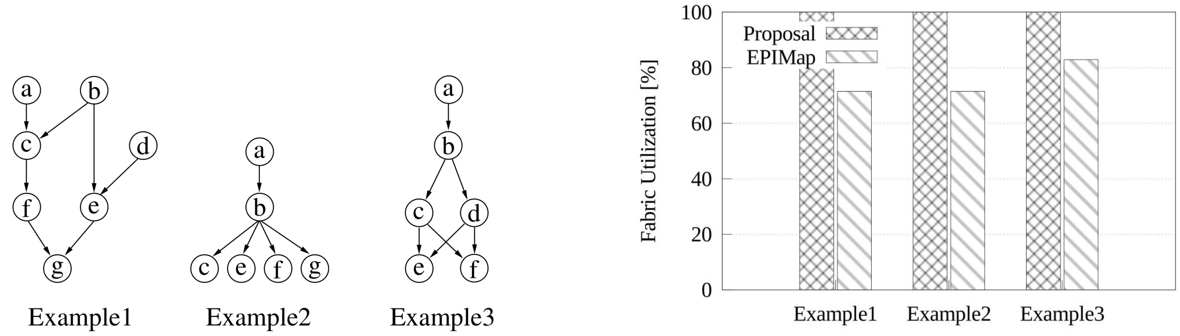
7.2.5 Conclusions and Outlook

7.2.5.1 Summary

An algorithm was proposed, which exploits Force-Directed Scheduling with careful consideration of architectural features, interconnect and processing element properties to schedule input DFGs efficiently. Some of the details applicable to a wide range of CGRAs such as inter-PE traffic, memory constraints and fabric utilization, as important metrics and factors which influence a good schedule were pointed out. Using examples, each of the proposed transformations is discussed, finally evaluating the scheduler with a set of real-life benchmark DFGs and comparing the proposed solution with the state of the art.

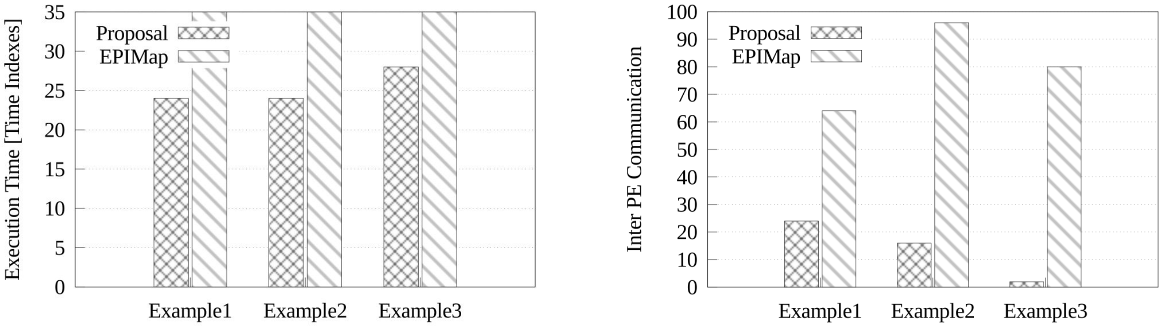
7.2.5.2 Future Work

The proposed algorithm can tackle a variety of applications efficiently even for large DFGs or for algorithms with little parallelism.



(a) Example graphs from EPIMap [73]

(b) Fabric utilization



(c) Overall execution time

(d) Inter-PE traffic

Figure 7.18: Comparison of EPIMap to the proposed algorithm, with the graphs of the examples. [59]

Adding preliminary temporal information by the FDS algorithm ensures that the required hardware resources are minimized without increasing t_{max}. In DFGs such as the 8×8 matrix multiplication, which consists of 64 subgraphs and 960 vertices in total, the FDS algorithm returns the result in 0.25 seconds on an Intel Core i5-3470 CPU operating at 3.2GHz. However, if the dependencies among the vertices increases or the DFG does not consist of independent subgraphs, the time taken by the FDS algorithm escalates to minutes or even hours. After placing a vertex v at a specific time index t_1, the impact on the DG needs to be evaluated by recursively calculating the forces for all other placement options for all vertices in the DFG. For the effects of placing v to $t_2 \neq t_1$ (given that mobility $m > 1$), all forces need to be recalculated.

To reduce the computation time, the recursion depth can be restricted as suggested by the authors of [124]. However in our experiments this resulted in situations in which dependent vertices were placed earlier in time than preceding operations. The succeeding stages following the FDS algorithm, consequently failed to compute a valid mapping, since they require a valid DFG.

Another approach would be to utilize the cores of a processor and to parallelize the FDS algorithm. Currently the algorithm runs only as a single thread on one core. Although the complexity of the FDS algorithm is not reduced, it would speed up the preliminary temporal time index assignments.

For future work, a more desirable approach will be to optimize the FDS algorithm so that it can place the vertices efficiently. The optimization would not only include improvements regarding the programming, but should also include more language-related architectural peculiarities distinct to CGRAs. One idea would be to consider segments of a DFG and optimize them individually. However determination of the sizes of these segments and its impact on the proposed algorithm needs to be investigated further.

7.3 Other Domain Applicability: Example for Cryptography

In this work, *Layers* was focused on the *dwarf* [22] of dense linear algebra, the flexibility of the design does not limit this architecture to one domain only. The processing elements can be easily exchanged to ones suitable for the target domain. Communication requirements, number of layers can also be adjusted.

A study on accelerating a family of cryptographich applications by using their common trait of Addition-Rotation-eXclusiveOr processing is conducted in [89]. While significantly different from *Layers* itself, this study was designed on the methodology concepts proposed in this work.

Similarly re-targeting of *Layers* to other domains is possible, some early experiments have been done for language processing and support vector machines.

7.4 Proposed SoC Integration

As for higher-level integration of the *Layers* accelerator as an SoC component, the proposed flow is illustrated in Fig. 7.19.

The flow starts from the SoC-level application, which is profiled and partitioned by a meta-compiler – one that can identify the kernels and mark them via `pragmas` to be executed on the accelerator. The precompiled code is then executed on the host CPU of the SoC system. *Layers* is connected to the system bus and external memory and receives stop/go commands via the respective `pragmas` from the host CPU. Once the instruction is received, the kernel part of the application is executed internally by *Layers*, returning control to the host CPU when execution completes.

As the internal processing remains intransparent to the host CPU, the users of such an kernel library-based accelerator do not require special effort in using its advantages.

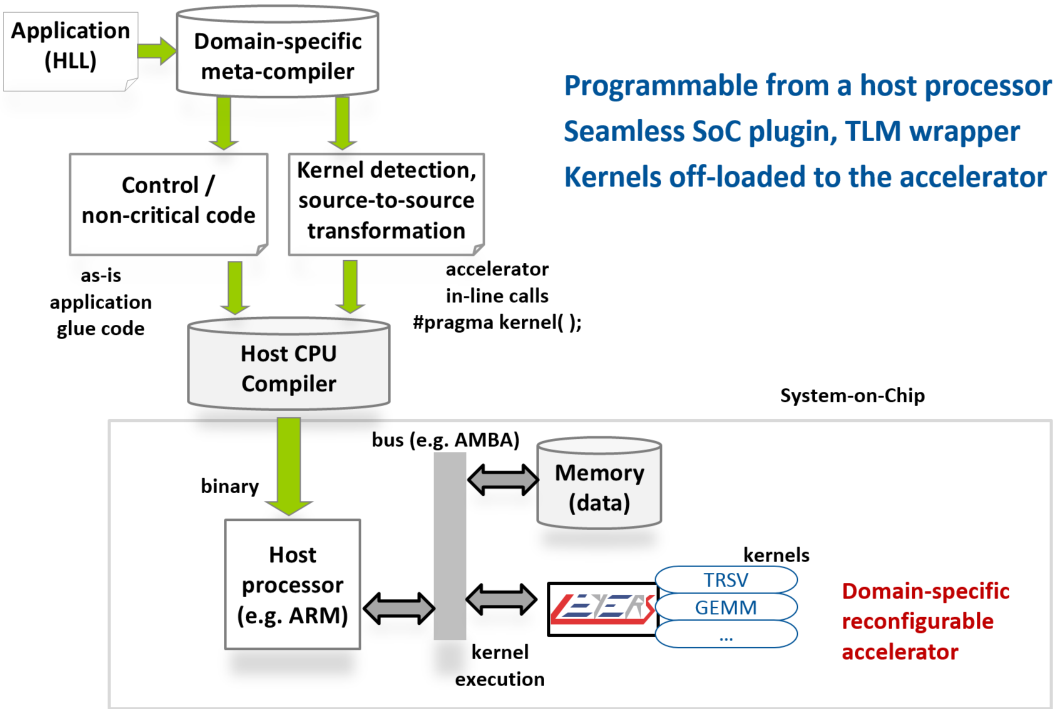


Figure 7.19: The proposed system-level integration of *Layers*, as a black-box library of accelerated kernels.

Chapter 8

Conclusions and outlook

This dissertation explores a new paradigm in designing highly efficient reconfigurable architectures proposing a new paradigm: functional reconfiguration. The key idea behind the proposed theory is that, flexibility can be exploited instead of being traded off to achieve efficiency, by creating a good match between the target application and the architecture. Several experiments across two design flow avenues are exploring the effectiveness of the proposed theory, proving that the proposed view, methodology and architectures can satisfy requirements of high energy-efficiency, quick design exploration and good performance. A more detailed summary follows.

8.1 Summary

Chapter 1 and 2 dive into the landscape of efficient computing and current industrial trends, offering a high level overview of the background. Several motivational vectors are identified and discussed and the problem is formulated. The necessity of highly efficient, low-energy and high-performance architectures is highlighted, coupled with the need of outlining a design methodology in line with current time-to-market constraints.

Chapter 3 invites the reader to experience a new view on designing and programming reconfigurable architectures. Especially for the coarse-grained reconfigurable architectures, the lack of a clean design and programming methodology makes the proposed theory appealing. In this view, the concept of functional programming is exploited to create *functional reconfiguration*, a view that enables architectures to adapt and closely match application requirements. Using functional reconfiguration, it is possible to achieve a high degree of architectural reconfigurability, especially when using the functional separation of 4 functional classes: memory access, data communication, data processing and control flow processing. This enables architectures to use hardware structures that can adapt and match application requirements to a high degree, achieving energy efficiency and high performance.

Chapter 4 explores a design methodology suggested by the theory, exploiting high level synthesis and design to quickly and efficiently cover the huge design space of reconfigurable computing. The methodology proposes design techniques and formulates guidelines which allow a quick design and evaluation of various structures at high abstraction level. Two different flows are proposed to exploit the theory: targeted and tunable architectural flexibility. With targeted architectural flexibility, the architecture is designed such that its language matches the target application's language as closely as possible, while minimizing overhead. This design direction uses hardware functions that reconstruct the application without any overheads, but sacrifices adaptability to other applications, resembling ASIC design philosophy. The second flow, exploits a well tailored pool of elementary hardware functions, to provide various ways of defining one architecture's language by reconfiguration and combinations of these functions. A well tailored language can then be formed for every new application, just be rearranging the way the elementary functions are called.

Chapter 5 dives deeply into the first proposed flow, exploring how well a language can be matched to a pair of WCDMA channel estimation algorithms. The two target algorithms are different from structural, complexity and computational point of view and an architecture is sought that can execute both efficiently. This is achieved by focusing architectural language and the underlying pool of elementary hardware functions to closely match common computation and structures required by the two algorithms. A hybrid architecture is the result, which maintains the performance when compared with two separate designs, without the overhead. Moreover, the hybrid architecture can save more than 88% of energy by dynamically switching between the algorithms, in a scenario where good and bad WCDMA signal conditions alternate.

Chapter 6 explores the second design flow: tunable flexibility. An architecture is proposed that follows closely the functional separation paradigm and creates elementary function pools for each class of computation, memory access, data movement and control flow in a 3D layered structure. By having a flexible pool of elementary functions, varied language constructs can be formed to adapt to 8 kernels from the linear algebra domain, making the new architecture an excellent and efficient domain-specific accelerator. A high degree of architectural flexibility permits thus to change applications on the fly just be re-adapting architectural language to fit exactly the application's required language patterns. A thorough evaluation is performed for each kernel and a performance evaluation is conducted, for comparison with other architectural platforms for this application domain.

Chapter 7 improves on the design of the 3D architecture by introducing two components. The first concerns automatic scheduling and mapping derivation for the computation layer via a force-directed scheduling heuristic approach. This enables a quicker adaptation of the architecture to new applications, by providing key application language requirements which then can be reproduced in the architecture. This approach is also valid for any other coarse-grained reconfigurable architecture. The second component automatically derives reconfigurable control flow structures using 3 different approaches: VLIW-like architecture, a reconfigurable array tailored for control flow and a graph theoretic approach, which generates the architecture automatically. Each solution is explored and evaluated in detail.

8.2 Conclusions

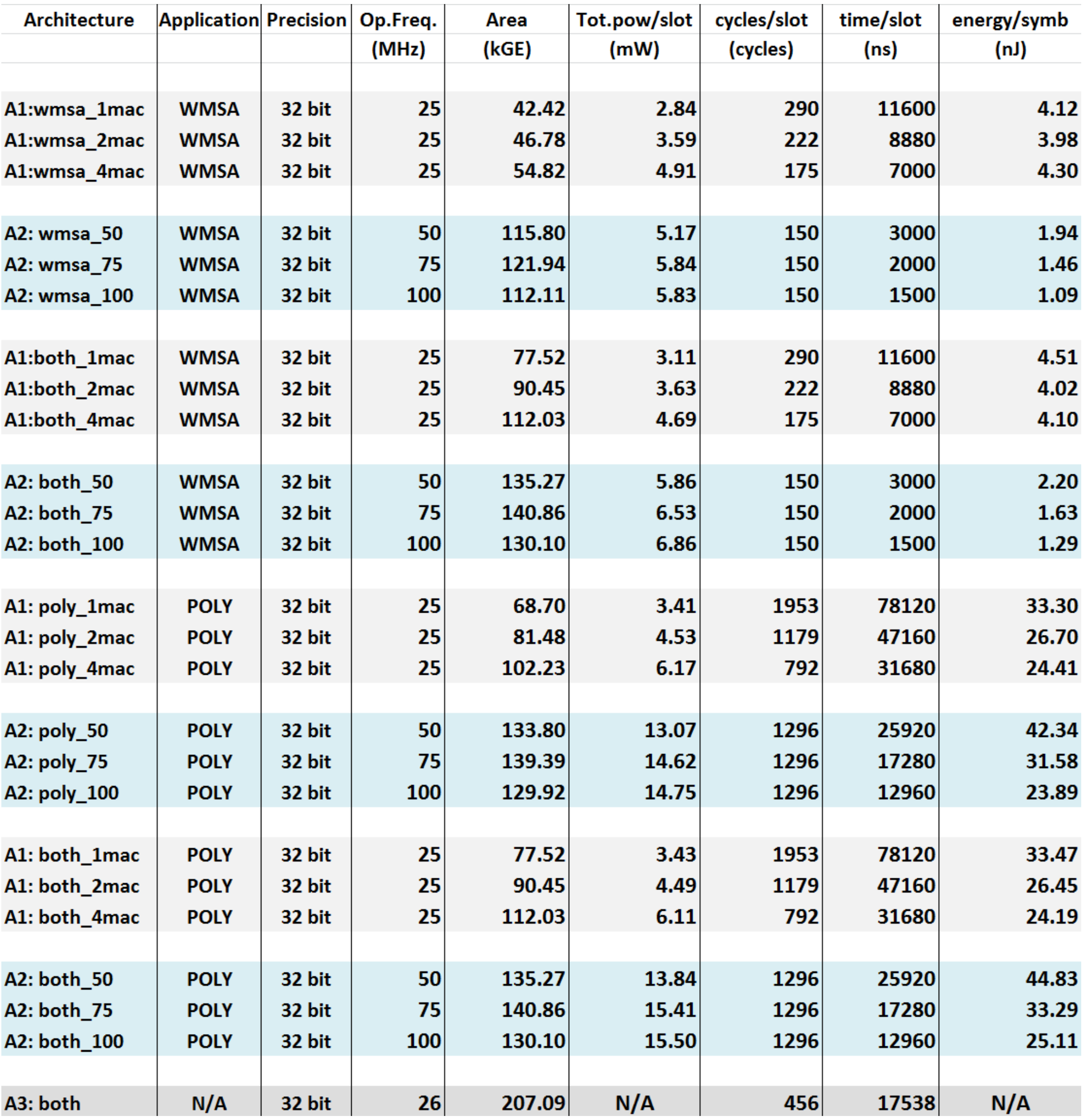
This dissertation proposes a novel view on designing and programming reconfigurable architectures. From the theory, two main design paradigms are derived and with a supporting methodology, these are deeply explored via various architectures. Furthermore, several enhancements are added to make this exploration a complete work, with a new view on reconfigurable computing.

Appendix A

Detailed Results Data for A1/A2 from Chapter 5

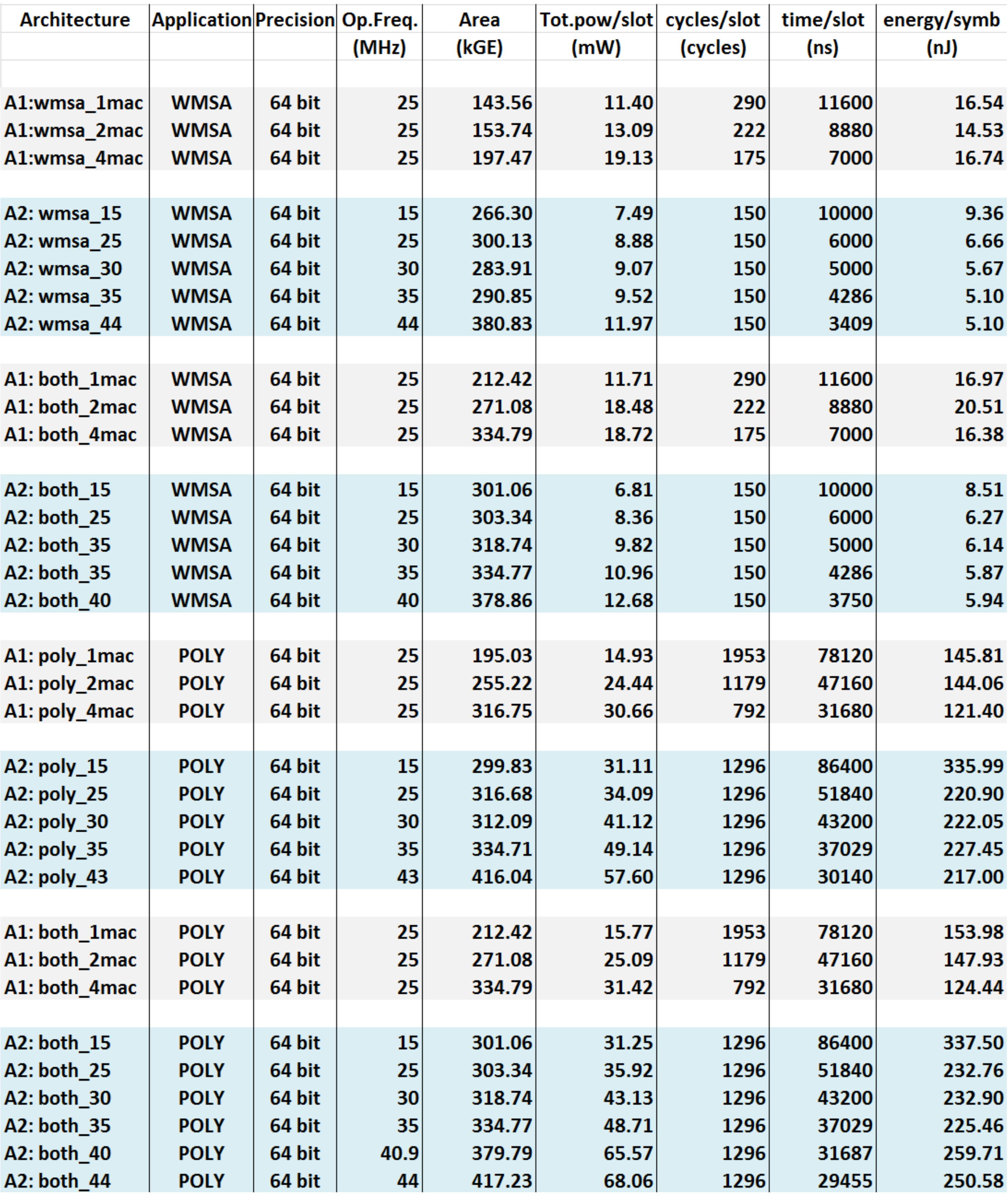
Detailed results for various design points of the architectures with targeted architectural flexibility.

Table A.1: Results for 32-bit architectures [142]



Architecture	Application	Precision	Op.Freq. (MHz)	Area (kGE)	Tot.pow/slot (mW)	cycles/slot (cycles)	time/slot (ns)	energy/symb (nJ)
A1:wmsa_1mac	WMSA	32 bit	25	42.42	2.84	290	11600	4.12
A1:wmsa_2mac	WMSA	32 bit	25	46.78	3.59	222	8880	3.98
A1:wmsa_4mac	WMSA	32 bit	25	54.82	4.91	175	7000	4.30
A2: wmsa_50	WMSA	32 bit	50	115.80	5.17	150	3000	1.94
A2: wmsa_75	WMSA	32 bit	75	121.94	5.84	150	2000	1.46
A2: wmsa_100	WMSA	32 bit	100	112.11	5.83	150	1500	1.09
A1:both_1mac	WMSA	32 bit	25	77.52	3.11	290	11600	4.51
A1:both_2mac	WMSA	32 bit	25	90.45	3.63	222	8880	4.02
A1:both_4mac	WMSA	32 bit	25	112.03	4.69	175	7000	4.10
A2: both_50	WMSA	32 bit	50	135.27	5.86	150	3000	2.20
A2: both_75	WMSA	32 bit	75	140.86	6.53	150	2000	1.63
A2: both_100	WMSA	32 bit	100	130.10	6.86	150	1500	1.29
A1: poly_1mac	POLY	32 bit	25	68.70	3.41	1953	78120	33.30
A1: poly_2mac	POLY	32 bit	25	81.48	4.53	1179	47160	26.70
A1: poly_4mac	POLY	32 bit	25	102.23	6.17	792	31680	24.41
A2: poly_50	POLY	32 bit	50	133.80	13.07	1296	25920	42.34
A2: poly_75	POLY	32 bit	75	139.39	14.62	1296	17280	31.58
A2: poly_100	POLY	32 bit	100	129.92	14.75	1296	12960	23.89
A1: both_1mac	POLY	32 bit	25	77.52	3.43	1953	78120	33.47
A1: both_2mac	POLY	32 bit	25	90.45	4.49	1179	47160	26.45
A1: both_4mac	POLY	32 bit	25	112.03	6.11	792	31680	24.19
A2: both_50	POLY	32 bit	50	135.27	13.84	1296	25920	44.83
A2: both_75	POLY	32 bit	75	140.86	15.41	1296	17280	33.29
A2: both_100	POLY	32 bit	100	130.10	15.50	1296	12960	25.11
A3: both	N/A	32 bit	26	207.09	N/A	456	17538	N/A

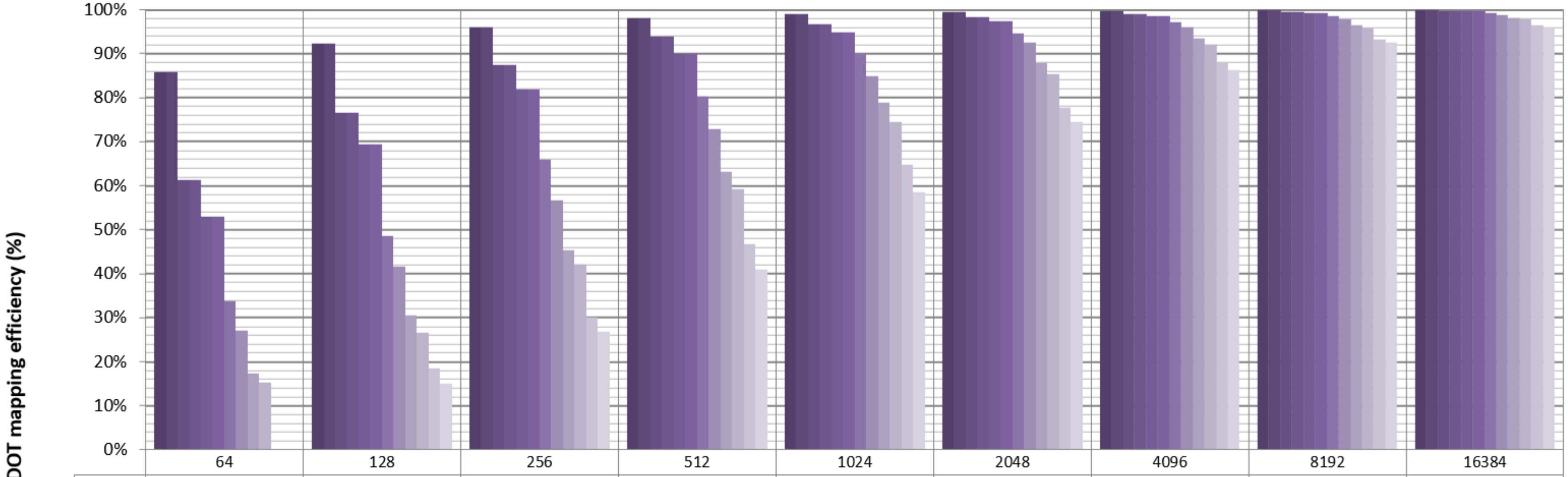
Table A.2: Results for 64-bit architectures [142]



Architecture	Application	Precision	Op.Freq. (MHz)	Area (kGE)	Tot.pow/slot (mW)	cycles/slot (cycles)	time/slot (ns)	energy/symb (nJ)
A1:wmsa_1mac	WMSA	64 bit	25	143.56	11.40	290	11600	16.54
A1:wmsa_2mac	WMSA	64 bit	25	153.74	13.09	222	8880	14.53
A1:wmsa_4mac	WMSA	64 bit	25	197.47	19.13	175	7000	16.74
A2: wmsa_15	WMSA	64 bit	15	266.30	7.49	150	10000	9.36
A2: wmsa_25	WMSA	64 bit	25	300.13	8.88	150	6000	6.66
A2: wmsa_30	WMSA	64 bit	30	283.91	9.07	150	5000	5.67
A2: wmsa_35	WMSA	64 bit	35	290.85	9.52	150	4286	5.10
A2: wmsa_44	WMSA	64 bit	44	380.83	11.97	150	3409	5.10
A1: both_1mac	WMSA	64 bit	25	212.42	11.71	290	11600	16.97
A1: both_2mac	WMSA	64 bit	25	271.08	18.48	222	8880	20.51
A1: both_4mac	WMSA	64 bit	25	334.79	18.72	175	7000	16.38
A2: both_15	WMSA	64 bit	15	301.06	6.81	150	10000	8.51
A2: both_25	WMSA	64 bit	25	303.34	8.36	150	6000	6.27
A2: both_35	WMSA	64 bit	30	318.74	9.82	150	5000	6.14
A2: both_35	WMSA	64 bit	35	334.77	10.96	150	4286	5.87
A2: both_40	WMSA	64 bit	40	378.86	12.68	150	3750	5.94
A1: poly_1mac	POLY	64 bit	25	195.03	14.93	1953	78120	145.81
A1: poly_2mac	POLY	64 bit	25	255.22	24.44	1179	47160	144.06
A1: poly_4mac	POLY	64 bit	25	316.75	30.66	792	31680	121.40
A2: poly_15	POLY	64 bit	15	299.83	31.11	1296	86400	335.99
A2: poly_25	POLY	64 bit	25	316.68	34.09	1296	51840	220.90
A2: poly_30	POLY	64 bit	30	312.09	41.12	1296	43200	222.05
A2: poly_35	POLY	64 bit	35	334.71	49.14	1296	37029	227.45
A2: poly_43	POLY	64 bit	43	416.04	57.60	1296	30140	217.00
A1: both_1mac	POLY	64 bit	25	212.42	15.77	1953	78120	153.98
A1: both_2mac	POLY	64 bit	25	271.08	25.09	1179	47160	147.93
A1: both_4mac	POLY	64 bit	25	334.79	31.42	792	31680	124.44
A2: both_15	POLY	64 bit	15	301.06	31.25	1296	86400	337.50
A2: both_25	POLY	64 bit	25	303.34	35.92	1296	51840	232.76
A2: both_30	POLY	64 bit	30	318.74	43.13	1296	43200	232.90
A2: both_35	POLY	64 bit	35	334.77	48.71	1296	37029	225.46
A2: both_40	POLY	64 bit	40.9	379.79	65.57	1296	31687	259.71
A2: both_44	POLY	64 bit	44	417.23	68.06	1296	29455	250.58

Appendix B

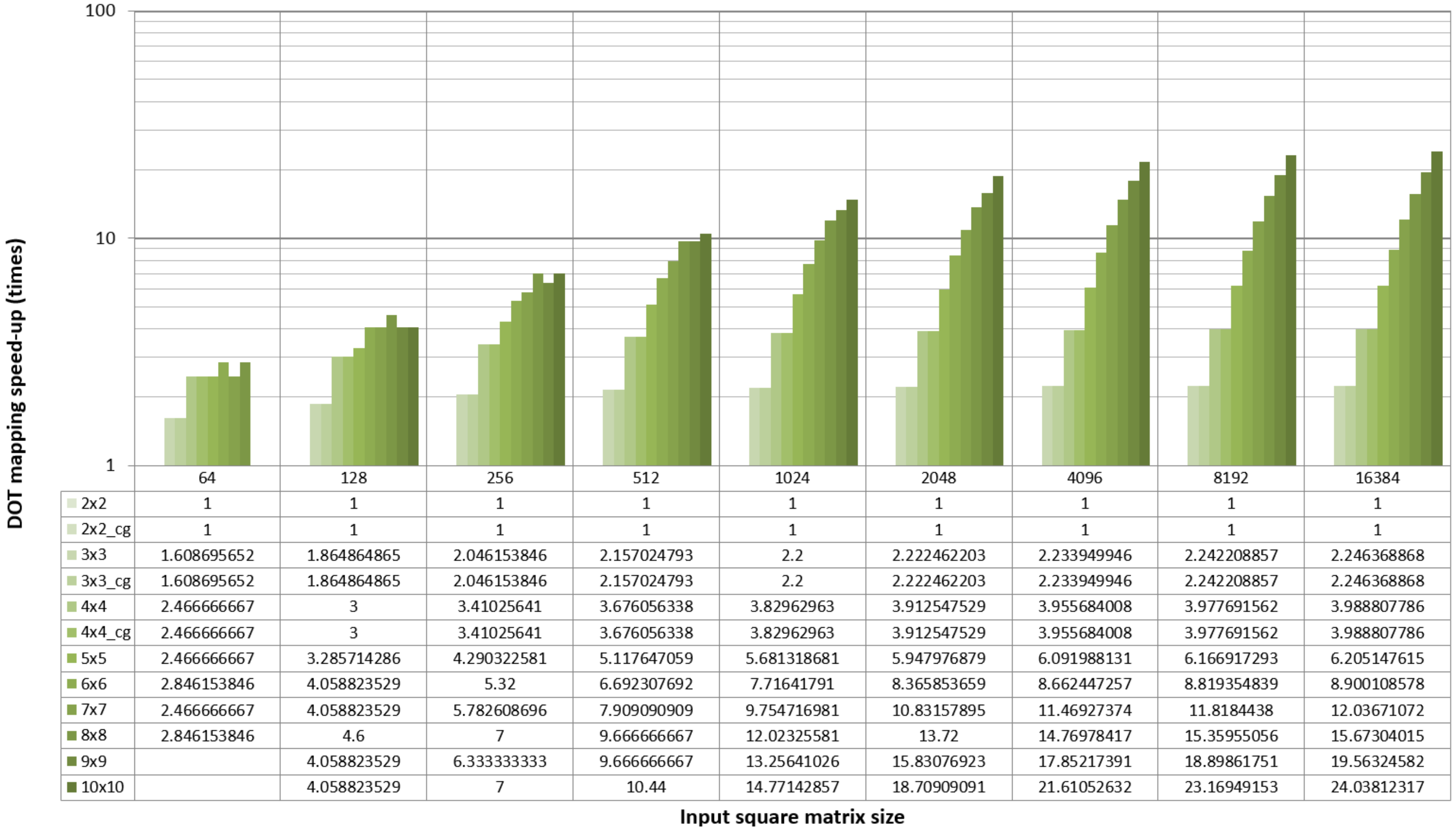
Detailed Data on Kernel Efficiency on *Layers*



	64	128	256	512	1024	2048	4096	8192	16384
■ 2x2	85.8108%	92.3913%	96.0526%	97.9885%	98.9845%	99.4898%	99.7443%	99.8720%	99.9360%
■ 2x2_cg	85.8108%	92.3913%	96.0526%	97.9885%	98.9845%	99.4898%	99.7443%	99.8720%	99.9360%
■ 3x3	61.3527%	76.5766%	87.3504%	93.9394%	96.7849%	98.2721%	99.0328%	99.5262%	99.7747%
■ 3x3_cg	61.3527%	76.5766%	87.3504%	93.9394%	96.7849%	98.2721%	99.0328%	99.5262%	99.7747%
■ 4x4	52.9167%	69.2935%	81.8910%	90.0528%	94.7685%	97.3146%	98.6392%	99.3150%	99.6563%
■ 4x4_cg	52.9167%	69.2935%	81.8910%	90.0528%	94.7685%	97.3146%	98.6392%	99.3150%	99.6563%
■ 5x5	33.8667%	48.5714%	65.9355%	80.2353%	89.9780%	94.6821%	97.2226%	98.5444%	99.2188%
■ 6x6	27.1368%	41.6667%	56.7778%	72.8632%	84.8673%	92.4797%	96.0033%	97.8674%	98.8268%
■ 7x7	17.2789%	30.6122%	45.3416%	63.2653%	78.8217%	87.9699%	93.3873%	96.3536%	98.1959%
■ 8x8	15.2644%	26.5625%	42.0230%	59.2014%	74.3823%	85.3125%	92.0751%	95.8743%	97.8938%
■ 9x9		18.5185%	30.0412%	46.7764%	64.7990%	77.7778%	87.9334%	93.2070%	96.5467%
■ 10x10		15.0000%	26.8947%	40.9200%	58.4857%	74.4545%	86.2211%	92.5593%	96.0909%

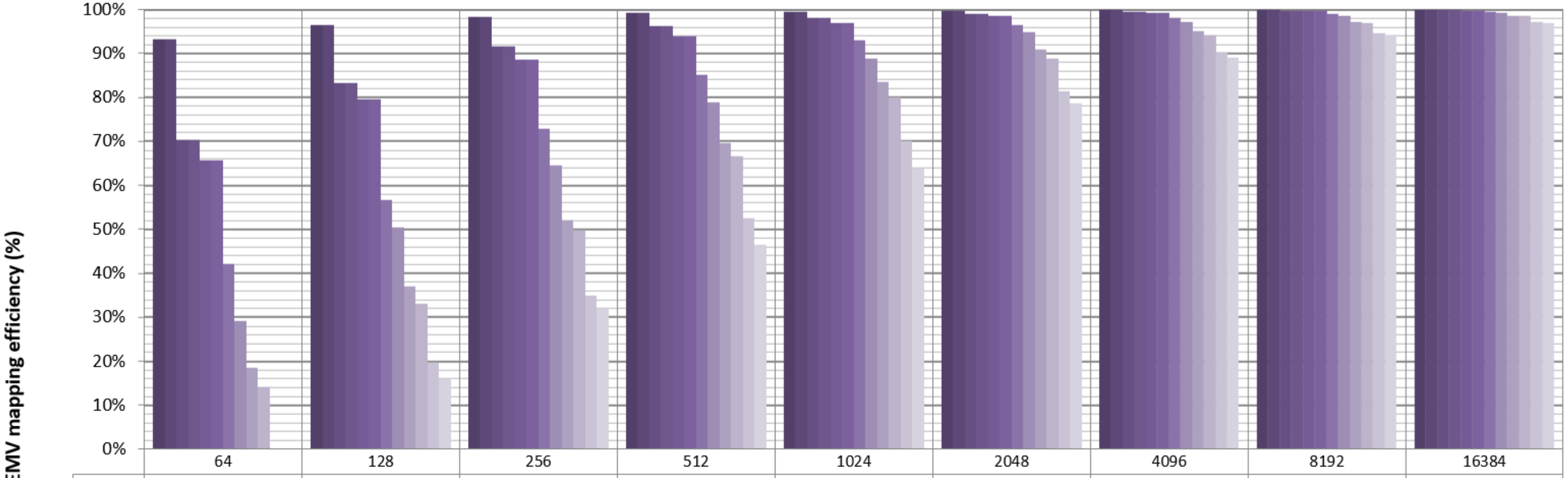
Input square matrix size

Figure B.1: DOT mapping efficiency up for various architectures (N=2..8) and data sizes (64..16384). [139]



	64	128	256	512	1024	2048	4096	8192	16384
2x2	1	1	1	1	1	1	1	1	1
2x2_cg	1	1	1	1	1	1	1	1	1
3x3	1.608695652	1.864864865	2.046153846	2.157024793	2.2	2.222462203	2.233949946	2.242208857	2.246368868
3x3_cg	1.608695652	1.864864865	2.046153846	2.157024793	2.2	2.222462203	2.233949946	2.242208857	2.246368868
4x4	2.466666667	3	3.41025641	3.676056338	3.82962963	3.912547529	3.955684008	3.977691562	3.988807786
4x4_cg	2.466666667	3	3.41025641	3.676056338	3.82962963	3.912547529	3.955684008	3.977691562	3.988807786
5x5	2.466666667	3.285714286	4.290322581	5.117647059	5.681318681	5.947976879	6.091988131	6.166917293	6.205147615
6x6	2.846153846	4.058823529	5.32	6.692307692	7.71641791	8.365853659	8.662447257	8.819354839	8.900108578
7x7	2.466666667	4.058823529	5.782608696	7.909090909	9.754716981	10.83157895	11.46927374	11.8184438	12.03671072
8x8	2.846153846	4.6	7	9.666666667	12.02325581	13.72	14.76978417	15.35955056	15.67304015
9x9		4.058823529	6.333333333	9.666666667	13.25641026	15.83076923	17.85217391	18.89861751	19.56324582
10x10		4.058823529	7	10.44	14.77142857	18.70909091	21.61052632	23.16949153	24.03812317

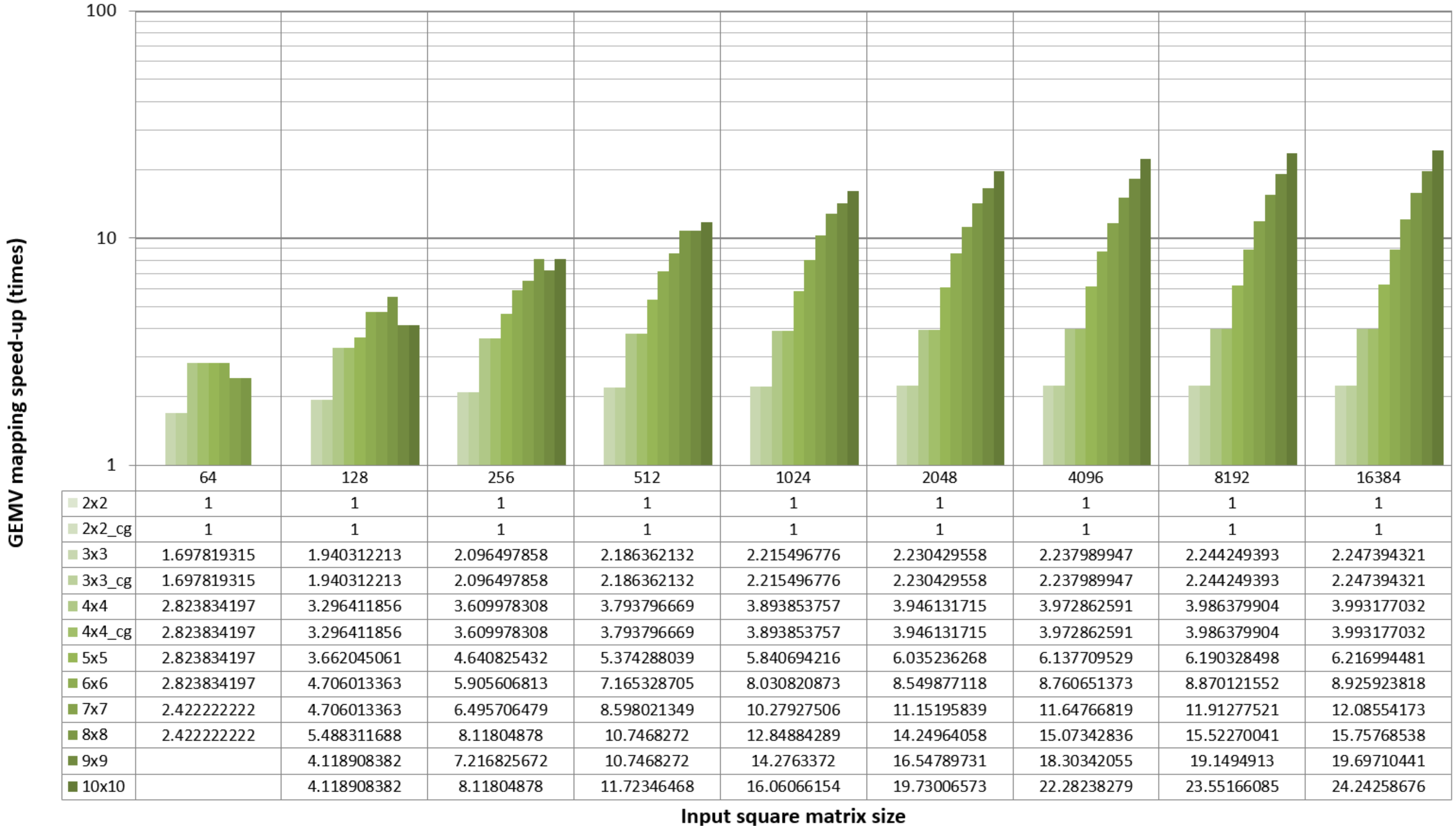
Figure B.2: DOT mapping-based speed-up for various architectures (N=2..8) and data sizes (64..16384). [139]



	64	128	256	512	1024	2048	4096	8192	16384
2x2	93.2110%	96.5452%	98.2574%	99.1249%	99.5615%	99.7805%	99.8902%	99.9451%	99.9725%
2x2_cg	93.2110%	96.5452%	98.2574%	99.1249%	99.5615%	99.7805%	99.8902%	99.9451%	99.9725%
3x3	70.3358%	83.2568%	91.5540%	96.3213%	98.0347%	98.9126%	99.3570%	99.6896%	99.8568%
3x3_cg	70.3358%	83.2568%	91.5540%	96.3213%	98.0347%	98.9126%	99.3570%	99.6896%	99.8568%
4x4	65.8031%	79.5632%	88.6768%	94.0149%	96.9195%	98.4368%	99.2125%	99.6048%	99.8020%
4x4_cg	65.8031%	79.5632%	88.6768%	94.0149%	96.9195%	98.4368%	99.2125%	99.6048%	99.8020%
5x5	42.1140%	56.5685%	72.9593%	85.2361%	93.0413%	96.3518%	98.0955%	98.9909%	99.4446%
6x6	29.2458%	50.4826%	64.4744%	78.9181%	88.8401%	94.7901%	97.2337%	98.5028%	99.1497%
7x7	18.4308%	37.0892%	52.1022%	69.5737%	83.5445%	90.8366%	94.9786%	97.1937%	98.6304%
8x8	14.1111%	33.1169%	49.8537%	66.5799%	79.9531%	88.8648%	94.1055%	96.9636%	98.4585%
9x9		19.6376%	35.0176%	52.6063%	70.1913%	81.5387%	90.2880%	94.5135%	97.2429%
10x10		15.9064%	31.9063%	46.4835%	63.9609%	78.7470%	89.0317%	94.1549%	96.9437%

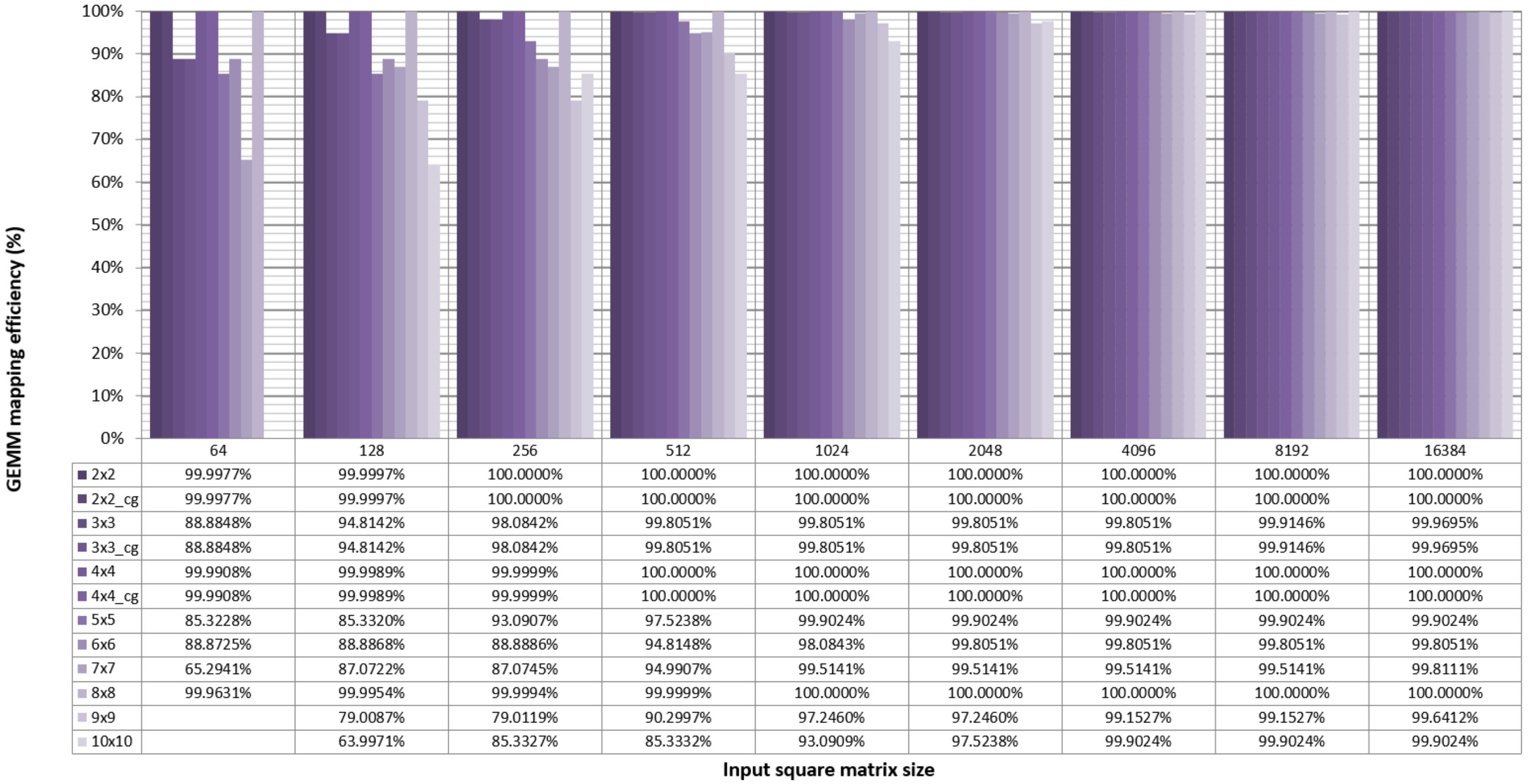
Input square matrix size

Figure B.3: GEMV mapping efficiency up for various architectures (N=2..8) and data sizes (64..16384). [139]



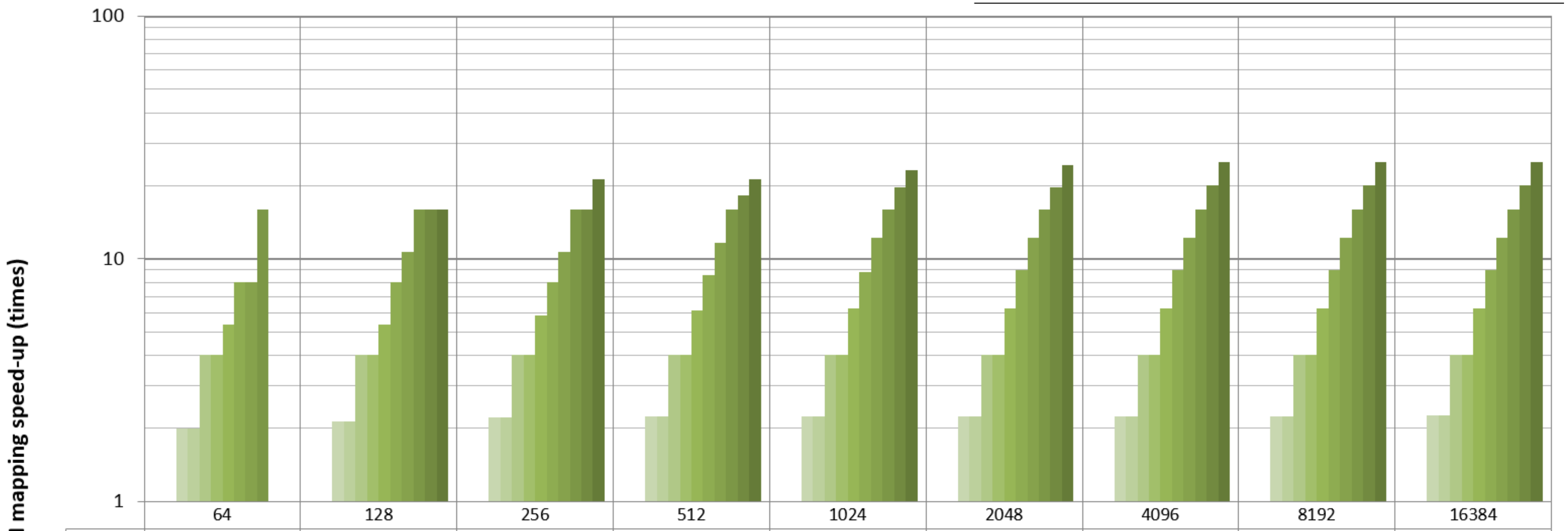
	64	128	256	512	1024	2048	4096	8192	16384
2x2	1	1	1	1	1	1	1	1	1
2x2_cg	1	1	1	1	1	1	1	1	1
3x3	1.697819315	1.940312213	2.096497858	2.186362132	2.215496776	2.230429558	2.237989947	2.244249393	2.247394321
3x3_cg	1.697819315	1.940312213	2.096497858	2.186362132	2.215496776	2.230429558	2.237989947	2.244249393	2.247394321
4x4	2.823834197	3.296411856	3.609978308	3.793796669	3.893853757	3.946131715	3.972862591	3.986379904	3.993177032
4x4_cg	2.823834197	3.296411856	3.609978308	3.793796669	3.893853757	3.946131715	3.972862591	3.986379904	3.993177032
5x5	2.823834197	3.662045061	4.640825432	5.374288039	5.840694216	6.035236268	6.137709529	6.190328498	6.216994481
6x6	2.823834197	4.706013363	5.905606813	7.165328705	8.030820873	8.549877118	8.760651373	8.870121552	8.925923818
7x7	2.422222222	4.706013363	6.495706479	8.598021349	10.27927506	11.15195839	11.64766819	11.91277521	12.08554173
8x8	2.422222222	5.488311688	8.11804878	10.7468272	12.84884289	14.24964058	15.07342836	15.52270041	15.75768538
9x9		4.118908382	7.216825672	10.7468272	14.2763372	16.54789731	18.30342055	19.1494913	19.69710441
10x10		4.118908382	8.11804878	11.72346468	16.06066154	19.73006573	22.28238279	23.55166085	24.24258676

Figure B.4: DOT mapping-based speed-up for various architectures (N=2..8) and data sizes (64..16384). [139]



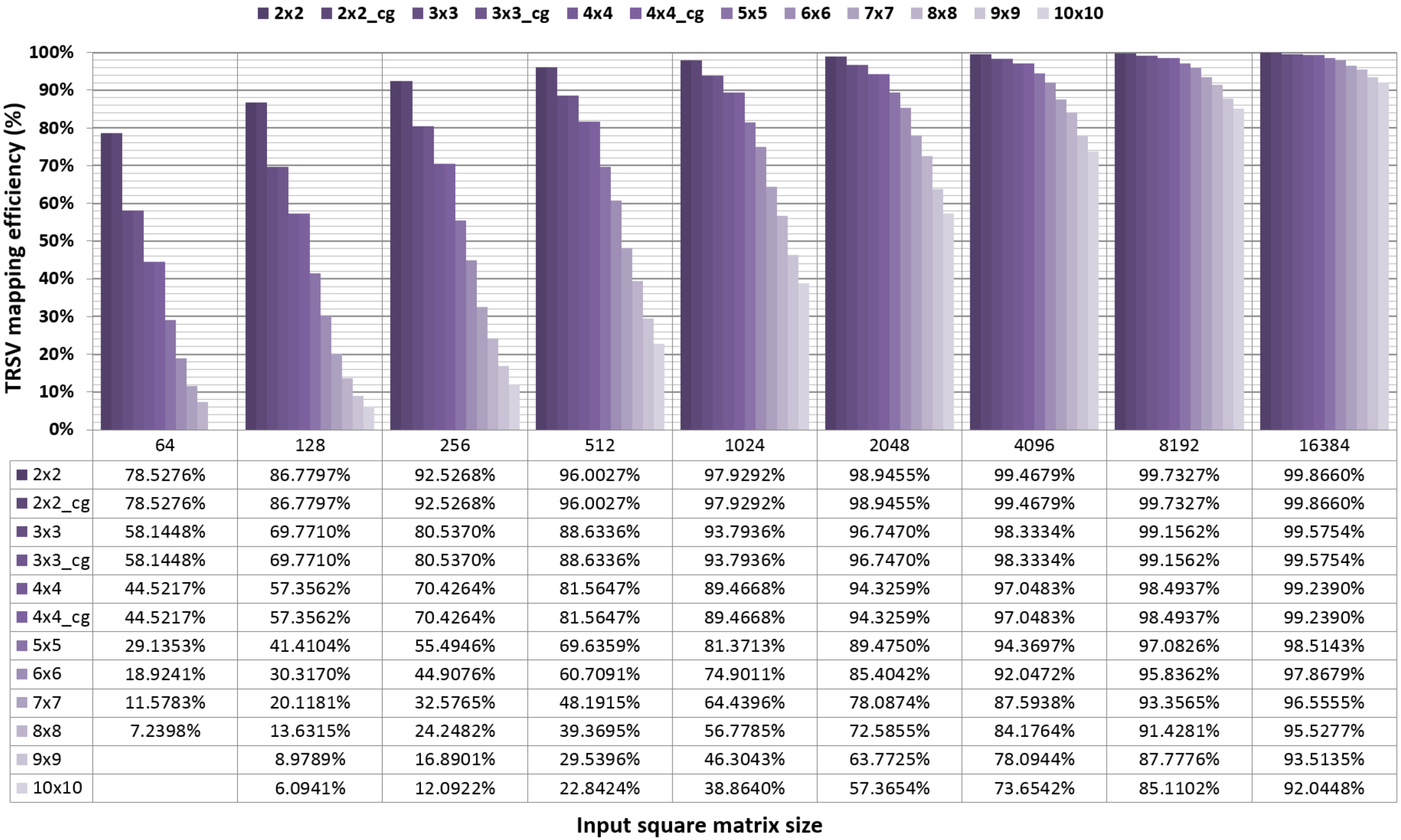
	64	128	256	512	1024	2048	4096	8192	16384
■ 2x2	99.9977%	99.9997%	100.0000%	100.0000%	100.0000%	100.0000%	100.0000%	100.0000%	100.0000%
■ 2x2_cg	99.9977%	99.9997%	100.0000%	100.0000%	100.0000%	100.0000%	100.0000%	100.0000%	100.0000%
■ 3x3	88.8848%	94.8142%	98.0842%	99.8051%	99.8051%	99.8051%	99.8051%	99.9146%	99.9695%
■ 3x3_cg	88.8848%	94.8142%	98.0842%	99.8051%	99.8051%	99.8051%	99.8051%	99.9146%	99.9695%
■ 4x4	99.9908%	99.9989%	99.9999%	100.0000%	100.0000%	100.0000%	100.0000%	100.0000%	100.0000%
■ 4x4_cg	99.9908%	99.9989%	99.9999%	100.0000%	100.0000%	100.0000%	100.0000%	100.0000%	100.0000%
■ 5x5	85.3228%	85.3320%	93.0907%	97.5238%	99.9024%	99.9024%	99.9024%	99.9024%	99.9024%
■ 6x6	88.8725%	88.8868%	88.8886%	94.8148%	98.0843%	99.8051%	99.8051%	99.8051%	99.8051%
■ 7x7	65.2941%	87.0722%	87.0745%	94.9907%	99.5141%	99.5141%	99.5141%	99.5141%	99.8111%
■ 8x8	99.9631%	99.9954%	99.9994%	99.9999%	100.0000%	100.0000%	100.0000%	100.0000%	100.0000%
■ 9x9		79.0087%	79.0119%	90.2997%	97.2460%	97.2460%	99.1527%	99.1527%	99.6412%
■ 10x10		63.9971%	85.3327%	85.3332%	93.0909%	97.5238%	99.9024%	99.9024%	99.9024%

Figure B.5: GEMM mapping efficiency up for various architectures (N=2..8) and data sizes (64..16384).



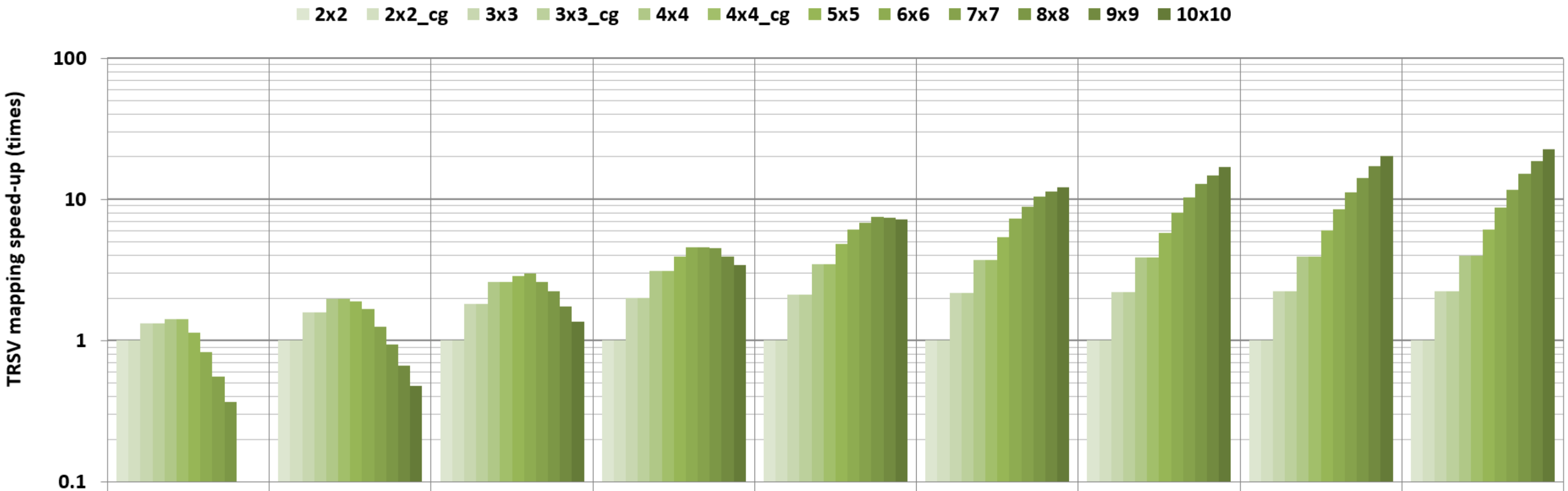
	64	128	256	512	1024	2048	4096	8192	16384
2x2	1	1	1	1	1	1	1	1	1
2x2_cg	1	1	1	1	1	1	1	1	1
3x3	1.999953865	2.133326389	2.206895597	2.24561391	2.245614019	2.245614033	2.245614035	2.248079034	2.249313564
3x3_cg	1.999953865	2.133326389	2.206895597	2.24561391	2.245614019	2.245614033	2.245614035	2.248079034	2.249313564
4x4	3.999723205	3.999965533	3.9999957	3.999999463	3.999999933	3.999999992	3.999999999	4	4
4x4_cg	3.999723205	3.999965533	3.9999957	3.999999463	3.999999933	3.999999992	3.999999999	4	4
5x5	5.332800262	5.333266954	5.818171773	6.095236706	6.243902256	6.243902416	6.243902436	6.243902439	6.243902439
6x6	7.998708408	7.999839158	7.999979934	8.533330457	8.827585821	8.98245609	8.982456134	8.98245614	8.98245614
7x7	7.998708408	10.66637052	10.66662972	11.6363581	12.19047543	12.1904761	12.19047618	12.19047619	12.22686567
8x8	15.99446563	15.99931069	15.999914	15.99998926	15.99999866	15.99999983	15.99999998	16	16
9x9		15.99931069	15.999914	18.28570014	19.69230563	19.69230744	20.07843134	20.07843137	20.1773399
10x10		15.99931069	21.3331779	21.33330544	23.27272437	24.38095198	24.9756097	24.97560975	24.97560976

Figure B.6: GEMM mapping-based speed-up for various architectures (N=2..8) and data sizes (64..16384).



	64	128	256	512	1024	2048	4096	8192	16384
■ 2x2	78.5276%	86.7797%	92.5268%	96.0027%	97.9292%	98.9455%	99.4679%	99.7327%	99.8660%
■ 2x2_cg	78.5276%	86.7797%	92.5268%	96.0027%	97.9292%	98.9455%	99.4679%	99.7327%	99.8660%
■ 3x3	58.1448%	69.7710%	80.5370%	88.6336%	93.7936%	96.7470%	98.3334%	99.1562%	99.5754%
■ 3x3_cg	58.1448%	69.7710%	80.5370%	88.6336%	93.7936%	96.7470%	98.3334%	99.1562%	99.5754%
■ 4x4	44.5217%	57.3562%	70.4264%	81.5647%	89.4668%	94.3259%	97.0483%	98.4937%	99.2390%
■ 4x4_cg	44.5217%	57.3562%	70.4264%	81.5647%	89.4668%	94.3259%	97.0483%	98.4937%	99.2390%
■ 5x5	29.1353%	41.4104%	55.4946%	69.6359%	81.3713%	89.4750%	94.3697%	97.0826%	98.5143%
■ 6x6	18.9241%	30.3170%	44.9076%	60.7091%	74.9011%	85.4042%	92.0472%	95.8362%	97.8679%
■ 7x7	11.5783%	20.1181%	32.5765%	48.1915%	64.4396%	78.0874%	87.5938%	93.3565%	96.5555%
■ 8x8	7.2398%	13.6315%	24.2482%	39.3695%	56.7785%	72.5855%	84.1764%	91.4281%	95.5277%
■ 9x9		8.9789%	16.8901%	29.5396%	46.3043%	63.7725%	78.0944%	87.7776%	93.5135%
■ 10x10		6.0941%	12.0922%	22.8424%	38.8640%	57.3654%	73.6542%	85.1102%	92.0448%

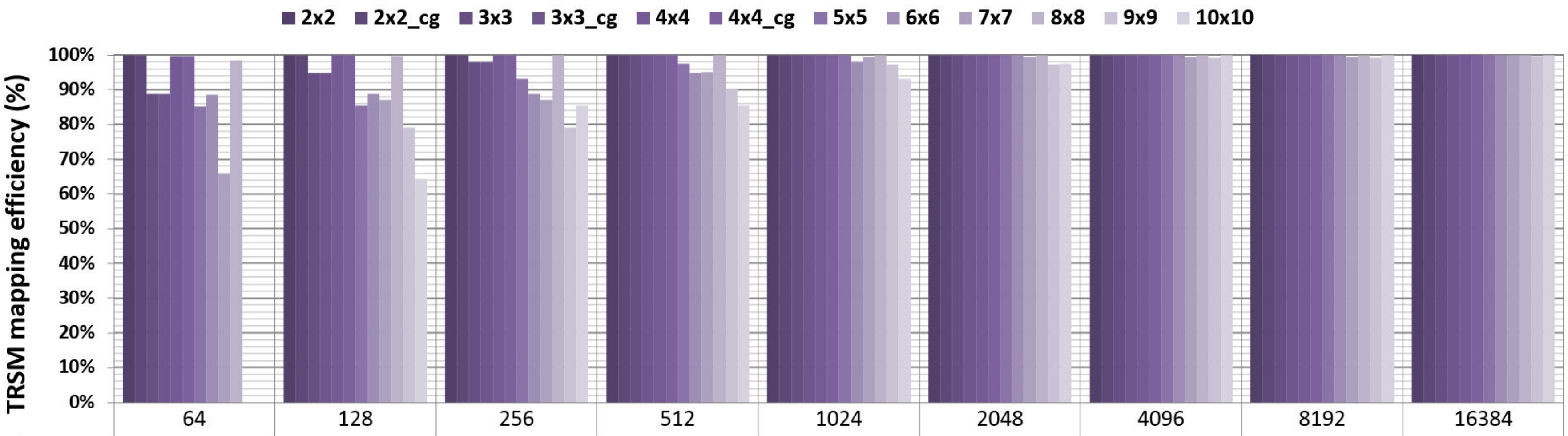
Figure B.7: TRSV mapping efficiency up for various architectures (N=2..8) and data sizes (64..16384).



	64	128	256	512	1024	2048	4096	8192	16384
2x2	1	1	1	1	1	1	1	1	1
2x2_cg	1	1	1	1	1	1	1	1	1
3x3	1.332788226	1.588393658	1.824299428	2.001478407	2.114321266	2.178892672	2.213571388	2.231557126	2.24072038
3x3_cg	1.332788226	1.588393658	1.824299428	2.001478407	2.114321266	2.178892672	2.213571388	2.231557126	2.24072038
4x4	1.417391304	1.982823002	2.587895461	3.115231436	3.493126044	3.726578692	3.857669841	3.92734052	3.963284627
4x4_cg	1.417391304	1.982823002	2.587895461	3.115231436	3.493126044	3.726578692	3.857669841	3.92734052	3.963284627
5x5	1.131158917	1.883646683	2.864058456	3.91121729	4.805139716	5.430762268	5.810951161	6.022295484	6.133988104
6x6	0.834186285	1.664576803	2.970318914	4.580817283	6.129272252	7.314672767	8.077142061	8.515578598	8.751619514
7x7	0.555744971	1.262182077	2.595392468	4.58589017	6.871477316	8.892172943	10.33522221	11.22075105	11.71533301
8x8	0.368778281	0.942492013	2.227563344	4.510957324	7.537288605	10.51479709	12.79357241	14.25100826	15.08419764
9x9		0.667504714	1.7335299	3.935267483	7.387169524	11.35682613	14.79084572	17.17800161	18.61234436
10x10		0.478809739	1.3546947	3.443792193	7.246002132	12.22058218	16.93095973	20.38108862	22.51488005

Input square matrix size

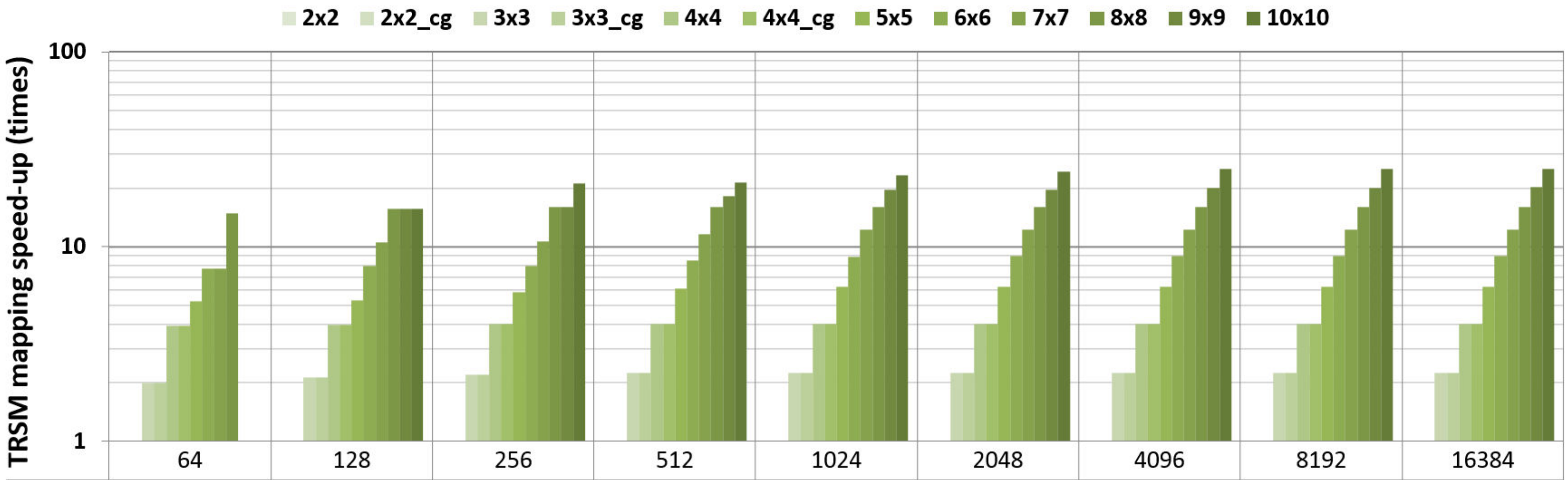
Figure B.8: TRSV mapping-based speed-up for various architectures (N=2..8) and data sizes (64..16384).



	64	128	256	512	1024	2048	4096	8192	16384
2x2	99.8740%	99.9690%	99.9923%	99.9981%	99.9995%	99.9999%	100.0000%	100.0000%	100.0000%
2x2_cg	99.8740%	99.9690%	99.9923%	99.9981%	99.9995%	99.9999%	100.0000%	100.0000%	100.0000%
3x3	88.7734%	94.7694%	98.0705%	99.8013%	99.8041%	99.8048%	99.8050%	99.9146%	99.9695%
3x3_cg	88.7734%	94.7694%	98.0705%	99.8013%	99.8041%	99.8048%	99.8050%	99.9146%	99.9695%
4x4	99.5751%	99.8945%	99.9738%	99.9935%	99.9984%	99.9996%	99.9999%	100.0000%	100.0000%
4x4_cg	99.5751%	99.8945%	99.9738%	99.9935%	99.9984%	99.9996%	99.9999%	100.0000%	100.0000%
5x5	85.1574%	85.2911%	93.0659%	97.5152%	99.9000%	99.9018%	99.9023%	99.9024%	99.9024%
6x6	88.5027%	88.7935%	88.8654%	94.8047%	98.0811%	99.8042%	99.8049%	99.8050%	99.8051%
7x7	65.8199%	86.9734%	87.0499%	94.9770%	99.5094%	99.5129%	99.5138%	99.5140%	99.8111%
8x8	98.4615%	99.6020%	99.9002%	99.9751%	99.9938%	99.9984%	99.9996%	99.9999%	100.0000%
9x9		79.0194%	79.0150%	90.2850%	97.2394%	97.2443%	99.1523%	99.1526%	99.6412%
10x10		64.2969%	85.2962%	85.3242%	93.0850%	97.5217%	99.9018%	99.9023%	99.9024%

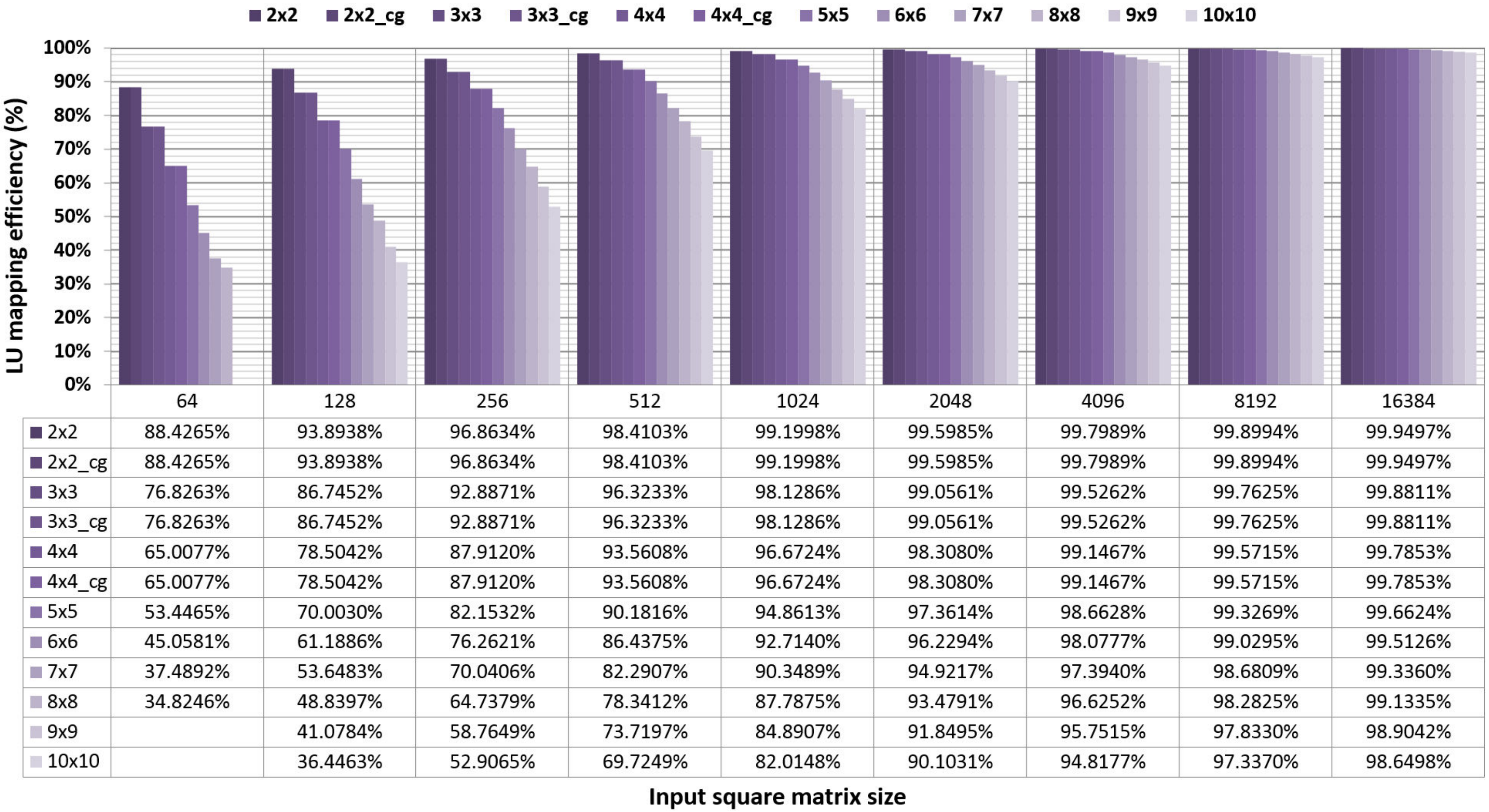
Input square matrix size

Figure B.9: TRSM mapping efficiency up for various architectures (N=2..8) and data sizes (64..16384).



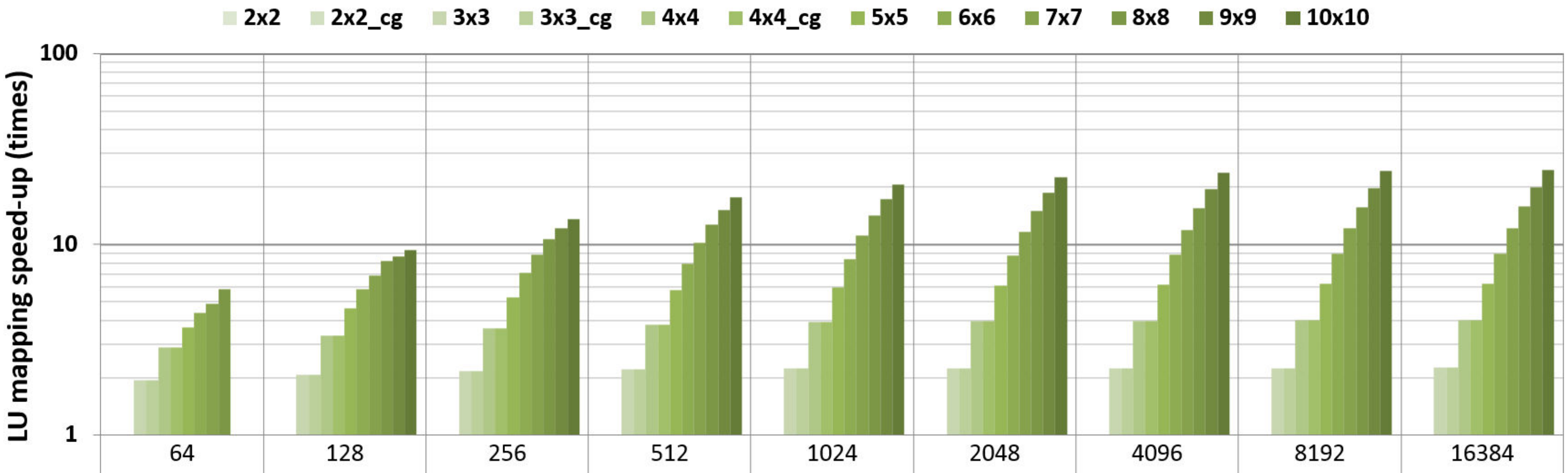
	64	128	256	512	1024	2048	4096	8192	16384
2x2	1	1	1	1	1	1	1	1	1
2x2_cg	1	1	1	1	1	1	1	1	1
3x3	1.990241397	2.130376028	2.206082374	2.245400415	2.245560654	2.245600693	2.2456107	2.248078198	2.249313355
3x3_cg	1.990241397	2.130376028	2.206082374	2.245400415	2.245560654	2.245600693	2.2456107	2.248078198	2.249313355
4x4	3.942014242	3.985355633	3.996333849	3.999083683	3.999771002	3.999942764	3.999985693	3.999996423	3.999999106
4x4_cg	3.942014242	3.985355633	3.996333849	3.999083683	3.999771002	3.999942764	3.999985693	3.999996423	3.999999106
5x5	5.222371968	5.305175184	5.809622122	6.092866994	6.243277636	6.24374627	6.243863402	6.24389268	6.243899999
6x6	7.734530938	7.931991599	7.982912179	8.528426289	8.826267705	8.982114153	8.982370653	8.98243477	8.982450798
7x7	7.734530938	10.54184907	10.63522896	11.62691821	12.18787332	12.18982555	12.19031355	12.19043553	12.22685544
8x8	14.90384615	15.7113399	15.92694482	15.98169044	15.99542108	15.99885534	15.99971386	15.99992847	15.99998212
9x9		15.7113399	15.92694482	18.26160468	19.68528538	19.69055213	20.07797467	20.0783172	20.17731107
10x10		15.7113399	21.20150687	21.30025396	23.26283919	24.37823369	24.97489584	24.97543129	24.97556514

Figure B.10: TRSM mapping-based speed-up for various architectures (N=2..8) and data sizes (64..16384).



	64	128	256	512	1024	2048	4096	8192	16384
■ 2x2	88.4265%	93.8938%	96.8634%	98.4103%	99.1998%	99.5985%	99.7989%	99.8994%	99.9497%
■ 2x2_cg	88.4265%	93.8938%	96.8634%	98.4103%	99.1998%	99.5985%	99.7989%	99.8994%	99.9497%
■ 3x3	76.8263%	86.7452%	92.8871%	96.3233%	98.1286%	99.0561%	99.5262%	99.7625%	99.8811%
■ 3x3_cg	76.8263%	86.7452%	92.8871%	96.3233%	98.1286%	99.0561%	99.5262%	99.7625%	99.8811%
■ 4x4	65.0077%	78.5042%	87.9120%	93.5608%	96.6724%	98.3080%	99.1467%	99.5715%	99.7853%
■ 4x4_cg	65.0077%	78.5042%	87.9120%	93.5608%	96.6724%	98.3080%	99.1467%	99.5715%	99.7853%
■ 5x5	53.4465%	70.0030%	82.1532%	90.1816%	94.8613%	97.3614%	98.6628%	99.3269%	99.6624%
■ 6x6	45.0581%	61.1886%	76.2621%	86.4375%	92.7140%	96.2294%	98.0777%	99.0295%	99.5126%
■ 7x7	37.4892%	53.6483%	70.0406%	82.2907%	90.3489%	94.9217%	97.3940%	98.6809%	99.3360%
■ 8x8	34.8246%	48.8397%	64.7379%	78.3412%	87.7875%	93.4791%	96.6252%	98.2825%	99.1335%
■ 9x9		41.0784%	58.7649%	73.7197%	84.8907%	91.8495%	95.7515%	97.8330%	98.9042%
■ 10x10		36.4463%	52.9065%	69.7249%	82.0148%	90.1031%	94.8177%	97.3370%	98.6498%

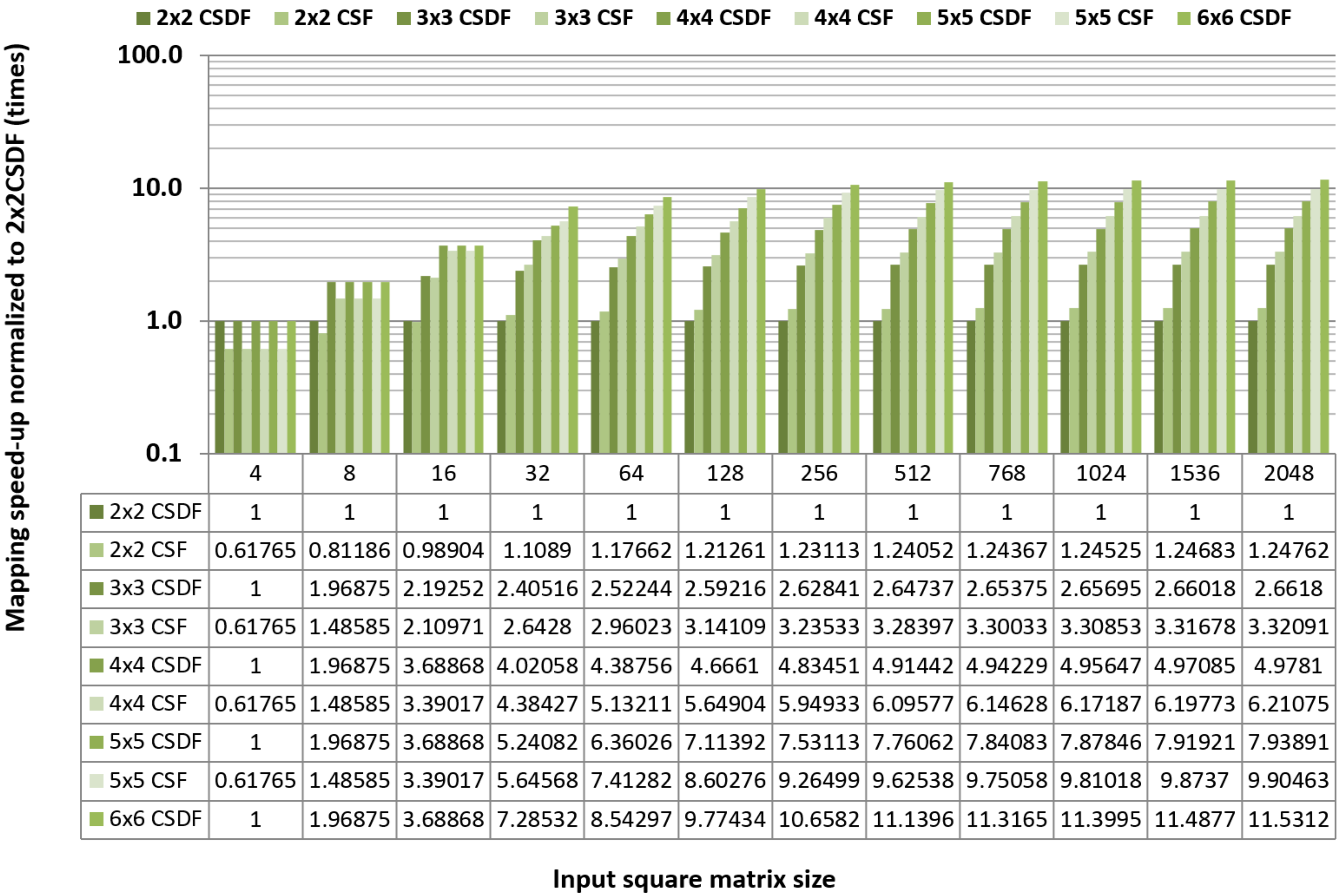
Figure B.11: LU mapping efficiency up for various architectures (N=2..8) and data sizes (64..16384). [140]



	64	128	256	512	1024	2048	4096	8192	16384
2x2	1	1	1	1	1	1	1	1	1
2x2_cg	1	1	1	1	1	1	1	1	1
3x3	1.940593268	2.074887089	2.156649314	2.20202995	2.225640552	2.237729357	2.243846505	2.246916286	2.248456471
3x3_cg	1.940593268	2.074887089	2.156649314	2.20202995	2.225640552	2.237729357	2.243846505	2.246916286	2.248456471
4x4	2.88974878	3.329712437	3.626360717	3.801838717	3.897823202	3.948101421	3.9738434	3.98686926	3.993421443
4x4_cg	2.88974878	3.329712437	3.626360717	3.801838717	3.897823202	3.948101421	3.9738434	3.98686926	3.993421443
5x5	3.66465348	4.624071285	5.290654175	5.724639863	5.975940063	6.109435232	6.178802868	6.214171604	6.232031124
6x6	4.380256271	5.797004311	7.065116279	7.899246894	8.410033316	8.695160507	8.84467388	8.921608443	8.960638678
7x7	4.871738258	6.885554102	8.821409257	10.23289371	11.15414268	11.67402834	11.95460978	12.10054206	12.17477274
8x8	5.791219279	8.143172877	10.63496944	12.71959367	14.15444548	15.01565055	15.49084512	15.74095485	15.86933095
9x9		8.615798335	12.19909237	15.14267448	17.32140915	18.67242904	19.42821213	19.83099622	20.03814825
10x10		9.373785173	13.5357712	17.67392759	20.65774574	22.61347228	23.75136672	24.35854196	24.67482028

Input square matrix size

Figure B.12: LU mapping-based speed-up for various architectures (N=2..8) and data sizes (64..16384). [140]

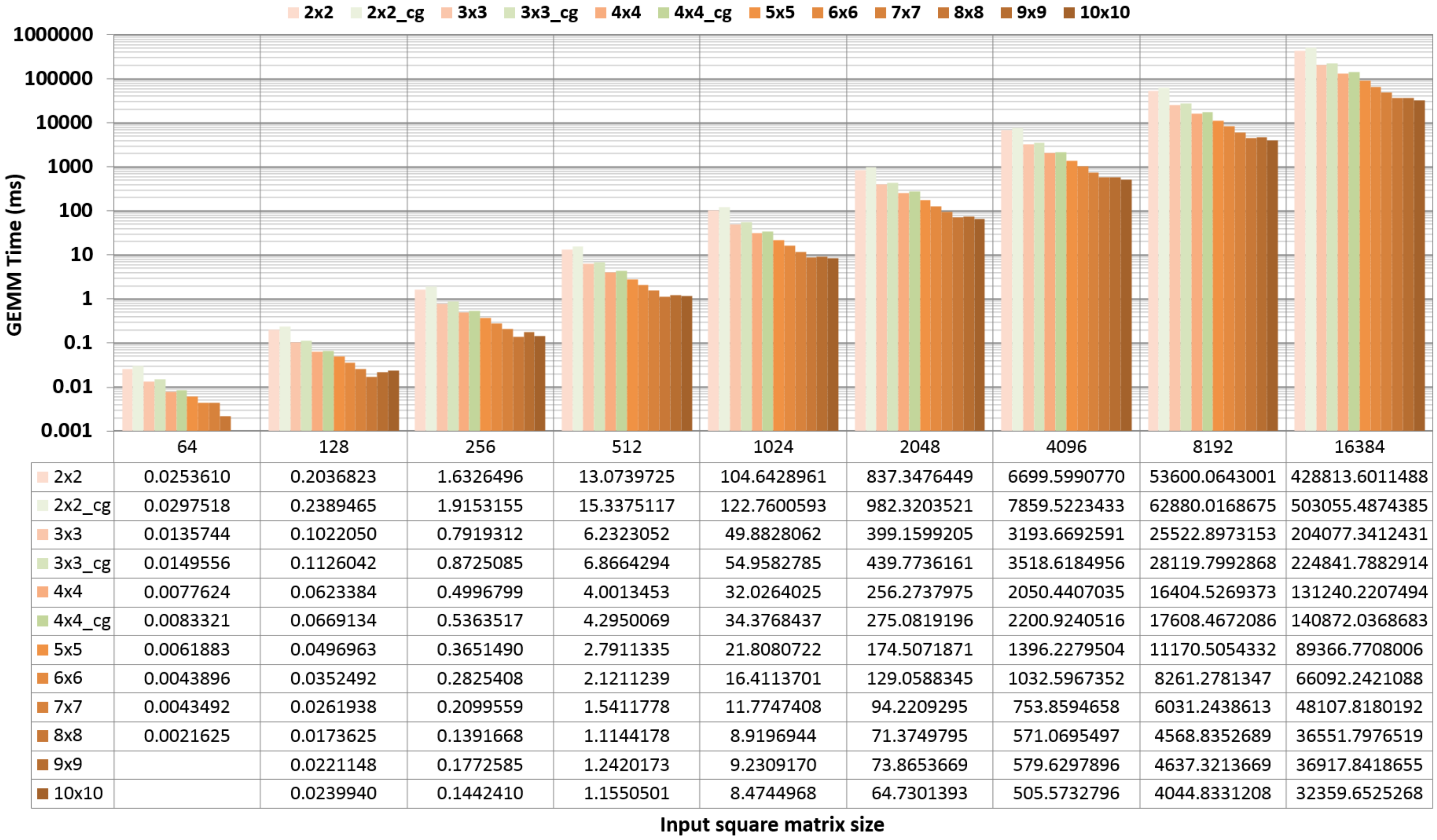


	4	8	16	32	64	128	256	512	768	1024	1536	2048
2x2 CSDF	1	1	1	1	1	1	1	1	1	1	1	1
2x2 CSF	0.61765	0.81186	0.98904	1.1089	1.17662	1.21261	1.23113	1.24052	1.24367	1.24525	1.24683	1.24762
3x3 CSDF	1	1.96875	2.19252	2.40516	2.52244	2.59216	2.62841	2.64737	2.65375	2.65695	2.66018	2.6618
3x3 CSF	0.61765	1.48585	2.10971	2.6428	2.96023	3.14109	3.23533	3.28397	3.30033	3.30853	3.31678	3.32091
4x4 CSDF	1	1.96875	3.68868	4.02058	4.38756	4.6661	4.83451	4.91442	4.94229	4.95647	4.97085	4.9781
4x4 CSF	0.61765	1.48585	3.39017	4.38427	5.13211	5.64904	5.94933	6.09577	6.14628	6.17187	6.19773	6.21075
5x5 CSDF	1	1.96875	3.68868	5.24082	6.36026	7.11392	7.53113	7.76062	7.84083	7.87846	7.91921	7.93891
5x5 CSF	0.61765	1.48585	3.39017	5.64568	7.41282	8.60276	9.26499	9.62538	9.75058	9.81018	9.8737	9.90463
6x6 CSDF	1	1.96875	3.68868	7.28532	8.54297	9.77434	10.6582	11.1396	11.3165	11.3995	11.4877	11.5312

Figure B.13: GR mapping-based speed-up for various architectures tailored for the two CSF and CSDF algorithms (N=2..6) and data sizes (4..2048). [137]

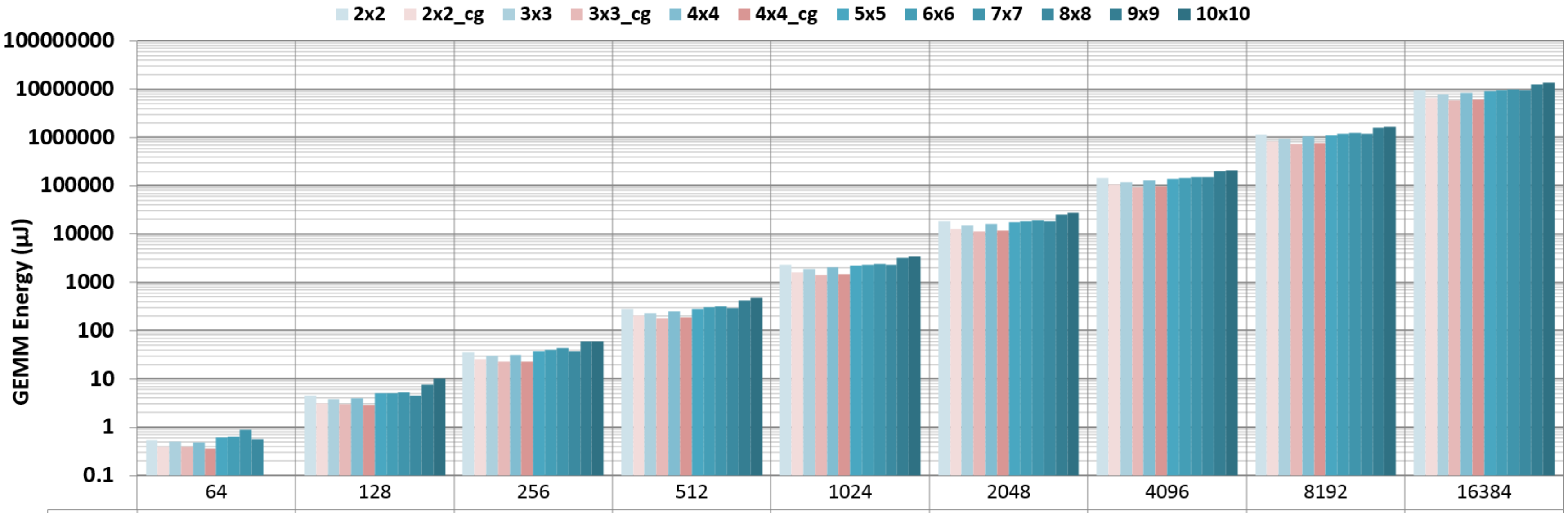
Appendix C

Detailed Timing and Energy Data on *Layers* Running NLA Kernels



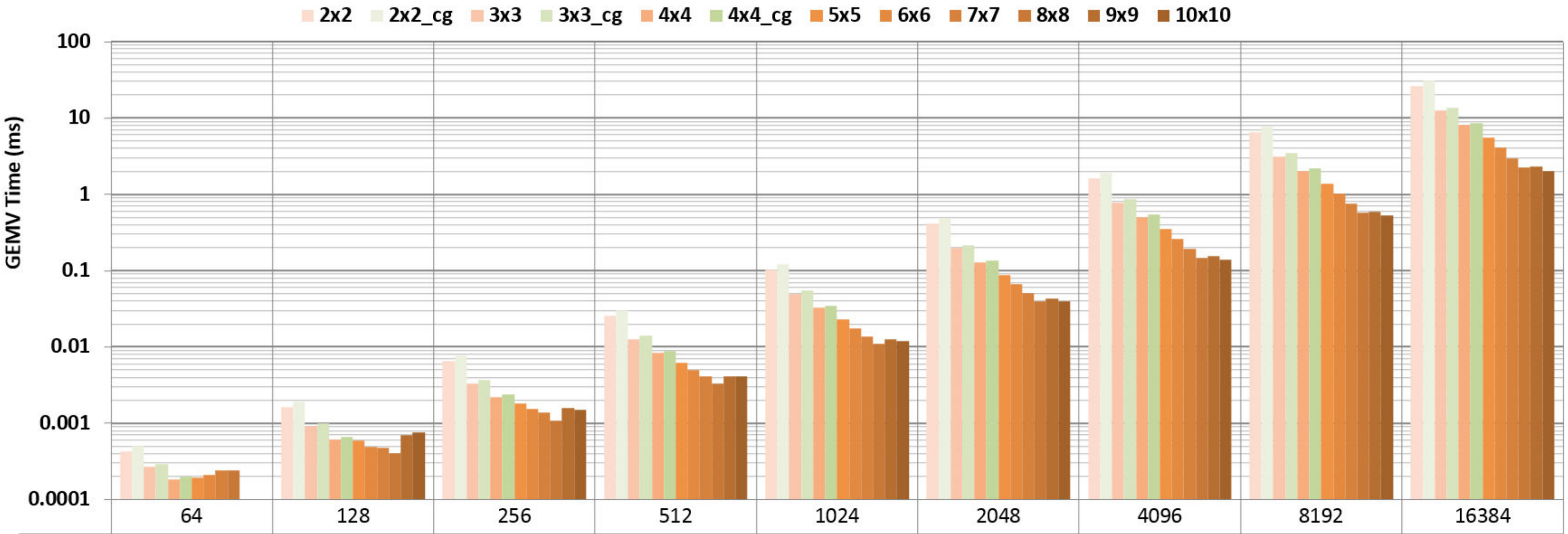
	64	128	256	512	1024	2048	4096	8192	16384
2x2	0.0253610	0.2036823	1.6326496	13.0739725	104.6428961	837.3476449	6699.5990770	53600.0643001	428813.6011488
2x2_cg	0.0297518	0.2389465	1.9153155	15.3375117	122.7600593	982.3203521	7859.5223433	62880.0168675	503055.4874385
3x3	0.0135744	0.1022050	0.7919312	6.2323052	49.8828062	399.1599205	3193.6692591	25522.8973153	204077.3412431
3x3_cg	0.0149556	0.1126042	0.8725085	6.8664294	54.9582785	439.7736161	3518.6184956	28119.7992868	224841.7882914
4x4	0.0077624	0.0623384	0.4996799	4.0013453	32.0264025	256.2737975	2050.4407035	16404.5269373	131240.2207494
4x4_cg	0.0083321	0.0669134	0.5363517	4.2950069	34.3768437	275.0819196	2200.9240516	17608.4672086	140872.0368683
5x5	0.0061883	0.0496963	0.3651490	2.7911335	21.8080722	174.5071871	1396.2279504	11170.5054332	89366.7708006
6x6	0.0043896	0.0352492	0.2825408	2.1211239	16.4113701	129.0588345	1032.5967352	8261.2781347	66092.2421088
7x7	0.0043492	0.0261938	0.2099559	1.5411778	11.7747408	94.2209295	753.8594658	6031.2438613	48107.8180192
8x8	0.0021625	0.0173625	0.1391668	1.1144178	8.9196944	71.3749795	571.0695497	4568.8352689	36551.7976519
9x9		0.0221148	0.1772585	1.2420173	9.2309170	73.8653669	579.6297896	4637.3213669	36917.8418655
10x10		0.0239940	0.1442410	1.1550501	8.4744968	64.7301393	505.5732796	4044.8331208	32359.6525268

Figure C.1: GEMM timing results for various architectures (N=2..8) and data sizes (64..16384). [139]



	64	128	256	512	1024	2048	4096	8192	16384
2x2	0.55160	4.43007	35.50996	284.35759	2275.97253	18212.22754	145715.60997	1165796.03852	9326652.94363
2x2_cg	0.39224	3.15018	25.25077	202.20372	1618.42036	12950.52537	103616.85299	828985.42418	6632085.79476
3x3	0.50840	3.82785	29.65992	233.41635	1868.24328	14949.59680	119611.37698	955900.14013	7643237.23451
3x3_cg	0.38938	2.93172	22.71633	178.77195	1430.87450	11449.79195	91609.51964	732117.25230	5853902.10533
4x4	0.49182	3.94973	31.65947	253.52323	2029.17685	16237.37967	129914.89776	1039382.62448	8315314.76657
4x4_cg	0.36034	2.89385	23.19593	185.74874	1486.71599	11896.63283	95184.68301	761523.94645	6092377.50167
5x5	0.61957	4.97557	36.55855	279.44703	2183.41438	17471.58105	139789.71409	1118385.97724	8947360.87751
6x6	0.62943	5.05442	40.51380	304.15008	2353.24278	18505.87534	148065.07845	1184592.93302	9477032.68822
7x7	0.88603	5.33627	42.77268	313.97220	2398.77669	19194.89976	153577.94656	1228698.56972	9800632.92946
8x8	0.56740	4.55559	36.51474	292.40228	2340.36012	18727.45278	149837.91374	1198776.48045	9590504.53005
9x9		7.58363	60.78552	425.91287	3165.46837	25329.93007	198766.79237	1590231.40246	12659875.56198
10x10		10.02902	60.28978	482.78714	3542.16510	27055.86479	211319.21604	1690656.92092	13525668.14737

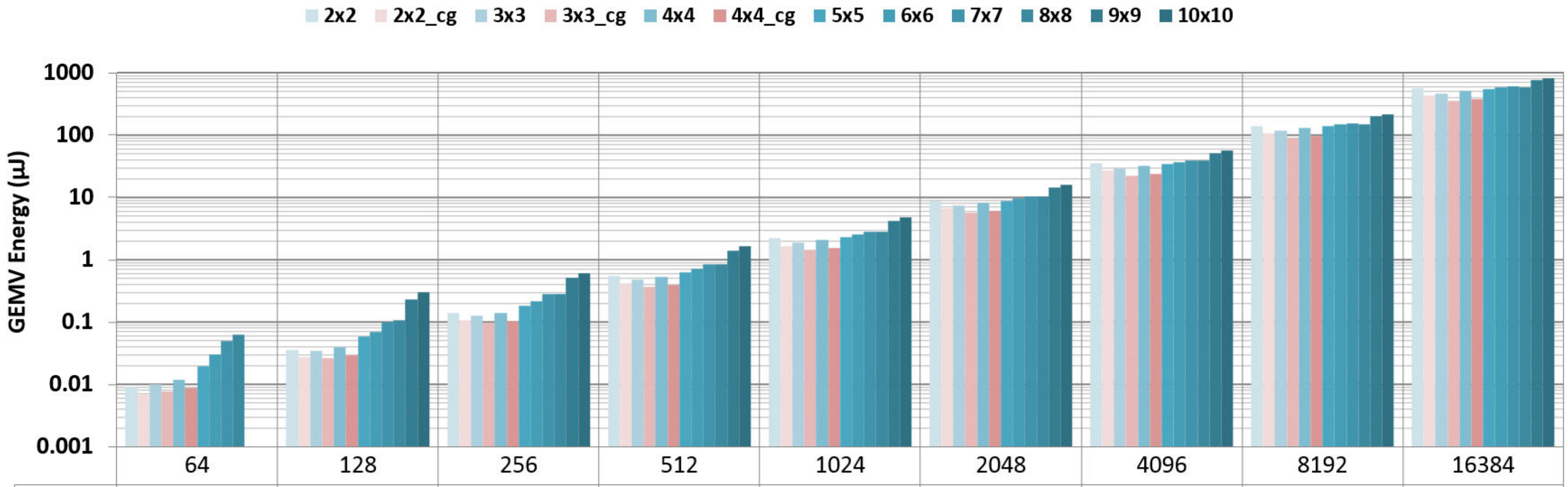
Figure C.2: GEMM energy results for various architectures (N=2..8) and data sizes (64..16384). [139]



	64	128	256	512	1024	2048	4096	8192	16384
■ 2x2	0.0004251	0.0016482	0.0064906	0.0257605	0.1026404	0.4097605	1.6374423	6.5465718	26.1798947
■ 2x2_cg	0.0004987	0.0019336	0.0076144	0.0302205	0.1204109	0.4807037	1.9209380	7.6800009	30.7125046
■ 3x3	0.0002680	0.0009093	0.0033141	0.0126127	0.0495934	0.1966608	0.7832206	3.1226194	12.4699541
■ 3x3_cg	0.0002953	0.0010018	0.0036513	0.0138960	0.0546394	0.2166707	0.8629117	3.4403395	13.7387461
■ 4x4	0.0001843	0.0006121	0.0022011	0.0083126	0.0322699	0.1271209	0.5045693	2.0104517	8.0261583
■ 4x4_cg	0.0001978	0.0006570	0.0023626	0.0089227	0.0346382	0.1364504	0.5416000	2.1580002	8.6152040
■ 5x5	0.0001959	0.0005857	0.0018199	0.0062373	0.0228674	0.0883486	0.3471549	1.3761439	5.4796275
■ 6x6	0.0002084	0.0004849	0.0015216	0.0049773	0.0176944	0.0663510	0.2587657	1.0217894	4.0606145
■ 7x7	0.0002407	0.0004804	0.0013706	0.0041098	0.0136968	0.0504012	0.1928365	0.7538123	2.9714199
■ 8x8	0.0002394	0.0004096	0.0010904	0.0032691	0.0108947	0.0392181	0.1481543	0.5751840	2.2658734
■ 9x9		0.0006951	0.0015623	0.0041640	0.0124892	0.0430149	0.1554051	0.5938659	2.3088577
■ 10x10		0.0007542	0.0015069	0.0041414	0.0120450	0.0391429	0.1385019	0.5238944	2.0353558

Input square matrix size

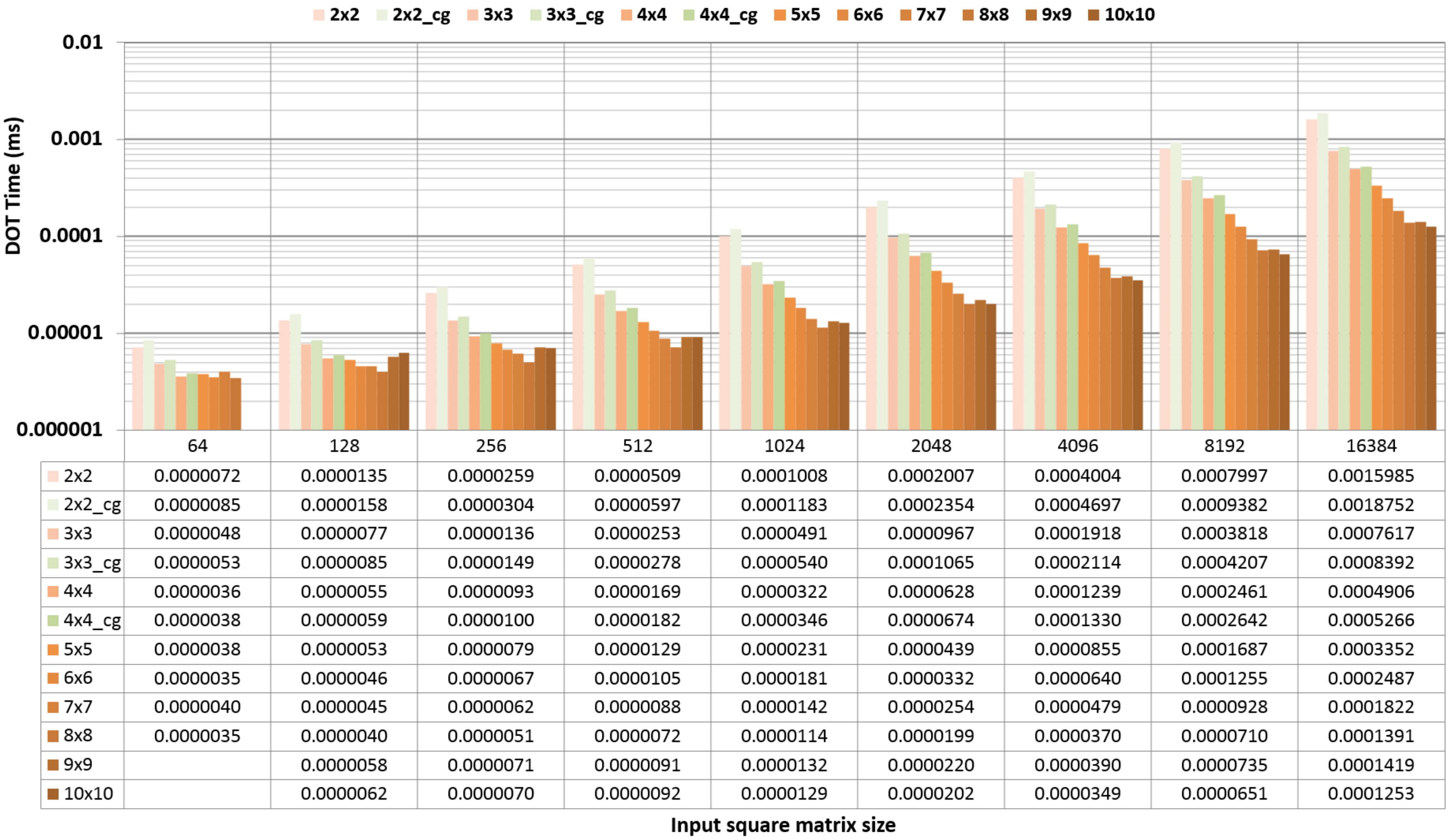
Figure C.3: GEMV timing results for various architectures (N=2..8) and data sizes (64..16384).



	64	128	256	512	1024	2048	4096	8192	16384
2x2	0.00909	0.03523	0.13875	0.55067	2.19408	8.75921	35.00262	139.94214	559.63190
2x2_cg	0.00694	0.02692	0.10602	0.42077	1.67652	6.69298	26.74579	106.93092	427.61927
3x3	0.01009	0.03423	0.12474	0.47474	1.86667	7.40221	29.47999	117.53368	469.36221
3x3_cg	0.00773	0.02621	0.09553	0.36357	1.42958	5.66894	22.57709	90.01253	359.45849
4x4	0.01181	0.03922	0.14104	0.53265	2.06774	8.14547	32.33109	128.82291	514.28894
4x4_cg	0.00871	0.02894	0.10408	0.39306	1.52586	6.01084	23.85829	95.06315	379.51266
5x5	0.01967	0.05880	0.18271	0.62620	2.29579	8.86981	34.85284	138.15886	550.13076
6x6	0.03002	0.06984	0.21915	0.71688	2.54850	9.55646	37.26973	147.16721	584.84584
7x7	0.04943	0.09864	0.28142	0.84381	2.81218	10.34825	39.59268	154.77075	610.08405
8x8	0.06272	0.10732	0.28571	0.85659	2.85464	10.27596	38.81953	150.71030	593.70642
9x9		0.23194	0.52130	1.38938	4.16724	14.35273	51.85383	198.15442	770.39344
10x10		0.30520	0.60981	1.67594	4.87433	15.84023	56.04854	212.00803	823.66167

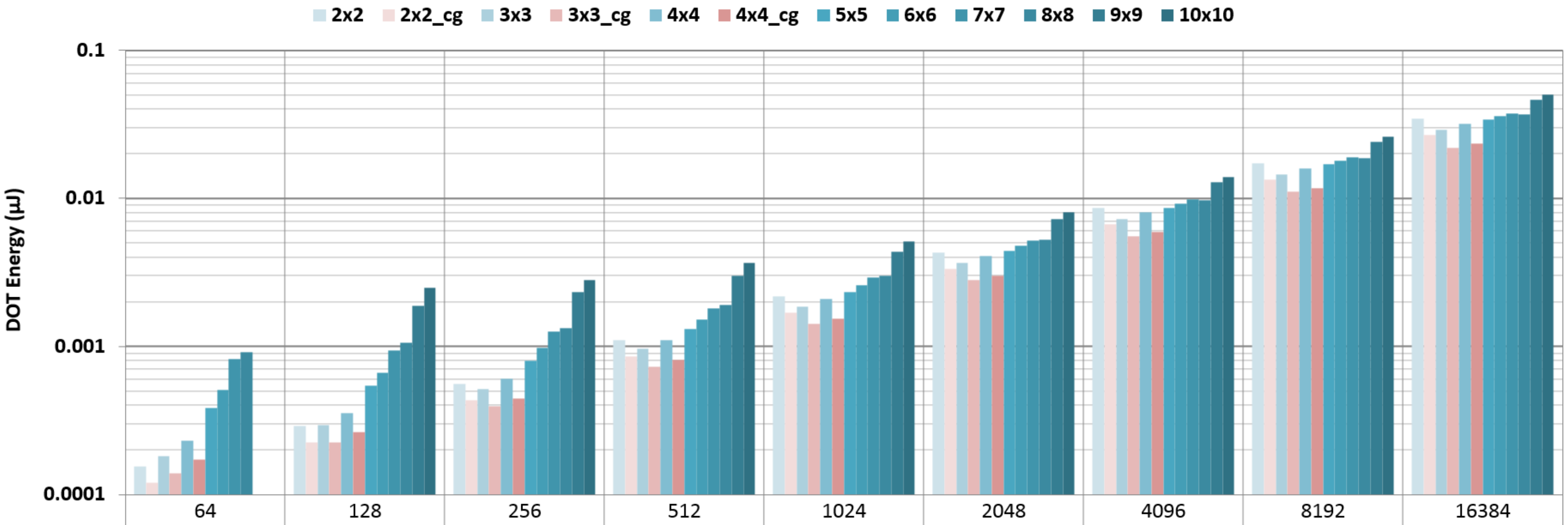
Input square matrix size

Figure C.4: GEMV energy results for various architectures (N=2..8) and data sizes (64..16384).



	64	128	256	512	1024	2048	4096	8192	16384
2x2	0.0000072	0.0000135	0.0000259	0.0000509	0.0001008	0.0002007	0.0004004	0.0007997	0.0015985
2x2_cg	0.0000085	0.0000158	0.0000304	0.0000597	0.0001183	0.0002354	0.0004697	0.0009382	0.0018752
3x3	0.0000048	0.0000077	0.0000136	0.0000253	0.0000491	0.0000967	0.0001918	0.0003818	0.0007617
3x3_cg	0.0000053	0.0000085	0.0000149	0.0000278	0.0000540	0.0001065	0.0002114	0.0004207	0.0008392
4x4	0.0000036	0.0000055	0.0000093	0.0000169	0.0000322	0.0000628	0.0001239	0.0002461	0.0004906
4x4_cg	0.0000038	0.0000059	0.0000100	0.0000182	0.0000346	0.0000674	0.0001330	0.0002642	0.0005266
5x5	0.0000038	0.0000053	0.0000079	0.0000129	0.0000231	0.0000439	0.0000855	0.0001687	0.0003352
6x6	0.0000035	0.0000046	0.0000067	0.0000105	0.0000181	0.0000332	0.0000640	0.0001255	0.0002487
7x7	0.0000040	0.0000045	0.0000062	0.0000088	0.0000142	0.0000254	0.0000479	0.0000928	0.0001822
8x8	0.0000035	0.0000040	0.0000051	0.0000072	0.0000114	0.0000199	0.0000370	0.0000710	0.0001391
9x9		0.0000058	0.0000071	0.0000091	0.0000132	0.0000220	0.0000390	0.0000735	0.0001419
10x10		0.0000062	0.0000070	0.0000092	0.0000129	0.0000202	0.0000349	0.0000651	0.0001253

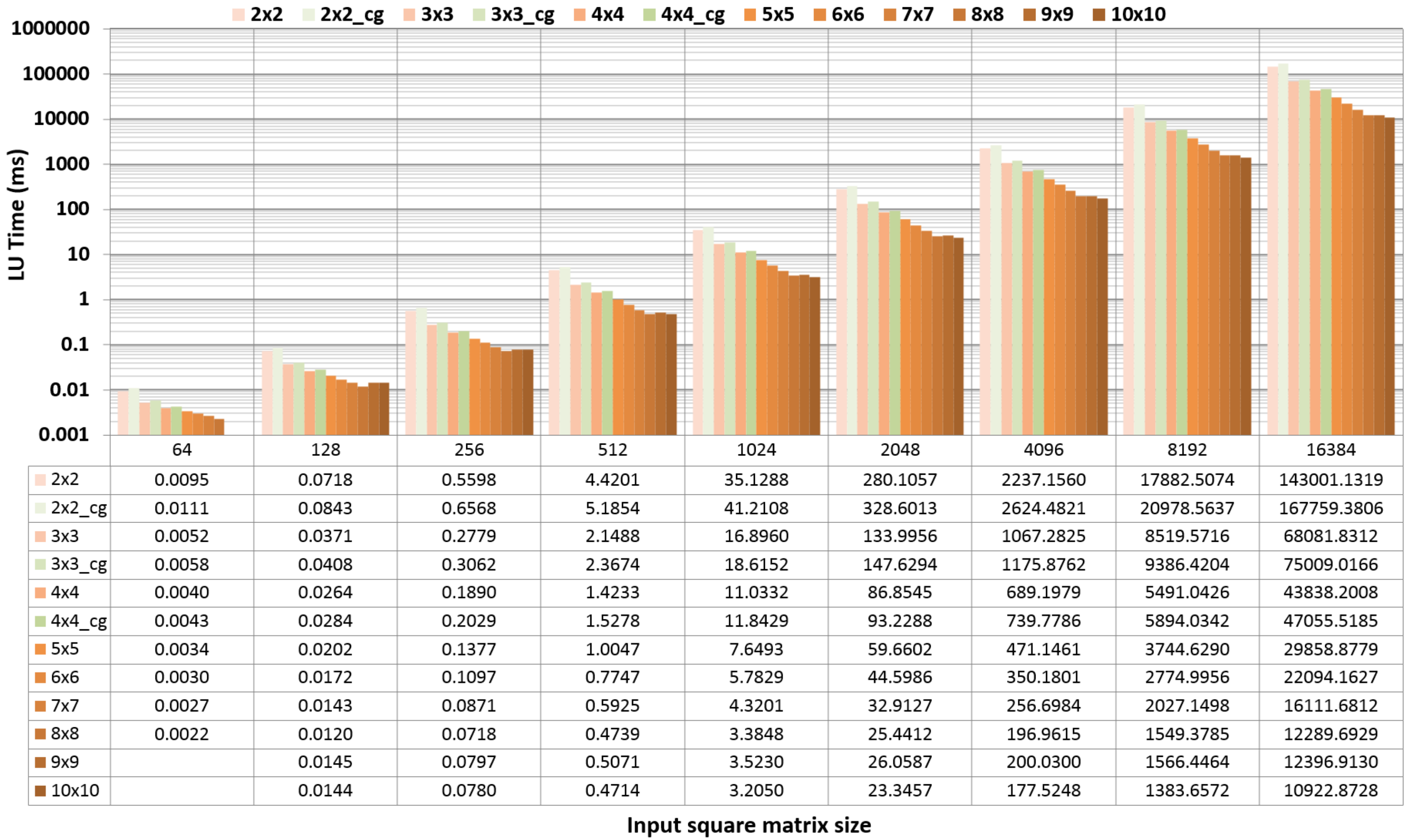
Figure C.5: DOT timing results for various architectures (N=2..8) and data sizes (64..16384).



	64	128	256	512	1024	2048	4096	8192	16384
2x2	0.00016	0.00029	0.00056	0.00109	0.00217	0.00432	0.00861	0.01720	0.03438
2x2_cg	0.00012	0.00022	0.00043	0.00085	0.00168	0.00335	0.00668	0.01334	0.02666
3x3	0.00018	0.00029	0.00051	0.00096	0.00186	0.00366	0.00727	0.01447	0.02887
3x3_cg	0.00014	0.00022	0.00039	0.00073	0.00142	0.00279	0.00554	0.01103	0.02200
4x4	0.00023	0.00035	0.00060	0.00109	0.00208	0.00405	0.00800	0.01589	0.03168
4x4_cg	0.00017	0.00026	0.00044	0.00081	0.00154	0.00300	0.00592	0.01176	0.02344
5x5	0.00038	0.00054	0.00080	0.00131	0.00234	0.00444	0.00865	0.01707	0.03391
6x6	0.00050	0.00066	0.00097	0.00151	0.00260	0.00477	0.00920	0.01805	0.03575
7x7	0.00082	0.00093	0.00126	0.00181	0.00291	0.00521	0.00982	0.01904	0.03737
8x8	0.00091	0.00105	0.00133	0.00189	0.00301	0.00525	0.00974	0.01871	0.03664
9x9		0.00189	0.00233	0.00300	0.00433	0.00721	0.01276	0.02408	0.04650
10x10		0.00249	0.00279	0.00367	0.00514	0.00807	0.01394	0.02597	0.05003

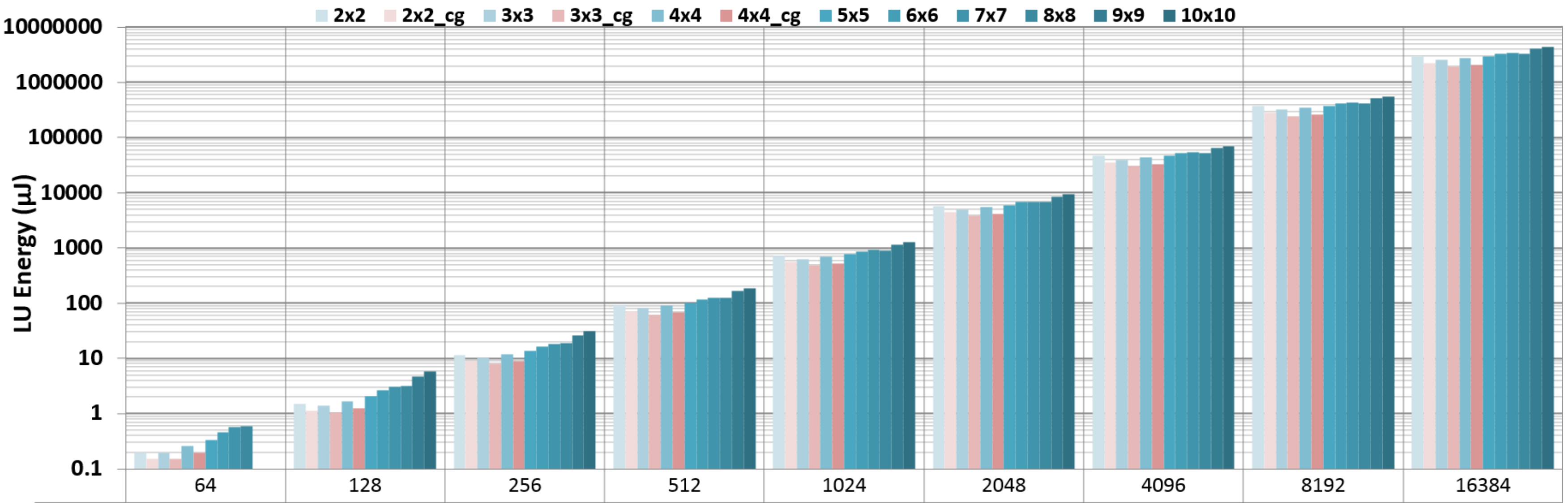
Input square matrix size

Figure C.6: DOT energy results for various architectures (N=2..8) and data sizes (64..16384).



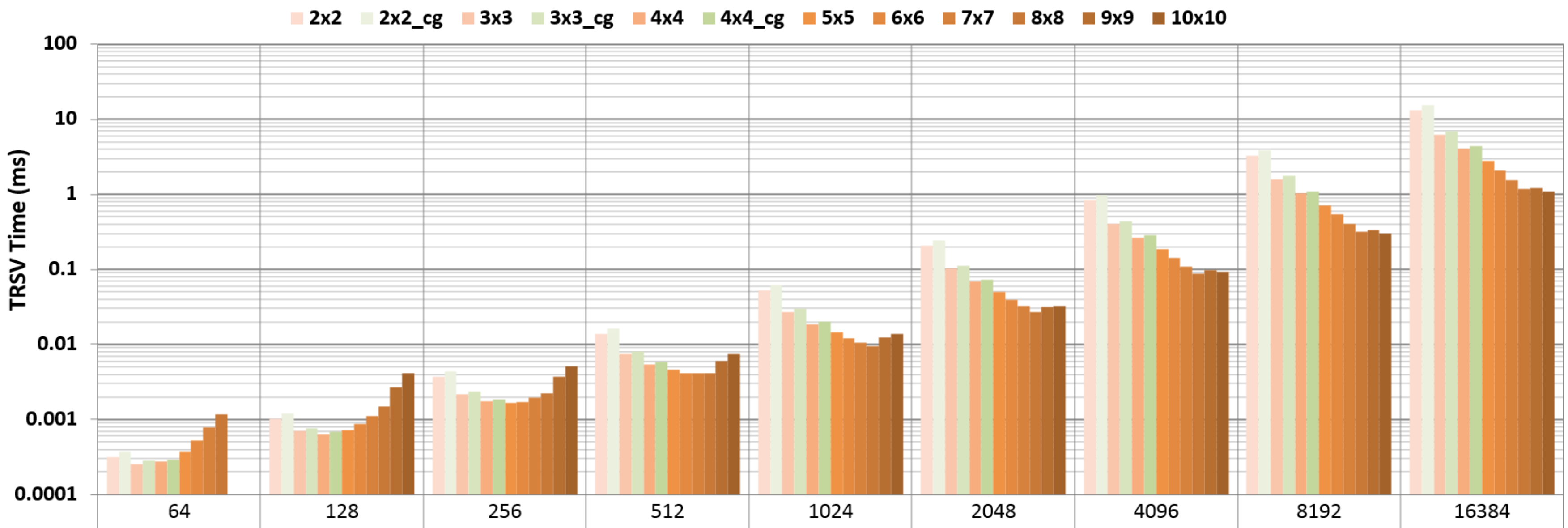
	64	128	256	512	1024	2048	4096	8192	16384
2x2	0.0095	0.0718	0.5598	4.4201	35.1288	280.1057	2237.1560	17882.5074	143001.1319
2x2_cg	0.0111	0.0843	0.6568	5.1854	41.2108	328.6013	2624.4821	20978.5637	167759.3806
3x3	0.0052	0.0371	0.2779	2.1488	16.8960	133.9956	1067.2825	8519.5716	68081.8312
3x3_cg	0.0058	0.0408	0.3062	2.3674	18.6152	147.6294	1175.8762	9386.4204	75009.0166
4x4	0.0040	0.0264	0.1890	1.4233	11.0332	86.8545	689.1979	5491.0426	43838.2008
4x4_cg	0.0043	0.0284	0.2029	1.5278	11.8429	93.2288	739.7786	5894.0342	47055.5185
5x5	0.0034	0.0202	0.1377	1.0047	7.6493	59.6602	471.1461	3744.6290	29858.8779
6x6	0.0030	0.0172	0.1097	0.7747	5.7829	44.5986	350.1801	2774.9956	22094.1627
7x7	0.0027	0.0143	0.0871	0.5925	4.3201	32.9127	256.6984	2027.1498	16111.6812
8x8	0.0022	0.0120	0.0718	0.4739	3.3848	25.4412	196.9615	1549.3785	12289.6929
9x9		0.0145	0.0797	0.5071	3.5230	26.0587	200.0300	1566.4464	12396.9130
10x10		0.0144	0.0780	0.4714	3.2050	23.3457	177.5248	1383.6572	10922.8728

Figure C.7: LU timing results for various architectures (N=2..8) and data sizes (64..16384). [140]



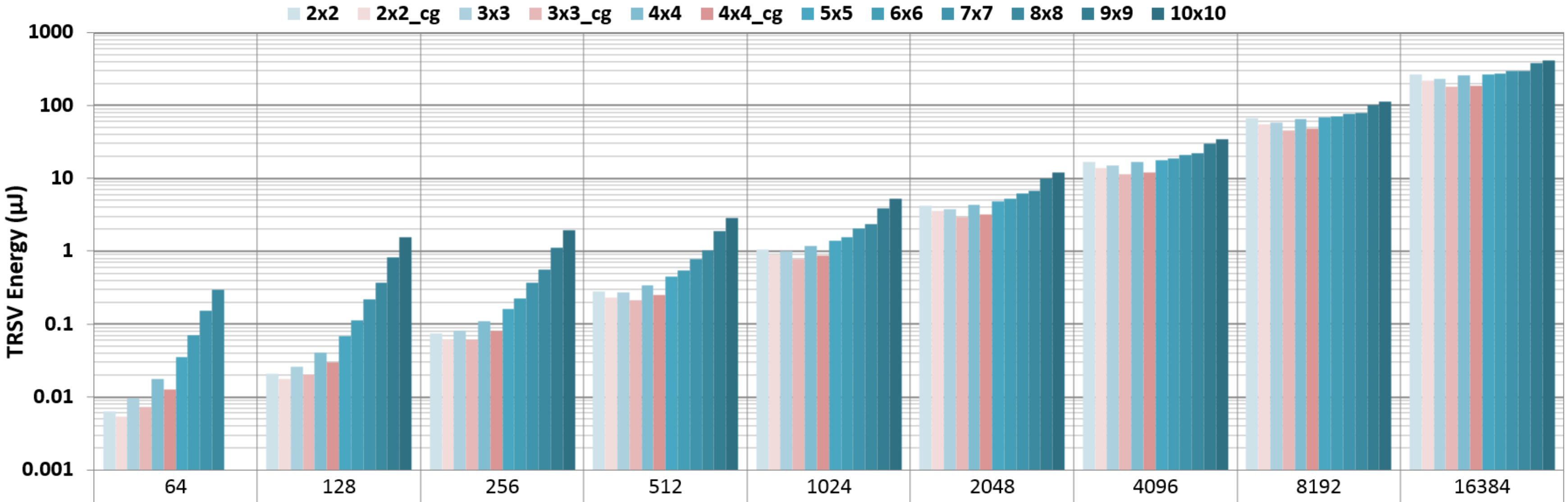
	64	128	256	512	1024	2048	4096	8192	16384
2x2	0.1967	1.4934	11.6361	91.8700	730.1313	5821.8296	46497.9461	371677.1864	2972192.7265
2x2_cg	0.1495	1.1348	8.8424	69.8126	554.8310	4424.0420	35334.0583	282439.6451	2258586.4545
3x3	0.1959	1.3909	10.4268	80.6257	633.9703	5027.7644	40046.4122	319670.0888	2554556.2593
3x3_cg	0.1496	1.0622	7.9628	61.5724	484.1516	3839.6125	30582.7183	244126.2459	1950868.2588
4x4	0.2547	1.6781	12.0059	90.4141	700.8657	5517.3075	43780.3292	348810.7916	2784760.3300
4x4_cg	0.1913	1.2604	9.0176	67.9098	526.4186	4144.0373	32883.3066	261990.9995	2091627.2080
5x5	0.3381	2.0340	13.8521	101.0748	769.5062	6001.7216	47396.5630	376703.8702	3003756.8404
6x6	0.4535	2.6010	16.6291	117.4267	876.5609	6760.2189	53079.9880	420631.3296	3349013.2998
7x7	0.5619	3.0178	18.3544	124.9235	910.8221	6939.1704	54121.1690	427395.4391	3396916.7825
8x8	0.5905	3.1872	19.0160	125.5298	896.5084	6738.4748	52168.0888	410375.1937	3255102.1275
9x9		4.7530	26.1568	166.3698	1155.9022	8549.9201	65630.2884	513954.5949	4067455.0464
10x10		5.7719	31.1458	188.3279	1280.5362	9327.5177	70928.2115	552826.0008	4364121.3046

Figure C.8: LU energy results for various architectures (N=2..8) and data sizes (64..16384). [140]



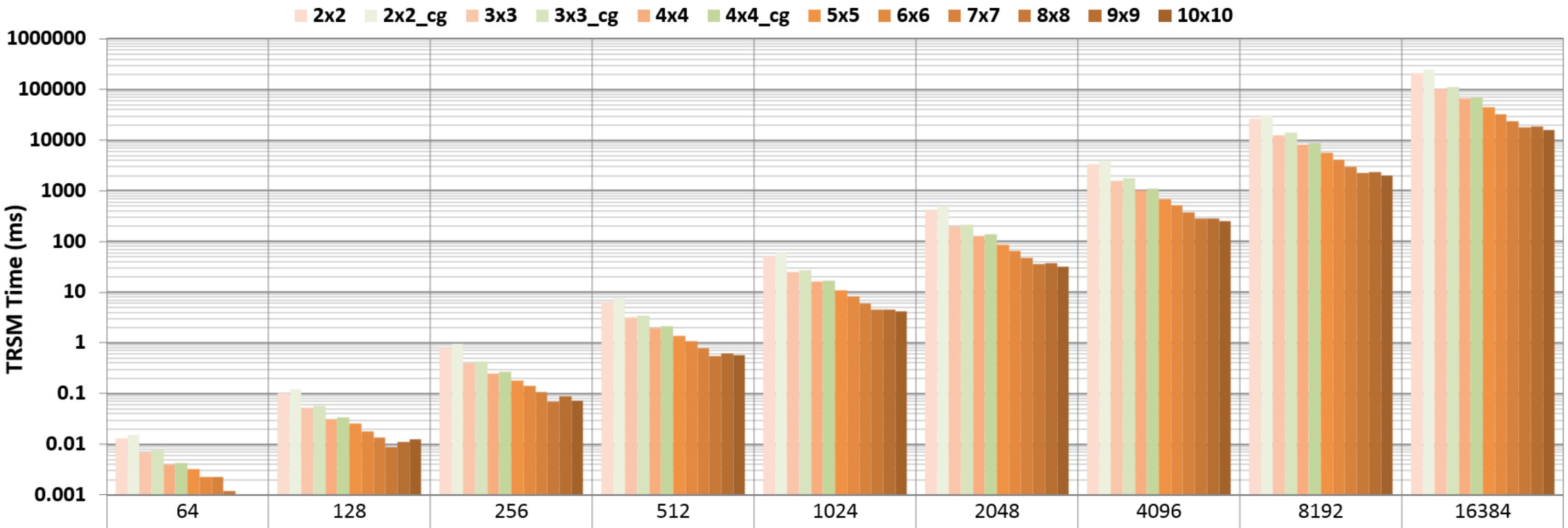
	64	128	256	512	1024	2048	4096	8192	16384
2x2	0.0003179	0.0010355	0.0036689	0.0137282	0.0530168	0.2082742	0.8255098	3.2868639	13.1171041
2x2_cg	0.0003729	0.0012148	0.0043041	0.0161050	0.0621957	0.2443334	0.9684329	3.8559293	15.3881108
3x3	0.0002553	0.0006979	0.0021528	0.0073424	0.0268422	0.1023236	0.3992132	1.5767032	6.2665216
3x3_cg	0.0002813	0.0007689	0.0023719	0.0080895	0.0295734	0.1127348	0.4398323	1.7371295	6.9041272
4x4	0.0002745	0.0006393	0.0017356	0.0053949	0.0185805	0.0684201	0.2619724	1.0245698	4.0517356
4x4_cg	0.0002947	0.0006862	0.0018630	0.0057908	0.0199441	0.0734415	0.2811987	1.0997637	4.3490954
5x5	0.0003657	0.0007153	0.0016669	0.0045673	0.0143572	0.0499043	0.1848582	0.7102043	2.7826505
6x6	0.0005275	0.0008612	0.0017100	0.0041490	0.0119752	0.0394201	0.1414951	0.5343731	2.0750405
7x7	0.0007846	0.0011253	0.0019391	0.0041063	0.0105834	0.0321285	0.1095632	0.4018112	1.5358386
8x8	0.0011755	0.0014984	0.0022463	0.0041505	0.0095931	0.0270144	0.0880016	0.3145548	1.1859761
9x9		0.0026948	0.0036765	0.0060600	0.0124671	0.0318574	0.0969529	0.3323845	1.2242456
10x10		0.0040760	0.0051044	0.0075132	0.0137901	0.0321214	0.0918950	0.3039525	1.0980440

Figure C.9: TRSV timing results for various architectures (N=2..8) and data sizes (64..16384).



	64	128	256	512	1024	2048	4096	8192	16384
2x2	0.00641	0.02088	0.07400	0.27688	1.06926	4.20056	16.64921	66.29079	264.55100
2x2_cg	0.00537	0.01750	0.06201	0.23203	0.89606	3.52013	13.95230	55.55273	221.69794
3x3	0.00949	0.02594	0.08004	0.27297	0.99793	3.80414	14.84177	58.61796	232.97392
3x3_cg	0.00731	0.01997	0.06162	0.21015	0.76825	2.92861	11.42590	45.12689	179.35438
4x4	0.01742	0.04056	0.11010	0.34222	1.17864	4.34019	16.61806	64.99297	257.01942
4x4_cg	0.01274	0.02967	0.08054	0.25033	0.86218	3.17485	12.15614	47.54246	188.01009
5x5	0.03538	0.06920	0.16126	0.44186	1.38898	4.82796	17.88396	68.70818	269.20543
6x6	0.06935	0.11322	0.22481	0.54544	1.57429	5.18229	18.60140	70.25040	272.79146
7x7	0.15069	0.21615	0.37244	0.78871	2.03279	6.17103	21.04419	77.17730	294.99397
8x8	0.29196	0.37215	0.55790	1.03085	2.38259	6.70943	21.85654	78.12449	294.55530
9x9		0.82865	1.13052	1.86344	3.83366	9.79618	29.81312	102.20858	376.45681
10x10		1.53467	1.92187	2.82883	5.19214	12.09413	34.59966	114.44204	413.42771

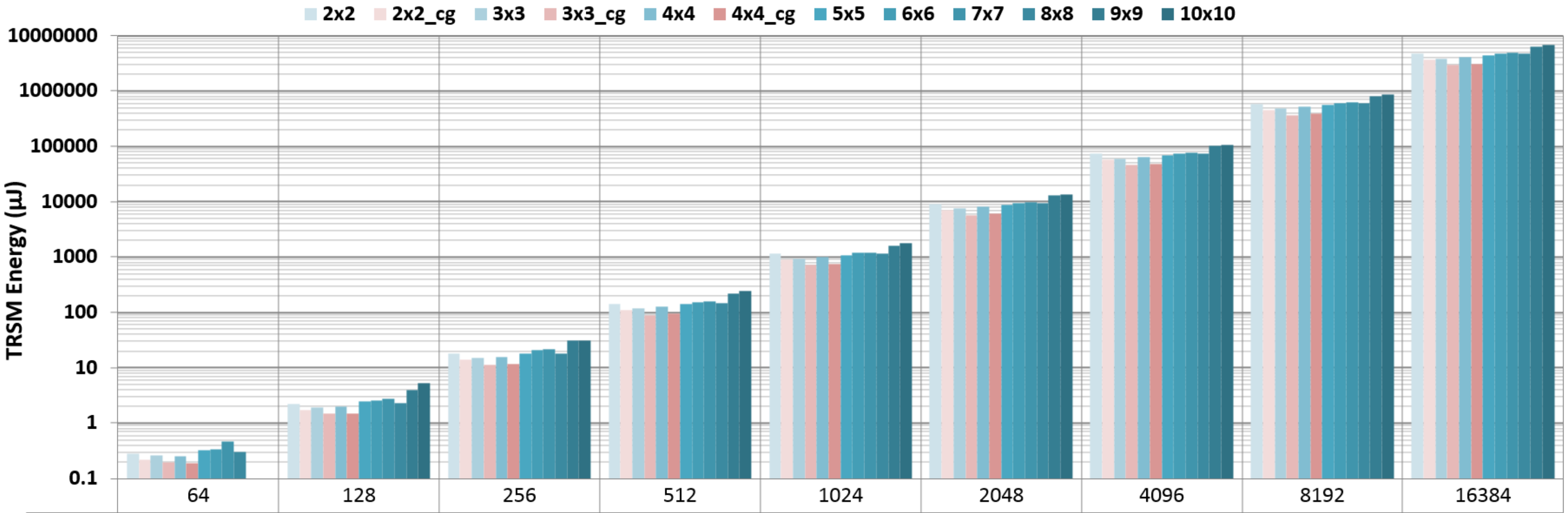
Figure C.10: TRSV energy results for various architectures (N=2..8) and data sizes (64..16384).



	64	128	256	512	1024	2048	4096	8192	16384
2x2	0.0128461	0.1023719	0.8181847	6.5439007	52.3480567	418.7781597	3350.2126933	26801.6763812	214413.3607245
2x2_cg	0.0150702	0.1200959	0.9598396	7.6768675	61.4112452	491.2825776	3930.2458572	31441.9373360	251535.4396493
3x3	0.0069094	0.0514400	0.3970140	3.1197453	24.9546804	199.6308832	1597.0339502	12762.2363431	102041.8021710
3x3_cg	0.0076125	0.0566739	0.4374094	3.4371730	27.4937675	219.9429124	1759.5288489	14060.7674742	112424.3443238
4x4	0.0039894	0.0314465	0.2506386	2.0032475	16.0222620	128.1706653	1025.3504667	8202.7740267	65622.1328044
4x4_cg	0.0042822	0.0337544	0.2690332	2.1502673	17.1981476	137.5772045	1100.6016900	8804.7816326	70438.1892915
5x5	0.0032009	0.0251099	0.1832597	1.3975863	10.9106730	87.2775206	698.2044435	5585.6041116	44684.7700251
6x6	0.0022994	0.0178680	0.1418950	1.0622943	8.2110842	64.5479557	516.3669603	4130.9023172	33047.1518129
7x7	0.0022782	0.0133207	0.1055278	0.7720303	5.8916221	47.1247199	376.9812516	3015.8170027	24054.6651022
8x8	0.0011755	0.0088864	0.0700614	0.5584370	4.4633923	35.6989370	285.5750987	2284.5680008	18276.4784348
9x9		0.0113188	0.0892381	0.6224858	4.6194526	36.9451806	289.8572324	2318.8161223	18459.5120915
10x10		0.0122806	0.0727338	0.5790330	4.2412048	32.3767451	252.8250441	2022.5550915	16180.3502172

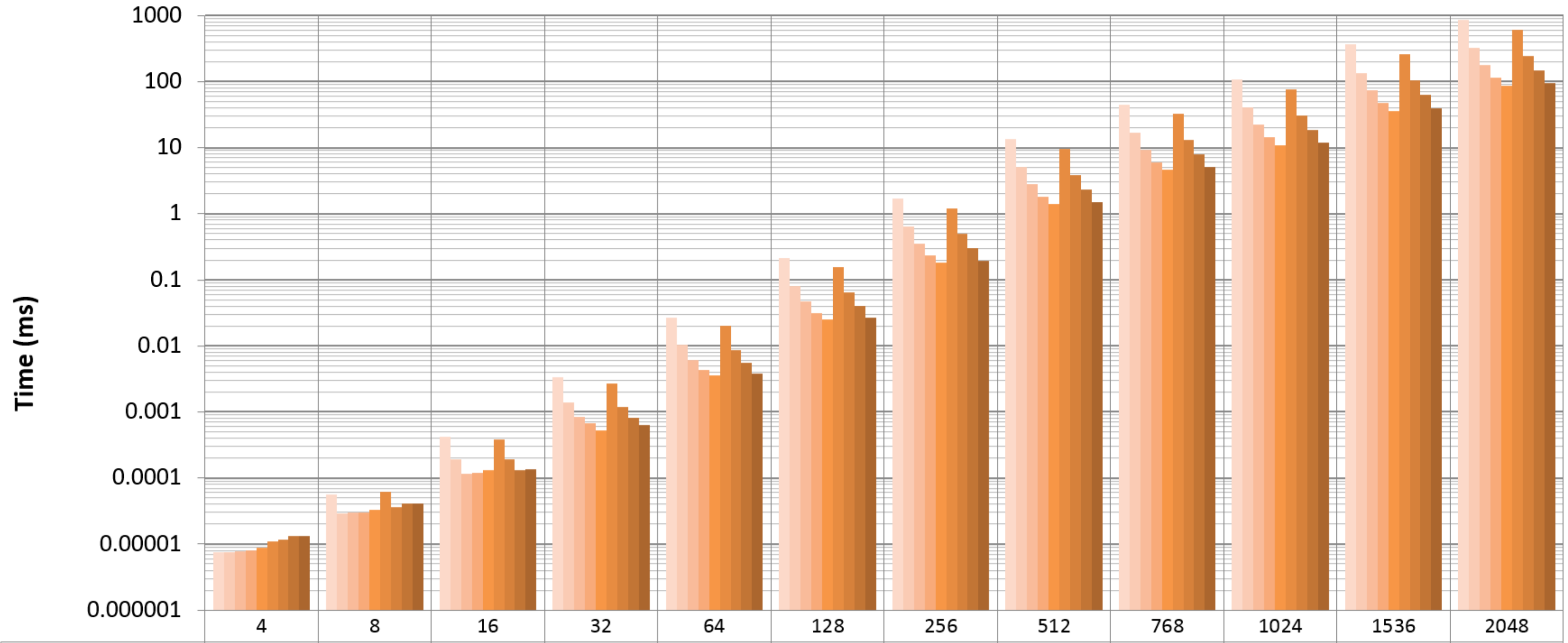
Input square matrix size

Figure C.11: TRSM timing results for various architectures (N=2..8) and data sizes (64..16384).



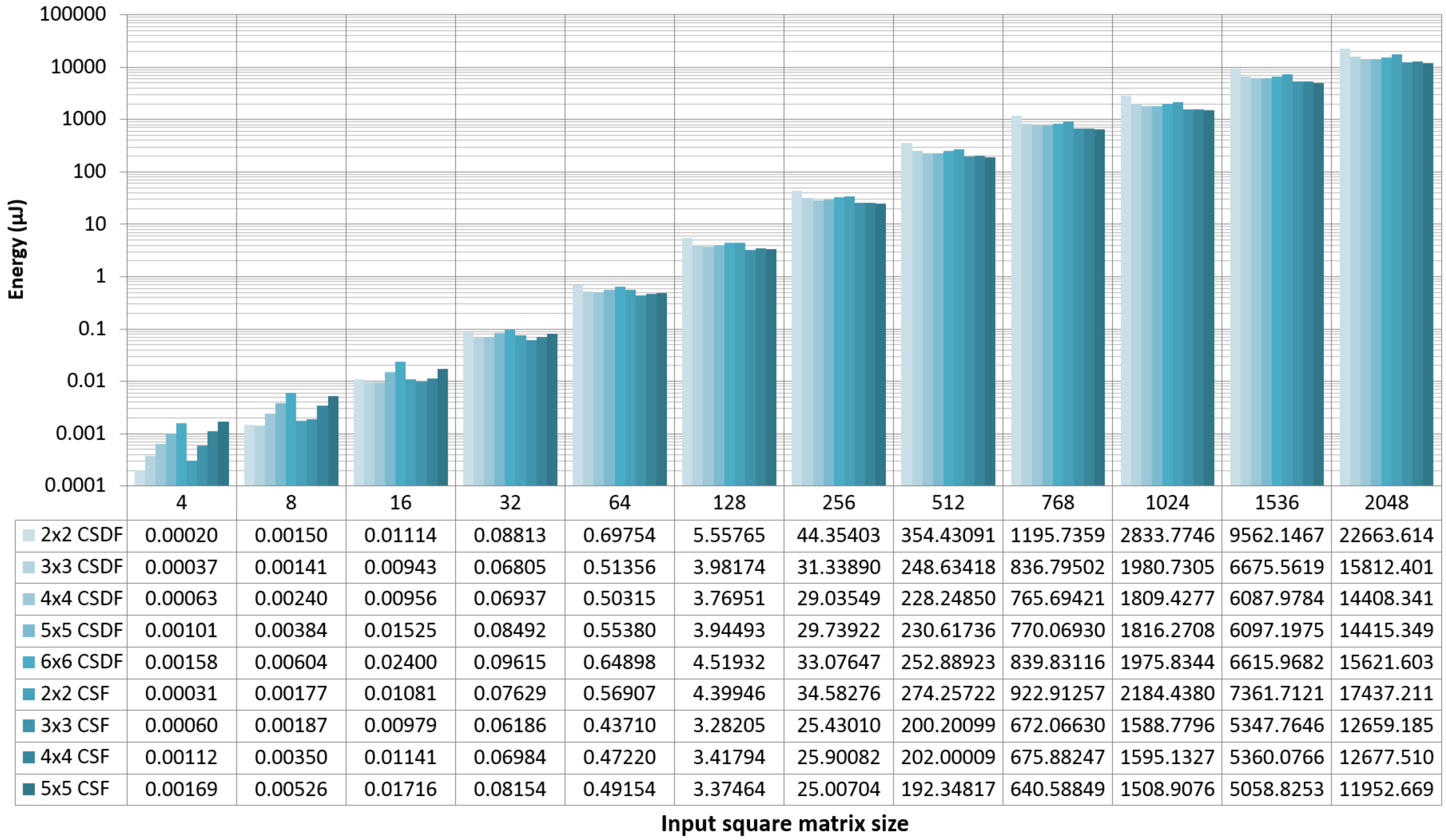
	64	128	256	512	1024	2048	4096	8192	16384
2x2	0.27940	2.22658	17.79543	142.32919	1138.56500	9108.38310	72866.79106	582933.78112	4663469.15442
2x2_cg	0.21712	1.73023	13.82850	110.60134	884.75751	7077.95371	56623.41701	452986.91073	3623894.43526
3x3	0.25878	1.92656	14.86923	116.84273	934.61891	7476.70560	59813.15357	477979.57098	3821735.90208
3x3_cg	0.19820	1.47554	11.38824	89.48903	715.81811	5726.35669	45810.47727	366081.22069	2927040.88029
4x4	0.25202	1.98649	15.83297	126.54614	1012.13430	8096.60502	64771.90166	518173.33665	4145382.94032
4x4_cg	0.18674	1.47193	11.73178	93.76713	749.96306	5999.35667	47994.15802	383951.87361	3071612.20807
5x5	0.31958	2.50704	18.29716	139.53886	1089.35159	8714.02767	69710.65170	557682.07491	4461450.32243
6x6	0.33241	2.58306	20.51286	153.56919	1187.02472	9331.29131	74647.91834	597178.52332	4777418.54054
7x7	0.46391	2.71241	21.48796	157.20370	1199.67418	9595.71219	76762.33613	614091.96741	4898101.11310
8x8	0.30696	2.32050	18.29499	145.82342	1165.51581	9321.98487	74571.59725	596564.21600	4772496.60544
9x9		3.93147	30.99604	216.21468	1604.52412	12832.56672	100679.22849	805419.33103	6411740.77448
10x10		5.17156	30.62940	243.84047	1786.04217	13634.38804	106468.84825	851731.72571	6813815.68832

Figure C.12: TRSM energy results for various architectures (N=2..8) and data sizes (64..16384).



	4	8	16	32	64	128	256	512	768	1024	1536	2048
2x2 CSDF@694MHz	0.0000076	0.0000567	0.0004223	0.0033416	0.0264477	0.2107206	1.6817011	13.4383934	45.3368214	107.443728	362.552720	859.300236
3x3 CSDF@694MHz	0.0000076	0.0000288	0.0001926	0.0013893	0.0104850	0.0812914	0.6398167	5.0761301	17.0840564	40.4387108	136.288664	322.826913
4x4 CSDF@675MHz	0.0000078	0.0000296	0.0001177	0.0008541	0.0061947	0.0464096	0.3574804	2.8101600	9.4271078	22.2773920	74.9542421	177.393255
5x5 CSDF@662MHz	0.0000079	0.0000302	0.0001200	0.0006685	0.0043598	0.0310565	0.2341228	1.8155409	6.0623898	14.2986373	48.0003393	113.485201
6x6 CSDF@602MHz	0.0000087	0.0000332	0.0001320	0.0005287	0.0035686	0.0248510	0.1818821	1.3905962	4.6180930	10.8647879	36.3801183	85.9006198
2x2 CSF @781MHz	0.0000109	0.0000621	0.0003795	0.0026786	0.0199802	0.1544656	1.2142044	9.6292019	32.4035637	76.6958634	258.470537	612.222463
3x3 CSF @729MHz	0.0000116	0.0000363	0.0001904	0.0012029	0.0084998	0.0638224	0.4945116	3.8930922	13.0689464	30.8952788	103.992196	246.169480
4x4 CSF @649MHz	0.0000131	0.0000408	0.0001332	0.0008150	0.0055105	0.0398868	0.3022582	2.3573075	7.8874361	18.6149346	62.5512034	147.944438
5x5 CSF @641MHz	0.0000133	0.0000413	0.0001349	0.0006412	0.0038651	0.0265351	0.1966326	1.5124509	5.0370047	11.8646763	39.7779969	93.9849126

Figure C.13: GR timing results for various architectures (N=2..6) and data sizes (4..2048). [137]



	4	8	16	32	64	128	256	512	768	1024	1536	2048
2x2 CSDF	0.00020	0.00150	0.01114	0.08813	0.69754	5.55765	44.35403	354.43091	1195.7359	2833.7746	9562.1467	22663.614
3x3 CSDF	0.00037	0.00141	0.00943	0.06805	0.51356	3.98174	31.33890	248.63418	836.79502	1980.7305	6675.5619	15812.401
4x4 CSDF	0.00063	0.00240	0.00956	0.06937	0.50315	3.76951	29.03549	228.24850	765.69421	1809.4277	6087.9784	14408.341
5x5 CSDF	0.00101	0.00384	0.01525	0.08492	0.55380	3.94493	29.73922	230.61736	770.06930	1816.2708	6097.1975	14415.349
6x6 CSDF	0.00158	0.00604	0.02400	0.09615	0.64898	4.51932	33.07647	252.88923	839.83116	1975.8344	6615.9682	15621.603
2x2 CSF	0.00031	0.00177	0.01081	0.07629	0.56907	4.39946	34.58276	274.25722	922.91257	2184.4380	7361.7121	17437.211
3x3 CSF	0.00060	0.00187	0.00979	0.06186	0.43710	3.28205	25.43010	200.20099	672.06630	1588.7796	5347.7646	12659.185
4x4 CSF	0.00112	0.00350	0.01141	0.06984	0.47220	3.41794	25.90082	202.00009	675.88247	1595.1327	5360.0766	12677.510
5x5 CSF	0.00169	0.00526	0.01716	0.08154	0.49154	3.37464	25.00704	192.34817	640.58849	1508.9076	5058.8253	11952.669

Figure C.14: GR energy results for various architectures (N=2..6) and data sizes (4..2048). [137]

Glossary

p	the set of physical functions f_{hw} that the architecture can perform
f_{hw}	an addressable physical operation or function of an architecture
Acronyms	
ADL	Architecture Description Language, a type of high level language specific for quick description of hardware
ALU	arithmetic-logic unit, performs arithmetic and logical operations on data
ASIC	application-specific integrated circuit
ASIP	application-specific instruction-set processor, some instructions can be customized to accelerate a given application
BER	Bit-Error Rate
CDFG	Control-Data Flow Graph, an intermediate representation of application or architectural features
CE	Channel Estimation
CISC	Complex Instruction-Set Computer
CPU	central processing unit of a GPP, usually featuring one or more ALUs and registers
CSDF	Column-wise Square-root and Division Free Givens rotation (algorithm)
CSF	Column-wise Square-root Free Givens rotation (algorithm)
DISE	Dual-Issue Single Execution, a predication method for control flow execution
DOT	Matrix dot product
DSP	digital signal processor, a customized processor with special instructions for signal processing
FDS	Force-Directed Scheduling, a heuristic algorithm
FU	functional units or processing elements in the reconfigurable architecture
GEMM	GEneral Matrix-Matrix multiplication
GEMV	GEneral Matrix-Vector multiplication
GPP	general purpose processor, with a generic instruction set, not optimized for any particular application

GR	Givens Rotation
HLD	High-Level Design, design from a high abstraction level
HLS	High-Level Synthesis, automatic generation of lower abstraction designs and components from a high abstraction description
I/O	input-output
ICE	Institute for Communication Technologies and Embedded Systems at the RWTH Aachen University
LISA	Language for Instruction-Set Architectures, a high-level architecture description language
LU	Lower-Upper matrix factorization
MAC	Multiply-ACcumulate (units)
MIMO	multiple input multiple output
MPSoC	multi-processor system-on-chip, a complex processing platform often integrating GPPs, DSPs, ASICs using buses or NoCs
NoC	network-on-chip, a scalable and flexible chip-level interconnect, often with higher level routing capability
OFDM	Orthogonal Frequency-Division Multiplexing
PE	processing elements or functional units in the reconfigurable architecture
PI	Polynomial Interpolation
f_a	the target function of an algorithm or application which is to be implemented in an architecture
rASIP	reconfigurable application-specific instruction-set processor, an ASIP extended with a reconfigurable fabric to accomodate post-fabrication changes of custom instructions
$\mathcal{F}$	how well an architecture can adapt to (a change in) the application
$\mathcal{L}$	represents the sum of all higher order functions that the architecture can perform by combining f_{hw} elements of the p
RISC	Reduced Instruction-Set Computer
RTL	Register Transfer Level (hardware description)
SDR	Software-Defined Radio, an adaptability and flexibility concept for the wireless domain
SFDG	Square-root and Division Free Givens rotation
SFG	Square-root Free Givens rotation
SQDA	Single Qualifier Double Address, a flavour of assembly code, that selects one of the two unconditional jump addresses, depending whether the qualifying condition is true or false
TRSM	TRiangular Solve Matrix
TRSV	TRiangular Solve Vector

TSV	Through Silicon Via, cross-die vertical interconnects in 3D process technologies
VLSI	Very Large-scale System Integration
WCDMA	Wide-band Code-Division Multiple Access, a wireless transmission air interface standard
WMSA	Weighted Multi-Slot Averaging
$\mathcal{F}_{\text{IC}}$	flexibility given by 1/re-implementation time

Notation (Integrated Circuits)

A	silicon area in mm^2
$A(1\,\mathrm{GE})$	silicon area of a two-input drive-one NAND gate for the used standard cell library
A_{GE}	equivalent gate count in units of two-input drive-one NAND gates with size $A(1\,\mathrm{GE})$
γ	cycles required by an architecture/software implementation to examine one node
E	electrical energy in J
$f_{\max}$	maximum clock frequency of a synchronous IC design
P	electrical power in W
P_{d}	dynamic CMOS power in W
P_{s}	static CMOS leakage power in W
T	task execution time in s
t_{p}	intrinsic CMOS inverter propagation delay
V_{dd}	supply voltage

List of Figures

List of Tables

Bibliography

[1] "http://www.itrs.net/ITRS%201999-2014%20Mtgs,%20Presentations%20&%20Links/2013ITRS/2013Chapters/2013SysDrivers_Summary.pdf", [Online] ITRS roadmap, System Drivers Summary.

[2] "http://www.research.ibm.com/cognitive-computing/brainpower/", [Online] IBM.

[3] "www.haskell.org", [Online] Haskell Programming Language.

[4] "http://www.wolfram.com/language/", [Online] Wolfram Language.

[5] "www.tensilica.com", [Online], Tensilica, part of Cadence IP.

[6] "www.arm.com", [Online], ARM.

[7] "https://www.synopsys.com/IP/Pages/default.aspx", [Online] Synopsys.

[8] "http://calypto.com/en/products/catapult/overview", [Online] Calypto Catapult.

[9] "https://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx", [Online] Synopsys Synphony-C Compiler.

[10] "http://www.xilinx.com/products/design-tools/vivado/integration/esl-design", [Online] Xilinx Vivado HLS.

[11] "http://www.recoresystems.com/", [Online], Recore Systems.

[12] "http://www.menta.fr", [Online], Menta eFPGA.

[13] "www.arc.com", synopsys Designware ARC.

[14] "The Mesa 3D Graphics Library," accessed: 2014-03-25. [Online]. Available: http://www.mesa3d.org/

[15] S. Abeta, M. Sawahashi, and F. Adachi, "Performance comparison between time-multiplexed pilot channel and parallel pilot channel for coherent rake combining in DS-CDMA mobile radio," *IEICE transactions on communications*, vol. 81, no. 7, pp. 1417–1425, 1998.

[16] M. Ali, E. Stotzer, F. D. Igual, and R. A. van de Geijn, "Level-3 BLAS on the TI C6678 multi-core DSP," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*. IEEE, 2012, pp. 179–186.

[17] V. Allada, T. Benjegerdes, and B. Bode, "Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–9.

[18] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1983, pp. 177–189.

[19] M. Alles, T. Vogt, and N. Wehn, "FlexiChaP: A reconfigurable ASIP for convolutional, turbo, and LDPC code decoding," in *Turbo Codes and Related Topics, 2008 5th International Symposium on*. IEEE, pp. 84–89.

[20] G. Ansaloni, P. Bonzini, and L. Pozzi, "EGRA: A coarse grained reconfigurable architectural template," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 6, pp. 1062–1074, 2011.

[21] W. Arden, M. Brillouët, P. Cogez, M. Graef, B. Huizing, and R. Mahnkopf, "Morethanmoore white paper," *Version*, vol. 2, p. 14, 2010.

[22] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from Berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[23] B. M. Baas, "A low-power, high-performance, 1024-point fft processor," *Solid-State Circuits, IEEE Journal of*, vol. 34, no. 3, pp. 380–387, 1999.

[24] J. Backus, "Can programming be liberated from the von neumann style?: a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[25] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Orti, "Evaluation and tuning of the level 3 CUBLAS for graphics processors," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–8.

[26] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP A self-reconfigurable data processing architecture," *the Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[27] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. Geijn, "The science of deriving dense linear algebra algorithms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 1, pp. 1–26, 2005.

[28] P. Biswas, K. Varadarajan, M. Alle, S. K. Nandy, and R. Narayan, "Design space exploration of systolic realization of QR factorization on a runtime reconfigurable platform," in *Embedded Computer Systems (SAMOS)*, 2010, pp. 265–272.

[29] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," in *ACM SIGPLAN Notices*, vol. 34, no. 1. ACM, 1998, pp. 174–184.

[30] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCauley, P. Morrow, D. W. Nelson, D. Pantuso *et al.*, "Die stacking (3d) microarchitecture," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE, 2006, pp. 469–479.

[31] B. Black, D. W. Nelson, C. Webb, and N. Samra, "3d processing technology and its impact on ia32 microprocessors," in *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*. IEEE, 2004, pp. 316–318.

[32] H. Blume, H. Feldkaemper, and T. G. Noll, "Model-based exploration of the design space for heterogeneous systems on chip," *The Journal of VLSI Signal Processing*, vol. 40, no. 1, pp. 19–34, 2005.

[33] H. Blume, H. Hübert, H. Feldkämper, and T. G. Noll, "Model-based exploration of the design space for heterogeneous systems on chip," in *Application-Specific Systems, Architectures and Processors, 2002. Proceedings. The IEEE International Conference on*. IEEE, 2002, pp. 29–40.

[34] T. Bollaert, "Catapult synthesis: a practical introduction to interactive c synthesis," in *High-Level Synthesis*. Springer, 2008, pp. 29–52.

[35] B. Bougard, B. De Sutter, D. Verkest, L. Van der Perre, and R. Lauwereins, "A coarse-grained array accelerator for software-defined radio baseband processing," *Micro, IEEE*, vol. 28, no. 4, 2008.

[36] H. Bunke, X. Jiang, and A. Kandel, "On the minimum common supergraph of two graphs," *Computing*, vol. 65, no. 1, pp. 13–25, 2000.

[37] J. Cartwright, "Intel enters the third dimension," *nature news*, 2011.

[38] A. Chattopadhyay, X. Chen, H. Ishebabi, R. Leupers, G. Ascheid, and H. Meyr, "High-level Modelling and Exploration of Coarse-grained Re-configurable Architectures," in *Design, Automation and Test in Europe, 2008. DATE '08*, March 2008, pp. 1334–1339.

[39] A. Chattopadhyay, "Ingredients of adaptability: a survey of reconfigurable processors," *VLSI Design*, vol. 2013, p. 10, 2013.

[40] A. Chattopadhyay, W. Ahmed, K. Karuri, D. Kammler, R. Leupers, G. Ascheid, and H. Meyr, "Design space exploration of partially re-configurable embedded processors," in *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*. IEEE, 2007, pp. 1–6.

[41] A. Chattopadhyay, X. Chen, Z. E. Rákossy, and G. Ascheid, "Partially reconfigurable processor for wireless receiver applications," in *Reconfigurable Logic: Architecture, Tools and Applications*. CRC Press, 2015.

[42] A. Chattopadhyay, H. Meyr, and R. Leupers, *LISA: A Uniform ADL for Embedded Processor Modelling, Implementation and Software Toolsuite Generation*. Morgan Kaufmann, jun 2008, ch. 5, pp. 95–130.

[43] X. Chen, A. Minwegen, Y. Hassan, D. Kammler, S. Li, T. Kempf, A. Chattopadhyay, and G. Ascheid, "FLEXDET: Flexible, Efficient Multi-Mode MIMO Detection Using Reconfigurable ASIP," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2012, pp. 69–76.

[44] X. Chen, S. Li, J. Schleifer, T. Coenen, A. Chattopadhyay, G. Ascheid, and T. G. Noll, "High-level Modeling and Synthesis for Embedded FPGAs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 1565–1570.

[45] Y.-K. Chen and S.-Y. Kung, "Trend and challenge on system-on-a-chip designs," *Journal of signal processing systems*, vol. 53, no. 1-2, pp. 217–229, 2008.

[46] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys (csuR)*, vol. 34, no. 2, pp. 171–210, 2002.

[47] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, "GAUT: A high-level synthesis tool for DSP applications," in *High-Level Synthesis*. Springer, 2008, pp. 147–169.

[48] P. Coussy and A. Morawiec, *High-level synthesis: from algorithm to digital circuit*. Springer, 2008.

[49] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco: Morgan Kaufmann Publishers, 2003.

[50] A. DeHon, "The density advantage of configurable computing," *Computer*, vol. 33, no. 4, pp. 41–49, 2000.

[51] R. H. Dennard, V. Rideout, E. Bassous, and A. Leblanc, "Design of ion-implanted mosfet's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, 1974.

[52] V. Derudder, B. Bougard, A. Couvreur, A. Dewilde, S. Dupont, L. Folens, L. Hollevoet, F. Naessens, D. Novo, P. Raghavan *et al.*, "A 200mbps+ 2.14 nj/b digital baseband multi processor system-on-chip for sdrs," in *VLSI Circuits, 2009 Symposium on*. IEEE, 2009, pp. 292–293.

[53] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[54] R. Döhler, "Squared givens rotation," *IMA Journal of numerical analysis*, vol. 11, no. 1, pp. 1–5, 1991.

[55] P. Eades, "A heuristic for graph drawing," *Congressus Numerantium*, vol. 42, pp. 149–160, 1984.

[56] C. Ebeling, D. Cronquist, and P. Franklin, "RaPiD—Reconfigurable pipelined datapath," *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, pp. 126–135, 1996.

[57] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE Micro*, no. 3, pp. 122–134, 2012.

[58] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proc. of the European conf. on Design and Test*. IEEE Computer Society, 1995.

[59] A. Fell, Z. E. Rákossy, and A. Chattopadhyay, "Force-Directed Scheduling for Data-Flow Graph Mapping on Coarse-Grained Reconfigurable Architectures," in *Reconfigurable Computing and FPGAs (ReConFig), 2014 International Conference on*. IEEE, 2014.

[60] Fonseca, Ma Nilma and Klautau, Aldebaro, "High speed GSGR matrix inversion algorithm with application to G.fast systems," in *Microwave & Optoelectronics Conference (IMOC), 2013 SBMO/IEEE MTT-S International*. IEEE, 2013, pp. 1–5.

[61] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: an architecture-adaptive CGRA mapping tool," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '09. New York, NY, USA: ACM, 2009, pp. 191–200.

[62] H. Fujisawa, M. Saito, S. Nishijima, N. Odate, Y. Sakai, K. Yoda, I. Sugiyama, T. Ishihara, Y. Hirose, and H. Yoshizawa, "Flexible Signal Processing Platform Chip for Software Defined Radio with 103 GOPS Dynamic Reconf1gurable Logic Cores," in *Solid-State Circuits Conference, 2006. ASSCC 2006. IEEE Asian*. IEEE, 2006, pp. 67–70.

[63] D. Gajski, T. Austin, and S. Svoboda, "What input-language is the best choice for high level synthesis (HLS)?" in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*. IEEE, 2010, pp. 857–858.

[64] M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, 2nd ed. New York, NY: Freeman, 1983.

[65] P. Garrou, C. Bower, and P. Ramm, *Handbook of 3d integration: volume 1-technology and applications of 3D integrated circuits*. John Wiley & Sons, 2011.

[66] C. Giraud-Carrier, "A reconfigurable dataflow machine for implementing functional programming languages," *ACM Sigplan Notices*, vol. 29, no. 9, pp. 22–28, 1994.

[67] J. Glossner, D. Iancu, M. Moudgill, G. Nacer, S. Jinturkar, S. Stanley, and M. Schulte, "The sandbridge sb3011 platform," *EURASIP journal on embedded systems*, 2007.

[68] G. H. Golub and C. F. Van Loan, *Matrix computations (3rd ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[69] J. Gonzalez and R. C. Núñez, "LAPACKrc: Fast linear algebra kernels/solvers for FPGA accelerators," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012042.

[70] J. Götze and U. Schwiegelshohn, "A square root and division free givens rotation for solving least squares problems on systolic arrays," *SIAM J. Sci. Stat. Comput.*, vol. 12, no. 4, pp. 800–807, May 1991. Online: http://dx.doi.org/10.1137/0912042

[71] O. Gustafsson, K. Amiri, D. Andersson, A. Blad, C. Bonnet, J. Cavallaro, J. Declerck, A. Dejonghe, P. Eliardsson, M. Glasse *et al.*, "Architectures for cognitive radio testbeds and demonstrators—an overview," in *Cognitive Radio Oriented Wireless Networks & Communications (CROWNCOM), 2010 Proceedings of the Fifth International Conference on*. IEEE, 2010, pp. 1–6.

[72] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in *Design, Automation, and Test in Europe*. Springer, 2008, pp. 31–45.

[73] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "EPIMap: Using Epimorphism to Map Applications on CGRAs," in *Proceedings of the 49th Design Automation Conference (DAC)*, 2012.

[74] M. Hamzeh, A. Shrivastava, and S. B. K. Vrudhula, "REGIMap: Register-Aware Application Mapping on Coarse-Grained Reconfigurable Architectures (CGRAs)," in *The 50th Annual Design Automation Conference 2013, DAC '13*. ACM, May 29–Jun. 7, 2013.

[75] K. Han, J. Ahn, and K. Choi, "Power-efficient predication techniques for acceleration of control flow execution on cgra," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 2, p. 8, 2013.

[76] K. Han, J. K. Paek, and K. Choi, “Acceleration of control flow on cgra using advanced predicated execution,” in *Field-Programmable Technology (FPT), 2010 International Conference on*. IEEE, 2010, pp. 429–432.

[77] R. Hartenstein, “A decade of reconfigurable computing: a visionary retrospective,” in *Proceedings of the conference on Design, automation and test in Europe*. IEEE Press, 2001, pp. 642–649.

[78] J. Henkel, “Closing the soc design gap,” *Computer*, vol. 36, no. 9, pp. 119–121, 2003.

[79] M. Hofmann and E. J. Kontoghiorghes, “Pipeline Givens sequences for computing the QR decomposition on a EREW PRAM,” *Parallel Computing*, vol. 32, no. 3, pp. 222 – 230, 2006. Online: http://www.sciencedirect.com/science/article/pii/S0167819105001638

[80] N. Horiuchi, “Quantum information: One-way quantum computer,” *Nature Photonics*, vol. 9, no. 1, pp. 7–7, 2015.

[81] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, “CULA: hybrid GPU accelerated linear algebra routines,” in *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2010, pp. 770 502–770 502.

[82] S. L. P. Jones, “Implementing lazy functional languages on stock hardware: the spineless tagless g-machine,” *Journal of functional programming*, vol. 2, no. 02, pp. 127–202, 1992.

[83] S. Kala, S. Nalesh, S. Nandy, and R. Narayan, “Energy efficient, scalable, and dynamically reconfigurable fft architecture for ofdm systems,” in *Electronic System Design (ISED), 2014 Fifth International Symposium on*. IEEE, 2014, pp. 20–24.

[84] U. Kapasi, W. Dally, S. Rixner, J. Owens, and B. Khailany, “The Imagine stream processor,” in *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*. IEEE, 2002, pp. 282–288.

[85] M. Karkooti, J. Cavallaro, and C. Dick, “Fpga implementation of matrix inversion using qrd-rls algorithm,” in *Signals, Systems and Computers, 2005. Asilomar Conference on*, 2005, pp. 1625–1629.

[86] K. Karuri, A. Chattopadhyay, X. Chen, D. Kammler, L. Hao, R. Leupers, H. Meyr, and G. Ascheid, “A Design Flow for Architecture Exploration and Implementation of Partially Reconfigurable Processors,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 10, pp. 1281–1294, oct 2008.

[87] A. Kerr, D. Campbell, and M. Richards, “QR decomposition on GPUs,” in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 71–78.

[88] S. Kestur, J. D. Davis, and O. Williams, “Blas comparison on fpga, cpu and gpu,” in *Proceedings of the IEEE Annual Symposium on VLSI*, ser. ISVLSI ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 288–293. Online: http://dx.doi.org/10.1109/ISVLSI.2010.84

[89] Khawar Shahzad and Ayesha Khalid and Zoltán Endre Rákossy and Goutam Paul and Anupam Chattopadhyay, “CoARX: A Coprocessor for ARX-based Cryptographic Algorithms,” in *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2013.

[90] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by Simulated Annealing,” *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983.

[91] R. Koenig, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker, and J. Henkel, “KAHRISMA: a novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture,” in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 819–824.

[92] A. N. Kolmogorov, “On tables of random numbers,” *Sankhyā: The Indian Journal of Statistics, Series A*, pp. 369–376, 1963.

[93] A. N. Kolmogorov, “Three approaches to the quantitative definition of information,” *International Journal of Computer Mathematics*, vol. 2, no. 1-4, pp. 157–168, 1968.

[94] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, “Basic linear algebra subprograms for fortran usage,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[95] D. Lee, M. Jo, K. Han, and K. Choi, “FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability,” in *2009 International Conference on Field-Programmable Technology*, 2009, pp. 376–379.

[96] M.-W. Lee, J.-H. Yoon, and J. Park, “High-speed tournament givens rotation-based QR Decomposition Architecture for MIMO Receiver,” in *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, may 2012, pp. 21 –24.

[97] Y. Lei, Y. Dou, Y. Dong, J. Zhou, and F. Xia, “FPGA implementation of an exact dot product and its application in variable-precision floating-point arithmetic,” *The Journal of Supercomputing*, vol. 64, no. 2, pp. 580–605, 2013. Online: http://dx.doi.org/10.1007/s11227-012-0860-0

[98] G. Levi, “A note on the derivation of maximal common subgraphs of two directed or undirected graphs,” *Calcolo*, vol. 9, no. 4, pp. 341–352, 1973.

[99] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, “SODA: A low-power architecture for software radio,” in *ACM SIGARCH Computer Architecture News*, vol. 34. IEEE Computer Society, 2006.

[100] Y.-W. Lin, H.-Y. Liu, and C.-Y. Lee, “A dynamic scaling fft processor for dvb-t applications,” *Solid-State Circuits, IEEE Journal of*, vol. 39, no. 11, pp. 2005–2013, 2004.

[101] L. Ma, K. Dickson, J. McAllister, and J. McCanny, “QR Decomposition-Based Matrix Inversion for High Performance Embedded MIMO Receivers,” *IEEE Transactions on Signal Processing*, vol. 59, no. 4, pp. 1858–1867, 2011.

[102] Mahdi Shabany and Dimpesh Patel and P. Glenn Gulak, “A Low-Latency Low-Power QR-Decomposition ASIC Implementation in 0.13um,” *IEEE Transactions on Circuits and Systems*, vol. 60-I, no. 2, pp. 327–340, 2013.

[103] S. Mahlke, R. E. Hank, J. E. McCormick, D. August, W.-M. W. Hwu *et al.*, “A comparison of full and partial predicated execution support for ILP processors,” in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*. IEEE, 1995, pp. 138–149.

[104] P. Marwedel, “The mimola design system: Tools for the design of digital processors,” in *Design Automation, 1984. 21st Conference on*. IEEE, 1984, pp. 587–593.

[105] S. Masekowsky, T. Schweizer, W. Rosenstiel *et al.*, “CGADL: an architecture description language for coarse-grained reconfigurable arrays,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 9, pp. 1247–1259, 2009.

[106] M. Massey, A. Kotsialos, F. Qaiser, D. Zeze, C. Pearson, D. Volpati, L. Bowen, and M. Petty, “Computing with carbon nanotubes: Optimization of threshold logic gates using disordered nanotube/polymer composites,” *Journal of Applied Physics*, vol. 117, no. 13, p. 134903, 2015.

[107] J. J. McGregor, “Backtrack search algorithms and the maximal common subgraph problem,” *Software: Practice and Experience*, vol. 12, no. 1, pp. 23–34, 1982.

[108] L. McMurchie and C. Ebeling, “PathFinder: A Negotiation-based Performance-driven Router for FPGAs,” in *Proceedings of the 1995 ACM Third International Symposium on Field-programmable Gate Arrays*, ser. FPGA ’95. New York, NY, USA: ACM, 1995, pp. 111–117. Online: http://doi.acm.org/10.1145/201310.201328

[109] B. Mei, A. Lambrechts, J. Mignolet, D. Verkest, and R. Lauwereins, “Architecture exploration for a reconfigurable architecture template,” *Design & Test of Computers, IEEE*, 2005.

[110] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, “Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study,” in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, ser. DATE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 21 224–. Online: http://dl.acm.org/citation.cfm?id=968879.969178

[111] F. Merchant, A. Chattopadhyay, G. Garga, S. K. Nandy, R. Narayan, and N. Gopalan, “Efficient QR Decomposition Using Low Complexity Column-wise Givens Rotation (CGR),” in *VLSI Design*, 2014, pp. 258–263.

[112] F. Merchant, A. Maity, M. Mahadurkar, K. Vatwani, I. Munje, M. Krishna, S. Nalesh, N. Gopalan, S. Raha, S. Nandy *et al.*, “Micro-architectural Enhancements in Distributed Memory CGRAs for LU and QR Factorizations,” in *VLSI Design (VLSID), 2015 28th International Conference on*. IEEE, 2015, pp. 153–158.

[113] M. Meredith, “High-level SystemC Synthesis with Forte’s Cynthesizer,” in *High-Level Synthesis*. Springer, 2008, pp. 75–97.

[114] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. New York: Mc Graw-Hill, 1994.

[115] J. Mitola, “Cognitive radio for flexible mobile multimedia communications,” in *Mobile Multimedia Communications, IEEE International Workshop on*, 1999.

[116] J. Mitola, “Cognitive radio architecture evolution,” *Proceedings of the IEEE*, vol. 97, no. 4, pp. 626–641, 2009.

[117] N. Miura, Y. Take, M. Saito, Y. Yoshida, and T. Kuroda, “A 2.7 gb/s/mm 2 0.9 pj/b/chip 1coil/channel thruchip interface with coupled-resonator-based cdr for nand flash memory stacking,” in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*. IEEE, 2011, pp. 490–492.

[118] T. Miyamori and K. Olukotun, “Remarc: Reconfigurable multimedia array coprocessor,” *IEICE Transactions on information and systems*, vol. 82, no. 2, pp. 389–397, 1999.

[119] G. E. Moore *et al.*, “Cramming more components onto integrated circuits,” *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[120] M. Naylor and C. Runciman, “The reduceron: Widening the von neumann bottleneck for graph reduction using an fpga,” in *Implementation and Application of Functional Languages*. Springer, 2008, pp. 129–146.

[121] B. Neumann, T. von Sydow, H. Blume, and T. G. Noll, “Design flow for embedded fpgas based on a flexible architecture template,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE ’08. New York, NY, USA: ACM, 2008, pp. 56–61. Online: http://doi.acm.org/10.1145/1403375.1403391

[122] R. S. Nikhil, “Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions,” in *High-Level Synthesis*. Springer, 2008, pp. 129–146.

[123] K. Patel, S. McGettrick, and C. J. Bleakley, “Rapid functional modelling and simulation of coarse grained reconfigurable array architectures,” *Journal of Systems Architecture*, vol. 57, no. 4, pp. 383–391, 2011.

[124] P. G. Paulin and J. P. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASIC’s,” *IEEE Transactions on CAD*, vol. 8, no. 6, pp. 661–678, Jun. 1989.

[125] A. Pedram, A. Gerstlauer, and R. Geijn, “A high-performance, low-power linear algebra core,” in *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*. IEEE, 2011, pp. 35–42.

[126] A. Pedram, A. Gerstlauer, and R. van de Geijn, “Algorithm, architecture, and floating-point unit codesign of a matrix factorization accelerator,” *IEEE Transactions on Computers*, vol. 63, no. 8, 2014.

[127] A. Pedram, A. Gerstlauer, and R. A. Van De Geijn, “Floating point architecture extensions for optimized matrix factorization,” in *Computer Arithmetic (ARITH), 2013 21st IEEE Symposium on*. IEEE, 2013, pp. 49–58.

[128] A. Pedram, J. D. McCalpin, and A. Gerstlauer, “A Highly Efficient Multicore Floating-Point FFT Architecture Based on Hybrid Linear Algebra/FFT Cores,” *Journal of Signal Processing Systems*, vol. 77, no. 1-2, pp. 169–190, 2014.

[129] A. Pedram, R. A. van de Geijn, and A. Gerstlauer, “Codesign tradeoffs for high-performance, low-power linear algebra architectures,” *IEEE Transactions on Computers*, vol. 61, no. 12, pp. 1724–1736, 2012.

[130] S. Rajagopal, S. Bhashyam, J. Cavallaro, and B. Aazhang, “Efficient VLSI architectures for multiuser channel estimation in wireless base-station receivers,” *J. of VLSI Signal Processing*, vol. 31, no. 2, 2002.

[131] S. Rajagopal, S. Bhashyam, J. Cavallaro, and B. Aazhang, “Real-time algorithms and architectures for multiuser channel estimation and detection in wireless base-station receivers,” *Wireless Communications, IEEE Transactions on*, vol. 1, no. 3, pp. 468–479, 2002.

[132] S. Rajagopal, B. Jones, J. Cavallaro *et al.*, “Task partitioning wireless base-station receiver algorithms on multiple DSPs and FPGAs,” in *International conference on signal processing, applications, and technology (ICSPAT),(Dallas, TX)*, 2000.

[133] S. Rajagopal, S. Rixner, and J. Cavallaro, “A programmable baseband processor design for software defined radios,” in *Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium on*, vol. 3. IEEE, 2002, pp. III–413.

[134] Z. E. Rákossy, A. Acosta-Aponte, G. Ascheid, R. Leupers, T. Noll, and A. Chattopadhyay, “Design and Synthesis of Reconfigurable Control-Flow Structures for CGRA,” in *Reconfigurable Computing and FPGAs (ReConFig), 2015 International Conference on*. IEEE, december 2015.

[135] Z. E. Rákossy, A. Acosta Aponte, and A. Chattopadhyay, “Exploiting architecture description language for diverse IP synthesis in heterogeneous MPSoC,” in *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*. IEEE, 2013, pp. 1–6.

[136] Z. E. Rákossy, F. Merchant, A. Acosta-Aponte, S. Nandy, and A. Chattopadhyay, “Efficient and scalable cgra-based implementation of column-wise givens rotation,” in *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*. IEEE, 2014, pp. 188–189.

[137] Z. E. Rákossy, F. Merchant, A. Acosta Aponte, S. Nandy, and A. Chattopadhyay, “Scalable and Energy-Efficient Reconfigurable Accelerator for Column-wise Givens Rotation,” in *22nd International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2014.

[138] Z. E. Rákossy, T. Naphade, and A. Chattopadhyay, “Design and analysis of layered coarse-grained reconfigurable architecture,” in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*. IEEE, 2012, pp. 1–6.

[139] Z. E. Rákossy, D. Stengele, A. Acosta-Aponte, S. Chafekar, P. Bientinesi, and A. Chattopadhyay, “Scalable and Efficient Linear Algebra Kernel Mapping for Low Energy Consumption on the Layers CGRA,” in *Applied Reconfigurable Computing*. Springer, 2015, pp. 301–310.

[140] Z. E. Rákossy, D. Stengele, P. Bientinesi, R. Leupers, G. Ascheid, and A. Chattopadhyay, “Exploiting Scalable CGRA Mapping of LU for Energy Efficiency using the *LAYERS* Architecture,” in *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, october 2015.

[141] Z. E. Rákossy, Z. Wang, and A. Chattopadhyay, “Adaptive energy-efficient architecture for wcdma channel estimation,” in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*. IEEE, 2011, pp. 309–314.

[142] Z. E. Rákossy, Z. Wang, and A. Chattopadhyay, “High-level design space and flexibility exploration for adaptive, energy-efficient WCDMA channel estimation architectures,” *International Journal of Reconfigurable Computing*, vol. 2012, p. 8, 2012.

[143] B. R. Rau, “Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops,” in *Proceedings of the 27th Annual International Symposium on Microarchitecture*. San Jose, California: ACM SIGMICRO and IEEE Computer Society TC-MICRO, Nov. 30–Dec. 2, 1994, pp. 63–74.

[144] G. Rauwerda, P. Heysters, and G. Smit, "Towards software defined radios using coarse-grained reconfigurable hardware," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 1, pp. 3–13, 2008.

[145] J. W. Raymond and P. Willett, "Maximum common subgraph isomorphism algorithms for the matching of chemical structures," *Journal of computer-aided molecular design*, vol. 16, no. 7, pp. 521–533, 2002.

[146] D. L. Rosenband and T. Rosenband, "A design case study: Cpu vs. gpgpu vs. fpga," in *Formal Methods and Models for Co-Design, 2009. MEMOCODE'09. 7th IEEE/ACM International Conference on*. IEEE, 2009, pp. 69–72.

[147] Schmidhuber, J, "Generalized algorithmic information." Online: http://people.idsia.ch/~juergen/kolmogorov.html

[148] M. A. Shami, "Dynamically reconfigurable resource array," 2012.

[149] M. A. Shami and A. Hemani, "Partially reconfigurable interconnection network for dynamically reprogrammable resource array," in *ASIC, 2009. ASICON'09. IEEE 8th International Conference on*. IEEE, 2009, pp. 122–125.

[150] M. A. Shami and A. Hemani, "Control Scheme for a CGRA," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*. IEEE, 2010, pp. 17–24.

[151] H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *Computers, IEEE Transactions on*, vol. 49, no. 5, pp. 465–481, 2000.

[152] I. Sobel, "An Isotropic 3×3 image gradient operator," *Machine Vision for three-dimensional Sciences*, 1990. Online: http://ci.nii.ac.jp/naid/10018992790/en/

[153] P. Sutton, J. Lotze, H. Lahlou, S. Fahmy, K. Nolan, B. Ozgul, T. Rondeau, J. Noguera, and L. Doyle, "Iris: an architecture for cognitive radio networking testbeds," *Communications Magazine, IEEE*, vol. 48, no. 9, pp. 114–122, 2010.

[154] K. Swee and L. H. Hiung, "Performance comparison review of 32-bit multiplier designs," in *Intelligent and Advanced Systems (ICIAS), 2012 4th International Conference on*, vol. 2, 2012, pp. 836–841.

[155] H. Tabkhi, R. Bushey, and G. Schirner, "Function-level processor (flp): Raising efficiency by operating at function granularity for market-oriented mpsoc," *25th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP*, 2014.

[156] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee *et al.*, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *Micro, IEEE*, vol. 22, no. 2, pp. 25–35, 2002.

[157] K. Van Berkel, F. Heinle, P. Meuwissen, K. Moerman, and M. Weiss, "Vector processing as an enabler for software-defined radio in handheld devices," *EURASIP J. on Applied Signal Processing*, 2005.

[158] F. G. Van Zee, E. Chan, R. A. Van de Geijn, E. S. Quintana-Ort, and G. Quintana-Ort, "The libflame library for dense matrix computations," *Computing in science & engineering*, vol. 11, no. 6, pp. 56–63, 2009.

[159] S. R. Vegdahl, "A survey of proposed architectures for the execution of functional languages," *Computers, IEEE Transactions on*, vol. 100, no. 12, pp. 1050–1071, 1984.

[160] T. Vogt and N. Wehn, "A reconfigurable ASIP for convolutional and turbo decoding in an SDR environment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 10, 2008.

[161] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 31.

[162] J. Von Neumann, "First draft of a report on the edvac," *IEEE Annals of the History of Computing*, no. 4, pp. 27–75, 1993.

[163] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua *et al.*, "Baring it all to software: Raw machines," *Computer*, vol. 30, no. 9, pp. 86–93, 1997.

[164] X. Wang and M. Leeser, "A truly two-dimensional systolic array FPGA implementation of QR decomposition," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 1, pp. 3:1–3:17, Oct. 2009. Online: http://doi.acm.org/10.1145/1596532.1596535

[165] Z. Wang, Z. E. Rákossy, X. Wang, and A. Chattopadhyay, "ASIC Synthesis using Architecture Description Language," in *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, 2012.

[166] E. M. Witte, "Efficiency and flexibility trade-offs for soft-input soft-output sphere-decoding architectures," Ph.D. dissertation, Hochschulbibliothek der Rheinisch-Westfälischen Technischen Hochschule Aachen, 2012.

[167] Q. Xia, W. Robinett, M. W. Cumbie, N. Banerjee, T. J. Cardinali, J. J. Yang, W. Wu, X. Li, W. M. Tong, D. B. Strukov *et al.*, "Memristor- cmos hybrid integrated circuits for reconfigurable logic," *Nano letters*, vol. 9, no. 10, pp. 3640–3645, 2009.

[168] G. Yue, X. Zhou, and X. Wang, "Performance comparisons of channel estimation techniques in multipath fading CDMA," *Wireless Communications, IEEE Transactions on*, vol. 3, no. 3, pp. 716–724, 2004.

[169] W. Zhang, V. Betz, and J. Rose, "Portable and scalable FPGA-based acceleration of a direct linear system solver," *ACM Trans. on Reconf. Techn. and Sys.(TRETS)*, vol. 5, no. 1, p. 6, 2012.

[170] Y. Zhang, Y. H. Shalabi, R. Jain, K. K. Nagar, and J. D. Bakos, "Fpga vs. gpu for sparse matrix vector multiply," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*. IEEE, 2009, pp. 255–262.

[171] L. Zhuo and V. K. Prasanna, "High-performance designs for linear algebra operations on reconfigurable hardware," *Computers, IEEE Transactions on*, vol. 57, no. 8, pp. 1057–1071, 2008.

Curriculum Vitae

Name	Zoltán Endre Rákossy
Date of birth	January 5th, 1982, Brasov (Kronstadt), Romania
May. 2010 – Dec. 2015	Research assistant at the Institute for Communication Technologies and Embedded Systems, Prof. Dr.-Ing. Anupam Chattopadhyay, Prof. Dr.-Ing. Gerd Ascheid, Prof. Dr. rer. nat. Rainer Leupers, RWTH Aachen
Apr. 2007 – Apr. 2010	Graduation as Master of Informatics (M.Sc.) and master studies at Processor Architectures and Systems Synthesis Laboratory, Prof. Takashi Sato, Ph.D., and Prof. Hiroyuki Ochi, Ph.D., at the University of Kyoto, Japan *M.Sc. thesis:* "Hot-Swapping Architecture for Mitigation of Permanent Faults in Arrays of Coarse-Grained Functional Units"
Oct. 2001 – Sep. 2006	Graduation as Dipl.-Ing. and study of Electrical Engineering and Computer Science at "Transilvania" University of Brasov, Romania *Diploma thesis:* "Automatic Generation and Optimization of Advanced VLIW Architectures from LISA"
Sep. 1988 – Aug. 2000	Graduation "Baccalaureat" (Abitur) degree, "Johannes Honterus" Lyzeum, Brasov, Romania

Aachen, December 2015 *Zoltán Endre Rákossy*