ENERGY OPTIMIZATIONS FOR SCRATCH PAD MEMORY BASED SIMD ARCHITECTURES

NAMITA SHARMA

AMAR NATH AND SHASHI KHOSLA SCHOOL OF IT
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI
APRIL 2015

ENERGY OPTIMIZATIONS FOR SCRATCH PAD MEMORY BASED SIMD ARCHITECTURES

by

NAMITA SHARMA

Amar Nath and Shashi Khosla School of IT
Department of Computer Science and Engineering

Submitted

in fulfillment of the requirements of the degree of
Doctor of Philosophy

to the

Indian Institute of Technology Delhi
April 2015

Certificate

This is to certify that the thesis titled **Energy Optimizations for Scratch Pad Memory based SIMD Architectures** being submitted by **Ms. Namita Sharma** for the award of **Doctor of Philosophy** in Amar Nath and Shashi Khosla School of IT is a record of bona-fide work carried out by her under my guidance and supervision at the Department of Computer Science and Engineering, Indian Institute of Technology Delhi. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Preeti Ranjan Panda
Professor
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi - 110 016

Acknowledgments

The satisfaction that accompanies the successful completion of this thesis would be incomplete without making mention of the people who made it possible, whose constant guidance and encouragement crown all the efforts. First and foremost, I would like to express my deepest gratitude to my supervisor Prof. Preeti Ranjan Panda, Department of Computer Science and Engineering, IIT Delhi for providing me an opportunity to work under his precious guidance. It is my fortune to work with Prof. Panda who has been enthusiastically guiding me at every stage of this work and encouraging me to achieve the targets set well in time. Working with him was a great pleasure and vast learning experience. His critical comments during the discussions enabled me to overcome my weaknesses and make valuable improvements. Thanks for your belief in my abilities and to direct my passion towards the right direction.

It is my fortune to have my thesis work in collaboration with IMEC, Belgium. Thanks to Prof. Francky Catthoor for agreeing to serve as my co-supervisor at IMEC. I really admire him for his patience and long future insight he has. I would like to thank him for keeping timely meetings that made this distant collaboration fruitful. I would also like to extend my thanks to my colleagues – Tom Vander Aa, Praveen Raghavan, Eddy De Greef, Amir Amin and Min Li whose timely assistance greatly helped in developing a deep understanding of the project. Special thanks to my PhD friends Prashant Agrawal, Matthias Hartmann, Raf Appeltans, Robert Fasthuber, and Halil Kukner for friendly discussions and support that made my stay at IMEC memorable.

My sincere thanks to Professors Anshul Kumar, G.S. Visweswaran and Vinay Ribeiro for serving on my Student Research Committee. Thanks to my labmates – Vaibhav Jain, Sharat Varma, G. Krishnaiah, Swanti Satsangi, Rajshekhar Kalayappan, Sandeep Chandran, and Prathmesh Kallurkar for their great company. I would also like to thank S.D. Sharma and Vandana Ahluwalia for their timely assistance in lab related issues.

Finally, I dedicate this thesis to my beloved parents Gajendra Sharma and Uma Sharma for their blessings, support, and persistence which has always been a source of inspiration that kept me going to complete this venture. Special thanks to my mom and brother Hardik Sharma who always encouraged me and shared my tough and sweet moments without any hesitation.

Namita Sharma

Abstract

Mobile communication devices have evolved tremendously in the last few decades. These devices are not bound to an application but are useful for a wide range of functionalities. For example, the modern smartphones provide not only the calling facility but also multiple other functionalities such as 3D gaming, Internet access, Bluetooth, and so on. These evolutions require higher data rates, and low latencies. To achieve these features, techniques such as OFDM and MIMO are being introduced leading to co-existence of multiple wireless standards. Separate standards exist for supporting varying connectivity ranges. Relying on hardware implementation to accommodate these features on mobile devices leads to design challenges such as higher area for multiple hardware units, costlier implementation, longer time to market and so on. To overcome these challenges, having the entire baseband running on a programmable architecture is an attractive alternative. This is possible if we have the wireless physical layer functions software defined and run on a re-configurable architecture. This is popularly known as Software Defined Radio (SDR) implementation. With SDR architectures, we overcome the drawbacks of custom hardware implementations by bringing down the implementation cost, reducing the size as resource sharing is applicable, lowering the time-to-market as standard revisions simply require software upgradation rather than building the customized hardware. SDR thus proves to be an attractive alternative as it enables the reuse of design efforts across generations.

Battery life limits the utility of handheld devices such as mobile phones. With more and more applications and features enabled in modern devices the energy consumption is increasing. Current smart phones suffer a major drawback of short battery life, with average lasting time of less than a day. This poses a need to lower the energy consumption so as to increase the battery life. The lower the energy consumption, the longer the battery life will be. Bigger batteries are not a good solution for this problem as the portability of handheld devices decreases with increasing device sizes and weights. Thus, there is a need to have energy efficient implementations for applications running on such architectures. Memory plays a dominating role in the system design leading to significant amount of research in data and memory optimizations. Optimizations related to memory accesses and data storage make a significant difference to the performance and energy of a wide range of data-intensive applications. Such strategies need to

evolve with modern SoC and processor architectures, which lead to new optimization opportunities. In this thesis, we propose some strategies that significantly reduce the memory accesses thereby reducing the overall implementation energy. The proposed strategies are targeted at embedded processor systems with features such as scratch pad memories (SPM), Single Instruction Multiple Data (SIMD) FUs, and vector register files with wide interfaces to both SPM and FUs. The main contributions of this thesis are:

- We study the inter- and intra-kernel data dependencies for the LTE MIMO wireless standard and propose and compare efficient data layouts for the application. Our focus is on the LTE downlink receiver as this is a data- and computation-intensive part of the LTE application with tight energy and latency constraints.

- Array Interleaving, the proposed data layout transformation for the LTE application, is further developed to make it a more broadly applicable compiler transformation strategy for SIMD architectures. Based on the estimations of the benefits and interleaving costs, we perform a global analysis of arrays accesses spanning multiple loops, and take interleaving decisions that are predicted to be globally optimal. We also incorporate the effect of interleaving granularity in our exploration.

- We propose an energy efficient data flow transformation for Givens Rotation based QR Decomposition, a widely used function for matrix inversion and triangularization, for mapping to SIMD architectures. The proposed sequencing strategy is compared with the conventional sequences for matrices of different sizes. We also explore different possible implementations for QRD of multiple matrices using the SIMD feature of the processor.

- We propose a strategy to select an energy optimal Register File (RF) size for FFT processing on input data having large percentage of zeros. FFT is widely used in signal processing and image processing domains to transform the inputs in the time domain to the frequency domain for simplifying the computations. The strategy, based on memory access and compute count estimates, results in an energy efficient RF configuration out of the large number of configurations possible with the variation in number of registers and the SIMD widths.

With the proposed data layout strategy – Array Interleaving, for LTE application, a reduction of 7-15% in memory energy consumption is obtained over a highly hand-optimized code. Through an exploration for inter- and intra- kernel data dependencies in the application we conclude that there exist no dependencies across the Physical Resource Blocks (PRBs) allowing the merging of some kernels in the data processing block, thereby reducing the overall memory access count by around 15%. Experimental evaluation of the proposed layout strategy on

several applications in the wireless communication, multimedia, and image processing domain showed a significant reduction (6% – 34%) in memory energy consumption.

Our proposed data flow transformation strategy for QR Decomposition results in up to 36% reduction in overall energy across different implementations. Up to 75% reduction in memory accesses is achieved over the conventional sequences. A comparison of these sequences with standard tiling transformation shows that tiling results in an energy efficient implementation with respect to conventional sequences but has higher energy consumption compared to the proposed sequencing.

Using our proposed exploration strategy for RF size selection for FFT, we obtain a reduction of up to 59% in data memory energy. This percentage variation is obtained when comparing over the range of possible RF configurations with varying number of registers and SIMD widths.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

The evolution of modern embedded system architecture has led to new challenges in the efficient utilization of the architectural flexibility. The architectural space of embedded processors has expanded to include features such as SIMD Functional Units (FUs performing the same operation on multiple input data in parallel), vector register sets, and wide interfaces with scratch pad memories (SPMs). The search for energy efficient application mapping to such architectures is intricately tied to an efficient exploitation of the memory accesses, because of the dominant role played by accesses to the wide memory and vector register files. Since a large class of embedded applications in the communications and multimedia domain involve the storage, access, and computation of large amounts of data, special attention needs to be paid to the way data is organized and accessed.

This chapter discusses the context of this thesis and motivation behind the work presented here. It also summarizes the scope and main contributions of the thesis. The chapter is organized as follows. Section 1.1 describes the need for Software Defined Radio (SDR) Architectures. Section 1.2 presents the energy related challenges in the battery operated devices, which motivates the need for energy efficient implementations on SDRs. Section 1.3 points out the application areas addressed by the proposed strategies. Motivations behind the proposed transformations and architectural exploration for data memory energy optimization are presented in Sections 1.4, 1.5 and 1.6. Finally, we summarize the contributions of this thesis in Section 1.7 with the organization details in Section 1.8.

1.1 Why SDR Architectures?

A rapid evolution in the mobile communication devices has taken place in the last few decades. Modern smart phones provide not only the calling facility but also multiple other functionalities such as 3D gaming, Internet access, and Bluetooth. This evolution in the smart phone is

accompanied by features such as higher data rates, low latencies, and higher flexibility. For the smart phones supporting applications such as 3D gaming, video streaming; there is a need for higher data rates. The data rates have evolved from Kilo bits per second (Kbps) in 2G cellular standards to Giga bits per second (Gbps) in 4G and 60GHz communication standards. These applications also require low latencies for its efficient use. Figure 1.1 summarizes the challenges faced by the mobile phone industry arising out of the present day requirements in hand-held devices.

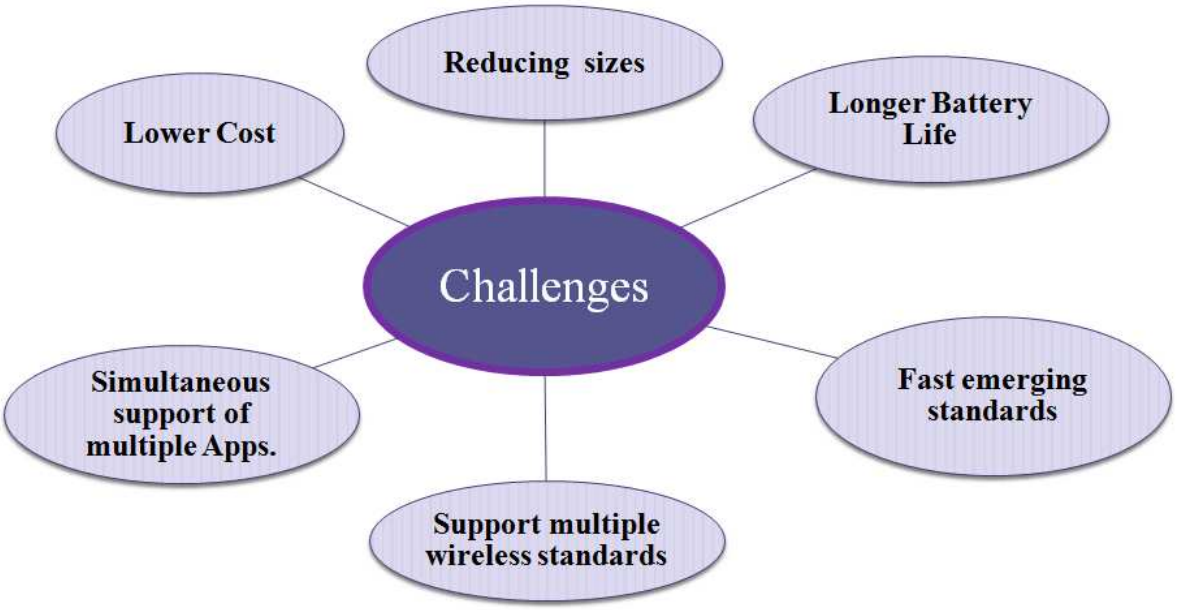


Figure 1.1: Challenges faced by the mobile industry

To enable efficient use of the available frequency spectrum, techniques such as Orthogonal Frequency Division Multiplexing (OFDM), Multiple Input and Multiple Output (MIMO) implementations have been introduced. Introduction of a multitude of such techniques is leading to a rapid increase in the number of wireless standards. Separate standards exist for supporting variable length connectivity (varying from short range to long range). Since the upgradation of base stations with these evolving standards would be very expensive, it is required by the mobile devices to support different standards. Accomodating these features in the traditional hardware implementation has some inherent drawbacks – higher area due to the multiple hardware units, costlier implementation, higher power dissipation, and longer time to market.

With the rapidly evolving wireless standards, custom hardware implementation becomes more difficult. The need to provide support for these different generation standards on a single hardware platform has led to Software Defined Radio (SDR) implementations, where the entire baseband runs on programmable or reconfigurable architectures. It is an attractive alternative as it enables the reuse of design efforts across generations. SDR is a communication system in

which some or all of the wireless physical layer functions are software defined to run on a reconfigurable architecture. With this feature, we overcome the drawbacks of custom hardware implementations. The cost is lower, with greater resource sharing leading to smaller sizes. The time-to-market is also lower, as revisions require a software upgradation with the evolving standards. Figure 1.2 shows a comparison of the hardware and software implementations in facing the implementation level challenges.

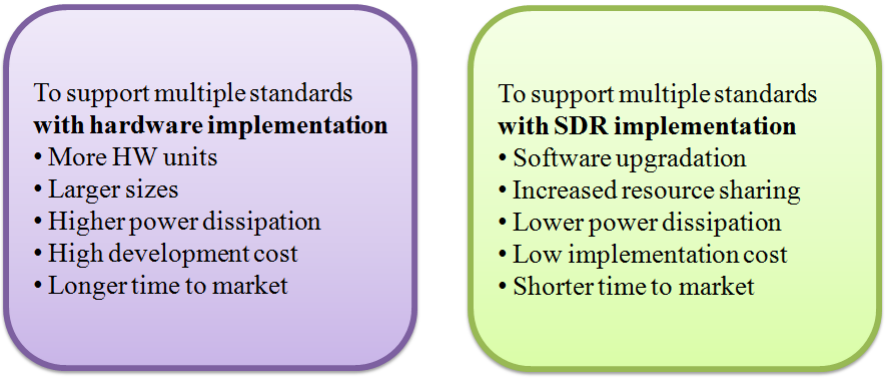


Figure 1.2: Comparison of the different implementations

1.2 Energy efficient implementation on SDRs

Battery operated handheld devices, such as mobile phones, require the energy consumption to be lowered so as to increase the battery life. To achieve longer battery life, we cannot opt for bigger batteries as the portability of handheld devices decreases with increasing device sizes and weights. Over the years, industry trends point towards reducing device sizes and weights to increase the portability. Thus, with improving technologies and reducing battery sizes the battery capacity does not show any significant improvement. On the other hand, with more and more applications and features enabled in modern devices the energy consumption is increasing. Currently used smart phones suffer a major drawback of short battery life, with average lasting time of less than a day. Thus, there is a need to have energy efficient implementations for applications running on such architectures.

Programmable processor based flexible baseband platforms have attracted extensive interest in both industry and academia. Many baseband processors are highly flexible and have SIMD support along with VLIW architectures to achieve high performance [1, 2, 3, 4] for applications involving streaming data. These architectures support scratch pad memories (SPM) and vector registers instead of cache based memory hierarchy. Figure 1.3 shows an instance of such an architecture on which the proposed strategies in this work are implemented.

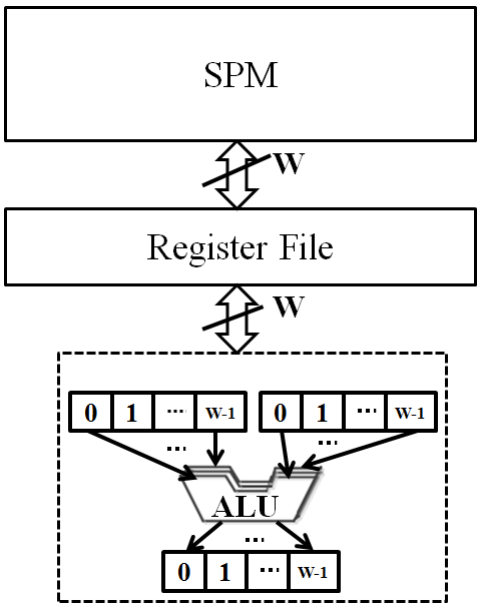


Figure 1.3: Architecture instance

As shown in the Figure 1.3, the register file has a wide interface to both memory and vector functional units (FUs). This makes both the memory accesses and computations expensive in terms of energy. Thus, there is a need to have implementations with lower memory access counts and operations to lower the overall implementation energy without any performance degradation. In this work, we propose data layout transformations to reduce the number of memory accesses by mapping the data in the memory such that the desired data is placed together and irrelevant data accesses are avoided. Also proposed is an application specific sequencing transformation that reduces the memory access and operation counts thereby significantly lowering down the overall implementation energy. The proposed strategies are applicable not only for SDR architecture under consideration, but for any processor with SIMD FUs, vector registers and no caches.

1.3 Application Domain

Data and memory optimizations have been the topic of a significant amount of recent research efforts because of the dominating role of memory in a large class of data-intensive applications. Embedded applications in the communication, multimedia and image processing domains fall under this class of applications. In this work, applications from these domains are explored to propose data memory energy efficient implementations for processor architectures with SIMD feature similar to the SDR architectures.

LTE is such a data-intensive standard for wireless communication with useful applications

in mobile phones and data terminals. LTE defines several channel bandwidths: 1.4 MHz, 3 MHz, 5 MHz, 10 MHz, 15 MHz, and 20 MHz [5]. The larger the bandwidth, the higher the channel capacity, and the greater the number of available sub-carriers for transmission. With the use of multiple antennas at both ends of the communication system (MIMO technique) a linear increase in the channel capacity can be obtained for the same bandwidth. Due to the processing requirement for a large number of sub-carriers, practical system implementations of this standard are subject to data memory organization and access bottlenecks.

QR Decomposition is a widely use matrix decomposition algorithm for implementing functionalities such as matrix triangularization and inversion in MIMO (Multiple Input and Multiple Output) detection systems [6, 7]. It finds applications in extracting principal components in data analysis, obtaining approximate solutions in image processing and coding, pattern recognition, and many other related areas. Exploration for energy efficient implementations of such a widely applicable functionality is an important area of research. QR Decomposition (QRD) is a typical matrix decomposition algorithm that shares many common features with other algorithms such as LU and Cholesky decomposition. The principle can be realized in several different implementations corresponding to a large number of valid processing sequences that differ significantly in the number of memory accesses and computations, and hence the overall implementation energy. With modern low power embedded processors evolving towards register files with wide memory interfaces and vector functional units (FUs), data flow in matrix decomposition algorithms needs to be carefully devised to efficiently utilize the costly wide memory accesses and the vector FUs.

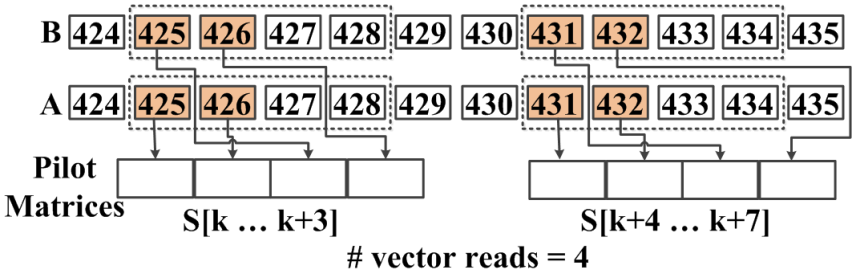
Discrete Fourier Transform (DFT) is commonly used in digital signal processing, image processing domains and for computing convolutions over large data points. Fast Fourier Transform (FFT) is an efficient algorithm for DFT computation. Transforming the input samples in the time domain to frequency domain lowers the computation complexity and FFT is an efficient methodology to implement this transformation. FFT is a memory intensive functionality as it involves operations over a large size of data and across multiple stages, thereby requiring a large number of memory accesses across the stages. There are multiple implementations for FFT with *radix-2 Cooley-Tukey algorithm* [8] being most commonly used. Memory accesses during the FFT processing are dependent on the RF size, thereby making it an important area of exploration for the target architecture.

1.4 Motivation for Data Layout Transformation

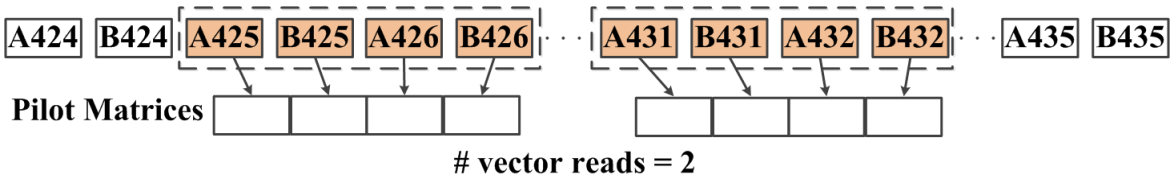
We illustrate the motivation behind the data layout transformation with the following example. Figure 1.4(a) shows a code snippet from the *Channel Estimation* stage of the LTE application

```
int k=0;
for(i=424; i<1624; i+=12) {
/* Extracting Pilot Matrix 1 */
    S[k]=A[i+1]*C0;
    S[k+1]=A[i+2]*C1;
    S[k+2]=B[i+1]*C2;
    S[k+3]=B[i+2]*C3;
/* Extracting Pilot Matrix 2 */
    S[k+4]=A[i+7]*C4;
    S[k+5]=A[i+8]*C5;
    S[k+6]=B[i+7]*C6;
    S[k+7]=B[i+8]*C7;
    k=k+8;
}
```

(a) Example loop



(b) Array access pattern (Layout 1)



(c) Array access pattern with interleaving (Layout 2)

Figure 1.4: Motivational example from LTE

that involves pilot extraction followed by product with constant conjugate pilots. Here, arrays *A* and *B* correspond to the sample space of the two antennas. Only a subset of these arrays is

used in the data processing block. Figure 1.4(b) shows the access pattern, with the accessed array elements highlighted.

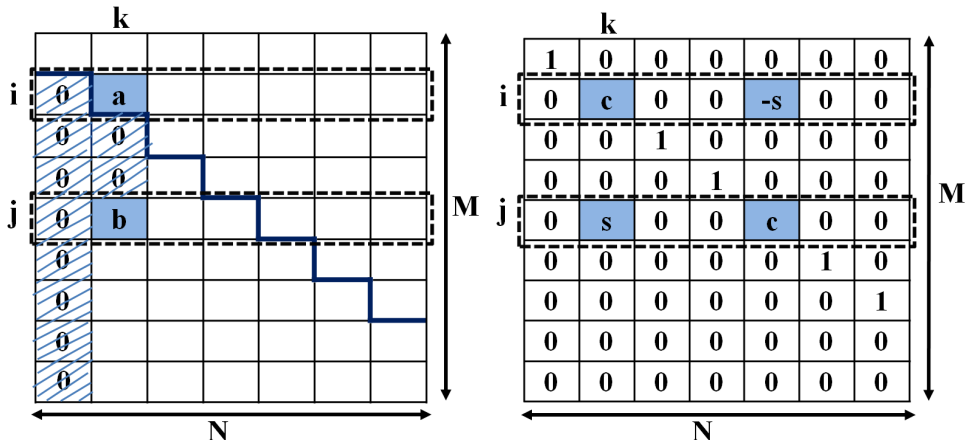
For these data sets, Figure 1.4 shows two possible layouts. Assuming that a vector register READ operation loads 4 consecutive words from the data memory into the vector register, in Layout 1 (Figure 1.4(b) which is the normal array storage strategy), two vector reads are required to load the elements for each pilot matrix. Here, the loads bring along the irrelevant data also, including A[427], A[428], etc. Layout 1 is conventionally chosen as the inputs and outputs of the OFDM demodulation block are all in sequential order. Consider Layout 2 shown in Figure 1.4(c) where elements of A and B are stored in an *interleaved* manner. Here, only one memory access per pilot matrix is required. Thus, a suitable layout in the memory reduces the number of memory accesses. This creates an important exploration problem, since the complexity increases if we have multiple arrays that can be interleaved. The choices of the subset of arrays to be interleaved and the granularity of interleaving, have to be comprehensively evaluated so as to produce energy-efficient implementations without compromising on performance.

1.5 Energy efficient sequencing strategy for Givens Rotation based QRD

QR Decomposition of a matrix involves the factorization of a $M \times N$ matrix A, where $M \geq N$, into an orthogonal matrix Q of size $M \times M$ and a $M \times N$ upper triangular matrix R, with all the elements in the bottom $(M-N)$ rows as zero. Thus, QRD can be expressed as $A_{M\times N} = Q_{M\times M} \times R_{M\times N}$. Givens Rotation based matrix decomposition [6] can be summarized as follows. With each rotation, a new element in the lower triangular part of the input matrix is turned zero. Consider an example matrix shown in Figure 1.5(a) where a part of the matrix is already zeroed. Given two elements, a and b, of a column k from rows i and j respectively (such that $i \neq j$ and $1 \leq i, j \leq M$, $1 \leq k < N$) of matrix A, in which b is to be zeroed, we define

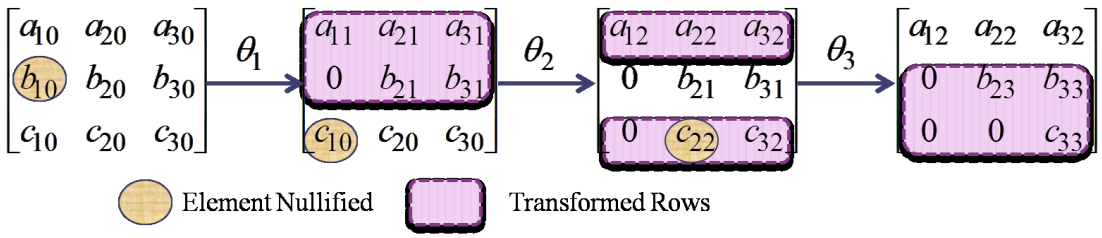
$$c = a/r, s = -b/r \text{ where } r = \sqrt{a^2 + b^2}$$

in the Givens Rotation (GR) matrix (as shown in Figure 1.5(b)). A sequence of such transformations on the input matrix A results in converting it to an upper triangular matrix R. The same rotations are also performed on an identity matrix Q that has one in the diagonal entries to begin with, and is eventually filled in (Figure 1.6). Two well known sequences of Givens Rotations for obtaining an upper triangular matrix from an input matrix A are: *row elimination sequence* and *column elimination sequence* (Figure 1.7).

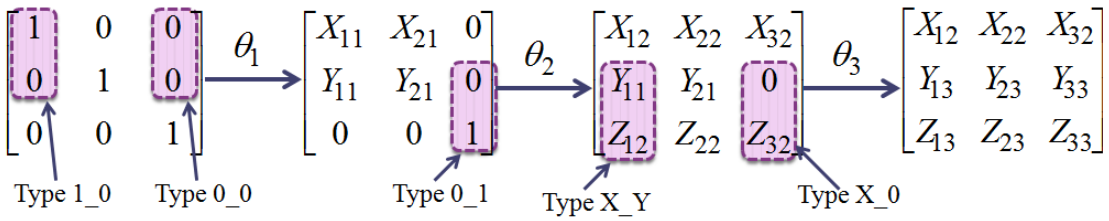


(a) Example matrix (b) GR matrix to nullify b

Figure 1.5: GR Matrix for a rotation in the example matrix



(a) Rotations on input matrix



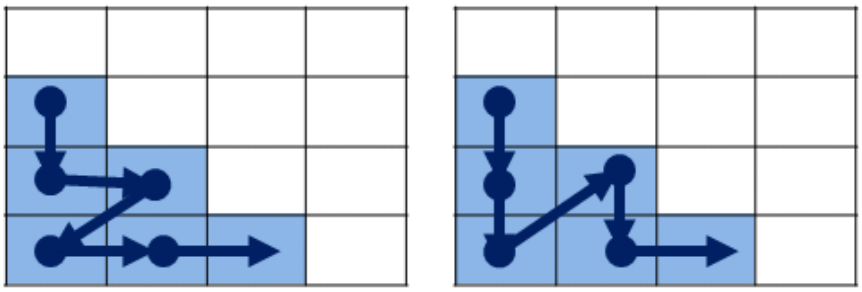
(b) Rotations on Q matrix

Figure 1.6: QRD with illustration for different types of operations

The number of Givens rotations to be applied on an $N \times N$ square matrix to obtain an upper triangular matrix R is

$$\#rotations = \frac{N \times (N-1)}{2}$$

These rotations involve different types of operations due to distinct operand types (as shown in Figure 1.6(b)) and can be categorized as:



(a) Row Elimination Sequence (b) Col Elimination Sequence

Figure 1.7: Conventional sequences for QR Decomposition

- Type X_Y: when both the entries on which rotation is performed are non-zero;
- Type X_0: when one of the entries is non-zero and the other is zero;
- Type 0_0: when both the entries are zero;
- Type 1_0 or 0_1: when one of the entries is one and the other is zero.

Each of these is associated with different sets of computations depending on the operation type, and hence, different energies. For example, the Type X_Y operation below requires 4 MULs and 2 ADDs.

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} c.X - s.Y \\ s.X + c.Y \end{bmatrix}$$

For other types, we replace X and Y by 0/1, etc., and determine the operations. Details are presented in Chapter 5. Table 1.1 summarizes the different operation and memory access counts when processing QRD for the conventional sequences (Figure 1.7). All the counts are in terms of scalar operations and individual memory word accesses. We observe that there is a trade-off between the memory accesses and computations – one sequencing strategy is better in one respect but worse in the other. Energy dissipation for the different categories are different and are architecture dependent; the final energy value depends on both counts. This analysis motivates us to determine an energy efficient data flow transformation for Givens Rotation based QRD.

Operations	Q Matrix		Input Matrix	
	Row Seq.	Col. Seq	Row Seq.	Col. Seq
X_Y	$\frac{2N^3-6N^2+4N}{6}$	$\frac{N^3-4N^2+5N-2}{2}$	$\frac{N^3-N}{3}$	$\frac{N^3-N}{3}$
X_0	$N(N-2)$	$N(N-2)$	0	0
0_0	$\frac{N^3-3N^2+2N}{6}$	$\frac{N^2-3N+2}{2}$	$\frac{N^3-3N^2+2N}{6}$	$\frac{N^3-3N^2+2N}{6}$
0_1	N	N	0	0
mem_acc	$2N^3-2N^2$	$2N^3-2N^2$	$\frac{4N^3-4N-6}{3}$	$\frac{4N^3-3N^2+5N-6}{3}$

Table 1.1: Memory access and operation counts for conventional rotation sequences

1.6 Effect of RF Size on FFT processing

An N point radix-2 FFT processing requires N memory reads, N memory writes, N addition/-subtraction operations and $N/2$ multiplications at every stage. Radix-2 FFT on input of length N is carried out in $log_2 N$ stages. Memory accesses being about 50 times more expensive in terms of energy consumption than the corresponding word length addition operation, leads to the memory energy contribution of up to 95% to the total energy consumption in FFT processing. Figure 1.8 shows the memory energy percentages for the number of inputs ranging from 50 to 500 in 1024 (= 1K) FFT for RF sizes varying from 4 to 64 (in powers of 2).

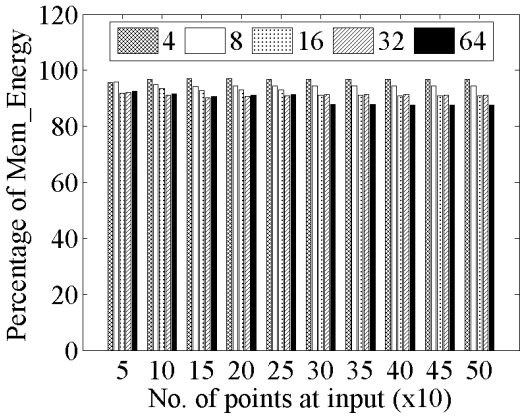
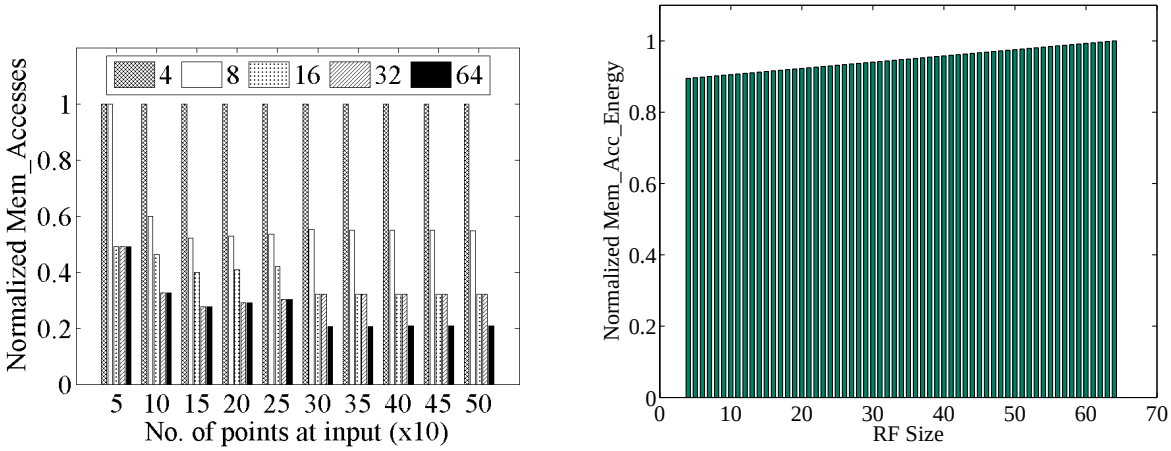


Figure 1.8: Percentage of memory energy in total energy consumption

We present an illustrative example on a 1K point FFT to motivate the need of RF size selection. We assume scalar registers in the configurable register file interfaced with SPM with the number of registers varying from 4 to 64. With the number of data points at input ranging from 50 to 500 (percentage of non-zero points being varied from 5% to around 50%), Figure 1.9(a) shows the variation of memory access counts for different RF sizes. We observe that the memory access count does not show a monotonic decrease with increasing RF sizes. Also the rate at which the memory access energy scales with increasing RF size (Figure 1.9(b)) is far lower than the rate at which the corresponding access counts decrease.



(a) Memory access count variation (b) Memory access energy variation

Figure 1.9: Motivational study

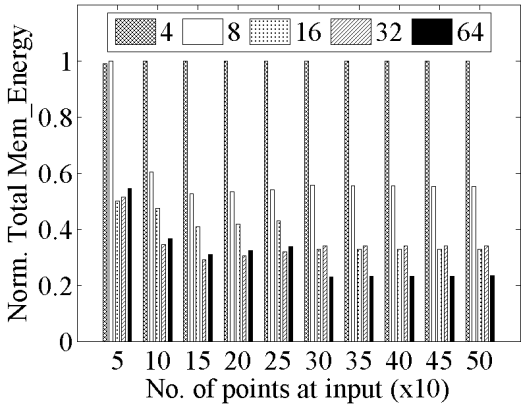


Figure 1.10: Memory energy variation with RF size

Taking into account both of these factors, we obtain the memory energy variations as shown in Figure 1.10. The percentage variation in memory energy is observed to be 50% to 75% over the range of input sizes for 1K FFT. Thus, the decision on energy optimal RF size is not straightforward and requires a detailed trade-off analysis. This observation motivates us to explore the RF size variations in more details. For the SIMD architecture under consideration, the decision on RF size will be a function of both the number of registers and the SIMD width W.

1.7 Contributions

In this thesis, we address the problem of data memory energy optimizations for SPM based SDR architectures supporting wide memory accesses. Efficient use of such architectures is possible only if the right data set is fetched from the memory, which in turn, requires an efficient data layout that aims at reducing the total number of memory accesses by maximizing reuse. We make the following contributions:

- Motivated by the need for an efficient data layout for SDR applications mapped to baseband architectures, we explore LTE (Long Term Evolution), a data intensive standard for wireless communication specified by 3rd Generation Partnership Project (3GPP), with useful applications in mobile phones and data terminals. Our focus is on the LTE MIMO downlink receiver. We study the inter- and intra- kernel data dependencies for the LTE MIMO 2x2 and 4x4 downlink receiver and propose and compare efficient data layouts for the application.

- We further extend the above study by investigating *Array Interleaving* – an energy efficient data layout transformation strategy, as a more broadly applicable compiler transformation, where the target system hardware consists of parallel SIMD functional units, vector registers, wide scratch pad memories and no hardware-controlled caches. We perform a global analysis of arrays accesses spanning different loops, and take interleaving decisions that are predicted to be globally optimal, based on estimations of the benefits and cost of interleaving. We also incorporate the effect of interleaving granularity in our exploration.

- We propose an energy efficient data flow transformation for Givens Rotation based QR Decomposition. We compare the proposed sequencing strategy with the conventional sequences for matrices of different sizes. We also explore different implementations for QRD of multiple matrices using the SIMD feature of the processor.

- A significant variation is observed in the data memory energy on varying the RF size for FFT on inputs having a large number of zeros. This motivated us to propose a strategy to determine the energy optimal RF size for the application. The strategy is defined for architectures supporting either scalar or vector registers. It is based on memory access count and compute count estimates across the FFT stages. We evaluate the proposed strategy by comparing the energy consumption over different register file sizes and varying SIMD width.

1.8 Thesis Outline

The rest of the thesis is organized as follows.

Chapter 2: Background: We discuss here some basic concepts in the context of this thesis. Beginning with an introduction to embedded system, we then introduce the different types of related architectures. This is followed by the related research on hardware and software based memory energy optimization techniques, which is the main target of this thesis. An overview of the polyhedral framework, used for automatic optimization and parallelization, is also presented.

Chapter 3: Long Term Evolution Wireless Standard: In this chapter the focus is on the LTE MIMO downlink receiver. We study the inter- and intra- kernel data dependencies for the LTE MIMO 2x2 and 4x4 downlink receiver and propose and compare efficient data layouts for the application.

Chapter 4: Array Interleaving – A data layout transformation: A technique for data memory energy optimization is presented here. We perform a global analysis of array accesses, and account for possibly different array behavior in different loop nests that might ultimately lead to changes in data layout decisions for the same array across program regions.

Chapter 5: Data Flow Transformation for QR Decomposition: In this chapter, we present an efficient data flow transformation strategy for the Givens Rotation based QRD that achieves energy efficiency through optimizing data memory accesses. We also explore different implementations for QRD of multiple matrices using the SIMD feature of the processor.

Chapter 6: RF Size Exploration for FFT: In this chapter, we present a strategy that determines an energy efficient Register File size based on memory access and compute count estimates. This exploration is especially relevant when there are large number of zeros at the input which is a common feature in our application domain.

Chapter 7: Conclusion and Future Work: We summarize our conclusions and identify possible directions for future research.

Chapter 2

Background

This chapter builds the background and context for the research addressed in this thesis. We begin with an introduction to embedded system in Section 2.1. We discuss the related architectures in Section 2.2. In Sections 2.3 and 2.4 we discuss some of the memory energy optimization techniques categorized as Hardware and Software based optimizations.

2.1 Embedded System

The optimizations presented in this thesis are targeted at low power embedded systems. We begin with an overview of an architectural model of an embedded system [9, 10]. It consists of Processor core; on-chip memory comprising data memory and instruction memory; and interconnect or communication network connecting the different components, which enable the data transfer between the cores and memories.

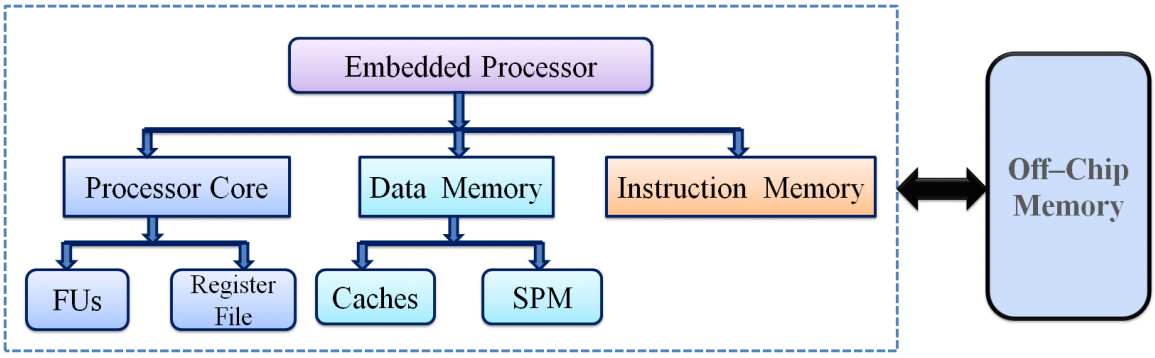
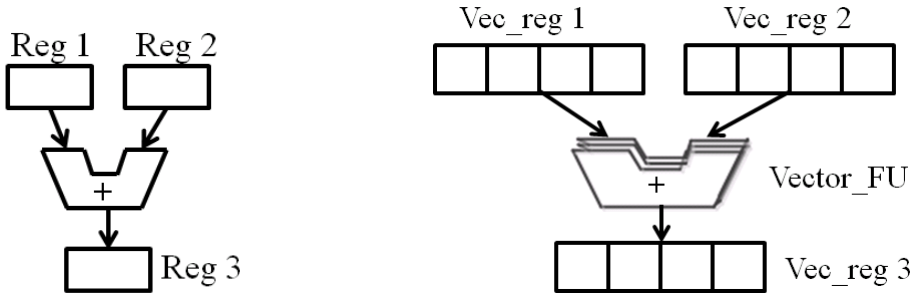


Figure 2.1: Embedded processor interfaced with off-chip memory

2.1.1 Components of an Embedded System

- *Functional Units* – The core consists of scalar or vector functional units capable of executing operations of different types, *e.g.*, addition, multiplication, and shifting. Scalar FUs are the hardware components that perform operations on a single operand while vector FUs can perform the same operation on multiple data in parallel. Figure 2.2 illustrates the mapping of data on different types of FUs.



(a) Scalar FU (b) Vector FU

Figure 2.2: Operation on Scalar and Vector FU in the datapath

- *Register File* – Register file is another component of the processor core. It may contain scalar registers or vector registers (registers capable of holding multiple words together). The vector registers are used to store operands for vector FUs and are thus useful for exploiting data level parallelism. Through the interconnect network the data operands are provided at the inputs of the FUs and the output is stored in the register file.

  The on-chip memory in an embedded system consists of data and instruction memory, each of which can be implemented as caches, or scratch pad memory or a combination of two. Other combinations such as ROM and embedded DRAM are also possible. The scratch pad SRAM has been presented to be both power and performance efficient solution [11, 12] for on-chip implementations.

- *Caches* – Cache Memory lies in between the processor and main memory in the computer memory hierarchy (Figure 2.3). The processor can access this memory much faster than main memory. Cache is relatively small and is used to keep a copy of recent data and instructions accessed by the CPU.

  Caches operate on the basic principle of *locality of reference* – programs tend to access only a small part of the address space at a given point in time. This principle involves two of these – temporal locality and spatial locality.

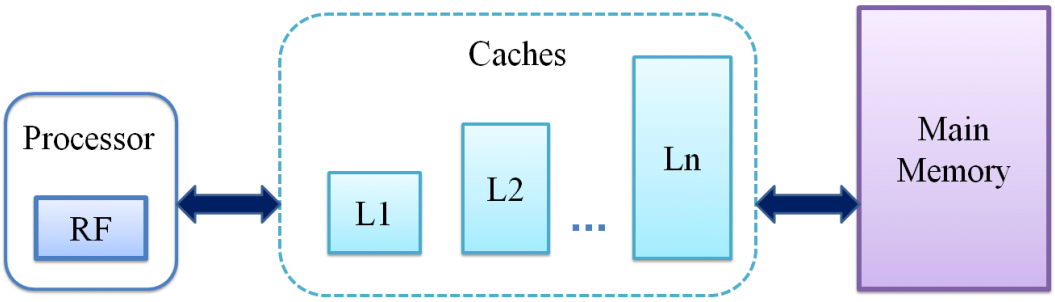


Figure 2.3: Memory hierarchy

- *Temporal locality* means the probability that the referenced memory will be referenced again soon is high.
- *Spatial locality* means that memory address in close proximity to the referenced address will be referenced.

Caches transfer a block of data (equal to cache line size) when an access to main memory is made. Line Size is the number of bytes per cache line, also referred to as block size. When a block of data is retrieved from main memory and placed in the cache, not only is the requested word loaded but also some number of adjacent words (those in the same block) are retrieved.

Each word in the main memory of 2^n words has a unique n bit address. This memory can be assumed to be composed of a number of fixed-length blocks of K words each. Thus, there are $M = \frac{2^n}{K}$ blocks. Figure 2.4 illustrates how a cache is split into C lines of K words each. Since the number of lines is considerably smaller than the number of main memory blocks ($C \ll M$), at any time, only a subset of the main memory blocks reside in the cache. If a word in a memory block is read, that block is transferred to one of the lines of the cache. Each line is associated with a tag that identifies which particular block of main memory is currently occupying that line of cache.

- *Scratch Pad Memory* – Scratch Pad Memory refers to an additional on-chip memory outside the cache hierarchy. It differs from the cache in that data storage is controlled by the programmer or compiler and the *hits* are guaranteed. Also SPM access times are more predictable while for caches the times may vary depending upon whether the access is a cache-hit or cache-miss. They are more commonly used in embedded systems and special purpose processors than in mainstream desktop processors.

  Effectiveness of the SPM was first studied by Panda et al. [13]. Data may be allocated to Scratch pad memories statically or dynamically. In static allocation, the mapping of data

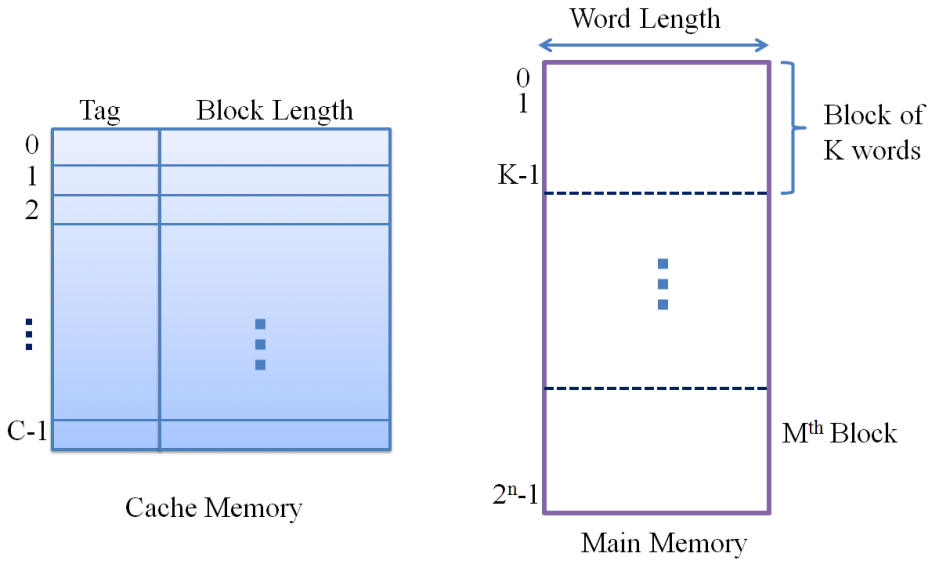


Figure 2.4: Cache / Main memory structure

to SPM remains invariant throughout the execution of the application. On the other hand, when loaded dynamically, the mapping changes during program execution. This feature is beneficial for irregular memory accesses [14].

Static allocation leads to reduced accesses to off-chip memory along with reduced energy consumption. Recent research has addressed dynamic allocation in SPM space. Approaches introduced in this field include run-time allocation using Presburger formulas [15] and block-wise copying of instructions to the SPM [16]. A liveness analysis based technique for dynamically overlaying the data and instructions on SPM is proposed by Verma et al. [17]. This is achieved by copying on to the SPM when required and copying-off otherwise. Such an analysis is carried out on the control flow graph for the code, where DEF-USE chains [18] are formulated for computing the liveness of the memory objects.

2.1.2 Pipelining and Parallelization Schemes

The execution of an instruction can be broadly divided into four stages - fetching, decoding, execution and write-back to memory [19]. With each stage being executed in one cycle, a processor may take four cycles to complete an instruction execution.

But what if the multiple stages are overlapped? That is, what if we have different instructions executing in different stages at the same time? This overlapping of multiple stages in every cycle is called pipelining and results in a maximum of n times speedup where n is the

number of stages in the pipeline. Figure 2.5 shows a pipelined micro-architecture with four stages.

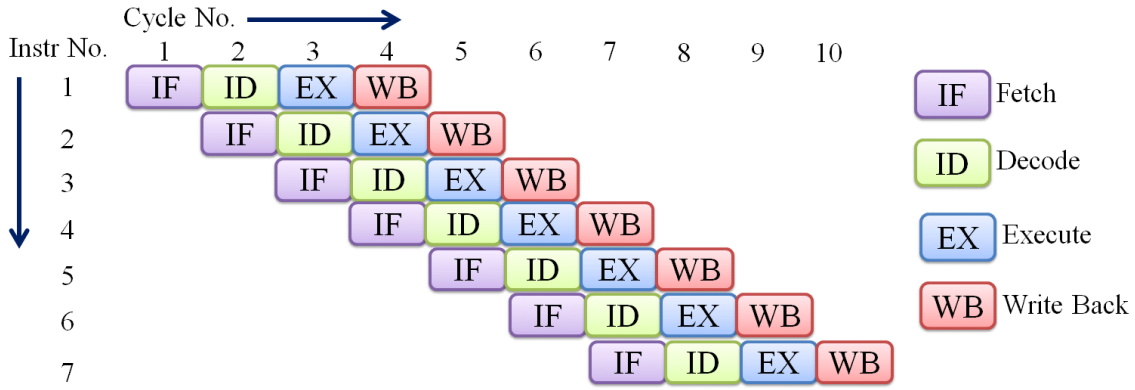


Figure 2.5: 4 stage pipeline

It is also possible to use the results available from the execute stage of one instruction for the next instruction in the pipeline without waiting for it to be written to memory and then accessed before its execution. This is achieved by providing a forwarding path/ bypass that allows data transfer in the earlier stages in the pipeline. Further sub-division of each stage may result in increased throughput by increasing the number of instructions completing per cycle. There may also be multiple instruction issue as in Superscalar processors. Aiming towards low power design and high performance for embedded systems has led to the development of many innovative architectures. To achieve these, it is required to make efficient use of available parallelization schemes in the system. Parallelism can be exploited at different levels – instruction level, task level, data level, bit level, etc. We briefly discuss the instruction and data level parallelization schemes.

- *Instruction Level Parallelism (ILP)* – This involves grouping of the instructions in the program that can be executed in parallel. For example, consider a sequence of instructions in a program.

```
1. A = B * C;
2. D = E * F;
3. H = A + D;
```

Instruction (3) depends on the outputs of instructions (1) and (2). It can be executed only after the completion of both, while the operands for instructions (1) and (2) are not dependent on any other instructions and can be executed in parallel. Thus, these

instructions can be executed in two cycles in parallel instead of three cycles when they are sequentially executed.

ILP can be exploited at two levels – Hardware and Software. Techniques such as out of order execution or speculative execution in superscalar processors are dynamic because of variable number of instructions scheduled per cycle and are implemented at hardware level. The software level approaches are static and are employed in Very Large Instruction Word (VLIW) processors. Here the compiler is responsible for packing a fixed number of instructions (that can be executed in parallel) into one large instruction.

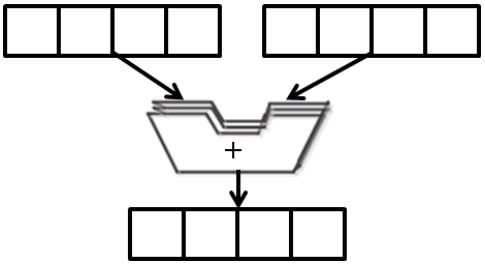
- *Data Level Parallelism (DLP)* – This is popularly known as loop-level parallelism. It is a form of parallel computing that focuses on distributing the data for computation across multiple processors. Data parallelism is achieved when different processors perform the same task but on different data. SIMD architectures exploit DLP. Consider an example loop shown in Figure 2.6(a). Here the operations in the i^{th} iteration are independent of the operations through iteration $(i+3)$. So all the four iterations can be executed in parallel, and the unrolled loop is of the form shown in Figure 2.6(b) and the instruction *addition* is executed as shown in Figure 2.6(c).

```
for(i=0; i<100; i++)
  A[i] = B[i] + C[i];
```

(a) Example Loop

```
for(i=0; i<100; i += 4)
  A[i] = B[i] + C[i];
  A[i+1] = B[i+1] + C[i+1];
  A[i+2] = B[i+2] + C[i+2];
  A[i+3] = B[i+3] + C[i+3];
```

(b) Unrolled Loop



(c) SIMD operation

Figure 2.6: Illustration for DLP

2.2 Related Architectures

Designing an embedded system requires us to consider different design metrics such as design and manufacturing costs, performance, power and time to market. System designing involves many trade-offs between performance and efficiency. Different architectures have different degrees of customization. On one end of the spectrum are the Application Specific Integrated Circuits (ASICs), that are most efficient for a given application but with least flexibility, while on the other end are the General Purpose Processors (GPPs) that possess the flexibility advantage but are the least efficient. Within this range, there exist multiple architectural possibilities such as Application Specific Instruction set Processor (ASIP), Field Programmable Gate Arrays (FPGA), Very Long Instruction Word (VLIW), and Coarse Grained Re-configurable Architectures (CGRA). For the desired performance in SDR applications, FPGAs and GPPs are not suitable because of high energy consumption, and are excluded from detailed discussion.

- *ASIC Architectures* – Application Specific Integrated Circuits (ASICs) refers to designs customized for a particular functionality and implementation of an application. The system implementation is divided into multiple functional blocks. For each block a dedicated unit is designed either manually at the Register Transfer Level (RTL) or by using High Level Synthesis tools.

- *ASIP Architectures* – Application Specific Instruction-set Processor (ASIP) architectures have instruction sets tailored for a special application. This makes the architecture less flexible than a general purpose CPU while the performance is better because of the use of application specific instructions. It can support multiple implementations of the same application. Compared to ASICs, these are programmable, enabling the support for different algorithms for the same functionality.

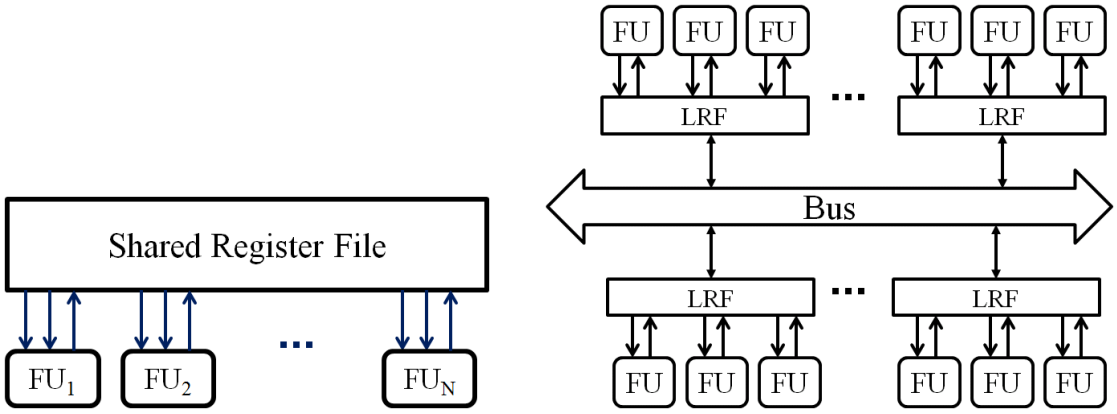
- *Baseband Processors* – Baseband processors are also ASIP architectures where the different functionalities in the baseband processing are incorporated. If an application is assumed to be divided into multiple functional blocks, the baseband processor provides support for multiple such blocks. These blocks can thus share the same hardware in the time domain. The Baseband processor instruction set also includes the domain specific instructions such as complex multiplications that are useful for the set of applications in a particular domain.

- *General purpose DSPs* – These signal processing oriented architectures are not designed for any particular application, and hence are called General Purpose DSPs. They find applications in multiple application domains, *e.g.*, in image processing, wireless and multimedia domains. Unlike ASIP Baseband processors, they do not include any domain

specific instruction set. Multiple functionality blocks of the application are implemented using a DSP and are shared among the blocks in the time domain. These are more widely applicable but due to absence of domain specific instruction set they are less efficient as compared to ASIPs.

- *VLIW Processors* – VLIW processors are designed to exploit Instruction Level Parallelism (ILP). Here, the decision on selection and execution order of the instructions is handled completely by the compiler. The VLIW approach does not require any additional scheduling hardware as required by Superscalar processors. As a result, these processors with less hardware complexity (and larger compiler complexity) offer high computational power. TriMedia media processors [20] and C6000 DSP series [21] are examples of VLIW processors.

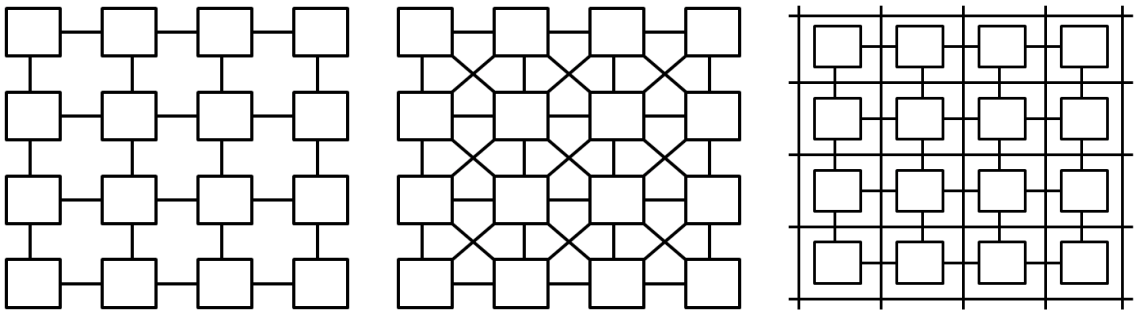
  The number of FUs in VLIW processors has to be carefully decided because of the impact on the delay, area, and power of the unified register file (Figure 2.7(a)). Rixner et al. [22] showed that for N FUs, the RF area grows as N^3, delay as $N^{\frac{3}{2}}$, and the power dissipated rises by N^3. This led to the clustered VLIW approach. In clustered VLIW processors, the processor is divided into multiple clusters, with each cluster consisting of a local register file and some FUs (Figure 2.7(b)).



(a) Unified VLIW Processor (b) Clustered VLIW Processor

Figure 2.7: VLIW Processors with Unified and Clustered RFs

- *Coarse Grained Re-configurable Architecture (CGRA)* – CGRAs are a class of re-configurable architectures with a configurable FU that can support a set of word or sub-word level operations (such as addition, subtraction, multiplication), selected by configuration bits stored in SRAM. CGRAs could comprise hundreds of FUs connected in topologies such as those shown in Figure 2.8.



(a) 4N× N (b) 8N× N (c) 4N× N with buses

Figure 2.8: Different inter-connect topologies and a re-configurable unit

Some of the early known CGRAs include Kress Array [23], a purely dataflow architecture that lacks support for applications with multiple re-configurable contexts, *e.g.*, the loop body containing if-else control flow. MorphoSys [24], another CGRA, consists of 8× 8 re-configurable cells (RCs) grouped into 4 tiles with each tile containing 4× 4 RCs. RaPiD [25] is another one-dimensional CGRA supporting pipelined dataflow. These architectures have distributed register files and a resource element in the mesh may be used for routing or computation. Having an efficient compiler for such architectures is a major challenge.

ADRES-DRESC Framework

ADRES is an Architecture for Dynamically Re-configurable Embedded Systems (Figure 2.9) and is supported by DRESC (Dynamically Re-configurable Embedded Systems Compiler) [26, 27, 28]. The main features of ADRES architecture include tight coupling between VLIW processor and the re-configurable array.

The ADRES architecture overcomes some of the common problems in CGRAs. A general trend in designing involves bottom-up style, *i.e.*, the architecture is followed by the compiler techniques. Thus, there exist architecture features that may not fit the compiler's needs. The ADRES-DRESC framework has been designed in the top-down style, *i.e.*, DRESC, the compiler, designed first followed by ADRES architecture incorporating the desirable features.

The configuration model supports two execution modes:

1. *VLIW mode* to execute control instructions, and

2. *Array mode* to execute kernels – functions with multiple loops applied to each element in the data stream.

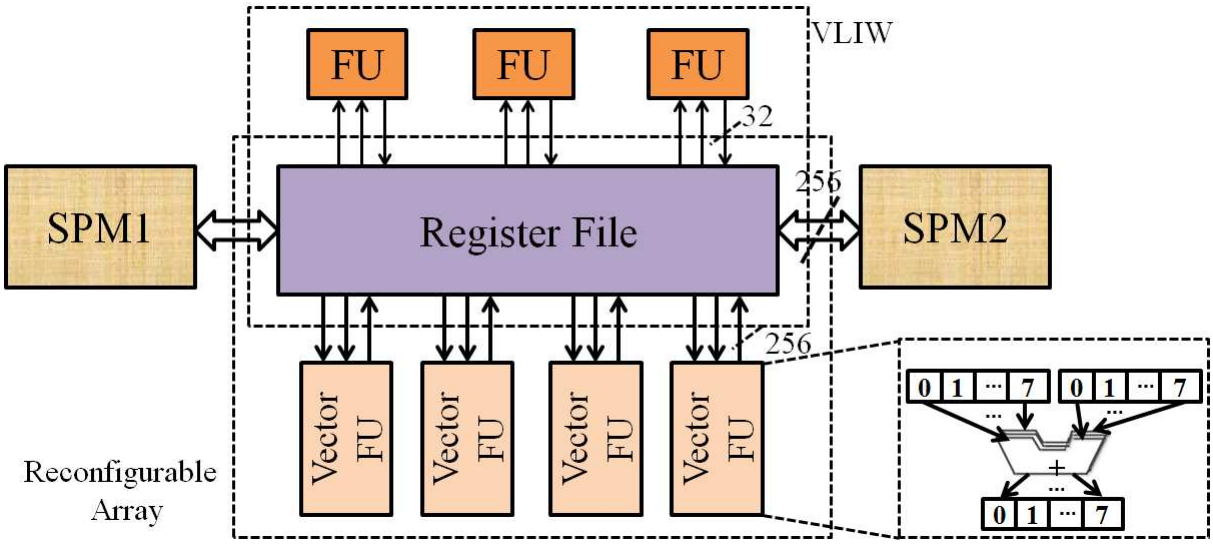


Figure 2.9: ADRES architecture template

The VLIW mode helps in exploiting Instruction Level Parallelism (ILP) while the array mode exploits Data Level Parallelism (DLP). For VLIW mode, instructions are fetched and executed in each cycle, while in array mode, execution is based on configured contexts in the configurable memory. Vander Aa et al. proposed a way to achieve maximum resource utilization in ADRES, by exploiting the available parallelism schemes in a power efficient way for SDR domain applications [29]. An XML based language is used to describe the architecture, and a cycle-accurate simulator of the processor is used to simulate the generated code on the architecture. Figure 2.9 gives an overview of an architecture instance [29, 30, 31] which represents an evolution over ADRES architecture [27] and is used for experimentation and validation of the proposed optimizations in this thesis. The CGRA part comprises N SIMD enabled functional units (FUs) and a central vector register file. The register file is also coupled to a VLIW processor with 3 FUs. For efficient utilization of the vector FUs, the register file has a wide interface with two scratch pad memories. The register file interface with the SIMD units is configurable while the interface with VLIW FUs is 32 bits wide.

2.3 Hardware Based Memory Energy Optimization

Power becomes an important constraint in the design specification of embedded systems. Several methodologies have been explored for building power efficient processors and memories. Focusing on the objective of this thesis, we now discuss the related research on energy efficient memory design and mapping. A large number of hardware and software based memory optimization techniques have been discussed in a survey on energy-aware design [32]. Hardware based optimization techniques include memory partitioning, extending the memory hierarchy

by introducing various levels in the entire address space, the width at the processor memory interface and so on. Different techniques may be used to perform data and memory optimizations to obtain a power efficient design. In this section we discuss some of the hardware based optimization techniques.

2.3.1 Register File Context

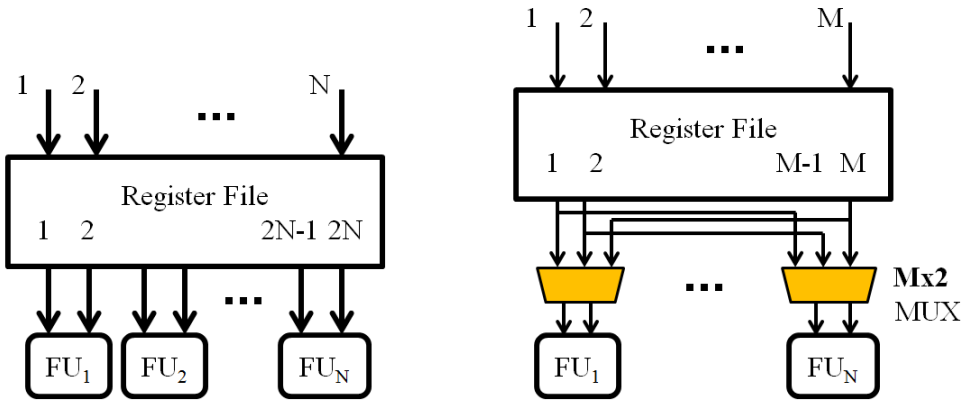
Register File is an array of registers used to store data between memory and function units while executing instructions in a processor. To explore parallelism, *i.e.*, to issue a large number of instructions in a single cycle, requires a correspondingly large number of operands to be available in the registers. They also need to be accessed simultaneously. This requires a multi-ported register file along with large register file size leading to complex addressing logic. All these contribute to longer access times and higher power dissipation. We describe some well known techniques to overcome these problems.

- *Port Reduction in Register File* – An *N*-issue processor requires *2N* read ports (considering two source operands per instruction) and *N* write ports in a register file. This number becomes very large for large values of *N*. Large number of ports leads to increased complexity and inter-connections, leading to large power dissipation.

  In an *N*-issue processor, there may be cases when the number of ports required is $\leq 3N$ [33]. This may arise due to several reasons.

  - Some instructions may require less than two operands.
  - Instructions such as stores and branches need not write to the register file.
  - In the pipelined execution, many operands can be read through the bypass network instead of the register file. This number can be increased by using auxiliary buffers to store some results for some cycles before sending to the register file. So, a *2M* ported register file (Figure 2.10) may be used instead which would lead to smaller area, lower access times as well as energy per access, though it may result in reduced instructions per cycle (IPC) because of the port conflicts that may arise. The overhead associated with this implementation is the extra MUXes required to direct the data from *M* read ports to the function units' inputs.

- *Banking Register File* – One architectural possibility is to split the register file into *N* banks each with *p* ports, which reduces the complexity and access energy (Figure 2.11). This banked RF architecture can provide $N \times p$ operands within a cycle, a feature similar to that of a single RF but with a limit on the number of operands from a single bank (here *p*). Thus, for optimal use of such an architecture a logical distribution of operands

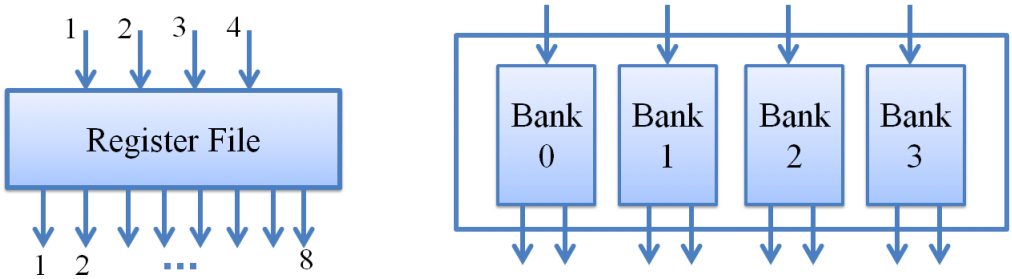


(a) N-issue RF with 3N ports

(b) RF with reduced number of ports (2M ≪ 3N)

Figure 2.10: Register File organization

among the banks is needed. Operands needed in the same instruction should be evenly distributed across the banks.



(a) 4-issue RF

(b) Banked architecture

Figure 2.11: (a) A 4-issue processor RF with 8 read ports and 4 write ports (b) A banked architecture with 4 banks, each bank having 2 read and 1 write port.

- *Asymmetric Register File* – Exploring for the energy efficient implementations of the register file, sometimes it is desirable to have a single ported register capable of exploiting spatial locality. This requires a wider interface of the register with the memory (as wide as the on-chip memory line size). However the data being accessed together need not undergo the same operation. This leads to a requirement of a narrower interface of the registers with the datapath. Raghavan et al. [34] proposed such an asymmetric register file architecture called *Very Wide Register (VWR)*. Such architectures could be power efficient for embedded applications, SDR and DSP applications in particular, that have streaming

data with high spatial locality. However because of dependencies in these applications data level parallelism is limited which limits the interface width towards the datapath. Figure 2.12 shows a VWR interfaced with both memory and datapath. VWR is a single ported register; thus both memory and datapath cannot be simultaneously accessed.

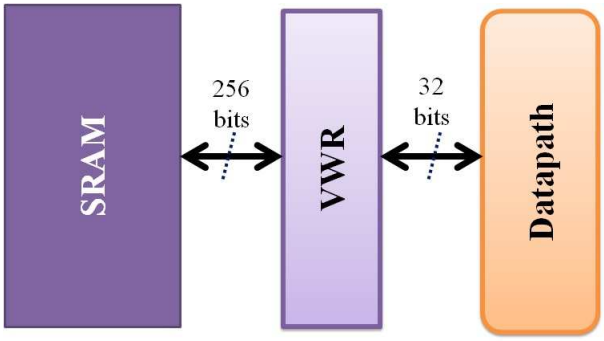


Figure 2.12: Very Wide Register (VWR)

2.3.2 Memory Modeling and Customization

The objective behind generating power efficient memories is low energy per access while maintaining the area and performance constraints. We discuss some of the related techniques proposed by researchers.

- *Additional Memories* – Memory space management plays a very important role in improving the performance of embedded systems. Adding small sized caches or buffers in the memory hierarchy (Figure 2.13) helps in achieving significant power savings, if they are relatively frequently accessed. Thus, by maintaining/ buffering proper data at the lower levels of the memory hierarchy, good power savings can be attained. Some of these additional caches include loop cache [35] and filter cache [36]. All these caches and buffers are dynamically assigned data and instructions.

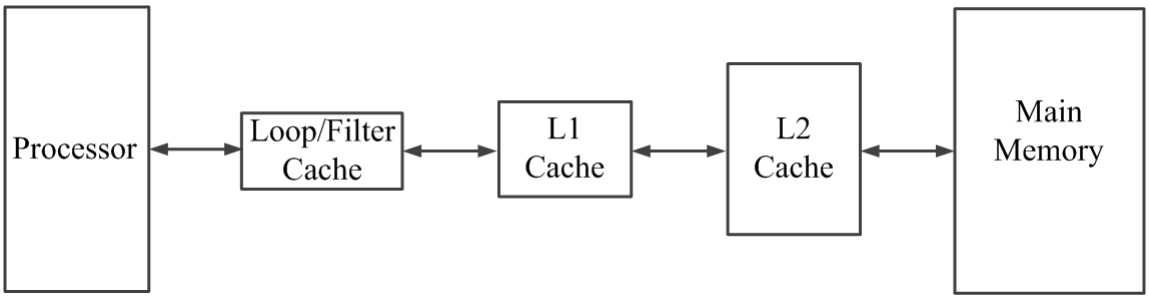


Figure 2.13: Additional caches in the memory hierarchy

Scratch Pad Memory (SPM) was introduced as a part of on-chip data memory along

with the hardware-controlled cache by Panda et al. [37] (Figure 2.14). It behaves differently from the cache by guaranteeing short access latency, as opposed to caches where the access latency varies depending on hits and misses. Mapping of frequently accessed application data on to SPM helps in minimizing the overall execution time of the application.

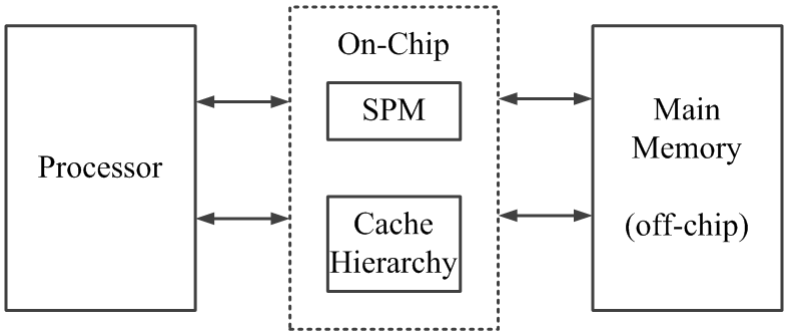


Figure 2.14: Off-chip and on-chip memory

- *Memory Banking* – Memory banking is a form of organizing the memory by partitioning the available memory/address space into multiple smaller banks/memories. This organization reduces the number of bits required to address a memory line or a location, thereby, leading to power savings. Also, the memory accesses per cycle can increase as the organization allows parallel access from all the banks (Figure 2.15). This memory organization strategy leads to improved performance.

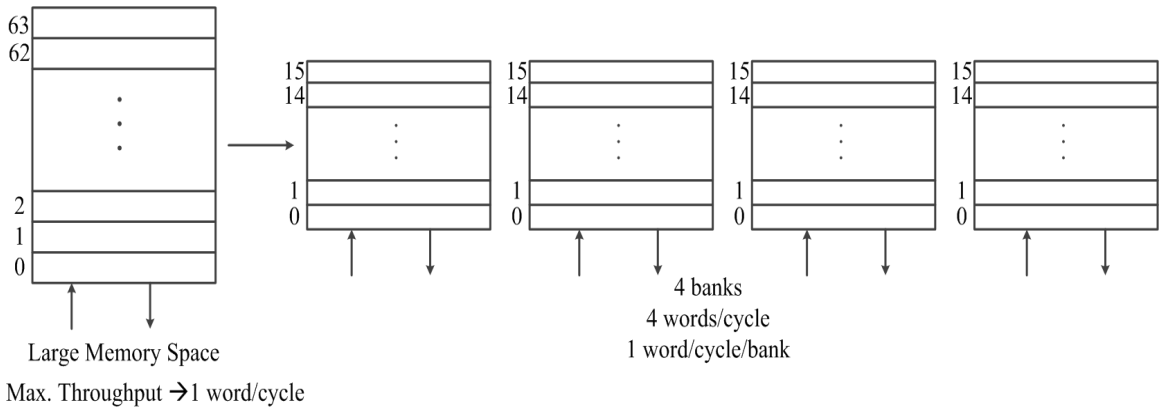


Figure 2.15: Memory banking

- *Memory Customization* – Memory can be customized for efficient utilization of the available memory for a specific application. Memory banks and variable assignment to the banks provide good customization opportunities. The problem of associating the variables to memory banks has been formulated using Kernigham-Lin graph partitioning

problem [38, 39]. The cost function used for partitioning of variables into groups is *delay* for the resulting schedule. The schedule is obtained using Memory Dependence Graph (MDG) for each basic block that contains the information regarding the memory dependencies in the respective blocks. Finally, a schedule is generated after partitioning the MDGs such that the number of page misses is minimized. It has also been observed that increasing the number of banks does not always decrease the delay or increase the performance due to the presence of data dependencies. Thus, the data dependencies impose a limit on the performance improvement with increasing number of memory banks.

Though increasing the number of banks reduces the energy per memory access, it comes with a large interconnect overhead. For applications with a large number of arrays, the energy consumed in memory accesses may be very large. Memory banking may prove to be useful by powering up and down the banks based on the requirement. This strategy for energy reduction is discussed by Kandemir [40]. The address space is partitioned into multiple banks. Each of these banks may be powered off if it is not active beyond a certain number of cycles. This may be useful provided data layout is done efficiently, so that repeated powering off and on of the memory banks does not waste cycles and energy.

A way to use the scratch pad memory in multiprocessor embedded systems to improve energy behavior is discussed by Kandemir et al. [11]. Each core has a local SPM. The sharing of the data in the shared SPM is achieved using a communication channel among the cores. This architecture improves both performance and energy consumption by reducing the accesses to off-chip memory for a loop level parallelized array-intensive application. The off-chip accesses are reduced by selecting proper data tile size and tile access pattern.

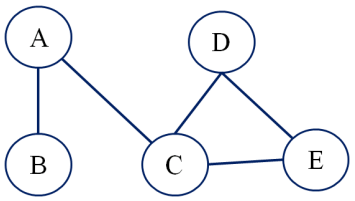
The work on memory banking discussed above assumed partitions or banks of identical sizes, thereby simplifying the problem. By employing non-uniform bank sizes memory energy consumption may be reduced [41]. Integer Linear Programming (ILP) is used to determine the size of the memory partitions with energy consumption as the objective function to be minimized under certain constraints. These constraints are updated after every step to account for data migration from one bank to another. Data migration to an active bank is sometimes more energy efficient than powering on and powering off of a bank. This concept of migrating the data block from one bank to another helps in reducing switching activity of the banks (thereby reducing the energy consumption) by keeping an already active bank active for a longer time and an inactive bank in sleep mode for longer duration.

2.4 Software Based Memory Energy Optimization

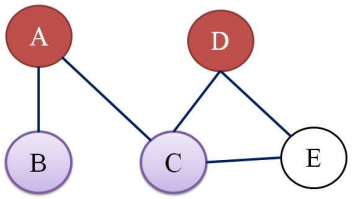
Compiler driven transformations may be architecture dependent or independent. In this section, we survey some of compiler based memory energy optimization techniques. Some of the early works in this area have been summarized by Panda et al. [42].

2.4.1 Register Allocation

In modern day computing systems, performance is highly dependent on the access latencies of the memory hierarchy. Registers, being closest to the processor, have the least access latency. Intelligent utilization of the available registers is thus very important for system performance. Thus, *Register Allocation*, the problem of assigning registers to a set of variables during the execution of a program, is one of the most important optimizations in the compiler domain. Assigning registers to the frequently accessed variables helps in minimizing the number of memory accesses, thereby improving performance.



(a) Interference Graph



(b) Graph Coloring

Register	Variables
R_1	A, D
R_2	B, C
R_3	E

(c) Register Allocation

Figure 2.16: Graph Coloring: A *Register Allocation* technique

Graph Coloring is the most popularly used technique for Register allocation. It involves building a variable-conflict graph (also called Interference Graph) where each program variable is represented by a node in the graph and an edge between the nodes indicates that the two variables are *live* simultaneously. The *Lifetime* of a variable is defined as the time between the generation of a value to its last use. Variables with non-overlapping lifetimes can share the same register. Once the graph is built it is then colored with n colors, given n is the number of registers available. Coloring is done in such a way that adjacent nodes (connected by an edge in the graph) are not assigned the same color. Figure 2.16 illustrates this register allocation algorithm with an example.

2.4.2 Code Transformations

Optimization is the process of code transformation to make it more efficient in terms of performance, memory space requirements, or power while maintaining the output unchanged. Code optimization is an active research field in the compiler domain.

Architecture independent transformations are effective on all platforms and their objective is to improve both the data as well as control dependence analysis. Induction variable elimination, false data dependency elimination, normalization, forward substitution, and loop fusion are some of the transformations in this category. Architecture dependent transformations use some architecture specific properties or depend upon parameters defining the architecture. Transformations such as loop interchange, loop reversal, skewing, permutation, strip mining, and tiling fall in this category [43, 44]. We discuss some of these code transformations that help in reducing the memory accesses.

- *Loop Unrolling* – This is a compiler optimization that can significantly speed up the program's execution by exploiting parallelism across loop iterations. The optimization has both advantages and disadvantages. Unrolling leads to performance improvement by reducing the control instructions, exposing parallelism across the iterations, although the process of duplicating the loop body multiple times leads to increase in code size that may result in increased I-Cache misses.

```
for(i=0; i<16; i++)
  A[i] = B[i] + C[i];
```

(a) Original loop

```
for(i=0; i<16; i+=2) {
  A[i] = B[i] + C[i];
  A[i+1] = B[i+1] + C[i+1];
}
```

(b) Loop unrolled once

Figure 2.17: Illustration for Loop Unrolling transformation

Figure 2.17 illustrates the unrolling transformation. The original loop (Figure 2.17(a)) forces sequential execution, while on unrolling (Figure 2.17(b)) multiple statements can be executed in parallel as there are no dependencies across iterations.

- *Loop Fission* – A loop body may have multiple statements accessing a wide range of memory address space across the statements. A small cache may incur a large number of conflict misses thereby degrading the system performance. If this loop is distributed into multiple loops with the same index range, it would result in more localized data accesses,

thereby reducing the number of cache misses. An example of this compiler optimization is shown in Figure 2.18.

```
for(i=0; i<500; i++) {
  A[i] = A[i] * 5;
  B[i] = B[i] + 10;
}
```

(a) Original loop

```
for(i=0; i<500; i++)
  A[i] = A[i] * 5;

for(i=0; i<500; i++)
  B[i] = B[i] + 10;
```

(b) Loop Fission applied on original loop

Figure 2.18: Example illustrating Loop Fission transformation

- *Loop Interchange* – This optimization changes the loop permutation by exchanging the nesting order of loops. It helps in improving the spatial locality for some array accesses in the nested loop depending on the array access indices.

```
for(j=0; j<100; j++)
  for(i=0; i<500; i++)
    A[i][j] = A[i][j] * 10;
```

(a) Original loop nest

```
for(i=0; i<500; i++)
  for(j=0; j<100; j++)
    A[i][j] = A[i][j] * 10;
```

(b) Transformed loop nest

Figure 2.19: Loop Interchange transformation example

Consider an example loop shown in Figure 2.19(a). The Loop interchange optimization transforms the loop to the one shown in Figure 2.19(b). With the array accesses being contiguous for the loop with index variable j, the transformation reduces the number of cache misses significantly, thereby improving the system performance.

- *Loop Fusion* – This is an optimization technique to reduce the loop overheads. If two loops have identical iteration count, loop bounds and increment factors, combining them results in reducing overheads associated with branching and conditional evaluation with separate loops.

- *Blocking (or Tiling)* – The above compiler optimizations reduce the cache misses, thereby memory accesses, by making use of spatial locality. Blocking is an optimization technique that exploits both spatial and temporal locality to reduce the miss count. The iteration space is divided into blocks such that the data required for computations for that

```
for(i=0; i<500; i++)
  A[i] = A[i] * 5;

for(i=0; i<500; i++)
  B[i] = B[i] + 10;
```

(a) Original loop

```
for(i=0; i<500; i++) {
  A[i] = A[i] * 5;
  B[i] = B[i] + 10;
}
```

(b) Loop Fusion applied on original loop

Figure 2.20: Example for Loop Fusion transformation

block fits in the data cache. Consider the example code of matrix multiplication (Figure 2.21) and the corresponding transformed code when the iteration space is divided into blocks of size $Bsize \times Bsize$. (Figure 2.21(b)). With arrays A, B, and C being too large to be held together in the cache, the original array accesses lead to a large number of main memory accesses. With efficient *Blocking*, the number of main memory accesses is significantly reduced.

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

(a) Original loop nest

```
for(jj=0; jj<N; jj+=Bsize)
  for(kk=0; kk<N; kk+=Bsize)
    for(i=0; i<N; i++)
      for(j=jj; j<min(jj + Bsize, N); j++)
        for(k=kk; k<min(kk + Bsize, N); k++)
          A[i][j] += B[i][k] * C[k][j];
```

(b) Transformed loop nest

Figure 2.21: Blocking transformation

2.4.3 Data Layout

Efficient data layout in memories may result in significant power savings. In this section we survey some of the techniques explored by researchers at various levels in the memory hierarchy that result in an efficient data mapping.

Manjikian and Abdelrahman [45] proposed a layout technique, called Cache Partitioning, for arrays to reduce cache conflicts. Applications consisting of nested loops with multiple large arrays incur a large number of cache conflicts. This leads to repeated fetching of the reusable data and thus degrades the performance. Cache Partitioning reduces cache conflicts by performing logical distribution of the cache capacity with memory allocation to arrays in a way that each array maps to a different cache partition. This ensures that reusable data is not ejected from the cache and hence maximizes the benefit when combined with loop transformations enhancing data locality. Various techniques such as introducing dummy words, and interleaving arrays have been introduced by Panda et al. [46] to reduce the cache conflicts. Sequential memory accesses also result in reducing the switching activity in the address bus as compared to the memory accesses at random addresses. Thus, memory data layout, if done using a prior knowledge of the application being mapped, can prove to be very power efficient.

Computation Decomposition and Alignment (CDA), a loop transformation technique [47], involves loop re-structuring at a very fine granularity. Breaking a statement into smaller statements provides additional optimization opportunities that may vary from statement to statement. The transformation modifies the loop structure which changes the execution order of iterations in a way that reduces cache conflicts.

Cierniak and Li [48] and Kandemir et al. [49] proposed methodologies aiming to reduce cache misses by combining loop and data layout transformations. Combining both transformations in a framework helps in improving both temporal and spatial locality thereby improving the overall performance. Several loop and data transformations leading to loop re-structuring that yield improved data locality and exploit parallelism have been discussed by Kulkarni and Stumm [43].

Energy reduction in memories with multiple banks is possible if data is assigned in such a way that minimum number of banks are kept active. The array interleaving approach [50] works out a transformation of arrays in such a way that most of the banks are in low power mode while maintaining the required performance. It makes sure that mapping is done such that the banks remain in low power mode as long as possible. Interleaving results in better spatial locality, and reduces the number of memory accesses, thereby giving the opportunity for banks to remain in low power mode for extended time periods.

Kim et al. [51] proposed a mathematical formulation that considers data locality in SDRAM. The idea here is to reduce the switching between the pages. It is based on minimizing the

number of page misses encountered during memory access. Here, instead of storing arrays in row major or column major form, they are divided into tiles. To ensure that the complete tile lies on the same page, the tile size should be appropriately chosen. This approach is observed to be efficient in video applications.

Most of the data layout related strategies proposed until the late 90's were performance oriented. Brockmeyer et al. [52] proposed an energy efficient data mapping strategy for an application mapping in the memory hierarchy. This optimization methodology for data memory hierarchy is known as Memory Hierarchy Layer Assignment (MHLA). It requires a global analysis of the arrays so that an appropriate layer is assigned to each data set in a way that minimizes the access cost. The methodology also determines the type of memory (single/dual port) to be assigned while taking data reuse into account.

Kulkarni et al. [53] addressed the problem for cache miss reduction for multimedia applications. The data layout problem is formulated in two stages – firstly, evaluating the tiling size for arrays; secondly, merging the arrays appropriately for each loop nest. The approach involves splitting the arrays into sub-arrays, each of which is a multiple of cache line size. Tile sizes are evaluated for each array depending on the array size in each of the loops, the number of simultaneously accessed arrays and the cache line size. This is followed by clustering/grouping the arrays in the loop according to their priority. The loops are prioritized based on the total number of array accesses involved in them (sum of all the array elements being accessed in the loop).

An intelligent mapping to SPM has been proposed where certain sharing strategies have been adopted for multitasking embedded systems [54]. The sharing strategies have been classified as: Non-Saving, Saving, and Hybrid. In the Non-Saving approach, each task is assigned a part of the SPM. The SPMs for processes/tasks are disjoint. In the Saving approach, only the overlapping region in all the processes is mapped to SPM. No process is individually allocated memory space. The Hybrid approach, as the name suggests, has features of both the strategies discussed above. Here each process is allocated a disjoint memory space and also a part of memory space is reserved only for overlapping regions of all processes.

Cho et al. [14] proposed data organization for irregular array access patterns. Power wasted in memory accesses can be minimized by making proper selection of the data to be mapped to Scratch Pad memory and at other levels in the memory hierarchy. Decisions in such cases are made based on the re-usability factor. The array elements with higher factor value are mapped on to on-chip memory thereby leading to improved power savings.

Memory and data layout optimizations developed for architectures supporting caches and scalar registers are not directly applicable for our target architecture that does not have caches, and instead supports wide scratch pad memories and vector registers. These differences in architectural parameters require new data layout strategies as the use of vector registers imposes

restrictions on the elements accessed together in the register. Elements in the vector register are subjected to the same operation in parallel using the SIMD FUs in the target architecture. This leads to a requirement on carefully defining the data layout for the target architecture to prevent redundant accesses and SIMD operations.

Chapter 3

Long Term Evolution Wireless Standard

In this chapter, we consider SDR baseband implementations for Long Term Evolution (LTE), a high speed cellular data communication standard, specified by 3rd Generation Partnership Project (3GPP). The LTE standard comes with a goal of achieving superior spectral efficiency and mobility support by using a large number of advanced air interface techniques. However, the superior performance comes at the cost of substantially increased computation and memory complexity, which motivates the need for highly optimized energy efficient implementations. This is especially important for SDR baseband processors aiming at high rate data transmission in the portable transreceiver.

We focus on the user data processing block in LTE MIMO downlink receiver as it is the data- and computation-intensive part with tight energy and latency constraints. Its SDR implementation involves kernels that have computationally intensive operations such as FFT, channel estimation, channel interpolation, MIMO equalizer computation, Log-Likelihood Ratio (LLR) computation, and Payload, along with complex memory access patterns due to the Multiple Input Multiple Output (MIMO) feature. All of these strongly motivate the need to explore the data and memory optimization opportunities for the standard.

Data and memory optimizations have been the topic of a significant amount of recent research efforts because of the dominating role of memory in system design. Data layout optimizations aim to arrange data in memory with the objective of improving system performance and energy through means such as reduced memory access count or reduced cache misses. Optimization of the memory architecture and hierarchy for application specific systems generally needs to be accompanied by appropriate data layout decisions for the architecture. Most of the early works in data layout address layout opportunities for cache based memory hierarchies [45, 47, 48, 49, 53, 52, 9, 55]. These approaches for memory hierarchy comprising caches and scalar registers are not directly applicable to scratch pad memory and vector register based SDR architectures as the difference in architectural aspects leads to variation in data layout

strategies.

We leverage the programmability of SDR platforms and propose systematic data layout optimizations for architectures supporting wide memory accesses while mapping for 3GPP-LTE. Exploration for an efficient data layout requires studying the data dependencies across and within the kernels. Following this study, decisions can be made on how the data should be read out for a kernel and written back in an efficient way for the next kernel. We study the inter- and intra-kernel data dependencies for the LTE MIMO downlink receiver, and proceed to propose and evaluate efficient data layouts for the application.

This chapter is structured as follows. We discuss the related research in Section 3.1. Sections 3.2 and 3.3 introduce the basic terminology associated with LTE and give an overview of the LTE MIMO applications. In Section 3.4, we discuss in detail the exploration for interleaving decision, covering its impact, and define the strategy for interleaving. Exploration results are discussed in Section 3.5. Finally, we conclude with a summary of this study in Section 3.6.

3.1 Related Work

LTE standard is an outcome of the commercialization of the MIMO technology. MIMO techniques enhance the system performance by improving the data throughput and spectral efficiency [56]. However, this is accompanied by an overhead of increased system complexity. With the use of multiple antennas at both ends of the communication system a linear increase in the channel capacity can be obtained for the same bandwidth. Significant research has been performed on improving system throughput by using multi-user MIMO networks. Optimizing such systems to control the transmitted power while maintaining a Quality of Service (QoS) for each user is challenging. Methods for spatial multiplexing in such channels have been developed [57, 58, 59] based on taking optimal decisions on transmit vectors, overcoming co-channel interference and exploiting multi-mode switching. Solutions for energy and performance improvement at different stages of the receiver have been proposed in the recent past. Schwarz et al. [60] have presented different methods for evaluating an optimal pre-coding matrix under different conditions. An iterative method for improvement in channel estimation has been proposed by Luis et al. [61]. The proposed schemes provide better performance and reasonable gain in spectral efficiency at a reduced complexity. None of the above works on LTE propose a data organization strategy for the application.

Several memory- and data-oriented optimizations targeting energy efficiency have been identified in the recent years [62, 63, 10, 64]. To the best of our knowledge, no work has been presented yet for an efficient data layout for LTE while considering cross-kernel dependencies. Compilation techniques presented earlier (in Section 2.4), including those developed

for GPUs and cache based memory hierarchy techniques, do not find a straightforward application in our context, where the target system hardware consists of parallel SIMD functional units, vector registers, wide scratch pad memories or software-controlled caches, and no hardware-controlled caches.

3.2 LTE Basics

This section introduces the basic terminology associated with the channel configuration for the LTE application [65, 5]. Channel allocation depends on the mode of transmission; it is made to a single user if it is a localized transmission or distributed among the multiple users for a predetermined amount of time in a distributed transmission mode. The communication channel is thus configured along both the time and frequency dimensions.

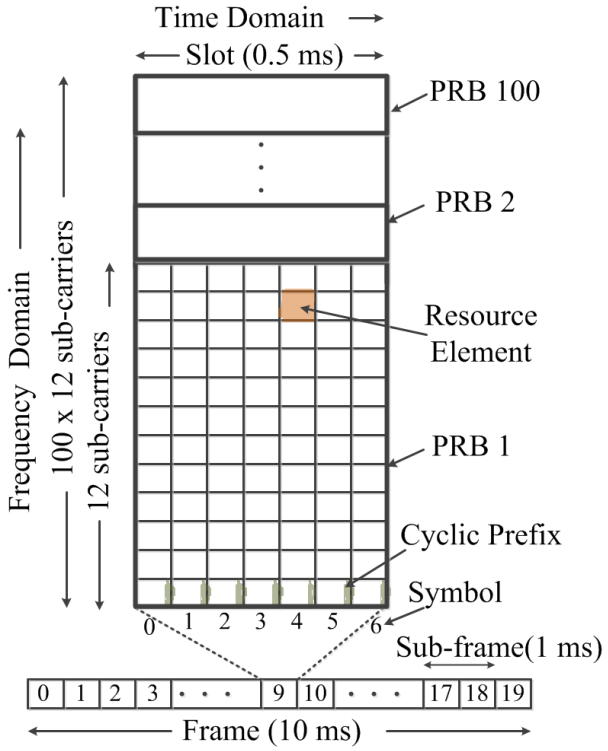


Figure 3.1: Downlink Resource Grid for channel bandwidth = 20 MHz, # PRBs=100

3.2.1 A Time Domain View

Transmission is based on *frames* of 10 ms duration. Each frame consists of 10 *sub-frames* that are divided into 2 *slots* each as shown in Figure 3.1. Each slot is made up of 6 to 7 OFDM *symbols* [65] depending on the *cyclic prefix* length. One sub-frame is the smallest unit of time

allocated to a user. In the next sub-frame, a different user may be allocated that frequency band. Each symbol is augmented with a cyclic prefix to prevent inter-symbol interference. At the user end the latency constraint for the processing is 5 ms.

3.2.2 A Frequency Domain View

The channel bandwidth is divided into *Physical Resource Blocks (PRBs)*. The lower the bandwidth, the lower the number of PRBs. Each PRB consists of 12 sub-carriers of 15 kHz bandwidth and is constant irrespective of the bandwidth under consideration. This splitting of the sub-carriers into resource blocks enables the system to divide the data across a pre-defined (standard) number of sub-carriers. *Resource element* is the smallest constituent of the resource grid and represents one single sub-carrier for one symbol period. The number of resource elements in a PRB is thus given by

$$\#elements\ in\ a\ PRB = 12 \times \#OFDM\ symbols\ in\ a\ slot \tag{3.1}$$

In order to facilitate channel estimation and timing synchronization, some of the resource elements are reserved for the transmission of special reference symbols (*pilots*) according to 3GPP standards and are not available for user data. Figure 3.2 shows their distribution in a PRB for the LTE 2x2 MIMO application. The highlighted resource elements are the pilot carriers. The corresponding pilot elements of two antennas when combined together constitute the pilot matrices. These matrices are then used in the follow up user data processing blocks (details covered in Section 3.4).

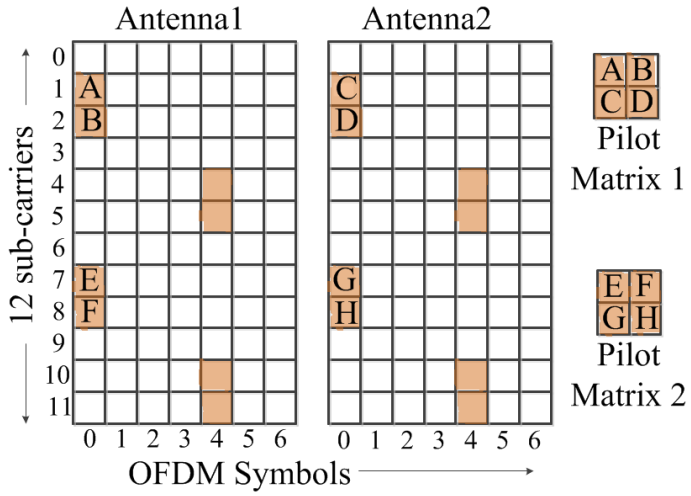


Figure 3.2: Pilot distribution for LTE2x2

3.3 Applications Overview

3.3.1 LTE 2x2 Application

In this section, we give an overview of the LTE 2x2 application, a MIMO, with two antennas at the transmitter as well as the receiver end. Pilots are positioned at Symbols 0 and 4 in each slot (Figure 3.2). Considering Symbol 0, sub-carriers at positions 1 and 2 (A/B and C/D) in the frequency domain of PRBs corresponding to both the antennas constitute one pilot matrix, while those at positions 7 and 8 (E/F and G/H) constitute the second matrix. Similarly, the pilot matrix elements for Symbol 4 are positioned at 4, 5 and 10, 11 in the PRBs for the two antennas. Thus, we have two pilot matrices associated with each of the pilot symbols 0 and 4.

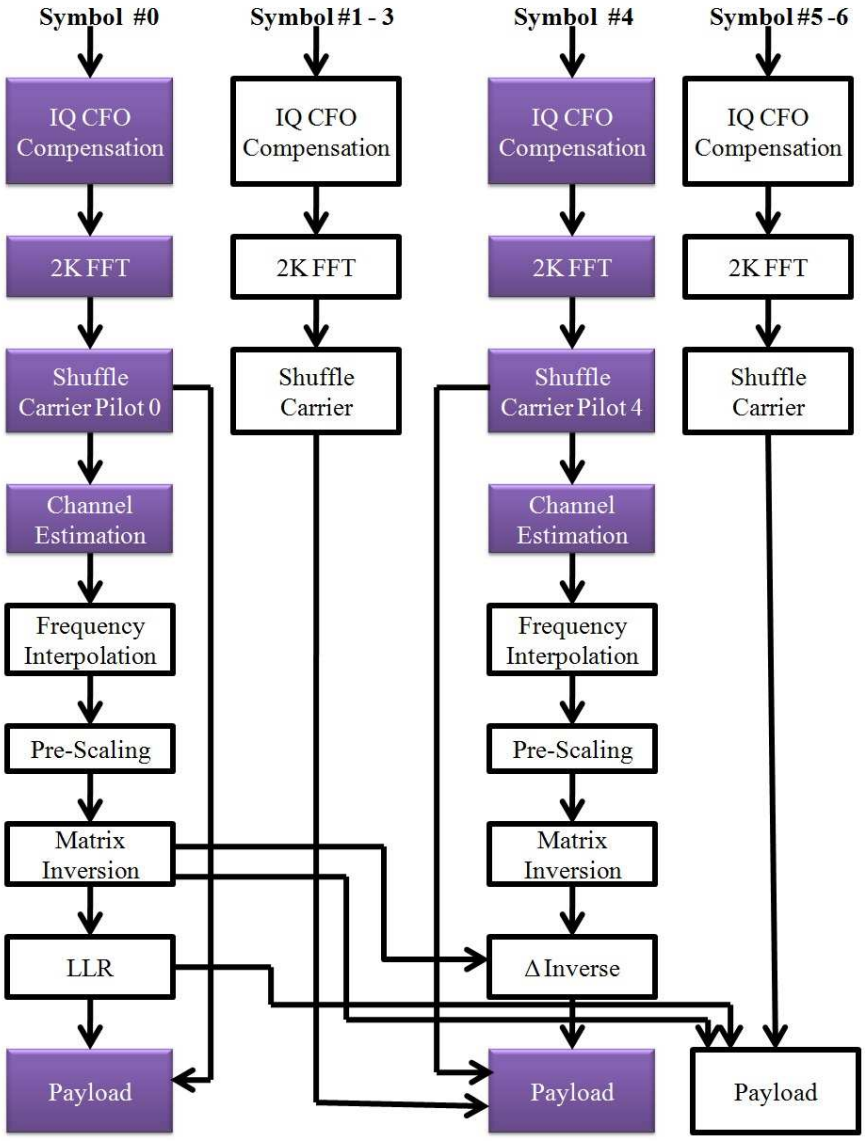


Figure 3.3: High level overview of the LTE 2x2 application

Figure 3.3 illustrates the complete application flow. The OFDM demodulation step consists of the *IQ CFO Compensation* kernel followed by *FFT* to move from time domain to the frequency domain. OFDM demodulation step is executed for all the symbols. The output of *FFT* goes to the *Shuffle Carrier Pilot0/4* kernel for the pilot symbols 0/4 and to *Shuffle Carrier* for non-pilot symbols. These kernels separate out the data and pilot carriers from the dummy sub-carriers. This operation is commonly called the *de-framing* operation. The kernel *Shuffle Carrier* executed for non-pilot symbols just separates out the data signals from the dummy sub-carriers both at the start and end of the band (424 on each side) and the DC carrier. Kernels *Shuffle Carrier Pilot0* and *Shuffle Carrier Pilot4* extract the pilot and data signals for Symbols 0 and 4 respectively.

The *Channel Estimation* kernel computes the channel estimate matrices and the rotational angles based on the pilot symbols. It does the preparation for interpolation along the frequency domain in the *Frequency Interpolation* block. Here, the channel estimates (H matrices) for reference carriers are linearly interpolated along the frequency domain to obtain H matrices for all the sub-carriers corresponding to each PRB. The *Pre-Scaling* kernel does a linear scaling of matrices by a computed pre-coding constant matrix. Following this, the QR decomposition method is used for matrix inversion to compute the equalizing matrix in the *Matrix Inversion* kernel. From the point of pilot extraction to the matrix inversion, the flow is the same for both pilot symbols.

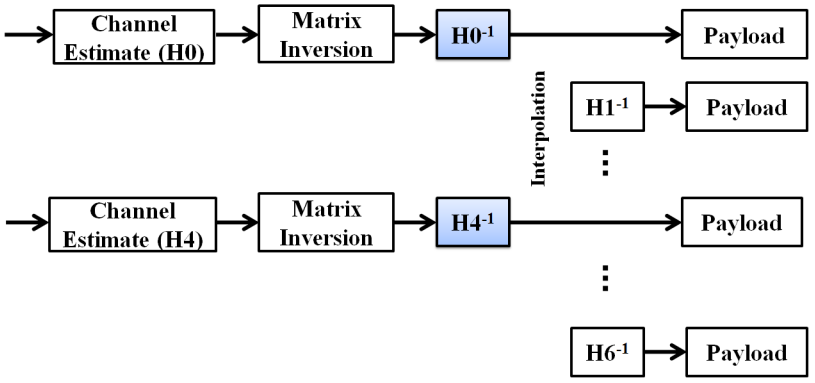
After this, the *LLR* block is executed where Log-Likelihood Ratios (LLR) are calculated based on the inverse channel matrices of Symbol 0. The Δ *Inverse* computation block for Symbol 4 uses the inverse matrices of both pilot symbols to prepare for interpolation along the time domain. Channel estimates are obtained for the intermediate non-pilot symbols in kernel *Payload*. These are then forwarded to *Payload* together with the received data signals for retrieval of the transmitted information bits. The *Payload* processing for Symbols 1-4 can be done only after the *Channel Estimation* for Symbol 4. This leads to a large storage requirement for data buffering. The *Payload* processing for Symbols 0, 5, and 6 can be done without any extra storage overhead. As the number of samples for FFT computation is very large and significant data buffering is required in the local memory in payload processing for non-pilot symbols, memory becomes an important unit of the CGRA processor for optimization. This motivates us to explore an energy efficient data layout transformation.

3.3.2 LTE 4x4 Application

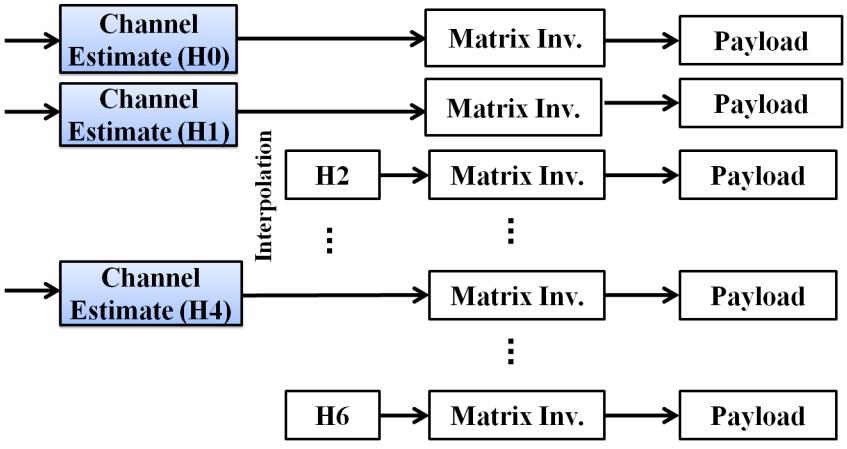
At a high level both the LTE MIMO applications are similar as they are intended for receiver implementation, but there are slight algorithmic differences. The differences between the two are discussed below.

Algorithmic Differences among the LTE applications

In LTE2x2, channel estimates or channel matrices are generated for pilot symbols. These channel matrices are interpolated along the frequency domain to obtain the H matrix for the 12 sub-carriers in a PRB. This is followed by evaluating H^{-1} or equalizing for the matrices corresponding to two symbols. The inverse matrices are then interpolated along the time domain to obtain H^{-1} for all the symbols. For LTE4x4, after obtaining the channel matrices (H matrices) for pilot symbols, we first interpolate the H matrix along the frequency domain to obtain H matrices for the 12 sub-carriers corresponding to a pilot symbol in a resource block. These matrices are then linearly interpolated along the time domain to obtain channel matrices for all the symbols in a PRB. Then, the inverse matrix or equalizing matrices are derived for each of the sub-carriers.



(a) LTE2x2



(b) LTE4x4

Figure 3.4: Algorithmic difference between the two LTE MIMO applications

Thus, LTE 4x4 is more computationally intensive than the 2x2 counterpart where the inverse matrices obtained corresponding to two pilot symbols are interpolated, thereby avoiding a large number of computations. The algorithm for LTE 2x2 gives higher degradation in output for LTE 4x4. Figure 3.4 gives an overview of the differences in the two LTE chains.

Computation Outline for LTE 4x4

Figure 3.5 shows the pilot distribution for the LTE4x4 application. The pilots are positioned at symbols 0, 1, 4, 7, 8 and 11 in accordance with the 3GPP LTE standards. For the LTE 4x4 case, pilot matrix size is 2x4 (= 8 elements per matrix). Samples/data associated with sub-carriers within a PRB at positions 2, 5, 8 and 11 are extracted to constitute the pilot matrices – two for each of the pilot symbols. Processing in the further kernels is carried out using these pilot matrices.

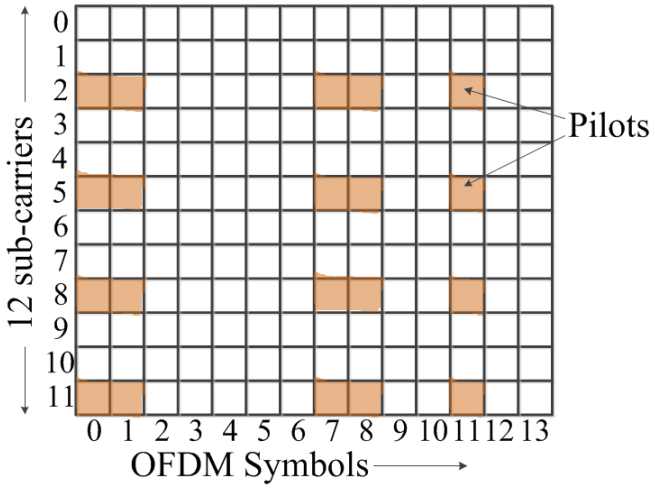


Figure 3.5: Pilot distribution for LTE4x4

Figure 3.6 illustrates the kernel calls in the LTE 4x4 downlink receiver mapping. Symbols 0-6 are used for pre-processing. In the OFDM demodulation block, we have *cpstIQCFO* for IQ imbalance and clock frequency offset compensation. This is followed by *FFT* for moving from time to frequency domain. The *shufflecarrier* kernels extract data and pilot carriers for the corresponding symbols. The only difference in the *shufflecarrierpilot* kernels is the different position of the pilot sub-carriers while defining the pilot matrix. Then, channel matrix (H) is estimated for each of the pilot symbols and interpolated along the frequency domain for a symbol. This is followed by time interpolation to obtain channel matrices for non-pilot symbols. Now, inverse matrix calculations (finding the equalizing matrix) needs to be done for each 'H' matrix, which makes it different from LTE 2x2 implementation, making it more computationally intensive. Finally, payload processing is done, which requires the data carriers (output of

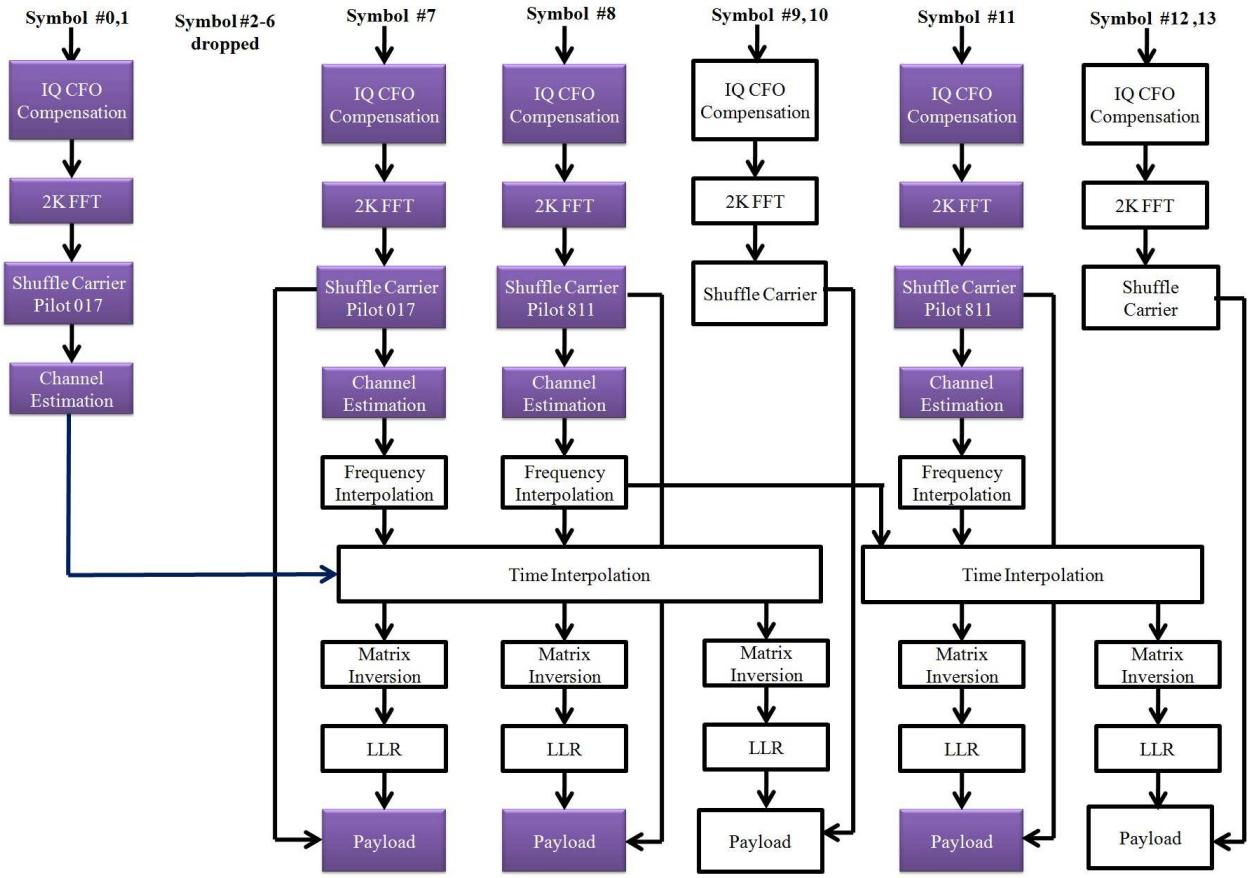


Figure 3.6: High level overview of the data flow in LTE4x4 application

shuffle kernel), the equalizing matrix (from *InvMtxQR* kernel), log likelihood ratio values (from *compLLR* kernel) and the equalizing matrices obtained as a result of time interpolation for non-pilot symbols to extract the transmitted carriers. The kernel processing details are similar to the implementation for LTE2x2.

3.4 Exploration for Interleaving Decision

3.4.1 Data Dependence Analysis of LTE2x2

Exploration for an efficient data layout requires studying the data dependencies across and within the kernels. Following this study, decisions can be made on how the data should be read out for a kernel and written back in an efficient way for the next kernel. Kernels *IQ CFO Compensation* and *FFT* form the time domain processing block for user data. In *IQ CFO Compensation* kernel, the outputs are simply a linear function of inputs. It has uniform dependencies throughout the sample space. The *FFT* kernel functionality can be divided into multiple stages

with the output of each stage being the input to the next. The final output of *FFT* is expected to be in sequential order for further processing of the data. In the implementation considered for analysis, the Decimation-in-Frequency (DIF) algorithm is used. Bit-reversal has been accounted for in the intermediate stages. *FFT* involves non-uniform but regular dependencies and operates on the entire sample space. The output of the *FFT* undergoes the de-framing operation in the LTE shuffle kernel. From here, the user data processing block begins, for which the dependencies are shown in Figure 3.7.

In kernel *Shuffle Carrier Pilot0/4*, the pilots and data signals are separated out. The dependencies in this kernel are also linear but non-uniform because of the conditionals to remove the DC carrier. The *Shuffle Carrier Pilot0/4* kernels differ in the position of reference elements (Figures 3.2) associated with the pilot matrix in the PRB but have similar dependencies. The pilot matrix extraction is followed by the *Channel Estimation* kernel where channel estimates for pilots (hp_1 and hp_2) are computed using the same indexed elements of both the pilot and conjugate pilot matrices. Also, angle shift values between the corresponding elements of two channel estimate matrices are evaluated. These channel estimates are linearly interpolated along the frequency domain using the angle shift values. The difference between the two is divided by the number of carriers in between to obtain the slope factor in the datapath unit DP2. This slope factor, multiplied by a scaling constant K_i, is added to the angle shifted channel matrix of Pilot 1 (*hp1*) to obtain the channel estimates (*hpi*) for all the 12 carriers in a PRB. In the *Pre-Scaling* kernel, the channel matrix elements (*hpi*) derived above are multiplied by the corresponding elements from the pre-coding constants matrix. Each output matrix from the *Pre-Scaling* kernel undergoes inversion by QR decomposition method in the *Matrix Inversion* kernel. For Symbol 0, the *Hinv0* matrices are forwarded for *LLR* computation. The element pairs (00, 01) and (10, 11) are used to compute the two elements of LLR vector for each interpolated carrier. In the DP4 unit, elements of each pair are squared, added, and inverted to compute one element of LLR vector.

For interpolation along the time domain, the Δ *Inverse* kernel uses the *Hinv* matrices computed for Symbols 0 and 4 of the same PRB. The *Hinv* matrices are multiplied element by element to compute the rotational angles. It is used in the DP5 unit, comprising vector multiplication and subtraction, with both the *Hinv* matrices to obtain ΔH^{-1} matrices. These ΔH^{-1} matrices, along with *Hinv* matrix for Symbol 0 and rotational angle matrix (*rot*), are passed to the *Payload* kernel, where the channel estimates for non-pilot symbols are derived by linear interpolation along the time domain. These estimates are then compared against the input data carriers of both the antennas in the DP_6 unit and the corresponding LLR vectors (*LLR_vec*), a measure of transmission accuracy, are derived. In the kernels starting from *Channel Estimation*, all the dependencies are local to a matrix. The difference is only in the matrix access patterns for different kernels corresponding to a symbol.

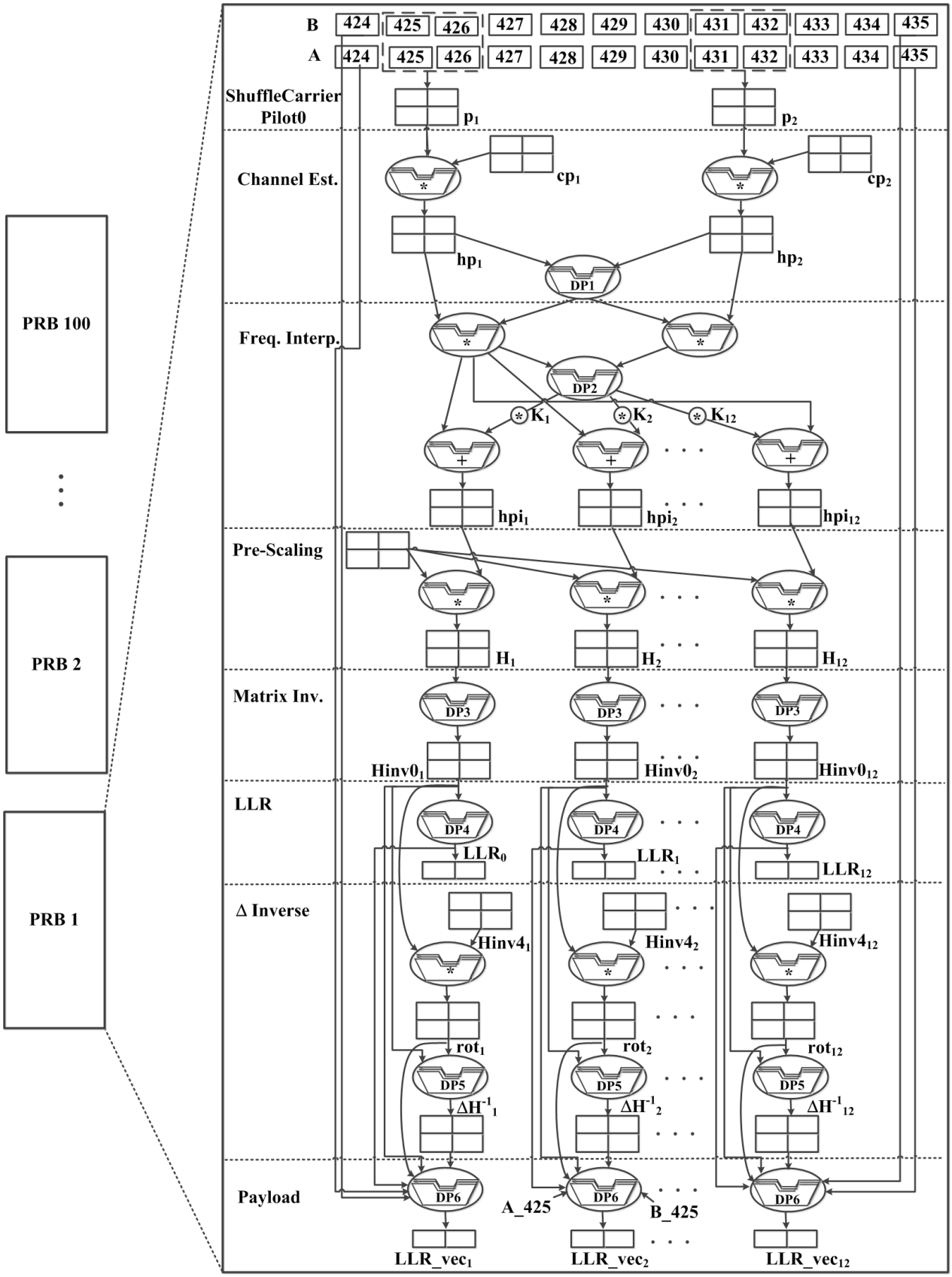


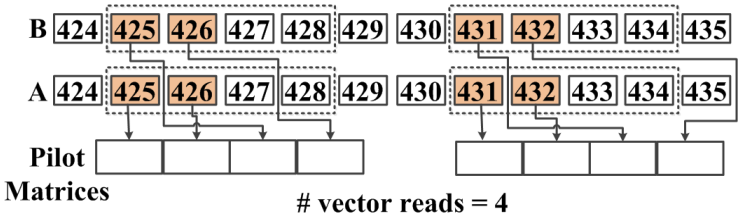
Figure 3.7: Data dependence analysis for Data Processing Block

Conclusions from Dependence Analysis

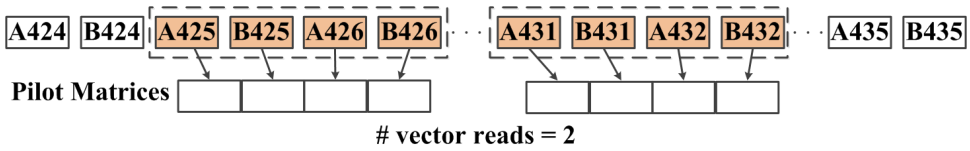
From the data dependence analysis, we observe that operations in the user data processing block (excluding *Payload* processing) are dependent only on the pilot elements. Another important observation is that in the user data processing stage, there are no dependencies crossing the PRB boundaries. Thus, the PRBs can be processed independently. This gives an opportunity to merge stages from *Channel Estimation* to *Matrix Inversion* and reduce the intermediate memory accesses.

3.4.2 Array Interleaving

Interleaving is a data layout transformation for combining the storage of multiple arrays, so that blocks of data from different arrays are stored contiguously, in order to achieve spatial locality in memory accesses. For the LTE application under consideration, the sample space of each antenna is represented by an array of size *2048*, the number of samples per antenna. By interleaving, the functionality of the *Shuffle Carrier Pilot* kernel to group the pilot elements from the different arrays can be achieved. Reading out elements for the pilot matrices requires loading of pilot elements from each of the arrays separately (Figure 3.8(a)). From the application



(a) Normal storage

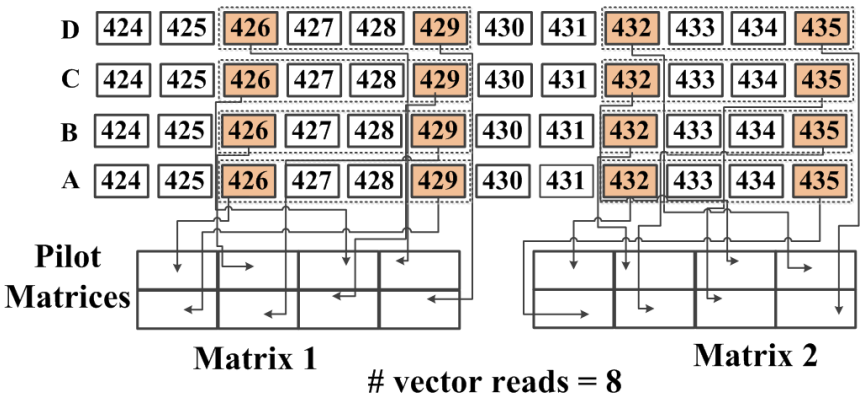


(b) Interleaved storage

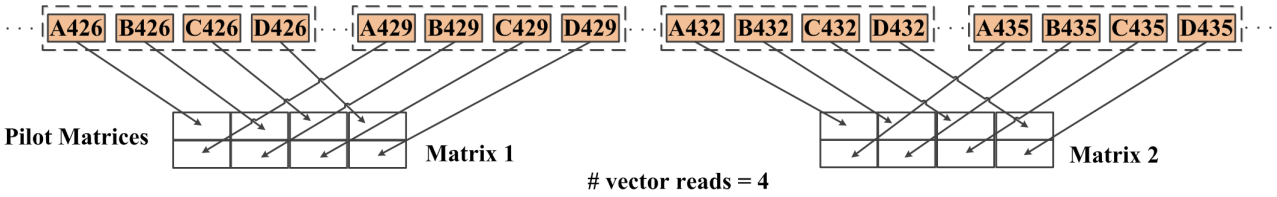
Figure 3.8: Pilot extraction for LTE MIMO 2x2 application (SIMD width = 4)

knowledge, the position of elements to be fetched is the same for all the antennas. By interleaving we are able to group the data to be extracted and thus reduce the number of memory accesses for loading the same pilot matrices as observed in Figure 3.8(b). With larger number of antennas, and correspondingly larger number of arrays, as shown in Figure 3.9(a), the spatial locality of extracted data set increases due to interleaving(Figure 3.9(b)) and the amount

of redundant data fetched reduces. The interleaving transformation, when combined with loop merging (Section 3.4.1), helps obtain additional energy savings (Figure 3.18).



(a) Normal storage



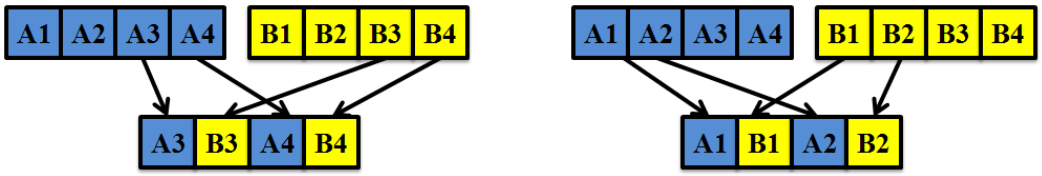
(b) Interleaved storage

Figure 3.9: Pilot extraction for LTE MIMO 4x4 application (SIMD width = 4)

3.4.3 Impact of Interleaving

Propagating the dependencies backwards from the point of extraction of pilot elements (*Shuffle Carrier Pilot* kernel), where we need an interleaved data set, the output of FFT is required to be in interleaved form. This poses a requirement of interleaved data samples at the input of FFT, that can be made possible by writing out the output of *IQ CFO Compensation* kernel in this form.

For this data re-arrangement we need interleave operations. If an interleaving instruction is capable of interleaving R registers, we need R interleave operations for each set of R vector loads. Figure 3.10 illustrates the interleaving operation on a pair of vector loads (i.e. $R = 2$). Here we need two interleave operations for each pair of vector loads – one each for least significant (LS) words and most significant (MS) words. Both these operations are single-cycled in the processor we consider.



(a) For LS words (b) For MS words

Figure 3.10: Interleave operation

The complete sample space to be re-arranged is given by:

$$\#elements = N \times ArraySize \tag{3.2}$$

where N is the number of arrays to be interleaved and $ArraySize$ is the number of elements associated with each of the arrays (=FFT size). With SIMD width W, the number of vector loads for shuffling the entire data becomes

$$\#Vector\ loads = \frac{\#elements}{W} \tag{3.3}$$

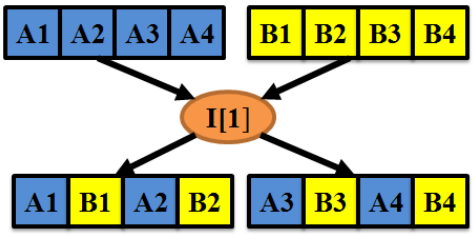
Since interleaving at each stage is performed between a set of R arrays, interleaving N arrays requires $k = log_R N$ stages. The interleaving instruction should be chosen such that N is a power of R. The total number of operations required for interleaving N arrays using R registers at a time becomes:

$$\#Extra\ Ops = R \times \frac{\#Vector\ loads}{R} \times log_R N \tag{3.4}$$

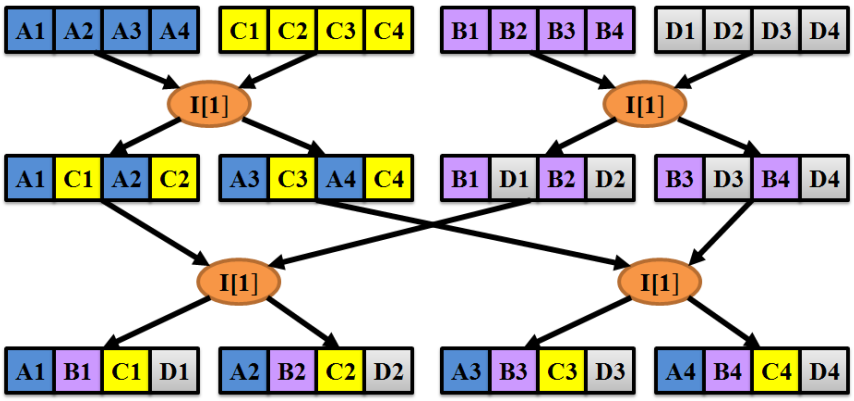
Figure 3.11(a) illustrates the interleaving of 2 arrays ($N = 2$) which is the consequence of interleave operations on LSBs and MSBs shown in Figure 3.10. This is the representation we use to illustrate interleaving of data in 2 vector registers. The $I[1]$ symbol refers to a granularity of 1 used in this interleaving. Figure 3.11(b) illustrates the case when $N = 4$ and $R = 2$. The number of stages (k) required for interleaving is 2. In Stage 1, arrays A, C and B, D are interleaved. With these interleaved pairs of arrays as input to Stage 2, we obtain interleaved arrays A, B, C, and D. At the output of *IQ CFO compensation* kernel, two interleave operations are required for each pair of vector loads, and the number of additional operations for interleaving becomes:

$$\#Extra\ Ops,\ A = 2 \times \frac{\#Vector\ loads}{2} \times log_2 N \tag{3.5}$$

After *IQ CFO compensation*, *FFT* is to be performed on each array. With interleaved sam-



(a) Interleaving 2 arrays ($N = 2$)



(b) Interleaving 4 arrays ($N = 4$)

Figure 3.11: Array Interleaving examples

ples at the input, now instead of performing FFTs for each antenna data set (*Array*) separately, we compute one FFT with 11 stages (for $2K$ size) on $2K \times N$ samples. As a result we obtain interleaved FFT outputs. Performing FFT on all arrays together reduces the number of coefficient (twiddle factor) accesses. With the re-arranged data set, the samples to be operated upon with the same twiddle factors are accessed together, thus preventing the repeated coefficient loads by a factor equal to the number of antennas, N.

Following *FFT* we have the *Shuffle Carrier/Shuffle Carrier Pilot* kernel which, after interleaving, has been removed, as its functionality of grouping the pilot elements is already realized by interleaving. Thus, we save on the memory accesses required to fetch all samples, vector pack operations (involving extraction and insertion of pilot elements), and finally writing back to memory the pilot and data sub-carriers.

Starting from the *Channel Estimation* kernel, the next few kernels upto *Payload* processing are relevant only for pilot symbols. In *Channel Estimation*, the variation in number of memory accesses, if any, is taken into account by the array access computations. Proceeding from *Channel Estimation*, all kernels until *Payload* have no impact due to interleaving.

In the *Payload* kernel, we need to de-interleave the data sub-carriers before processing. De-

interleaving is also carried out by interleaving in multiple stages. In any stage k, the elements of an array separated by W/R^k are brought together. Thus, to obtain contiguous arrangement interleaving is done in $log_R W$ stages, leading to additional operations given by:

$$\#Extra\ Ops = R \times \frac{\#Vector\ loads}{R} \times log_R W \tag{3.6}$$

For *Payload* kernel, the number of these de-interleaving operations (when $R = 2$) is given by:

$$\#Extra\ Ops,\ B = 2 \times \frac{\#Vector\ loads}{2} \times log_2 W \tag{3.7}$$

Figure 3.12 shows the de-interleave operation for two arrays using vector registers with SIMD width, $W = 4$. Interleaving in the first stage brings the elements of the arrays A and B separated by 2 ($=\frac{W}{2}$) . In the second stage, elements of the arrays separated by 1 ($=\frac{W}{4}$) get grouped together. This is how de-interleaving is effectively achieved by interleaving in multiple stages.

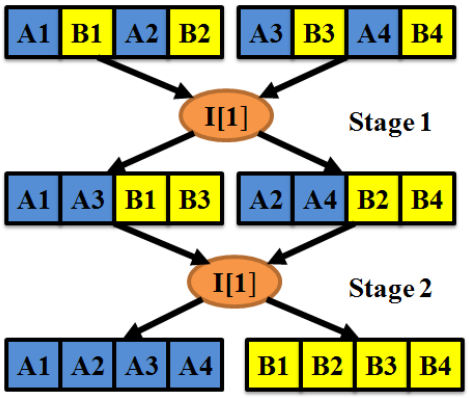


Figure 3.12: De-interleave operation

3.4.4 Global Trade-off Analysis and Overall Strategy

Interleaving may not always result in improved performance or energy. The layout decision needs to be made taking into account the overall objectives and constraints. For the wireless communication standard under consideration, our focus is to reduce energy consumption without performance degradation. We propose a data mapping strategy, keeping in view the goal of energy optimization for wireless devices without compromising on device performance and latency constraints. Implementing interleaving requires additional operations that may lead to overall performance degradation. Thus, there is a need for a quantitative estimate of its impact. The subset of arrays to be interleaved, and the granularity of interleaving, have to be chosen after a proper estimation of the expected savings and associated overheads. The kernels (highlighted in Figure 3.3) are re-written for studying the effect of varying different parameters.

3.4.5 Memory Access Estimates

We first compute the total memory accesses for the kernel mappings for both the cases—without and with interleaving—as follows.

$$tot\_mem\_acc = \sum_{i=1}^{K} \sum_{j \in arr_i} arr\_acc(l_i, j) \tag{3.8}$$

$$tot\_mem\_acc\_int = \sum_{i=1}^{K'} \sum_{j' \in arr'_i} arr\_acc(l'_i, j') \tag{3.9}$$

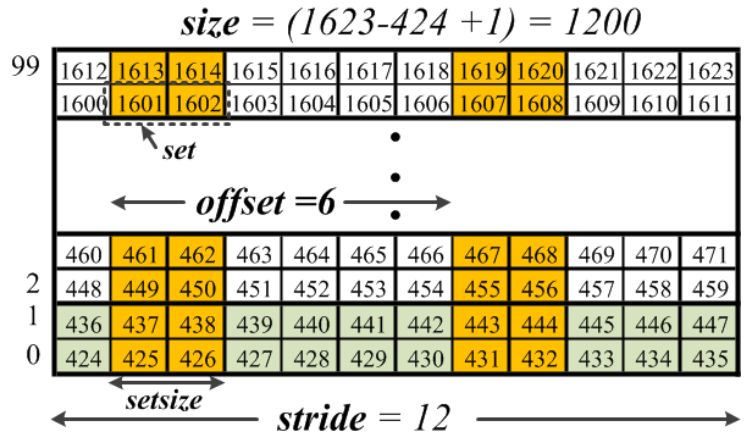
where, tot_mem_acc and $tot\_mem\_acc\_int$ represent the total estimated memory accesses associated in the application mapping without and with interleaving optimization respectively; $l_1 ... l_K$ denote the original set of K loop kernels; $l'_1 ... l'_{K'}$ denote the transformed set of kernels; arr_i denotes the set of arrays accessed in kernel l_i; and $arr\_acc(l_i, j)$ denotes the estimated number of memory accesses to array j in kernel l_i. Note that, due to interleaving, the set of arrays (arr'_i) in the transformed kernels l'_i may be different from the respective original sets (arr_i).

The memory access count within each kernel, is computed as the sum of the accesses for each array of the kernel, from the following parameters: *sets* of *setsize* consecutive elements at intervals of *offset* accessed in each loop iteration, the loop stride (*stride*), SIMD width (W) and the array size (*size*). The estimation presented here assumes that the array accesses in the loop kernel can be organized in this way, and an appropriate set of (*setsize, offset*) parameters can be extracted from the loop. While this is not true in the most general case, the LTE receiver application does satisfy this template.

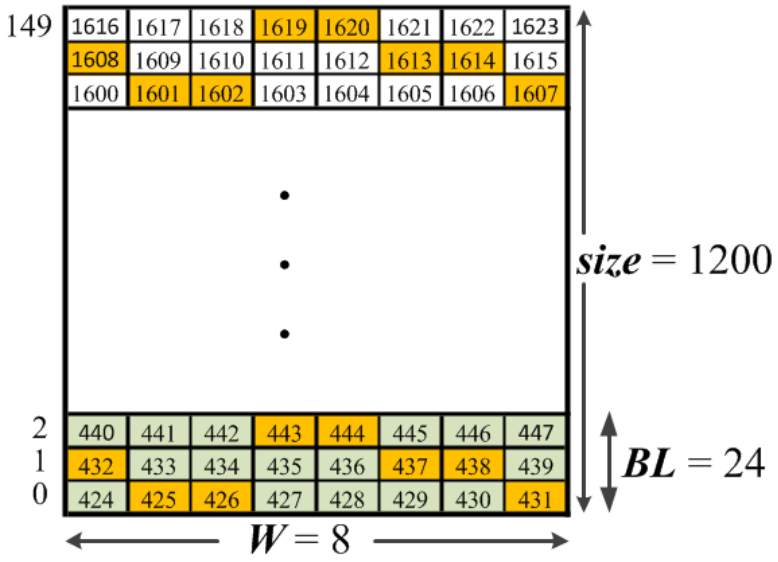
Figure 3.13(a) illustrates the accesses for an array (A or B) corresponding to the loop shown in Figure 1.4(a) with an iteration count of 100 and a *stride* of 12. We have 100 rows (=loop iterations) and 12 columns (=loop stride). Figure 3.13(b) shows the accesses made with vector registers of size W for the same array. This re-arranges the elements in Figure 3.13(a) to reflect the physical parameter W. Now, we have 150 rows (= iterations after re-arrangement) and 8 columns (= W). Comparing the logical and physical access patterns, we identify *blocks* with repeating access patterns, of length BL given by:

$$BL = L.C.M.(stride,\ W) \tag{3.10}$$

In the example of Figure 3.13, we have $BL = L.C.M.(12, 8) = 24$. That is, 2 rows with 12 elements each of Figure 3.13(a) are mapped as 3 rows with 8 elements each. This blocking pattern is repeated over the entire array space. An array of $size (= UB - LB + 1$; where UB and LB denote the upper and lower loop bounds respectively) elements, is sub-divided into $\left\lceil \frac{size}{BL} \right\rceil$ blocks.



(a) Logical access pattern (1 row = 1 loop iteration)



(b) Physical access pattern (1 row = 1 memory access)

Figure 3.13: Access patterns for loop in Figure 1.4(a)

Memory accesses per block can be computed using the parameters defining the array access pattern. We have, in general, multiple *sets* of consecutive elements, with length given by *setsize*, positioned at intervals of *offset*. The number of *sets* per block is given by $= BL/offset$. We can have two cases.

- $offset \geq W$. In this case only one set can be loaded per set of memory accesses of W width. Number of accesses per set is given by $\left\lceil \frac{setsize}{W} \right\rceil$ and the memory accesses per block

$$acc\_per\_blk = \frac{BL}{offset} \times \left\lceil \frac{setsize}{W} \right\rceil \tag{3.11}$$

- *offset* $< W$. Here multiple sets may be accessed per vector load. Assuming f sets are accessed in l vector loads, the desired values of (l, f) will be the minimum of the positive integral solutions for Equation 3.12 with l bounded by the maximum number of accesses per block (= BL/W)

$$setsize + (f-1)offset = l \times W \tag{3.12}$$

  With f *sets* accessed in l vector loads, the number of accesses corresponding to sets in a block becomes

$$acc\_per\_blk = \left\lceil \frac{BL}{offset} \times \frac{l}{f} \right\rceil \tag{3.13}$$

  If there does not exist an integral solution to Equation 3.12 then the memory access count per block is bounded by BL/W, which leads to loading all the elements of a block.

In both cases, the total number of memory accesses for an array is given by

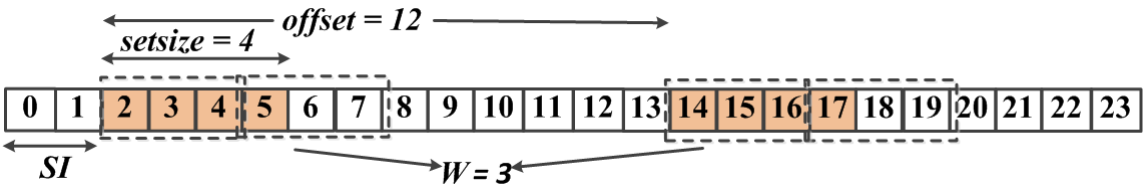
$$acc\_arr = \left\lceil \frac{size}{BL} \right\rceil \times acc\_per\_blk \tag{3.14}$$

Equations 3.8 and 3.9 are now used to estimate the memory accesses for all arrays in all kernels.

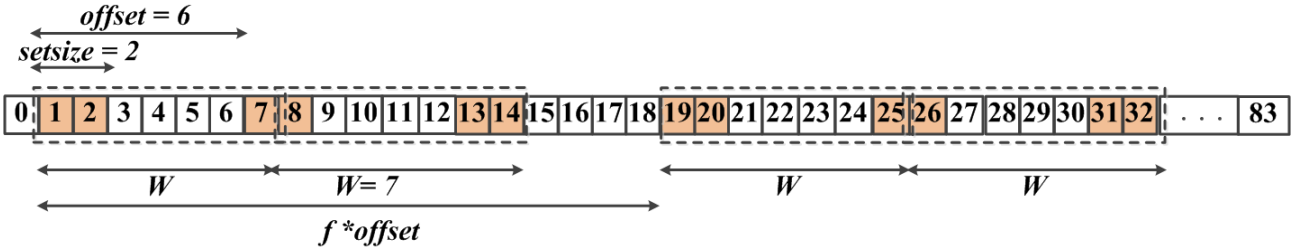
3.4.6 Examples for Memory Access Computations

Consider an example with *offset* = 12, W = 3, *setsize* = 4, *stride* = 24, *size* = 2400 as shown in Figure 3.14(a). There are 100 blocks with block length (BL = 24), and 2 *sets* per block. Here, *offset* $\geq W$ and number of accesses per *set* is 2. Thus, using Equation 3.11 we evaluate the number of memory accesses per block of length BL to be 4. With 100 such blocks covering the entire array space, the total number of memory accesses is 400 (= 100$\times$ 4 using equation 3.14).

Figure 3.14(b) illustrates an example where *offset* $< W$ with *offset* = 6, W = 7. The other parameters are as follows: *setsize* = 2, *stride* = 12, *size* = 1200. There are 15 blocks with block length (BL=84), and 14 *sets* per block. Using Equation 3.13 the number of memory accesses per block evaluates to 10 with k and f being 2 and 3 respectively. With 15 blocks covering the entire array space, the total number of memory accesses is 150.



(a) *offset* $\geq W$



(b) *offset* $< W$ ($f = 3$)

Figure 3.14: Memory access computation examples

3.4.7 Interleaving Granularity and Candidate Arrays Selection

Interleaving is beneficial when the number of consecutive elements (*setsize*) to be accessed in an array is not a multiple of the vector register width, W. If *setsize* is a multiple of W, then the layout is already optimized for vector loading and will not result in any further gain in terms of memory accesses. *Interleaving granularity (g)*, the number of array elements in each unit to be interleaved, should be an integral factor of both W and the array's initial index, *SI*. Factor of W ensures that loads in a vector register are uniformly distributed from multiple arrays. Factor of *SI* ensures that the initial skipped elements are grouped together and the accessed elements are not scattered across multiple vector words. We select the highest such common factor. Once the granularity g is known we can find the number of arrays, N to be interleaved as $N = \left\lfloor \frac{W}{g} \right\rfloor$ so that the combined sets of g elements accessed from each of the N arrays fits into the vector register width. Candidate arrays for interleaving in a particular loop nest are considered compatible if they possess the following properties:

- **Similar access pattern.** Two arrays are compatible if their index expressions are identical for every distinct reference to the arrays in the innermost loop.
- **Operands of same operation type.** Since we perform vector operations in the processor, the same operation will be applied to every element of the vector.
- **Not operands of the same instruction.** The operands for the vector FU are loaded from

different vector registers. If we interleave the operands of the same instruction, additional operations are necessary for de-interleaving and re-arranging the data before proceeding with the SIMD operation. To reduce these overheads, we prune out such pairs from the interleaving candidates.

Accesses corresponding to an interleaved array can be estimated using the above strategy, with each input parameter (except SIMD width W) multiplied by the number of interleaved arrays, N.

3.4.8 Decision for Interleaving

The decision for interleaving is justified only if the overhead induced by interleaving is less than the cycles saved by the data layout transformation. For each symbol s, we compute the memory accesses saved from the difference in non-optimized and optimized access counts (Equations 3.8 and 3.9), and multiply by the memory access latency (C_{mem_acc}) to obtain the estimated cycles saved.

$$gain = (tot\_mem\_acc - tot\_mem\_acc\_int) \times C_{mem\_acc} \tag{3.15}$$

We then estimate the *Overhead* by computing the cycles consumed for the mappings – without and with data layout optimization – as follows.

$$tot\_cycles = \sum_{i=1}^{K} \sum_{j=1}^{typ} Ops(l_i, j) \times C_j \tag{3.16}$$

$$tot\_cycles\_int = \sum_{i=1}^{K'} \sum_{j=1}^{typ} Ops(l'_i, j) \times C_j + A + B \tag{3.17}$$

where, tot_cycles and tot_cycles_int represent the cycles consumed for the application mapping without and with interleaving optimization respectively; $Ops(l_i, j)$ denotes the number of operations of type j in the original set of K loop kernels $l_1, \cdots l_K$; $1 \cdots typ$ denote the different operation types; $Ops(l'_i, j)$ denote the number of operations of type j in the transformed set of kernels $l'_1, \cdots l'_{K'}$; A and B are the single cycle operation overheads estimated in Section 3.4.3; and C_j denotes the number of cycles required for operation type i. The *Shuffle Carrier/Shuffle Pilot Carrier* kernels do not affect the datapath operation counts. These involve only memory accesses for separating out the data and pilot carriers from the dc carrier, the effects of which have been taken into account by the memory access estimates. The overhead due to the data layout transformation is given by:

$$Overhead = tot\_cycles\_int - tot\_cycles = A + B \tag{3.18}$$

The decision for interleaving is justified if the estimated overhead is less than the estimated cycles saved due to the memory accesses, i.e., $Overhead \leq gain$.

3.5 Experimental Results

3.5.1 Framework

We used an extension of the IMPACT compiler framework [66] for mapping the reference application to the architecture template shown in Figure 2.9. An XML based language is used to describe the architecture. The processor simulator provides run-time statistics that are used to compare the memory accesses and performance for the kernel mappings. We synthesized the HDL models of the processor using Cadence RTL compiler for TSMC 40nm standard library. Power simulations were carried out on the synthesized design using Synopsys Primepower. Delay and energy numbers for memory structures such as register file and SPM were derived from commercial memory compilers.

3.5.2 The Starting Point

The reference implementation we use is an already optimized mapping on the baseband processor under consideration. It takes into account the architectural aspects to have best performance, reduced memory footprint and memory accesses. Some of these optimizations are as follows. Loop transformations such as loop splitting and loop unrolling are considered to give an optimal mapping for the SIMD architecture. From the *Channel Estimation* kernel, processing of two PRBs is done together to have the total carriers being operated upon as a multiple of the SIMD width (i.e., 24 carriers corresponding to 2 PRBs for SIMD width 8). This helps in avoiding repeated access of a subset of carriers while moving from one resource block to another. In FFT, to obtain the in-order output in final stage, bit shuffling is taken into account while moving across the stages instead of additional memory accesses and cycles for re-arranging after the last stage. Also, the coefficient accesses in FFT are reduced by reproducing a part of the coefficients by rotating one set by 90 degrees. For the kernels processed in the frequency domain, the active resource blocks are considered for simulation. The optimized code, thus obtained, is used as the reference implementation.

3.5.3 Exploration Experiments for LTE

Based on our interleaving decision strategy discussed in Section 3.4.4, we conclude that interleaving should be done only for the pilot symbols in a slot. For non-pilot symbols, though

there is a gain with respect to memory access energy, the performance overhead is high. Thus, kernels *IQ CFO Compensation, FFT*, and *Channel Estimation* are modified for pilot symbols. Interleaving implicitly implements one of the functions of the *Shuffle* kernel to group the pilots. The other one for extracting the data carriers can be postponed until *Payload* processing. Thus, the *Shuffle* kernel can be completely removed for all the symbols. The kernels are re-written for studying the effect of varying different parameters. We study the effect of interleaving on the memory access variations due to different application and architectural parameters when the LTE 2x2 application is mapped on to the CGRA processor.

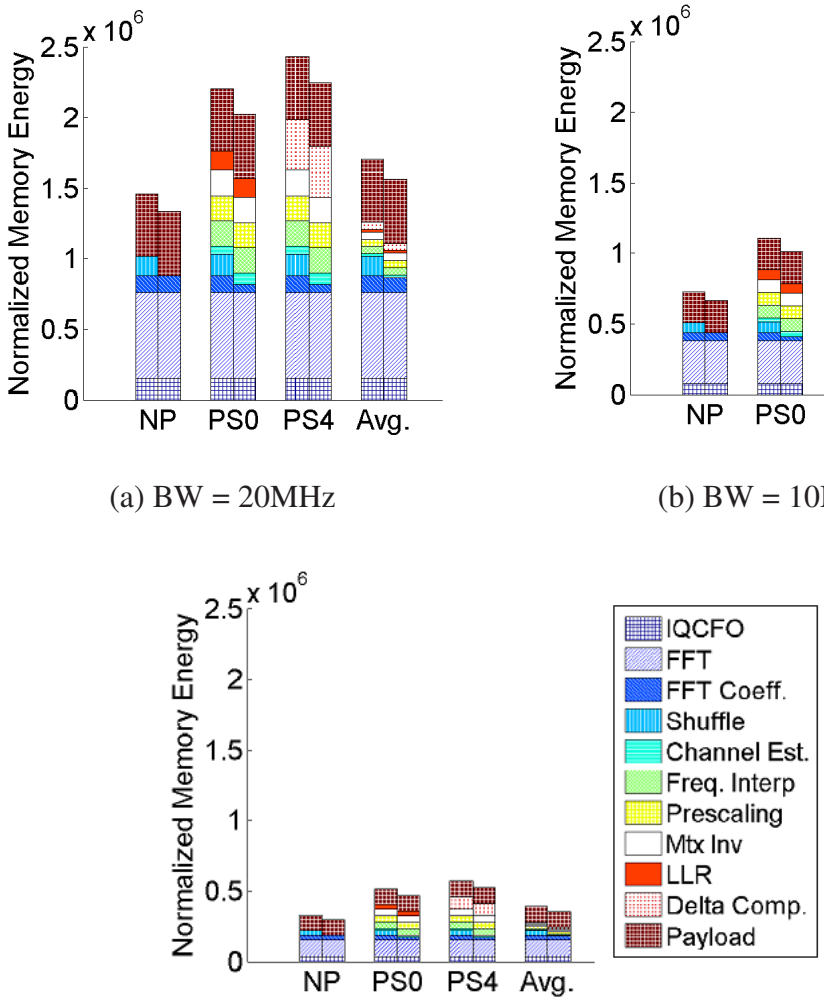
In our experiments the percentage reduction in energy is directly proportional to the corresponding reduction in memory access count. However, interleaving leads to additional vector register accesses, with a corresponding energy increase. The energy values reported here take this overhead into consideration. We observe that in the worst case, when all PRBs are occupied at maximum bandwidth of 20 MHz, the additional register accesses lead to a 1.07% reduction in the improvement due to reduced memory accesses. This additional effect of vector register accesses becomes negligible with reducing number of PRBs because of the reduced interleave operations in *Payload*, with an unchanged gain from *Shuffle* kernel. We compare the energy efficiency of our proposed strategy against the conventional data layout, where the array data is stored in contiguous locations, in the sequence determined by the order of declaration. An explicit comparison against the approaches discussed in Section 3.1 could not be performed since the cache- and GPU-oriented techniques target different architectures and are sensitive to unrelated parameters (such as line size). As mentioned in Section 3.4.7, our interleaving decision strategy ensures that the new mappings do not degrade the system performance. The energy numbers reported here are normalized with respect to the average energy of a 32 bit adder. The reference case for all the exploration results discussed below is the configuration with fully occupied channel of bandwidth 20MHz and mapping on to the architecture with SIMD width of 8. The average energy value used in the discussion here is computed as:

$$Avg = \frac{Energy_{NP} \times \#NP + Energy_{P0} + Energy_{P4}}{\#symbols} \tag{3.19}$$

where $Energy_{NP}$ represents the energy for a non-pilot symbol. $Energy_{P0}$ and $Energy_{P4}$ represent the energy values for Pilot symbols 0 and 4 respectively. For LTE 2x2, the number of non-pilot symbols (#NP) is 5 and the total number of symbols (#symbols) is 7.

- *Variation with channel bandwidth.* LTE defines several channel bandwidths. The higher the bandwidth, the larger the channel capacity. The number of sub-carriers varies with the available bandwidth. Since the memory access count is proportional to the number of sub-carriers for all kernels except FFT, it varies by the same factor. In FFT, the number of loops associated with merged stages changes, thereby affecting intermediate memory

accesses. This variation is not uniform, as the number of loops associated with stages does not scale by a constant factor. Thus, the percentage reduction in memory access count due to interleaving with varying bandwidth does not show a uniform increase in Figure 3.15. It is 8.51% for channel bandwidths of 20 and 10 MHz, and increases to 9.33% for bandwidth of 5 MHz.



(a) BW = 20MHz

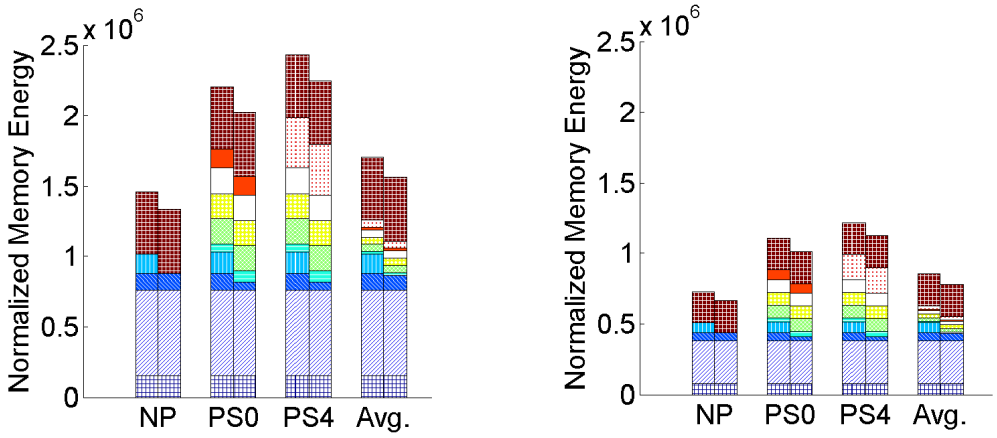
(b) BW = 10MHz

(c) BW = 5MHz

Figure 3.15: Non-optimized vs. Optimized storage: variation with channel bandwidth (SIMD width = 8, 100% PRB occupation). NP: non-pilot symbol, PS0 and PS4: pilot symbols 0 and 4, Avg.: Average over all symbols

- *Effect of variation in SIMD width.* Memory access count is inversely proportional to the SIMD width. With varying SIMD width, the memory access count associated with each kernel changes by the same factor. Thus, the overall reduction remains the same with SIMD width variations as shown in Figure 3.16. For each non-pilot (NP) symbol,

the reduction is 8.68%, while the reductions of 8.37% and 7.72% are achieved for pilot symbols 0 and 4 respectively. Averaging out over the time slot (*Avg.* in Figure 3.16), the reduction in memory accesses is 8.51%.



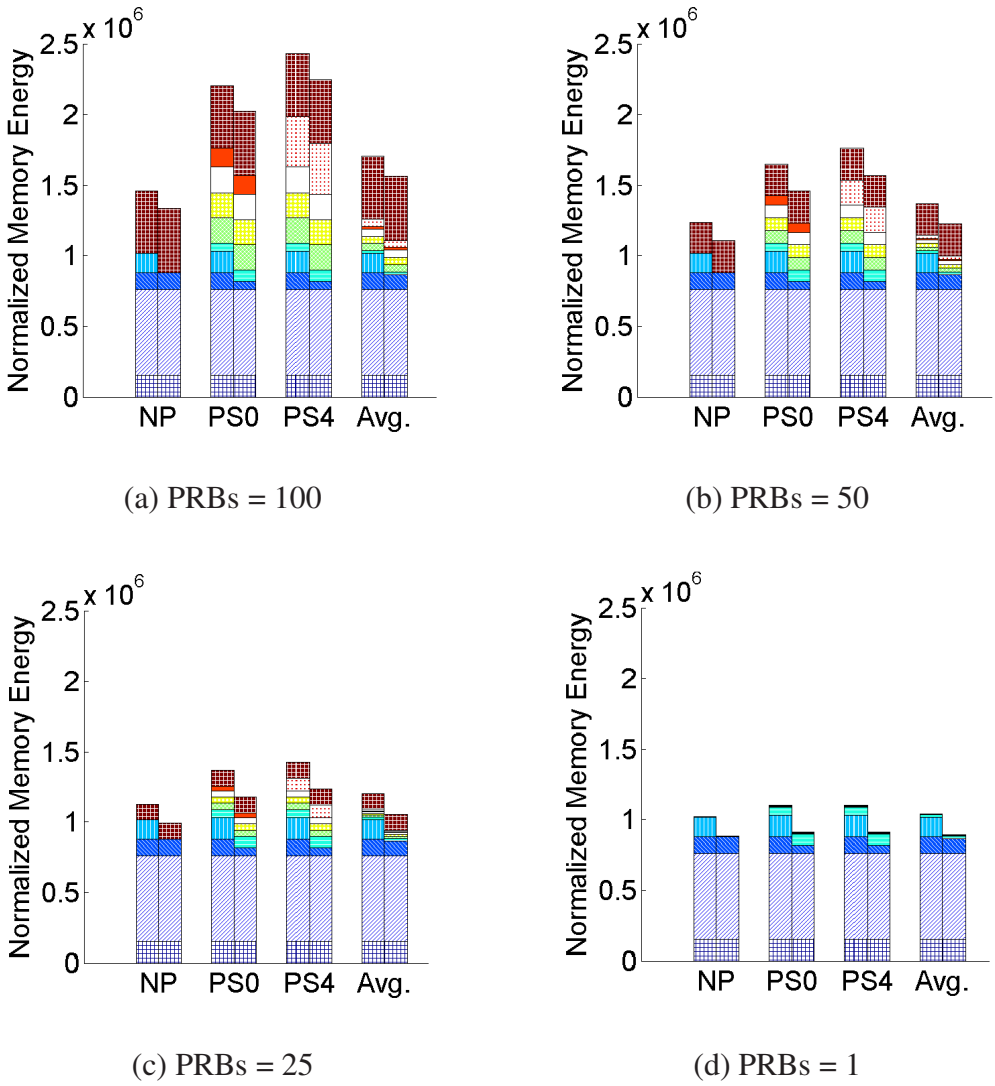
(a) SIMD width = 8

(b) SIMD width = 16

Figure 3.16: Non-optimized vs. Optimized storage: variation with SIMD width (BW = 20MHz, 100% PRB occupation)

- *Effect of variation in number of occupied PRBs.* The maximum number of PRBs associated with 20 MHz channel bandwidth is 100. However, since 100% PRB occupation occurs rarely in practice, we study the effect of interleaving with the number of occupied PRBs as a parameter. A substantial improvement in gain is achieved with reducing number of occupied PRBs (Figure 3.17). Since all the data samples have to be processed until the *Channel Estimation* stage, the memory access count does not vary with PRBs until channel estimation. Proceeding towards the MIMO detection stage, the lower the number of occupied PRBs, the lower the computations and the corresponding memory accesses. Thus, the percentage reduction in memory access count shows an approximately linear increase for all symbols. The average reduction over a time slot varies from 8.57% for 100 PRBs to 14.33% when the number of occupied PRBs is reduced to one.

- The observation that there are no dependencies crossing the PRB boundaries for the user data processing block, provides us an opportunity to merge the kernels beyond *Channel Estimation* until the equalizing matrix generation (*Matrix Inversion*) block. By doing this, we further reduce the redundant memory accesses across the different stages. Figure 3.18 shows the reduction in memory access count with merging (15.21%) and a small increase in average energy reduction percentage from 8.5% to 9% by interleaving, achieved over the mapping without merging.



(a) PRBs = 100

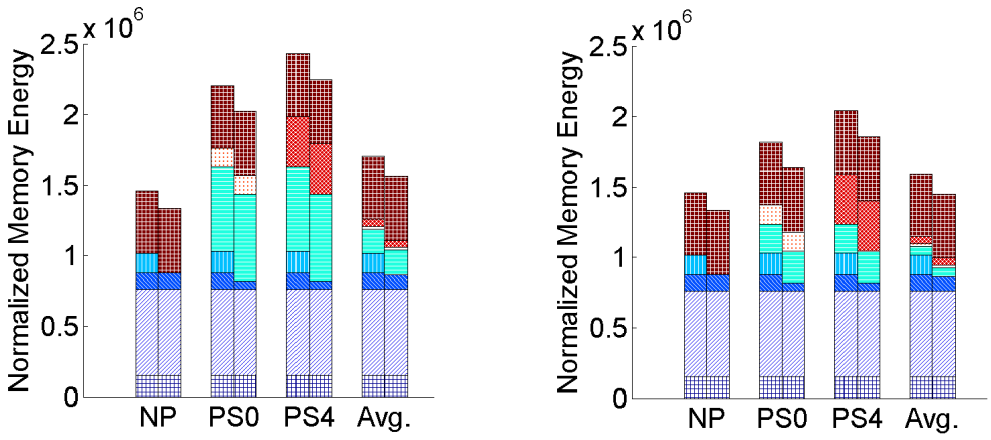
(b) PRBs = 50

(c) PRBs = 25

(d) PRBs = 1

Figure 3.17: Non-optimized vs. Optimized storage: variation with #occupied PRBs (BW=20MHz, SIMD width = 8)

For the processor under consideration, the memory subsystem studied above accounts for roughly 50% of the energy, with the datapath elements accounting for the remaining 50%. Since our transformations have almost no impact on datapath, and the extra energy due to additional register energy in the interleaving/de-interleaving instructions has already been accounted for in the memory energy improvement computation, the improvement in the total system energy can be approximated as half the energy saved due to the proposed storage optimization.



(a) Without Merging

(b) With Merging

Figure 3.18: Non-optimized vs. Optimized storage: variation due to merging kernels (BW=20MHz, 100% PRB occupation, SIMD width= 8)

3.6 Conclusion

We presented an approach to improve memory energy by reducing memory accesses using an efficient data layout targeting a CGRA with SIMD structure. The LTE downlink was used as a demonstrator example, as it is a memory dominant application with SDR because the number of samples for FFT computation is very large, with significant data buffering in the local memory in payload processing for non-pilot symbols. We conclude from a global analysis of kernels that significant reduction (7-15%) in data memory energy consumption can be obtained if array data are interleaved, with no performance overhead. From the data dependence analysis, we also conclude that there are no dependencies crossing the PRB boundaries, giving an opportunity to process each PRB separately in the user data processing block. This allows merging a few kernels in data processing block, which reduces the overall memory access count by around 15%.

In this chapter, our memory optimization analysis focused on different configurations of the LTE 2x2 application. Our overall approach is also suitable for a wide range of embedded applications in the communication, multimedia, and image processing domains. In the next chapter, we discuss the generalization steps for the interleaving scheme to be used as a more broadly applicable compiler transformation and present the mapping results on applications with varying complexity, ranging from single loop kernels to complex applications from different domains.

Chapter 4

Array Interleaving – A Data Layout Transformation

In this chapter we build upon the data layout transformation – *Array Interleaving*, proposed in Chapter 3, to make it a more widely applicable compiler transformation. The transformation reduces the number of memory accesses by loading the right set of data into vector registers, thereby minimizing redundant memory fetches. This data layout transformation, when combined with architectural parameters such as vector register width, results in a new exploration space. We perform a global analysis of array accesses, and account for possibly different array behavior in different loop nests that might ultimately lead to changes in data layout decisions for the same array across program regions. The technique relies on detailed estimates of the savings due to interleaving, and also the cost of performing the actual data layout modifications. We also account for the vector register widths and the effect of interleaving granularity in our exploration. The number of interleaving possibilities grows exponentially with the number of live arrays. To reduce the search space, we incorporate pruning strategies that significantly reduce the number of possibilities.

This chapter is organized as follows. Related research is discussed in Section 4.1. Section 4.2 introduces the problem with illustrative examples. Section 4.3 formulates the problem and introduces the mechanisms for estimating the savings and overheads. Section 4.4 discusses the exploration strategy for making the globally optimal decision on array interleaving. The experimental validation is discussed in Section 4.5. Section 4.6 concludes the chapter with some key observations.

4.1 Related Work

Memory and data management in general, and specifically, data layout has been an important topic of research in embedded systems. Several aggressive techniques have been employed to improve system energy and performance through a combination of data and program transformations and architecture exploration. The strategy in [67] partitions the variables to Scratch Pad Memory and DRAM in a way that interference among the different variables is minimized. The partitioning is guided by data characteristics such as access frequency of variables and lifetimes. Strategies for re-arranging aggregate data structures such as *struct* for improving cache performance also fall in this category [55, 68, 69].

Polyhedral models are also being used to address the problems of in-place data mapping and determining minimum storage requirement. Polyhedral Parallel Code Generator (PPCG) [70] introduces additional memory access related efficiencies by defining code generation schemes that improve data locality at different levels in the memory hierarchy. Algorithms for counting the number of memory locations accessed in a loop using a set of polynomials representing the iteration space have been proposed [71, 72]. An approach similar to Barvinok's decomposition [73] is used for counting using the polyhedral model. These point counting methods are not directly applicable for the SIMD architectures because we have to incorporate the effect of the multiple word accesses in the wider memory format instead of scalar accesses. The number of useful words that are read/written is not constant and that means we have to modify the way we count. We also need to have a very fast method because it is used only for relative cost comparisons in the design exploration space. We present a memory access count estimation approach for SIMD architectures in Section 4.3.2.

The concept of interleaved storage of arrays has been proposed in different embedded system contexts earlier. Interleaved storage has been suggested for array data with the objective of minimizing off-chip memory address bus transitions [74]. Data is stored in row-major order, column-major order, organized into tiles, or interleaved depending on the sequence of addresses generated, so that the resulting spatial locality causes minimal number of bit transitions on the memory address bus. Array interleaving can also help in achieving spatial locality that improves data cache performance because array data is tightly grouped into fewer cache lines [75]. The concept has also been applied to the problem of storing data in DRAM banks. Instead of spreading out data accesses into several memory banks, and thereby incurring the penalty of wasted power due to keeping all the banks active, array interleaving can help restrict the number of active banks, which creates opportunities for turning a few DRAM banks to low power mode [50]. These objectives are very different from the one targeted in our work due to the different architectural assumptions. The interleaving decision is constrained by the architectural parameters such as vector register width and SIMD FUs. The different operands for the

vector FU are loaded from different vector registers. Thus, the operands of the same instruction should not be interleaved so that they are not accessed in the same vector register. Otherwise, separating them out would lead to large overheads. Another constraint is imposed by vector FU implementation, i.e., the same operation is applied to all the vector elements. Thus, the arrays being interleaved should be operands of the same type. Thus, the array interleaving decisions are different when the objective is to reduce data transfers between wide interface scratch pad memories and vector registers. Our work is also the first to perform a global analysis of the effect of interleaving across all the loops by estimating the costs and benefits throughout the application program.

4.2 Illustrative Examples

We present some illustrative examples to motivate the data layout transformation presented in this chapter, and highlight the issues involved. We assume an embedded processor architecture (Figure 1.3) in which we have a wide interface between a register file and SPM, with W elements being simultaneously transferred into a vector register, and a vector FU in which we operate on all W elements in parallel. This processor belongs to the class of re-configurable architectures.

Figure 4.1(a) shows a code snippet from one of the benchmark applications that involves extracting a few elements from data stored in an array, corresponding to the sampling operation. Vectors A and B are read and copied to generate the output vector C. Figure 4.1(b) shows the access pattern in the first iteration, with the accessed array elements highlighted.

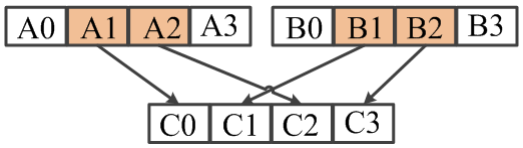
Different data layout styles could be used for storing the arrays in Figure 4.1(a). The default layout is that all array elements are stored in contiguous locations, as shown for arrays A and B in Figure 4.1(c). Since the vector READ operation loads 4 consecutive elements from the data memory into the vector register, this layout results in two vector reads in each loop iteration, one corresponding to each array. The array elements accessed in one vector load are shown enclosed in the dotted rectangles. Note that such loads fetch unnecessary elements (such as $A[0]$ and $A[3]$) into the registers. An alternative layout is shown in Figure 4.1(d), where the storage of the array elements is *interleaved*. The vector READ operation loads $A[1]$, $A[2]$, $B[1]$, and $B[2]$ in the first iteration, all of which are accessed in the loop, eliminating any redundant fetches. For the embedded processor under consideration, the interleaved layout results in a 34% reduction in memory energy, arising out of the reduced number of vector READs. This provides us the motivation to investigate the array interleaving transformation in more detail.

Figure 4.2 shows the code for a more complex example, where array interleaving is still useful, but the interleaving decisions in different loops may be conflicting.

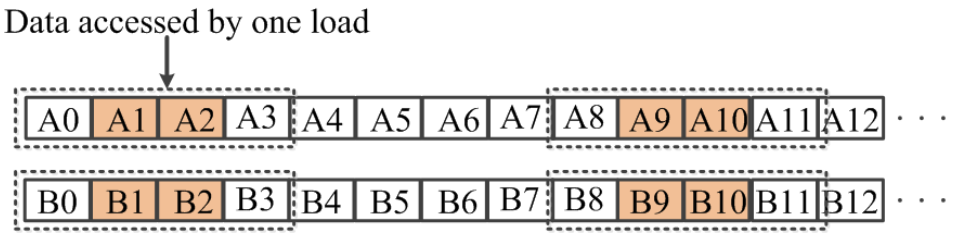
Assuming that arrays E and F are stored in interleaved form, we now discuss the possible

```
double A[480], B[480];
double C[480];
int j=0;
for(i=0; i<480; i+=8) {
  /* Sampling */
   C[j]=A[i+1];
   C[j+1]=B[i+1];
   C[j+2]=A[i+2];
   C[j+3]=B[i+2];
   j=j+4;
}
```

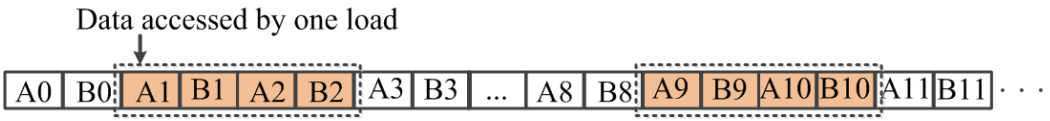
(a) Example loop



(b) Access Pattern in first iteration



(c) Conventional storage



(d) Interleaved storage

Figure 4.1: Motivational Example

data organization for arrays *C* and *D*. In loops L2 and L3, we prefer an interleaved storage for arrays *C* and *D*. As argued earlier, such a storage pattern leads to fewer vector memory accesses. However, for loop L1, we prefer normal storage for *C*. Interleaving *C* and *D* is worse in L1 because the *D* elements would be unnecessarily fetched from memory and actually interfere with the vector operations. One possibility is to interleave *C* and *D* arrays *after loop L1 completes*. This layout change incurs additional cost, and needs to be traded-off against the savings obtained in the later loops. We observe in Figure 4.3 that the decision to perform the

```
int k, max = 1000;
int C[max], D[max], E[max], F[max];
int A[max], B[max], M[max], N[max];
int R1[max], R2[max];
L1: for(i=0; i<1000; i+=1) {
      A[i] = C[i] + B[i];
      M[i] = D[i] * N[i];
    }
L2: for(i=0; i<500; i+=2) {
      R1[k]=C[2i] * E[2i];
      R1[k+1]=D[2i] * F[2i];
      R1[k+2]=C[2i+1] * E[2i+1];
      R1[k+3]=D[2i+1] * F[2i+1];
      k=k+4;
    }
L3: for(i=0; i<1000; i+=4) {
      R2[k]=C[i];
      R2[k+1]=D[i];
      R2[k+2]=C[i+1];
      R2[k+3]=D[i+1];
      k=k+4;
    }
```

Figure 4.2: Example with multiple loops

interleaving of C, D after loop L1 results in one fewer memory READ operation per iteration in both L2 and L3, leading to a total of 500 (= 250 + 250) reduced memory accesses (shown as Δmem_access). The number of memory reads (MR) and writes (MW) per iteration is shown for the non-interleaved (NI) and interleaved (WI) cases. For the processor used in our experiments, this amounts to a reduction of 15% in memory and register file energy after accounting for the energy overhead due to performing the array interleaving.

The above examples highlight the non-trivial nature of the array interleaving optimization, and the need for a global analysis that considers both the savings and possible overheads due to interleaving. Earlier approaches to the interleaving transformation are not applicable for the context described here. Since they target different objectives and none of them is designed to anticipate the class of application behaviors exemplified by Figure 4.2. We consider sequen-

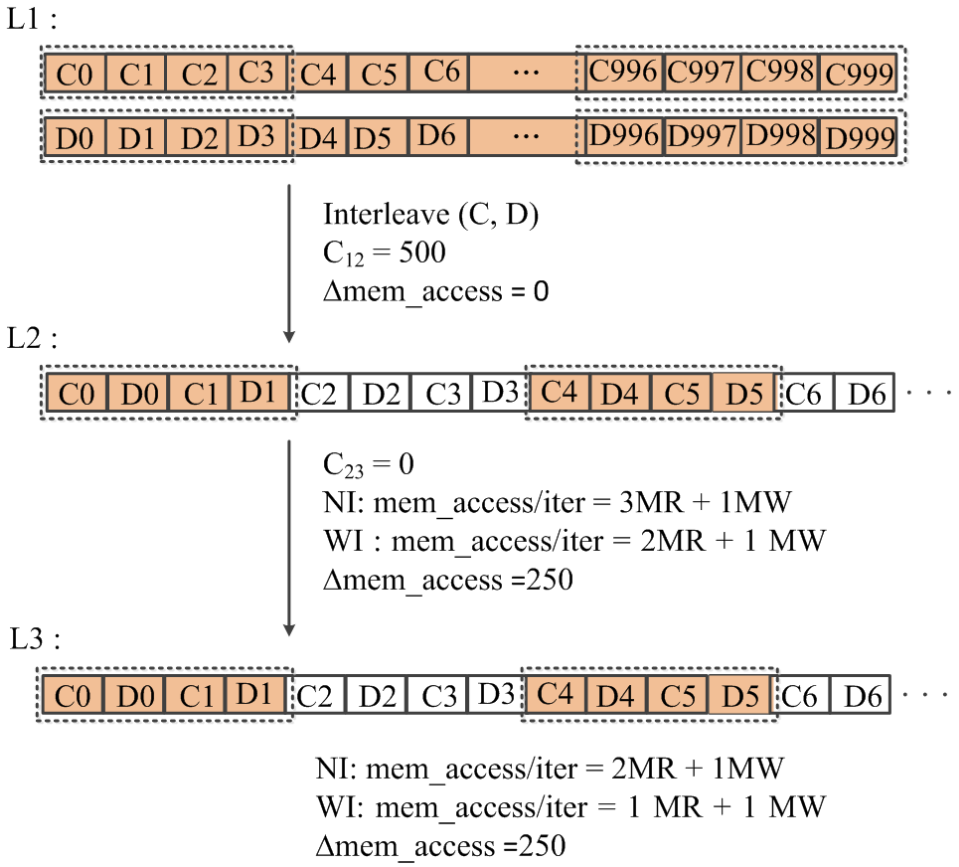


Figure 4.3: Access pattern with interleaved layout for the example loops in Figure 4.2

tially executing loops and restrict our exploration to loops with affine indices. Different loop execution orders are assumed to have been explored before this step begins, for improving access locality; we consider the execution order to be fixed. More complex non-manifest loops with data-dependent conditions are currently not handled in our formulation. As future work, we could combine this with the system scenario approach [76], where each system scenario may lead to possibly different interleaving decisions.

4.3 Problem Formulation and Cost Estimation

In this section, we define the interleaving decision problem that considers both the savings and interleaving overheads, to account for interleaving decisions at various loop levels. We present estimates for the savings as well as overheads due to a specific interleaving possibility under evaluation. In Section 4.4, we use these estimates in an overall exploration strategy for determining the final layout.

4.3.1 Problem Definition

We model the application as a sequence of n loop nests $L_1, L_2, \cdots, L_n$, with m arrays represented by set $A = \{A_1, A_2, \cdots, A_m\}$ accessed in these loops. Each loop L_i can have a subset $S_i \subseteq A$ of these arrays as *live* during its execution. Within loop L_i, the data layout for some arrays in set S_i can be interleaved. This is represented by *partitioning* of S_i into disjoint subsets. A specific partitioning candidate in loop L_i is represented as P_k^i (the k^{th} partitioning possibility of live arrays in loop L_i) given by:

$$P_k^i = \{P_{k_1}^i, P_{k_2}^i, \cdots, P_{k_z}^i\}$$

such that $z \in Z^+, P_{k_r}^i \subseteq S_i, S_i = \bigcup_{r=1}^{z} P_{k_r}^i$, and $P_{k_x}^i \bigcap P_{k_y}^i = \phi, \forall x \neq y$. If $|P_{k_r}^i| > 1$, the arrays included in set $P_{k_r}^i$ have their storage interleaved during the execution of loop L_i. If $|P_{k_r}^i| = 1$, then the conventional storage is used for the single array. Every valid partitioning candidate P_k^i represents a distinct interleaving possibility for the live arrays during loop L_i.

For example, consider a set of live arrays $S_i = \{Q, R, S, T, U\}$ accessed in loop L_i. One candidate partition of S_i is given by: $P_k^i = \{\{Q,R\},\{S,T\},\{U\}\}$, where $P_{k_1}^i = \{Q,R\}$, $P_{k_2}^i = \{S,T\}$, and $P_{k_3}^i = \{U\}$. This candidate partition corresponds to array Q being interleaved with R; S being separately interleaved with T; and U not being interleaved, during loop L_i.

For every candidate partition P_k^i, we can associate a cost given by $cost(P_k^i)$, which is the estimated cost of memory accesses to the arrays in S_i, as a result of the interleaving scheme implied by P_k^i. Since the interleaving choice can be different in different loops for the same array, there may be a transition cost given by $transition(P_k^i \rightarrow P_{k'}^{i+1})$ as we proceed from partition P_k^i in loop L_i to partition $P_{k'}^{i+1}$ in loop L_{i+1}. For example, let $P_k^i = \{\{Q,R\},\{S,T\},U\}$ and $P_{k'}^{i+1} = \{\{Q\},\{R\},\{S\},\{T,U\}\}$. The transition cost $transition(P_k^i \rightarrow P_{k'}^{i+1})$ would be given by the overhead involved in rearranging the data after the completion of loop L_i and before starting loop L_{i+1}. The rearrangement involves de-interleaving operations for (Q,R) and (S,T), and the new interleaving operations for arrays T and U.

The Array Interleaving problem can now be defined as choosing the best set of partitions $\{P_{K1}^1, P_{K2}^2, \cdots, P_{Kn}^n\}$ that minimizes the total cost defined as:

$$\begin{aligned} Total\ Cost = {} & cost(P_{K1}^1) + transition(P_{K1}^1 \rightarrow P_{K2}^2) + cost(P_{K2}^2) + transition(P_{K2}^2 \rightarrow P_{K3}^3) \\ & + \cdots + transition(P_{Kn-1}^{n-1} \rightarrow P_{Kn}^n) + cost(P_{Kn}^n)) \end{aligned} \tag{4.1}$$

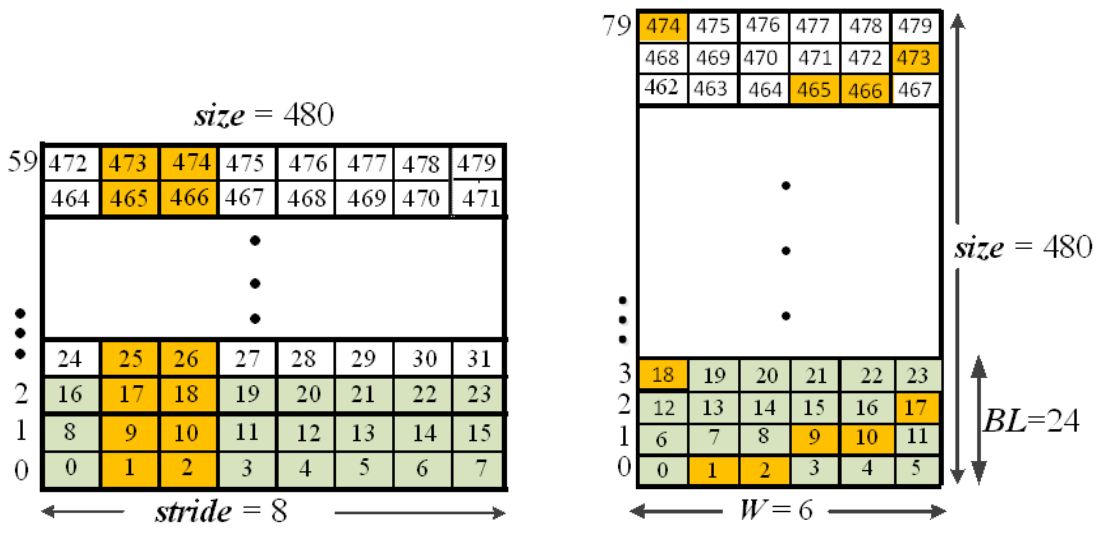
4.3.2 Memory Access Cost Estimation

In general, when there are several arrays and loop nests in an application, a large number of interleaving possibilities could exist. The number of candidate partitions P_k^i at each loop

grows exponentially with $|S_i|$, and is given by the Bell number $B_{|S_i|}$ [77]. The total number of possible candidate solutions for the whole application is the product $\prod_{i=1}^{n} B_{|S_i|}$. We use an estimation based strategy to evaluate the energy cost attributed to a candidate interleaving possibility, without explicitly going through detailed simulation.

Since the primary target of the data layout transformation is the number of memory accesses, we first attempt to estimate the number of memory accesses in a given loop L_i, due to the interleaving suggested by a partition P_k^i. Memory size remains constant in our exploration, so memory size estimation is not considered. We assume in this section that the *granularity* of interleaving, $g = 1$. That is, when p arrays are interleaved, we store the first element from all p arrays in sequence, then the second element from all p arrays, and so on. We generalize this scheme later in Section 4.4.4.

For every distinct array accessed in loop L_i, we attempt to estimate the number of memory accesses through the wide memory interface to the vector register of width W. We begin by studying the arrays access patterns in the loop. Figure 4.4(a) illustrates the accesses for array A (or B) corresponding to the loop shown in Figure 4.1(a), with an iteration count of 60 and a loop *stride* of 8. We have 60 rows (= loop iterations) and 8 columns (= loop stride). In Figure 4.4(b) we show the same array accesses visualized according to the memory accesses made, assuming vector registers of width $W = 6$ elements (SIMD factor). A memory access is made to fetch a row into a vector register if there is an access to one or more elements in the row.



(a) Array access pattern

(b) Visualizing memory accesses with $W = 6$

Figure 4.4: Access pattern for loop in Figure 4.1(b)

We first divide the array A in loop L_i into blocks of length BL_A^i, where

$$BL_A^i = L.C.M.(max\_array\_span_A^i, W) \tag{4.2}$$

$max\_array\_span_A^i$ represents the maximum span of elements of array A accessed in one iteration of loop L_i, and is given by:

$$max\_array\_span_A^i = max\_coeff^i{}_A \times loop\_stride_i \tag{4.3}$$

and $max\_coeff^i{}_A$ is the maximum of the loop index co-efficients of array A accessed in the loop L_i. The LCM is used since the access pattern repeats after BL_A^i elements in both the loop specification, as well as the memory access sequence. In our example, we have $max\_coeff^i{}_A = 1$ and $BL_A^i = L.C.M.(1 \times 8, 6) = L.C.M.(8, 6) = 24$, i.e., 3 rows with 8 elements each of Figure 4.4(a) are mapped as 4 rows with 6 elements each as in Figure 4.4(b). This blocking pattern is repeated over the entire array space.

The number of blocks into which the array A of size $size_A$ is sub-divided in loop L_i is given by:

$$\#Blk_A^i = \left\lceil \frac{size_A}{BL_A^i} \right\rceil \tag{4.4}$$

We now proceed to find the number of memory accesses associated with a block of length BL_A^i. Separate lists are generated corresponding to different loop index co-efficients for array A in loop L_i, with one list node corresponding to each distinct offset. Figure 4.5 illustrates this with an example loop accessing an array A with multiple reference patterns. As a simplification, the list includes elements accessed in the block length BL_A^i for array A in loop L_i after $\left(\frac{BL_A^i}{max\_array\_span_A^i}\right)$ loop iterations to avoid the error in access count estimation due to the offset values associated with the array A accesses in the loop. Considering only READ accesses in BL_A^i, we have two lists for the loop index coefficients 1 and 3 respectively. Algorithm 1 outlines the memory access count computation in a block, consisting of a virtual execution. It takes vector register width W and the sorted list of elements accessed in BL_A^i as input. The list $block\_elems[BL_A]$ is traversed from head to tail (Line 4), for estimating the access count from wide memory interface to vector registers. A counter, $count_A^i$, is initialized to 0 (Line 3) and incremented by 1 (Line 7) if the separation between the element associated with the last *count* is greater than the SIMD width W, as a vector access to one element would also account for the next $(W-1)$ consecutive elements (Line 6).

For cases where an array has multiple loop index co-efficients or access patterns (p), $count(W, BL)$ is computed as:

$$count_A^i = \sum_{m=1}^{p} count_{A,m}^i \times \frac{C_{A,m}^i}{max\_coeff^i{}_A} \tag{4.5}$$

Algorithm 1 *block_accesses*: Memory Access Count

Input: Vector register width W, elements of array A accessed in a block, $block\_elems[BL_A]$
Output: Memory accesses count for the block
$k = 0, count^i_A = 0$;
while $(k < BL^i_A)$ **do**
  if $(block\_elems[k] == TRUE)$ **then**
    $k = k + W$
    $count^i_A = count^i_A + 1$
  else
    $k = k + 1$
  end if
end while
Return $count^i_A$

where $count^i_{A,m}$ is the count value obtained for co-efficient m, $C^i_{A,m}$ is the loop index co-efficient value for the m^{th} list, and $max\_coeff^i_A$ is the maximum of the index co-efficients corresponding to array A accesses. Fraction $\frac{C^i_{A,m}}{max\_coeff^i_A}$ is multiplied with count value for each m as with iterations corresponding to BL_A, only this fraction of the elements in m^{th} list is actually used and are to be accessed.

Thus, the total number of memory accesses for an array partition ($P^i_{k_r}$) is given by

$$mem\_acc(P^i_{k_r}) = \#Blk^i_A \times count^i_A \tag{4.6}$$

We illustrate the above memory access computation using the example in Figure 4.5. There are two reference patterns associated with READ accesses to array A. $A[i+k_1]$ covers the accesses $A[i+10]$ and $A[i+11]$. A separate list is used for references of the form $A[3i+1]$. Thus we have $p = 2$ with index coefficients $C_{A,1} = 1$ and $C_{A,2} = 3$. With $W = 6$, *loop_stride* as 3, the array span values ($array\_span = access\_coeff \times loop\_stride$) are 3 and 9. Thus, the block length $BL_A = L.C.M.(max(3,9),6) = 18$. The $count_A$ values corresponding to $C_{A,1}, C_{A,2}$ are 3 and 2 respectively as highlighted in Figure 4.5(b). Thus, $count_A = 3 \times \frac{1}{3} + 2 \times \frac{3}{3} = 3$ and $mem\_acc(A) = 5 \times 3 = 15$ (with $\#Blk_A = \lceil \frac{75}{18} \rceil = 5$).

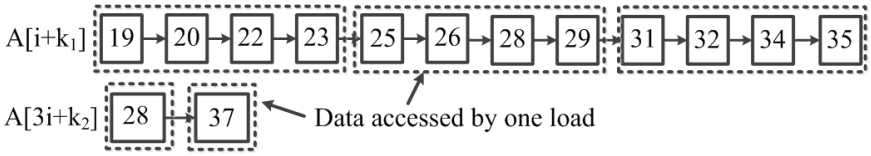
The above computation assumes that the iteration space is rectangular. If the space is not rectangular, and all array blocks are not accessed, then we multiply the estimated access count by a factor f_A, where:

$$f_A = \frac{\text{Accessed volume of Array } A}{\text{Total \# elements in } A} \tag{4.7}$$

The numerator is the volume of the polytope enclosing the accessed points in the iteration

```
for ( i =0;  i <25;  i +=3)  {
   READ  A[ i  +  10]
   READ  A[ i  +  11]
   READ  A[3 i  +  1]
   WRITE  A[ i ]
}
```

(a) Example loop



(b) Representation for READ accesses in BL_A

Figure 4.5: Access count computation example

space, and is estimated using the polytope based integer point counting method [78, 79]. The adjusted memory accesses count for partition ($P^i_{k_r}$) is given by

$$mem\_acc(P^i_{k_r}) = f_A \times \#Blk^i_A \times count^i_A \tag{4.8}$$

Thus, the estimated memory access count for the partition P^i_k becomes:

$$mem\_acc(P^i_k) = \sum_r mem\_acc(P^i_{k_r}) \tag{4.9}$$

where the summation is over the different subsets of interleaved arrays in partition P^i_k. This gives us the estimated number of memory accesses for a candidate partition. This computation is repeated separately for READ accesses and WRITE accesses, and combined with the respective memory energy per access values, to generate the estimated memory energy spent in the loops due to the candidate set of interleaved array accesses. The gain with the proposed interleaving is defined as

$$gain = mem\_acc_{unopt} - mem\_acc_{opt} \tag{4.10}$$

where the mem_acc_{unopt} and mem_acc_{opt} are the memory access counts for the conventional mapping and the proposed layout strategy respectively. Both of these can be estimated using

the approach discussed above.

4.3.3 Estimation of Overheads

As shown in the example of Figure 4.3, the optimal set of interleavings in one loop could be different from that in another, and the solutions could be conflicting in that the same array could be interleaved in different ways in the two loops. It is necessary to evaluate the cost of change in the data layout across different loops, so that the global estimated energy cost could account for both the energy cost of array accesses within the loops, and also the impact of any dynamic layout changes. In the general case, we may need to de-interleave arrays after a loop ends and interleave them in a different way before the next loop begins.

Interleaving Cost Estimate

We first estimate the number of vector register operations required to perform the array interleaving operation. Consider a scenario where we aim at interleaving four arrays A, B, C, and D. Figure 4.6 shows an example of the type of operations involved in the process of interleaving the first 4 elements of each array. Two stages of register interleaving are necessary; in the second stage, the register operands are modified to achieve the desired effect. Generalizing to N arrays each of size $size(A)$, vector width W, and instructions capable of interleaving R vector registers simultaneously, the number of interleaving stages, is $\lceil \log_R N \rceil$, and the total number of elements participating in the interleaving is $N \times size(A)$. The total number of vector register operations is given by:

$$\#Reg\_ops\_int = \frac{N \times size(A)}{W} \times \log_R N \tag{4.11}$$

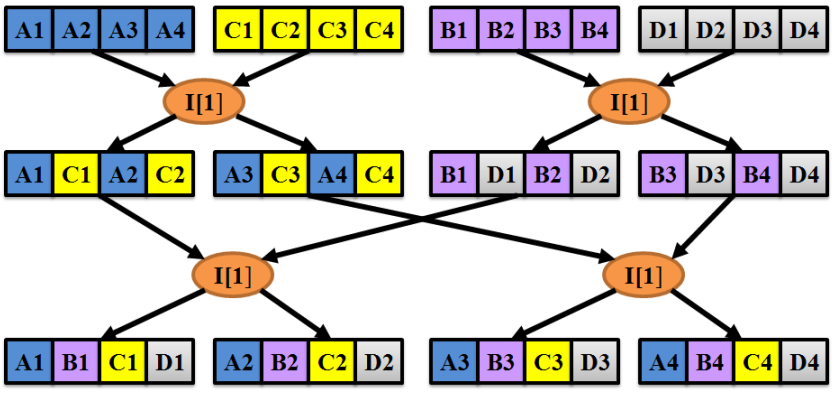


Figure 4.6: Interleaving multiple arrays ($N = 4$) with 2 stages

The energy overhead attributed to the interleaving operations can now be estimated by using the operation count and the energy models of the register file and datapath. For estimating the

performance overhead caused by interleaving, we use the same approach and substitute the cycle counts for the register operations and also take into account the degree of parallelism available for performing the register operations. In our example processor, the interleaving operations shown in Figure 4.6 require 2 cycles, with each stage requiring one cycle.

Two important issues related to implementation efficiency arise in the context of an accurate cost estimation in practical situations:

- The interleaving operation required may be partial. For example, we may need to derive the interleaving cost between loop L_i where arrays $\{A,B\}$ are already interleaved, to loop L_{i+1} where we need the interleaving of four arrays $\{A,B,C,D\}$. Since the first stage of interleaving A and B is skipped, the estimate in Equation 4.11 has to be modified suitably.

- In addition to the register operation related cost indicated by Equation 4.11, we may also have additional costs due to the vector load and store operations for the initial transfer of memory data into the vector registers, and the final transfer back to memory. In Figure 4.6, these would appear before the first stage and after the final stage, to complete the functionality. However, it is a frequent practical occurrence that the interleaving operation need not be instantiated as an explicit data re-arrangement between loops, and instead, can be integrated into the array WRITE operations in a preceding loop. This situation lends itself to a substantial reduction in the interleaving cost because we can avoid a large number of redundant memory READs and WRITEs needed for interleaving. In our strategy, we attempt to integrate the interleaving into the loop where the constituent array subsets were last written.

Adjustments are made in Equation 4.11 to account for reduced register operations in case of partial interleaving, and additional memory access costs applicable. If the set of arrays to be interleaved has a compatible layout, then we can further interleave the two. Otherwise, the sets need to be de-interleaved first, followed by interleaving. For interleaving cost computation, N in Equation 4.11 is taken to be the number of sets. If an array has to be explicitly fetched from memory before interleaving, then an additional number of memory accesses ($= size(A)/W$) is added.

De-Interleaving Cost Estimate

A *de-interleaving* operation is sometimes necessary when a new interleaving structure requires us to first undo the effect of a previous interleaving. For example, assume that arrays $\{A,B\}$ are already interleaved, and we need to generate a new interleaving $\{A,C\}$. The $\{A,B\}$ subset has to be de-interleaved first, followed by the $\{A,C\}$ interleaving.

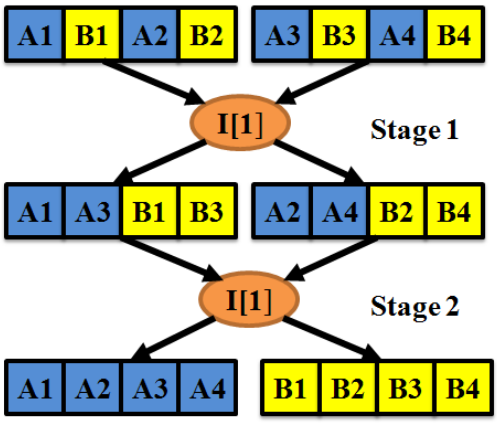


Figure 4.7: De-interleave Operation

De-interleaving is implemented using the same interleaving instructions in multiple stages. In each stage, the elements of an array separated by (W/R) are brought together. Figure 4.7 illustrates the de-interleaving operation with $R = 2$ and $W = 4$. The arrays are re-arranged contiguously in $\log_R W$ stages. The overhead of additional register operations needed for de-interleaving is given by:

$$\#Reg\_ops\_deint = \frac{N \times size(A)}{W} \times \log_R W \tag{4.12}$$

4.4 Exploration for Interleaving Decision

4.4.1 Representation

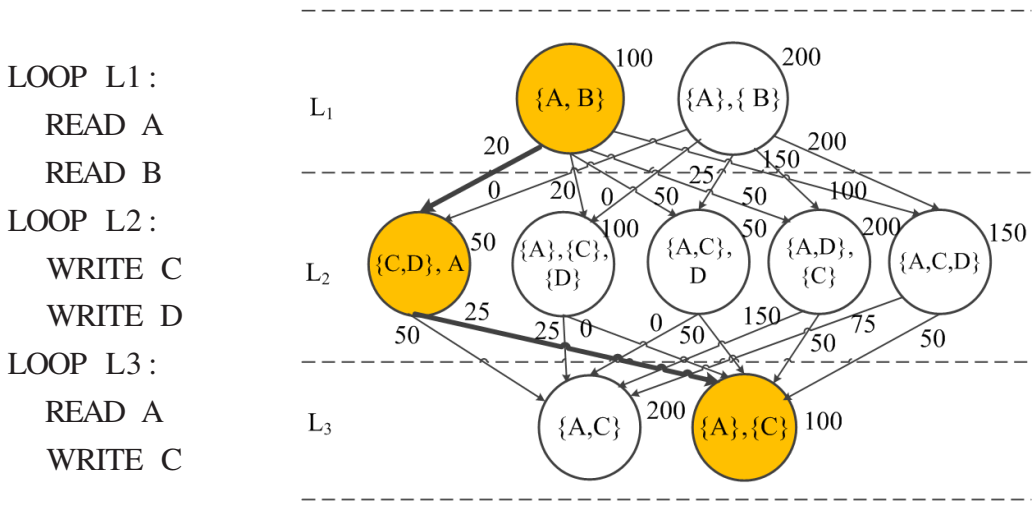
We aim at a global search space exploration where potentially all valid interleaving choices in an application code are explored but with pruning strategies (discussed next in this section) to bound the exploration time. We begin by building an Array Interleaving Graph (AIG) to represent the loops, live arrays in each loop, and the cost of candidate partitions and transitions between partitions as the execution proceeds through the partitions. An array is live in the loop, if there is a use of that array in the path without redefining the array from the loop to the end of the control flow graph of the program. By redefining the array, we mean that all the elements of the array are reassigned. For an application with loop sequence $L_1, L_2, ..., L_n$, the AIG has n levels, with each level containing a set of nodes, with each node representing a candidate partition P_k^i. Every node has an associated node cost $(= cost(P_k^i))$, representing the estimated memory cost in loop L_i due to the partitioning suggested by the node. An edge is introduced between node P_k^i to node $P_{k'}^{i+1}$ at the next level, with an annotated transition cost $(= transition(P_k^i \rightarrow P_{k'}^{i+1}))$.

With the above graph construction the objective is to find the optimal set of partitions in all the loops in such a way that the *Total Cost* defined in Equation 4.1 is minimized. This translates to finding the lowest cost path in the AIG from any node at the first level, to any node at the last (n^{th}) level, where the path cost includes the weights of all the selected nodes and all the edges that constitute the path.

Consider an example program P with 3 loops $L1$, $L2$, and $L3$; and 4 arrays A, B, C, and D. Let arrays $\{A,B\}$ be accessed in loop $L1$; arrays $\{C,D\}$ be accessed in loop $L2$, and $\{A,C\}$ be accessed in $L3$. The set of *live* arrays S_i at each loop is given by: $S_1 = \{A,B\}$; $S_2 = \{A,C,D\}$; $S_3 = \{A,C\}$. A is live during $L2$ because it is read in $L3$. Figure 4.8(a) illustrates the situation. The set of candidate partitions at each loop are as follows:

$$
\begin{aligned}
P_1^1 &= \{A,B\} \\
P_2^1 &= \{\{A\},\{B\}\} \\
P_1^2 &= \{\{A,C\},D\} \\
P_2^2 &= \{\{A,D\},C\} \\
P_3^2 &= \{\{C,D\},A\} \\
P_4^2 &= \{A,C,D\} \\
P_5^2 &= \{\{A\},\{C\},\{D\}\} \\
P_1^3 &= \{A,C\} \\
P_2^3 &= \{\{A\},\{C\}\}
\end{aligned}
$$

Each partition is represented by a node in the Array Interleaving Graph shown in Figure 4.8(b). After populating the graph with node and edge weights, we need to determine the optimal (lowest cost) path from any node at Level 1 to any node at Level 3. For the example weights indicated in Figure 4.8(b), the optimal path is highlighted.



(a) Example loop

(b) Representation and Optimal solution

Figure 4.8: Array Interleaving Graph

4.4.2 Search Space Pruning

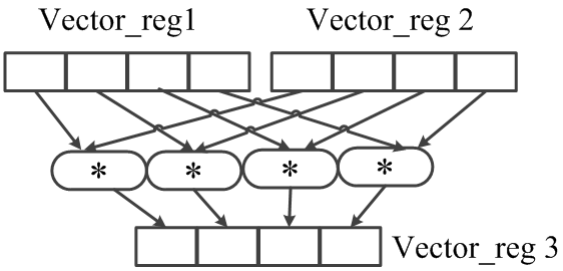
Before proceeding with the interleaving, we take an important intermediate step—the pruning of the search space, which significantly reduces the size of the array interleaving graph. The number of partitioning possibilities, as indicated earlier, grows exponentially with the number of live arrays in a loop. However, in practice, we can significantly reduce the number of partitions to be considered by employing some simple rules for pruning the search space. Since our exploration is mainly intended for SIMD architecture platforms, these pruning strategies are oriented primarily for applications subject to vector accesses.

Compatible arrays for interleaving are identified by ensuring that they satisfy all the following criteria.

- *Similar Reference Pattern* – Since interleaved arrays are accessed through similar memory load and store instructions, and are operands to the same vector operations, we first ensure that their array reference patterns are identical in the loop. If the iteration vector for a loop nest is I, and the co-efficient matrix C gives the array reference expression by the matrix-vector product $C.I$, then compatible arrays A and B are identified by ensuring that every reference $C.I$ for array A in the loop is matched by a reference $C.I$ for array B. For the example loop shown in Figure 4.9(a), this compatibility test divides all the arrays into two sets of arrays $X = \{E,F,G,L\}$ and $Y = \{A,B,C,D,H\}$, based on their index expressions. Arrays in set X need not be interleaved with those in Y. Other rules are used to further discriminate between arrays within the two sets.

```
for(i=1; i<=100; i+=2) {
    E[i]=A[4i] * B[4i];
    F[i]= C[4i] * D[4i];
    L[i]= G[i] + H[4i];
}
```

(a) Example loop



(b) Vector multiplication

Figure 4.9: Example loop with an illustration for vector operation

Further, if the co-efficient matrices C for two array references are identical throughout all the loops, then the two need not be distinguished when constructing partition candidates. For example, if arrays A and B have identical co-efficient matrices, then the partitioning $\{\{A,C\},B\}$ need not be distinguished from the partitioning $\{\{B,C\},A\}$. To generalize, if there is a set of arrays being considered for interleaving, and the arrays have identical co-efficient matrices in all loops, then the candidate partitions represented as explicit nodes in the AIG can be distinguished only in the number of such arrays present in the partition.

- *Same Operation Type* – Since vector operations are performed on the interleaved arrays, the same operation is applied to all vector elements, as shown in Figure 4.9(b). Thus, the arrays being interleaved need to be operands to the same type of operation. Based on this criterion, we can divide the arrays in our example into two groups: $\{A,B,C,D\}$ and $\{G,H\}$. Note that G and H were already identified to be incompatible based on the similar reference pattern rule.

- *Operands of the same instruction* – The set of different operands for the vector FU are loaded from different vector registers, as shown in Figure 4.9(b). If the operands of the same instruction are interleaved, further de-interleaving instructions would be necessary to align the vector operands correctly before proceeding with the SIMD operation. To reduce these overheads that degrade system performance, we prune out such pairs and disallow their interleaving. Taking such conflicts into consideration, the set $\{A,B,C,D\}$ is refined into just the following compatible pairs: $\{A,C\}$, $\{A,D\}$, $\{B,D\}$ and $\{B,C\}$. The pairs $\{A,B\}$ and $\{C,D\}$ are excluded.

- *Compatible Loads and Stores* – Since loads and stores occur at widths equal to the vector register width, the interleaved arrays must be either all loaded together or stored together. If an array is only read in a loop, and a different array is only stored, then the two are not candidates for interleaving. In the example under discussion, the set $\{E,F,G,L\}$ is split

into the following compatible sets: $\{E,F\}$, $\{G\}$ and $\{L\}$. The pair $\{E,F\}$ is compatible because both E and F are outputs (*Stores*) of the same operation type (multiplication). L being the output of a different operation type (addition), is placed in a different set from $\{E,F\}$. G is the only *Load* operation in the set $\{E,F,G,L\}$ and is placed in a separate set.

- *SIMD Factor* – The SIMD factor W imposes a natural upper limit to the set of interleaving candidate arrays. Since the vector FUs can execute only W operations simultaneously, we can restrict the candidate partitions to interleave no more than W arrays.

The above pruning rules help in significantly reducing the number of candidate partitions at each loop. This is very important from the point of view of keeping the AIG small. The pruning criteria related to compatible loads/stores and SIMD factor are natural consequences of SIMD/vector architectures. The operation and instruction type criteria are specific to the architecture we are using, and help reduce the search space, although these are also properties of the SIMD architectures being targeted. However, they do not influence our overall formulation in any way. If these restrictions are absent in other architecture, they would be removed from the pruning pattern list. In that sense, the pruning criteria represent the incorporation of architecture specific constraints in the formulation.

4.4.3 Optimal Path Through AIG

As discussed earlier, the optimal solution to the global array interleaving problem corresponds to least cost path from any node at Level 1 to any node at Level n in the AIG, with the path cost being defined as the sum of node weights and connected edge weights. Figure 4.10 outlines the scenario. A dynamic programming formulation can be used to obtain the optimal path. The overall strategy is outlined in Algorithm Optimal_AIG_Path. At every node p in the graph, we maintain the best path terminating at p and the associated path cost *opt_partial_cost*. Consider the node corresponding to partition P_k^i at level i. The optimal path ending at this node must have one of the nodes at level $i-1$ as the preceding node. Because of the nature of the graph, we can obtain the best path as follows (Line 12-14 in Algorithm 2)

$opt\_partial\_cost(P_k^i) = \min(opt\_partial\_cost(P_r^{i-1}) + transition(P_r^{i-1} \rightarrow P_k^i) + cost(P_k^i))$ $\forall$ nodes r at level $i-1$

This is illustrated in Figure 4.10. When this is completed for all k (Line 17), we have solved the problem for all nodes up to level i. When the traversal is completed for all levels i, the minimum cost path among all nodes at level n gives us the optimal path cost in the graph

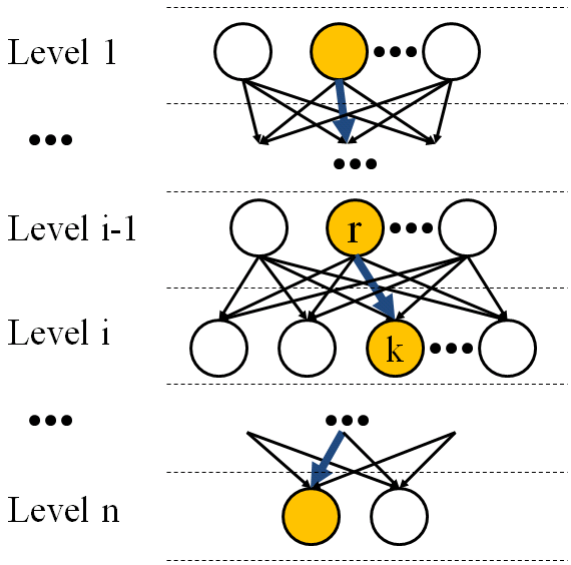


Figure 4.10: Optimal path through AIG

Algorithm 2 Optimal_AIG_Path

1: **Input:** AIG
2: **Output:** Optimal Path
3: **for all** nodes P_k^1 at level 1 **do**
4: $Pred(P_k^1) = \phi$
5: $opt\_partial\_cost(P_k^1) = cost(P_k^1)$
6: **end for**
7: **for** level $i = 2$ to n **do**
8: **for all** nodes P_k^i at level i **do**
9: $opt\_partial\_cost(P_k^i) = \infty$
10: **for all** edges $P_r^{i-1} \rightarrow P_k^i$ from P_r^{i-1} to node P_k^i **do**
11: $cost = opt\_partial\_cost(P_r^{i-1}) + transition(P_r^{i-1} \rightarrow P_k^i) + cost(P_k^i)$
12: **if** $cost < opt\_partial\_cost(P_k^i)$ **then**
13: $opt\_partial\_cost(P_k^i) = cost$
14: $Pred(P_k^i) = P_r^{i-1}$
15: **end if**
16: **end for**
17: **end for**
18: **end for**
19: **return** Path with minimum $opt\_partial\_cost(P_k^n)$ among all nodes terminating at level n

(Line 19). The path itself can be traced through the *Pred* values recorded at each node. The optimal path corresponds to the globally optimal set of interleaving decisions for all arrays accessed in all loops of the application.

The algorithm is illustrated with an example in Figure 4.8(b). The optimal partial costs for one node at Level 2 are computed as follows:

$opt\_partial\_cost(\{C,D\},A)$ = min(100+20+50,200+0+50) = 170

Node $\{A,B\}$ is noted as $Pred(\{C,D\},A)$. On completing the partial cost computation for all nodes, we select the node with the minimum *opt_partial_cost* at Level 3. The optimal path is highlighted. The implied optimal array interleaving decisions are:
Loop 1: $\{A,B\}$
Loop 2: $\{\{C,D\},A\}$
Loop 3: $\{\{A\},\{C\}\}$

4.4.4 Effect of Interleaving Granularity

As discussed earlier, the *granularity* g is another parameter to the interleaving transformation. The granularity could be fine ($g = 1$, the default granularity) or coarse ($g > 1$). Coarser granularities could be preferred when consecutive elements from the same array undergo the same operation in the same loop, particularly because the cost of de-interleaving is lesser for higher granularities.

Granularity Parameter

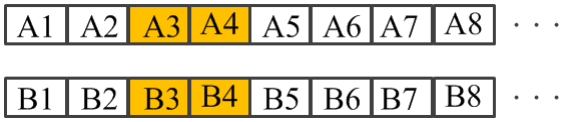
The interleaving granularity, g and number of arrays N that can be interleaved must satisfy the relationship:

$$N \times g = W \tag{4.13}$$

Figure 4.11 shows the resulting memory access patterns due to different interleaving granularities. Clearly, not all values (such as $g = 3$) are suitable for optimizing memory accesses (Figure 4.11(d)). We need to consider only integral factors of W as granularity values. The optimal value of g for an array in a loop is subject to the same global analysis as earlier, because the decision taken for one loop can affect the behavior in a different loop. To accommodate this, we can just treat g as an orthogonal parameter, and replicate each AIG node to represent different g values for the candidate interleavings (all interleaved arrays must have the same g value). However, an elementary pruning strategy helps in significantly reducing the number of such replications. Granularity values $g > 1$ need to be considered only if multiple elements of the same array are operands of the same vector instruction.

Incorporating Granularity into Cost

Finally, we need to estimate the overhead due to interleaving and de-interleaving operations when the granularity exceeds 1. Considering the instructions we have used in our processor



(a) Arrays A and B

(b) Granularity $g = 1$

(c) Granularity $g = 2$

(d) Granularity $g = 3$

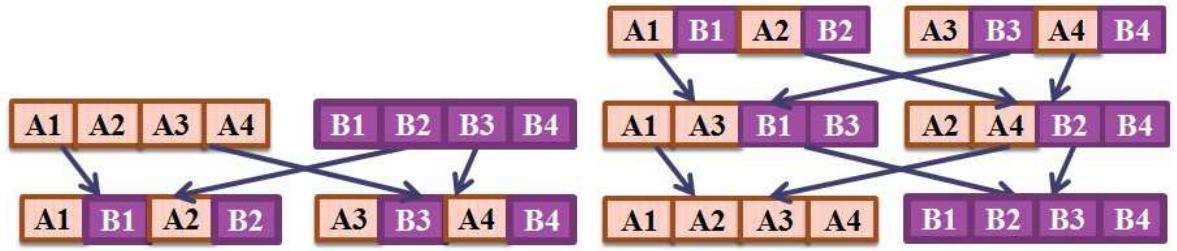
Figure 4.11: Access patterns with different interleaving granularity

for interleaving, the granularity does not change the number of stages and operations while interleaving. However it reduces the number of operations for de-interleaving. Instead of W elements being re-arranged, we have W/g sets of elements undergoing re-arrangement. Thus, the number of stages reduces from $log_R W$ for re-arrangement of W elements to $log_R(W/g)$, thereby reducing the number of operations to:

$$\#Reg\_ops\_deint = \frac{N \times size(A)}{W} \times \log_R \frac{W}{g} \tag{4.14}$$

Node cost estimates do not change with granularity.

Figure 4.12 illustrates the effect of granularity on the number of stages in interleaving/de-interleaving. Figure 4.12(a) shows that interleaving W elements of two arrays with granularity $g = 1$ is a single stage operation. While de-interleaving, the resulting layout requires 2 stages. In stage n, elements $\frac{W}{R^n}$ apart are made adjacent. In Figure 4.12(b), with $W = 4$, $R = 2$ in Stage 1, elements $A1$, $A3$ at distance 2 ($=\frac{4}{2^1}$) are made adjacent. In Stage 2, $A1$, $A2$, $A3$, $A4$ are grouped together as the separation distance between the interleaved elements reduces to 1 ($=\frac{4}{2^2}$). Considering granularity $g = 2$ for both interleaving and de-interleaving, we observe that interleaving still requires one stage (Figure 4.12(c)) while the number of stages for de-interleaving reduces to one (Figure 4.12(d)). This is because with $g = 2$ we have 2 ($\frac{W}{g} = \frac{4}{2}$)



(a) Interleaving with granularity $g = 1$

(b) De-interleaving with granularity $g = 1$

(c) Interleaving with granularity $g = 2$

(d) De-interleaving with granularity $g = 2$

Figure 4.12: Interleaving/De-interleaving with different granularities

sets of elements undergoing re-arrangement in one stage. Thus, the number of stages reduces from $log_2 4$ (= 2) to $log_2 2$ (= 1), since the number of element sets to be re-arranged in a register reduces from 4 to 2.

4.4.5 Analysis

The following comments are noteworthy regarding our proposed exploration algorithm for array interleaving.

- Algorithm 2 gives the optimal path in $O(e)$ time, where e is the number of edges in the AIG, since each edge is traversed exactly once in the exploration algorithm.
- Although the exploration algorithm itself is efficient, we need to ensure that the size of the AIG does not explode due to the possibly large number of candidate partitions to be explored. We have found in our experiments that the pruning strategy significantly reduces the actual number of nodes in the AIG for practical examples.
- The costs and the exploration target energy reduction through the interleaving process. However, the strategy can also be re-used for performance optimization. The node and transition cost estimates would need to be minimally modified to generate the performance costs due to the interleaving, by multiplying the memory access count by the cycles required for each access instead of energy per access. Where both energy and perfor-

mance are important parameters, a practical methodology could generate energy-optimal solutions, while using the adapted strategy to also generate a performance overhead estimate. The final choice can be made by the designer based on whether the performance overhead is acceptable. The converse approach, where performance is the primary optimization criterion, is also possible. The general approach is to generate a Pareto curve under different constraints so that a designer can manually select the preferred operation point.

- We have assumed that we can model the application as an ordered sequence of loop nests, and that loop and array access parameters such as bounds, strides, and co-efficient matrices, can be statically determined. A large number of embedded systems applications in the communication and multimedia domain indeed satisfy these conditions. Section 4.5 discusses the exploration results on different applications from these domains. For more general behaviors, the problem formulation would need to be extended.

4.5 Experimental Results

4.5.1 Compilation and Simulation Framework

We have implemented our array interleaving strategy using the framework discussed in Section 3.5. Since the target architecture is a configurable processor that can be customized in various ways, the standard evaluation and execution mechanism is to run the programs on a processor simulator. An XML based language is used to describe the architecture, and a cycle-accurate simulator of the processor is used to simulate the generated code on the architecture. Simulation is used to evaluate the two versions: the unoptimized (baseline) and the optimized implementation (after data layout transformation), and statistics reported by the simulator are collected for both versions. We present a comparison of these two implementations (for memory energy and performance) in this section.

Our proposed algorithm has been implemented in a compiler framework, with one manual step involved. The LLVM based framework takes a C-style input and collects relevant parametric details for computing the different cost estimates. This is followed by a compiler pass that performs a detailed evaluation of these equations and computes the data layout decisions for the arrays across different program regions. Based on these results we manually re-write the original application to obtain the optimized version in high level language using the intrinsic operations for the SIMD architecture under consideration; this reflects the interleaving decision suggested by the algorithm. The two versions (original and optimized) are then compiled with DRESC an extension of the IMPACT compiler targeting the ADRES architecture instance.

4.5.2 Exploration Experiments

We evaluated our approach on different applications with varying complexity, ranging from single loop kernels to complex applications such as *LTE* downlink receiver. We measured the improvement in terms of memory access energy by comparing the implementations with optimized (IL) against the corresponding un-optimized ones (Non-IL). These unoptimized implementations are the ones being used in practice. The energy values reported here include the energy difference due to the array interleaving strategy, and also the overheads due to the additional vector register accesses and operations for interleaving and de-interleaving. The normalized energy values in the graphs are the energy values normalized with respect to the energy dissipation of a 32-bit adder.

- *Successive Over Relaxation (SOR)* – The SOR algorithm is a method for evaluating partial differential equations or solving linear system of equations [80]. It has a single nested loop with 8 arrays. Figure 4.13 shows a comparison of the energy results due to the unoptimized and optimized storage decisions.

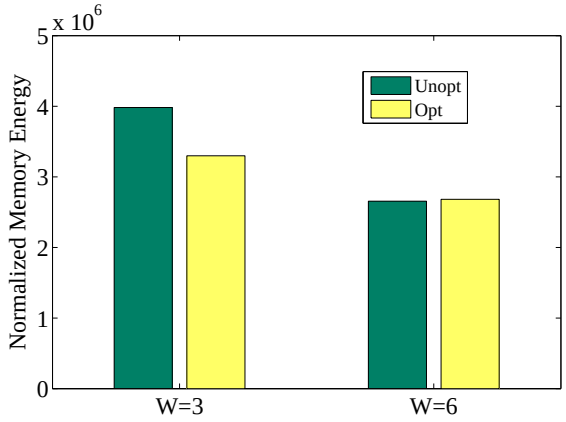
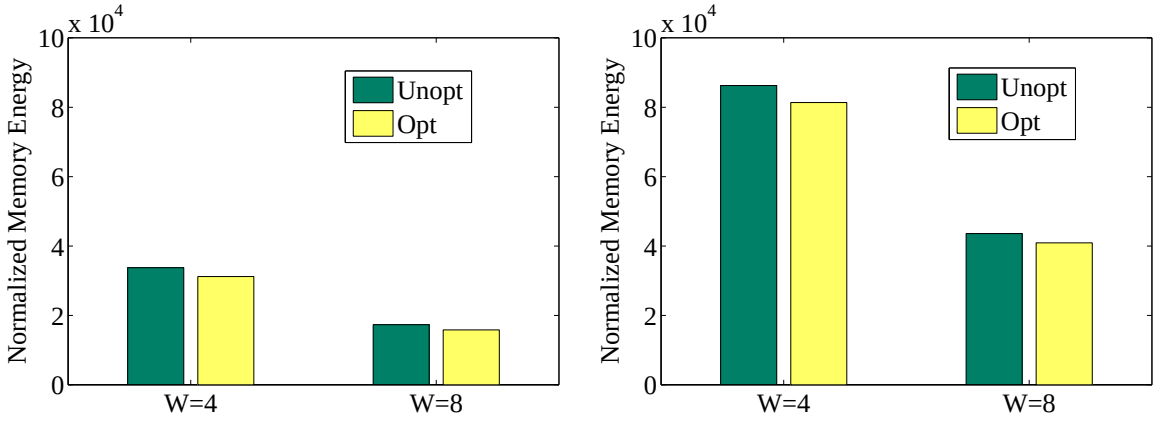


Figure 4.13: Non-optimized vs. Optimized storage for SOR

Interleaving decision for arrays results in memory energy reduction of 17.2% for an implementation with SIMD width $W = 3$. The same decision does not hold good for $W = 6$ as it results in no gain with respect to data accesses but proves costlier once overheads associated with interleaving are accounted for. This is correctly predicted by our exploration technique, and no interleaving is performed in this case.

- *Channel Estimation* – This application implements the DCT (Discrete Cosine Transform) based Channel Estimation in MIMO 2x2 802.11n standard. Of the several available implementations of DCT, the DCT-II [81] is frequently used in practice. We use the performance efficient implementation for N-point DCT using N-point FFT [82]. For the 20

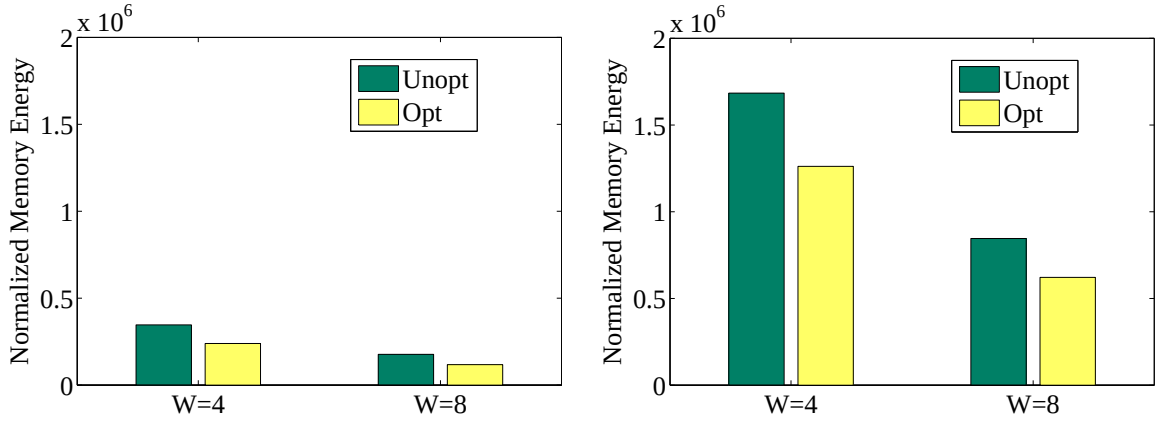
MHz channel bandwidth, we have N = 64, while for 40 MHz channel bandwidth, we have N = 128. The implementations involve three and four loops for the 20 and 40 MHz channel bandwidths respectively, with each loop accessing 5 arrays. By having interleaved arrays at the input to the kernel, 8.6% reduction in memory accesses is obtained for $W = 8$, and 7.5% for $W = 4$ for 20MHz bandwidth. A similar exploration for 40 MHz bandwidth (Figure 4.14(b)) resulted in 5.7% and 6.2% energy savings with SIMD widths of 4 and 8 respectively. The interleaving decisions help in reducing the co-efficient accesses and also group the information carrying sub-carriers from multiple transmitters together, to be used for further processing.



(a) 20 MHz bandwidth (b) 40 MHz bandwidth

Figure 4.14: Non-optimized vs. Optimized storage for channel estimation in IEEE 802.11n for different channel bandwidths

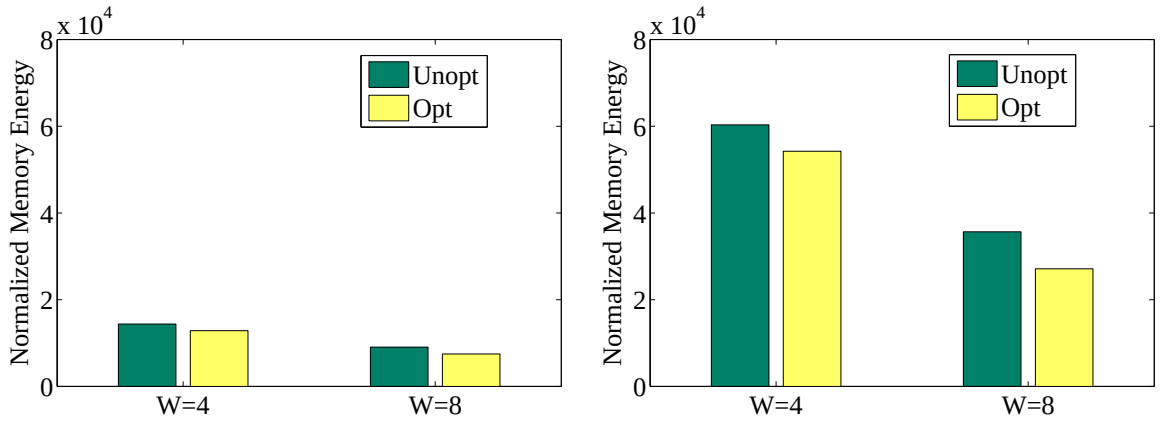
- *Vector Based Image Compression* – The Vector based image compression application involves the following steps [83]. The image is first divided into $M \times M$ sub-images, each divided into blocks of $N \times N$ pixels. Transformations on each block is carried out using an $N \times N$ DCT. The resulting DCT co-efficient sets, one set corresponding to each sub-image, are grouped into $M \times M$ vectors. Our algorithm returns a global decision for the interleaved storage of the sub-images stored as arrays. The algorithm implementation with $M = 4$ involves 18 arrays in 4 loops with $N = 8$, and in 5 loops with N = 16 because of the variation in DCT size. The exploration results in 33.5% reduction in memory access energy for $W = 8$, and 30.9% reduction for $W = 4$ (Figure 4.15(a)). With sub-image division into blocks of 256 (=16×16) pixels, energy savings of 26.5% and 25.1% are obtained for W = 8 and 4 respectively. This significant gain is possible since the output DCT co-efficients are stored in interleaved form, thereby reducing redundant loads and stores when grouped together in the third stage.



(a) Block of 8 x 8 pixels (b) Block of 16 x 16 pixels

Figure 4.15: Non-optimized vs. Optimized storage for image compression algorithms

- *Subband Coding* – Subband coding is also an image application involving the division of the original image into $M \times M$ sub-images [84]. Each sub-image is further decomposed into a set of sub-bands using a filter bank (e.g., Debaucles 4-Coefficient Wavelet filter). The resulting sub-bands are re-arranged in the form of vectors in such a way that components with the same index are grouped together. Each sub-image is stored in memory as a separate array (4 arrays when $M = 2$ and 16 when $M = 4$). We start with an already optimized implementation by merging the sub-band re-arrangement step with the filtering thereby reducing the redundant memory accesses for grouping the subbands with same indices into vectors.



(a) With 4 sub-images (b) With 16 sub-images

Figure 4.16: Non-optimized vs. Optimized storage for subband coding applications

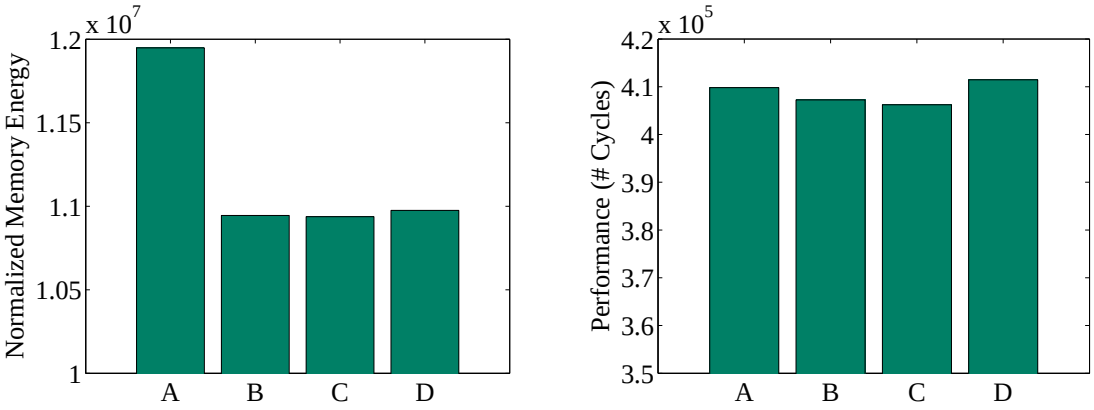
Our exploration results show that storing arrays in interleaved form during the filtering operations results in 11.2% energy savings when mapped with $W = 4$ while, the savings increase to 17.5% with $W = 8$ (Figure 4.16(a)). When the image is decomposed into 16 sub-images ($M = 4$), energy savings of 10.6% and 29.8% are achieved for SIMD widths (W) of 4 and 8 respectively (Figure 4.16(b)). In both cases, the gain is mostly due to reduction in the number of vector register accesses because of multiple extract and insert operations in the original application, which are avoided with interleaved array mapping.

4.5.3 Exploration results for LTE MIMO applications

We carried out experiments on LTE MIMO applications with Software Defined Radio (SDR) specified by 3GPP (3^{rd} Generation Partnership Project). The focus is on a series of kernels from the LTE downlink receiver, a data and compute intensive part of the application. The kernels include Orthogonal frequency-division multiplexing (OFDM) demodulation (comprising channel offset compensation and FFT), Shuffle, Pilot symbol processing (involving channel estimation, Frequency interpolation, Pre-Scaling and Matrix Inversion) covering the preamble part of the data processing block [65]. Our test cases are categorized as follows.

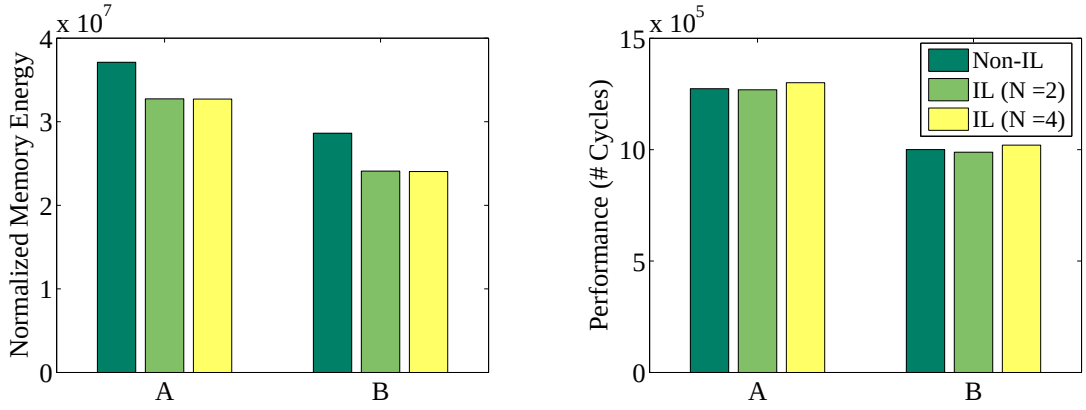
- *Effect of interleaving granularity and remapping in MIMO 2x2* – This configuration of LTE supports two antennas at the transmitter as well as the receiver end. A SIMD width $W = 8$ is used. Our exploration strategy resulted in best energy savings for interleaving the input arrays with granularity $g = 2$. A comparison among the different mapping strategies is shown in Figure 4.17. We observe that the mapping done by interleaving with $g = 2$ results in energy savings of 8.5% (Case C) while 8.3% reduction is obtained for $g = 1$ (Case B) with respect to a non-optimized mapping (Case A). Both mappings result in no performance overhead. A slight improvement in performance and energy is obtained with $g = 2$ because of reduced number of de-interleave operations while re-mapping. If there is no re-mapping, i.e., only one global layout is used, we have 8.1% energy savings but a slight performance overhead (Case D).

- *MIMO 4X4* – LTE 4x4 is a MIMO configuration of LTE supporting 4 antennas both at the transmitter as well as the receiver end. It is more computationally intensive than the 2x2 counterpart. Although the LTE 4x4 receiver is similar in functionality, the algorithm used here is different; the LTE 2x2 algorithm, when used directly, leads to higher degradation in LTE 4x4. Hence, we performed a separate exploration of LTE 4x4 to determine the interleaving decisions.



(a) Memory access energy variation (b) Performance variation

Figure 4.17: Non-optimized vs. Optimized storage for LTE2x2 with channel bandwidth=20MHz Cases: (A) Non-interleaved (B) Interleaved (g=1) (C) Interleaved (g=2) (D) Without remapping



(a) Memory access energy variation (b) Performance variation

Figure 4.18: Non-optimized vs. Optimized storage in LTE 4x4: (A) BW=20 MHz, 100% occupied, (B) BW=20 MHz, 1% occupied

Figure 4.18 compares the reduction in memory access energy and the performance difference for two different interleaving possibilities in the main computation kernels – interleaving 2 arrays (IL N = 2), and interleaving 4 arrays (IL N = 4). We obtain 11.8% reduction in terms of energy when at most 2 arrays are interleaved and 11.9% when 4 arrays are interleaved with respect to non-interleaved layout for case A; and reductions of 15.8% and 16.1% respectively for case B. The memory energy reduction is marginally higher when 4 arrays are considered for interleaving, but there is a 2% performance over-

head as observed in Figure 4.18(b). The final decision can be taken based on whether the slight performance degradation is acceptable to the designer.

4.5.4 Execution Times

An alternative to the estimation-based exploration could be the exhaustive simulation of the application with the different interleaving possibilities explicitly implemented. However, a single simulation run of an application takes several hours for the benchmarks we considered. The simulation time varies from approximately 3 hours for SOR, to more than a day for larger applications such as LTE. With the very large number number of interleaving possibilities, an explicit evaluation of each possible data layout quickly becomes infeasible. On the other hand, our exploration strategy returns an energy efficient data layout in less than a second. Table 4.1 gives a comparison of the respective execution times of an exhaustive simulation of an application and our estimation-based exploration. The relative execution times validate the need for our proposed data layout transformation strategy.

Table 4.1: Comparison of execution times

Applications	**Simulation Time (approx.)**	**Exploration Time**
SOR	3h	0.01s
Channel Est.	5h	0.02s
Vector based image compression	6h	0.02s
Subband Coding	3h	0.01s
LTE MIMO Applications	>24h	0.03s

4.5.5 Other Applications of Array Interleaving

In terms of application classes where interleaving could help, the intuition is as follows. The class of applications that benefits most from the interleaving transformation is the one where the memory accesses have strides other than 1. This is a consequence of either

- The loop index co-efficient being greater than 1, e.g., a[3i+1], or
- The loop stride being greater than 1, e.g., $i = i + 4$;.

In both cases, memory accesses are not contiguous, and standard vector transfers fetch useless data into the registers. Array Interleaving overcomes the problem by ensuring that the vector fetched into registers always contains useful data elements.

The above characteristic is not really contrived in any way, and naturally occurs in several classes of real-life applications. Other than the wireless applications extensively covered, we

notice such patterns in a wide range of other domains where large arrays are used. A look at the Livermore Loops set of scientific applications shows that this pattern occurs in constituent components such as: Cholesky Conjugate Gradient and Banded Linear Equations. Numerical computing applications such as Gauss-Seidel Relaxation show such patterns. The image processing domain frequently exhibits such patterns in kernels such as the Debaucles 4-Coefficient Wavelet filter used in image compression.

4.6 Conclusion

We presented array interleaving, a data layout technique to improve memory access energy, for architectures consisting of SIMD FUs, wide interface SPMs, and vector registers. Efficient data layouts are important for such systems with wide memory interfaces, to help reduce memory access energy by reducing the memory access count. Our array interleaving technique helps reduce memory energy significantly by improving the data locality that helps in loading relevant data sets to the vector registers. The interleaving decisions for different possible sets of candidate arrays are taken based on cost estimates for the benefits of interleaving as well as the overhead involved in performing the layout changes. Our global analysis is aware of arrays being accessed in different ways in different loops, and generates interleaving decisions that are globally optimal.

We also considered the possibility of remapping of the arrays across the loops if the estimated energy reduction is greater than the overhead involved in the layout changes. Our exploration also considers the interleaving granularity as a parameter. Experimental evaluation on several applications in the wireless communication and image processing domain showed a significant reduction (6–34%) in memory energy consumption.

Chapter 5

Data Flow Transformation for QR Decomposition

Embedded applications in the communication and multimedia domain widely use matrix decomposition as a sub-routine. Matrix decomposition finds application in MIMO (Multiple Input and Multiple Output) detection systems in wireless communication standards [6, 7, 85] for implementing functionalities such as matrix triangularization and inversion. The specific choice of matrix decomposition algorithms depends on the type of matrix and accuracy requirement in the results. Techniques such as LU decomposition and Cholesky decomposition are applicable for square matrices only, while QRD is applicable additionally to rectangular matrices also. Matrix decomposition is also popularly used in the domain of numerical analysis for solving a set of simultaneous linear equations and for computing eigen values. This finds applications in extracting principal components in data analysis, obtaining approximate solutions in image processing and coding, in pattern recognition, and many other related areas. Exploration for energy efficient implementations of such a widely applicable functionality is an important area of research.

5.1 Introduction

QR Decomposition of a matrix involves the factorization of a $M \times N$ matrix A, where $M \geq N$, into an orthogonal matrix Q of size $M \times M$ and a $M \times N$ upper triangular matrix R, with all the elements in the bottom $(M-N)$ rows as zero. Thus, QRD can be expressed as

$$A_{M\times N} = Q_{M\times M} \times R_{M\times N}$$

Out of the several methods for computing QRD, Givens Rotation based QRD is usually preferred for its implementation efficiency [6] since it can be easily parallelized and also exploits

the advantage of multiple zero entries in sparse matrices.

Algorithm 3 Column wise Givens Rotation Based QR Decomposition

```
1: Input: Matrix A (M × N)
2: Output: Matrices Qt, R
3: for i_col = 1 to N do
4:    for i_row = M downto i_col + 1 do
5:       row_a = i_col; row_b = i_row;
6:       GR((row_a, i_col)( row_b, i_col)); /* Computing Givens Rotation matrix */
7:       /* updation of corresponding rows in matrix R and Qt */
8:       R([row_a row_b], :) = GR * R([row_a row_b], :);
9:       Qt([row_a row_b], :) = GR * Qt([row_a row_b], :);
10:   end for
11: end for
```

The column-wise Givens rotation based matrix decomposition strategy is outlined in Algorithm 3. The sequence of rotations is such that it transforms the input matrix A to an upper triangular matrix R by zeroing out elements in the lower half of the matrix column by column, starting from the first column. *row_a* and *row_b* correspond to the rows which have to be accessed to zero out element (*row_b, i_col*) of the input matrix. Givens rotation is computed using elements a (*row_a, i_col*) and b (*row_b, i_col*), in which element b is to be zeroed, as follows:

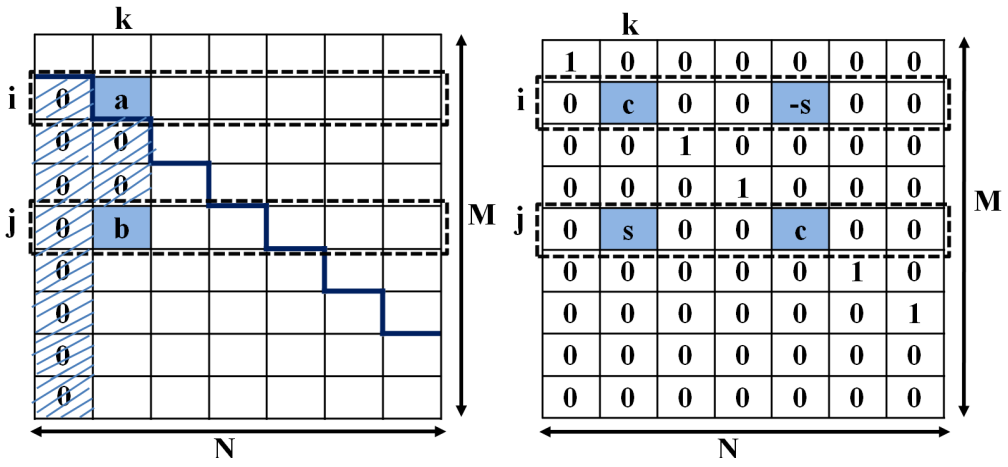
$$r = \sqrt{a^2 + b^2}, c = a/r, s = -b/r$$

such that

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

The corresponding Givens rotation (GR) matrix is shown in Figure 5.1(b). This is followed by updating of corresponding rows in R and Qt matrices. A sequence of such transformations on the input matrix A results in converting it to upper triangular matrix R. The same rotations are also performed on an identity matrix Q that has one in the diagonal matrix to begin with, and is eventually filled in. At the end, we obtain an upper triangular matrix R and transpose of the orthogonal matrix Q. Figure 5.2 shows the sequence of rotations for QRD of a 3×3 input matrix A. In the example, **a**, **b**, and **c** refer to elements of different rows. The first subscript on each element represents the column and the second subscript denotes the rotation sequence number generating the element.

Modern embedded system processor architectures include features such as vector register files, SIMD FUs (functional units with the ability to operate on multiple input data in parallel), and scratch pad memories with wide interfaces to the register file. Energy efficient implementation on such architectures requires intelligent use of wide memory accesses and vector FUs.

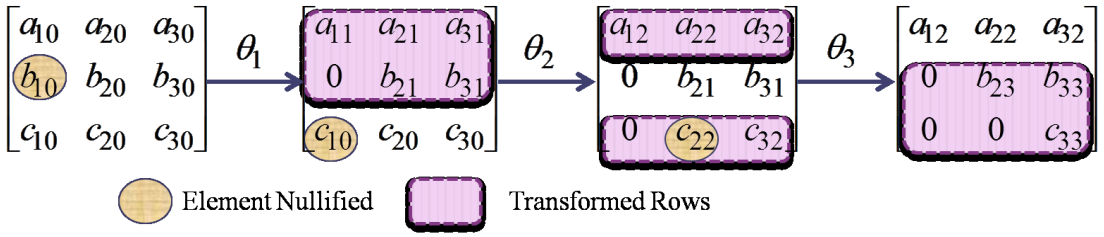


(a) Example matrix (b) GR matrix to nullify *b*

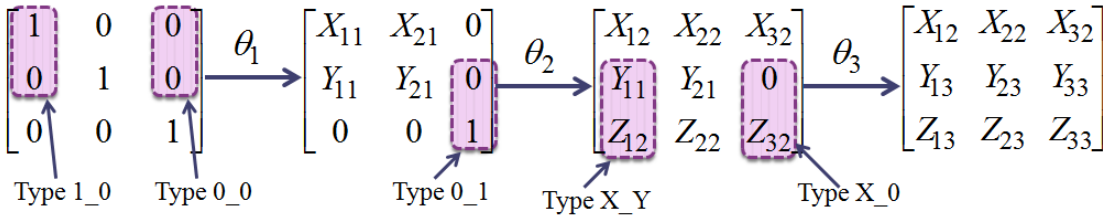
Figure 5.1: GR Matrix for a rotation in the example matrix

Since different Givens Rotation sequences lead to wide variations in the number of memory accesses, computations, and thereby the overall energy (discussed in Section 5.3), there is a need to find an energy efficient rotation sequence for such architectures. We propose an energy efficient data flow transformation for QRD in the context of SPM based SIMD architectures for embedded processors. We also explore different possible implementations using the SIMD feature of the processor. The results show our proposed strategy achieves a reduction of over 75% in memory access energy and consumes up to 36% less overall energy than the conventional sequences.

This chapter is structured as follows. We discuss the related research in Section 5.2. Section 5.3 discusses the motivation behind the work. In Section 5.4, we discuss our sequencing strategy. Different possible implementations using the SIMD feature of the architecture are presented in Section 5.5. Section 5.6 presents the supporting experimental results. Finally, we conclude with possible directions for future work in Section 5.7.



(a) Rotations on input matrix



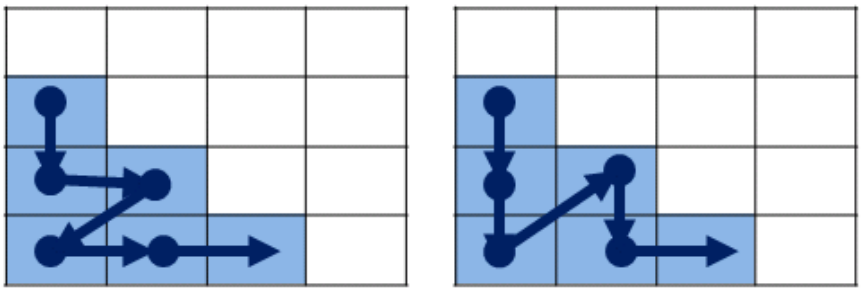
(b) Rotations on Q matrix

Figure 5.2: QRD with illustration for different types of operations

5.2 Related Work

The adoption of MIMO systems in several modern wireless communication standards such as WLAN, LTE, and WiMAX has led to increasing computation complexity at the receiver end. QRD is a popularly used technique in the MIMO-OFDM detection kernel for matrix triangularization and inversion because of its relatively higher reliability and lower computational complexity [6, 7, 85]. For MIMO receivers, QRD is a significant unit, as its frequency of execution is very high depending on the channel utilization. The number of matrices to be factorized increases with increasing number of sub-carriers. Power and energy constraints on the processors executing such systems make energy-efficient implementation of the matrix triangularization kernel (through QRD) an important point to be addressed for modern communication standards.

Previous research on QRD implementations has focused on hardware simplifications [86, 87, 88], computation complexity reduction by using real decomposition method [89], and algorithmic choices such as Gram Schmidt [90], Householder transformations, and Givens Rotation [6]. The latter algorithm is generally considered the most efficient from an implementation point of view [6]. Two standard orderings/sequences of Givens Rotations for obtaining an upper triangular matrix from an input matrix A are: *row elimination sequence* and *column elimination sequence*. Figure 5.3 shows the sequence of rotations for these conventional sequences. In both cases, the k^{th} row is used to nullify the element in the k^{th} column.



(a) Row Elimination Sequence (b) Col Elimination Sequence

Figure 5.3: Conventional Sequencing orders for a 4x4 matrix

For sparse matrices, George et al. [91] propose a methodology to determine good row orderings using a bipartite graph model for analyzing the number of Givens rotations and multiplication operations. This analysis is performed in the context of individual memory word accesses, and does not hold for our target architecture having register files with wide interface to SPMs and SIMD FUs. Architectures with the ability to perform operations on multiple pivots in parallel [92] or in pipelined fashion [93] have also been proposed to increase the throughput. None of these ASIC style implementations focuses on memory energy optimization which is very important in the context of our target architecture which has a processor with wide memory access interface.

Park et al. [94] evaluated different data optimizations on matrix manipulation kernels such as LU and Cholesky Decomposition, but not QRD. Using well known transformations such as tiling and blocking, a block size selection algorithm was proposed for efficient utilization of memory hierarchy. Tiling transformation, combined with an appropriate block size, resulted in performance improvements on different platforms.

Within the extensive literature on QRD, little research has focused on data memory energy optimization. Also, none of the works focuses on QRD implementation on scratch pad memory based SIMD architectures. These architectures do not have caches and have wide memory interface, and thereby, a large memory access energy. Hence, there is a need to efficiently utilize these memory accesses, which requires a careful ordering of the Givens rotations on a matrix. Our contributions in this chapter can be summarized as follows.

- We propose an energy efficient data flow transformation for Givens Rotation based matrix decomposition that reduces the memory accesses and thus the overall energy for the architecture under consideration.

- We present different implementations exploiting the SIMD feature of the architecture.

- Validation of the proposed strategy for different SIMD implementations on matrices of different sizes, and with variation in architectural parameters such as number of registers and SIMD width.

5.3 Operation and Memory Access Counts in Conventional QRD sequences

In this section we discuss how the ordering of the Givens rotations during matrix decomposition affects the number of memory accesses and computation cost, and hence, the overall energy. The number of Givens rotations to be applied on an $N \times N$ square matrix to obtain an upper triangular matrix R is:

$$\#rotations = \frac{N \times (N-1)}{2} \tag{5.1}$$

These rotations involve different types of operations due to distinct operand types (as shown in Figure 5.2(b)) and can be categorized as:

1. Type 0_0: when both the entries are zero.
2. Type 1_0 or 0_1: when one of the entries is one and other is zero.
3. Type X_0: when one of the entries is non-zero (with value other than 1) and the other is zero.
4. Type X_Y: when both the entries on which rotation is performed are non-zero.

Each operation type is associated with a different set of computations, as shown in Figure 5.4, and hence, different energies.

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} c.X - s.Y \\ s.X + c.Y \end{bmatrix}$$

(a) Operation Type X_Y (4 MULs + 2 ADDs)

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} X \\ 0 \end{bmatrix} = \begin{bmatrix} c.X \\ s.X \end{bmatrix}$$

(b) Operation Type X_0 (2 MULs)

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

(c) Operation Type 0_0

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -s \\ c \end{bmatrix}$$

(d) Operation Type 0_1

Figure 5.4: Compute complexity of different operations

To motivate the need for effective ordering of the rotations, we compare the effect of ordering on the overall memory accesses and compute the cost for the two conventional sequences – Row and Column elimination sequence. These memory access and operation counts are architecture dependent (e.g. a function of number and type (scalar or vector) of registers) and are also implementation dependent for vector architectures (details in Section 5.5). To make the motivational analysis independent of these parameters , we present the analysis for scalar architecture.

5.3.1 Analysis for the Q matrix

Considering the Q matrix, operation counts of Type 0_1 and X_0 are independent of the rotation sequence. Operations of Type 0_1 occur once for each row in the identity matrix. We have:

$$\#\text{Type } 0\_1 = N \tag{5.2}$$

Considering the column elimination sequence, the Type X_0 operations occur while processing elements in Columns 1 and 2 only. Applying rotations in Column 1, their number increases from 2 to (N - 1) as we move down the column with the increasing number of non-zero entries. In column 2, to begin with we have (N - 2) such operations which finally reduce to one as we move down the column nullifying the entries. We have:

$$\begin{aligned}\#\text{Type X\_0} &= \{2+3+\cdots+(N-1)\}+\{(N-2)+(N-3)+\cdots+1\} \\ &= \frac{N(N-1)}{2}-1+\frac{(N-2)(N-1)}{2} = N(N-2)\end{aligned} \tag{5.3}$$

This computation also holds true for the row elimination sequence. The Type 0_0 operation count depends on the elimination sequence. For the row elimination sequence, any rotation that turns the matrix entry *A(j, i)* to zero is associated with $(N-j)$ operations of this type. We have:

$$\#\text{Type } 0\_0_{row} = \sum_{j=2}^{N}\sum_{i=1}^{j-1}(N-j) = \frac{N^3-3N^2+2N}{6} \tag{5.4}$$

In the column sequence, Type 0_0 operations arise only while processing elements in Column 1. To begin with we have $(N-2)$ such operations, which finally reduce to one as we move down the column nullifying the entries. We have:

$$\#\text{Type } 0\_0_{col} = (N-2)+(N-3)+\cdots+1 = \frac{N^2-3N+2}{2} \tag{5.5}$$

Each rotation is associated with N operations, thereby making the total operation count

$= N \times \frac{N(N-1)}{2}$. Using the counts of Type 0_1, Type X_0, and Type 0_0 operations (Equations 5.2-5.5), the Type X_Y operation count can be computed.

Coming to the memory accesses associated with the Q matrix, all the entries for the two rows of the Q matrix ($= 2 \times 2 \times N$) are accessed (read and written) for each rotation. We have:

$$\#\text{mem\_acc}_Q = 2 \times 2N \times \frac{N(N-1)}{2} = (2N^3 - 2N^2) \tag{5.6}$$

5.3.2 Analysis for the Input matrix A

For the input matrix A, the computations are of Type X_Y and Type 0_0 only and are the same for both the sequences. The number of rotations in any column j is $(N-j)$ and each rotation is associated with $(N-j+1)$ operations of Type X_Y. We have:

$$\#\text{Type X\_Y}_A = \sum_{j=1}^{N-1} (N-j)(N-j+1) = \frac{N^3 - N}{3} \tag{5.7}$$

With the total operation count being $\left(= N \times \frac{N(N-1)}{2}\right)$, Type 0_0 count can be computed. The memory accesses for the input matrix A are computed as follows. For the column sequence, the accesses for elements in any column j is $(N-j+1)$ as there is one access to the pivot element (which is used to nullify other elements) and $(N-j)$ accesses for other non-zero elements. With $(N-j)$ rotations for any column j, each involving access to $(N-j)$ additional entries from the two rows under consideration, the total memory accesses can be computed as:

$$\begin{aligned} \#\text{mem\_acc}_{col} &= 2 \times \left\{ (N-j+1) + \sum_{j=1}^{N-1} (2(N-j)(N-j)) \right\} \\ &= \frac{4N^3 - 3N^2 + 5N - 6}{3} \end{aligned} \tag{5.8}$$

The factor of two accounts for the reads and writes corresponding to each element. In the row sequencing, the pivot element generally changes for each rotation. Each rotation in any column j involves operations on $(N-j+1)$ non-zero elements, since $(j-1)$ elements have already been nullified. Thus, the corresponding memory access count becomes:

$$\begin{aligned} \#\text{mem\_acc}_{row} &= 2 \times \left\{ \sum_{j=1}^{N-1} (2(N-j+1)(N-j)) - 1 \right\} \\ &= \frac{4N^3 - 4N - 6}{3} \end{aligned} \tag{5.9}$$

Table 5.1 summarizes the different operation and memory access counts for processing

QRD for both the conventional sequences. All the counts are in terms of scalar operations and individual memory word accesses. The variation in compute cost is due to the Type X_Y and Type 0_0 operation counts. The larger the Type X_Y count, the larger is the energy dissipation, and vice versa for Type 0_0 operation count. The memory access energy is directly proportional to the memory access counts. We observe that there is a trade-off between the memory accesses and computations – one sequencing strategy is better in one respect but worse in the other. Energy dissipation for the different categories are different and are architecture dependent; the final energy value depends on both counts. This analysis motivates us to determine an energy efficient data flow transformation for Givens rotation based QRD.

Operations	**Q Matrix**		**Input Matrix**	
	Row Seq.	**Col. Seq**	**Row Seq.**	**Col. Seq**
X_Y	$\frac{2N^3-6N^2+4N}{6}$	$\frac{N^3-4N^2+5N-2}{2}$	$\frac{N^3-N}{3}$	$\frac{N^3-N}{3}$
X_0	$N(N-2)$	$N(N-2)$	0	0
0_0	$\frac{N^3-3N^2+2N}{6}$	$\frac{N^2-3N+2}{2}$	$\frac{N^3-3N^2+2N}{6}$	$\frac{N^3-3N^2+2N}{6}$
0_1	N	N	0	0
mem_acc	$2N^3-2N^2$	$2N^3-2N^2$	$\frac{4N^3-4N-6}{3}$	$\frac{4N^3-3N^2+5N-6}{3}$

Table 5.1: Memory access and operation counts for conventional rotation sequences

5.4 Proposed Data Flow Transformation

For Givens Rotation based matrix decomposition of a $M \times N$ matrix, in column k, $(M-k)$ elements have to be nullified. Each rotation can be done by selecting a pair from any of the $(M-k+1)$ rows. The total number of possible orderings is given by the product of the possible choices for each column. Thus, there are an exponential number of possible rotation sequences with significant differences in number of memory accesses and computations (as discussed in Section 5.3). For a processor architecture with a memory interface of width W between the vector register file and SPM (W consecutive elements can be loaded into a vector register), and SIMD FUs, the individual memory accesses and vector computations are expensive from an energy viewpoint. Thus, there is a need for a careful sequencing of the rotations to minimize the number of accesses and vector computations.

5.4.1 Processing Sequence

Some pruning strategies need to be adopted to arrive at a solution in the exponentially large exploration space for the rotation sequence in matrix decomposition. One important pruning strategy is: to nullify the element $A(i,j)$ of the input matrix, the pivot row (row used to null

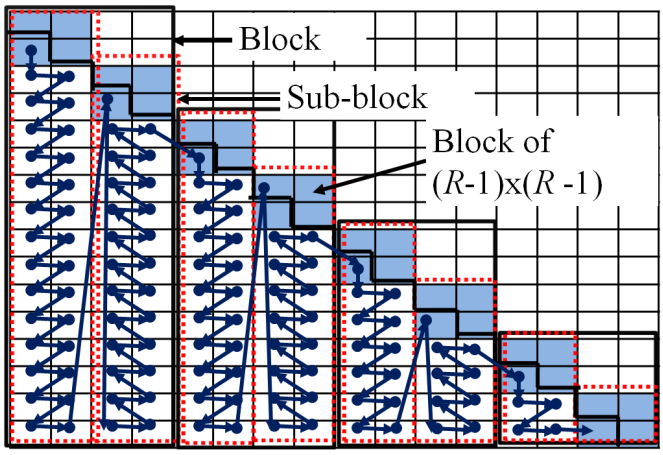
the element in another row) chosen should be such that all the elements $A(i,k)$ in the row with $k < j$ are already zero. Otherwise the already nullified entries will become non-zero. Another important requirement is to use the wide memory accesses judiciously. The memory access count depends on the number of available registers, and the rotation sequence should be such that these vector registers should be efficiently utilized so as to minimize memory accesses. Availability of $2R$ vector registers (of width W) allows storing W consecutive elements from not more than R rows each of the input and Q matrices in the registers. The memory access counts discussed below are the same for both Q and R matrices as the corresponding rows are considered together.

In our proposed ordering, the matrix is divided into blocks in such a way that $(R-1)$ rows accessed in a block are not re-loaded in other blocks. We assume that the vector register width W is greater than $R-1$ so that the elements of the row in the sub-block are accessed together. This assumption is reasonable as the number of vector registers in the register file is small. With vector register width W being 8 or higher, for the assumption to hold true, requires the register file size ($2R$) to be less than 18, which is already quite large. As the processing progresses from block i to block $(i+1)$, the block size reduces by a square sub-matrix of $2(R-1)$ rows from the previous block (as shown in Figure 5.5). Each block is further divided into two sub-blocks, each with its own uniform nulling sequences (Figure 5.5). For the first sub-block, the nulling sequence proceeds from top to bottom nulling $(R-1)$ columns in each row while in the second sub-block of any block i the traversal is from the bottom to top for $(N-2i(R-1))$ rows; this is done to re-use the last accessed row. At the beginning of each sub-block, $(R-1)$ rows are nullified at the desired positions for triangularization using column elimination sequence over the square block of size $(R-1)\times(R-1)$ (shaded squares in Figure 5.5). These are then used across the sub-block to nullify the entries in $(R-1)$ consecutive columns. This sequencing pattern is repeated over all blocks. Examples of the proposed rotation sequence for values of R as 3 and 5 are shown in Figure 5.5.

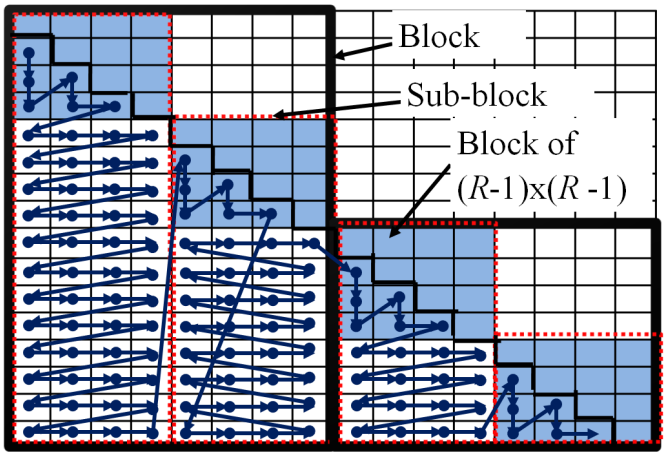
The memory accesses associated with the proposed optimized sequence ($\#mem\_acc_{opt}$) can be estimated as:

$$\begin{aligned}\#mem\_acc_{opt}(R) &= 2\left\lceil\frac{N}{W}\right\rceil\left(\sum_{i=1}^{b}(N-(i-1)(R-1))-(b-1)\right)\\ &\approx \left\lceil\frac{N}{W}\right\rceil\times\left(\frac{N^2}{R-1}+\frac{N(R-3)}{R-1}+2\right)\end{aligned} \tag{5.10}$$

where $b=\left\lfloor\frac{N}{R-1}\right\rfloor$ is the number of sub-blocks into which the proposed sequence divides the rotations for matrix decomposition. A factor of $\left\lceil\frac{N}{W}\right\rceil$ is included as there are N elements in a row out of which W elements are accessed in a single load. Across the blocks one memory



(a) $R = 3$



(b) $R = 5$

Figure 5.5: Proposed sequences for 16×16 matrix

access is re-used without any extra memory access, so a factor of $(b-1)$ is subtracted. A factor of two accounts for reads and writes corresponding to each access.

In the proposed sequencing, the rotation sequence changes with the number of registers, which leads to different computation counts for Q matrix. Operations of Type 0_0 arise only in the first block of $(R-1)$ register access. The subsequent rotations that are carried out using these over the remaining rows along with the rotation with $i = R$ and $j = (R+1)$ only contribute

to Type 0_0 count. The total count can thus be estimated as:

$$\begin{aligned}\#\text{Type } 0\_0_{opt}(R) &= \sum_{j=2}^{N} \sum_{i=1}^{min(j-1,R-1)} (N-j)+(N-(R+1)) \\ &= \frac{3N^2(R-1)+3N(1-R^2)+(R^3-R)}{6}\end{aligned} \tag{5.11}$$

With the total number of operations remaining the same at $N \times \frac{N(N-1)}{2}$, the increase in Type 0_0 count with the increasing number of registers causes a corresponding decrease in Type X_Y count (the sum of the two is constant). Type 0_1 and Type X_0 operation counts are same as that for conventional sequences.

5.4.2 Comparison with the conventional sequences

To compare the proposed strategy for defining the rotation sequence for matrix decomposition, we extend the discussion for conventional sequences in Section 5.3 to account for the number of registers. The rotation sequence for the conventional sequences is independent of the number of registers. Thus, the operation counts remain the same (Table 5.1) with the variation in memory accesses as follows.

- *Row Elimination Sequence* – The analysis for the memory accesses based on the number of registers (R) is as follows. Elements from R rows can be operated upon by loading R registers. For subsequent row entries to be zeroed in a row i, the number of memory accesses required is $(i-(R-2))$ for $i > R$. For others the data can be retained in the registers. Since for the rows $i = (R+1)$ and $i = N$ one repeated access is reduced so, we subtract a factor of 2. Thus,

$$\begin{aligned}\#mem\_access_{row}(R) &= 2 \times \left\lceil \frac{N}{W} \right\rceil \times \left\{ R + \sum_{i=R+1}^{N} (i-(R-2)) - 2 \right\} \\ &= \left\lceil \frac{N}{W} \right\rceil \times (N^2 + N(5-2R) + (R^2-3R-4))\end{aligned} \tag{5.12}$$

- *Column Elimination Sequence* – The number of memory accesses required to nullify the entries falling in the lower triangular matrix for any column i is $(2+(N-1-i))$. Since $(R-1)$ accesses can be retained for any column the remaining memory accesses required for each column becomes $((2+(N-1-i))-(R-1))$ which holds good until the expression is positive, beyond which the required data is already in registers and no further accesses are required. Hence, the upper bound of the summation is the maximum value of i that satisfies

$$(2 + (N - 1 - i)) - (R - 1) \geq 0$$
$$\Rightarrow i_{max} = (N + 2 - R)$$

Thus, the memory access count per matrix for column elimination sequence is:

$$\begin{aligned} \#mem\_access_{col}(R) &= 2 \times \left\lceil \frac{N}{W} \right\rceil \times \{(R-1) + \sum_{i=1}^{i_{max}} (2 + (N-i-1) - (R-1))\} \\ &= \left\lceil \frac{N}{W} \right\rceil \times (N^2 + N(3-2R) + (R^2 - R)) \end{aligned} \tag{5.13}$$

Comparing the memory accesses for different sequences discussed above, we conclude that $\#mem\_access_{opt} < \#mem\_access_{col} < \#mem\_access_{row}$. Thus, the proposed strategy is superior to both the conventional sequences in terms of memory accesses. However, it has an additional compute overhead with respect to row sequence, as the Type 0_0 operation count goes as: #Type 0_0_{col} < #Type 0_0_{opt} < #Type 0_0_{row} (from Equation 5.11 and Table 5.1) with reverse ordering for Type X_Y. This reduces the overall energy savings when computations are considered depending upon the implementations discussed in Section 5.5.

5.4.3 Comparison with the standard tiling transformation

Loop Tiling is a well known compiler optimization with the objective of improving cache performance by reducing cache misses. This in turn reduces the memory access energy. Reduction in cache misses is achieved by maximizing the usage of the data accessed in the cache. This requires a logical division of the loop iteration space into *blocks* or *tiles* where computations in the inner loops can be carried out without cache conflicts. Several techniques have been proposed in the literature for the optimal tile size selection.

We use a standard tiling approach that divides the matrix into square tiles of dimension $T \times T$ where T (the tile width) is a multiple of SIMD width W for the SIMD processor under consideration. For QR Decomposition the lower part of the matrix (shown shaded in Figure 5.6) is to be nullified. To estimate the memory access count for tiled loops, we categorize the tiles as:

- Complete tiles – tiles in which all elements have to be nullified;
- Diagonal tiles – tiles in which only the lower diagonal elements are to be reduced to zero as shown in Figure 5.6.

For an $N \times N$ matrix, the number of columns of tile width T is N/T with each column sub-divided into N/T tiles. In each column, there is one diagonal tile and $N/T - k$ complete

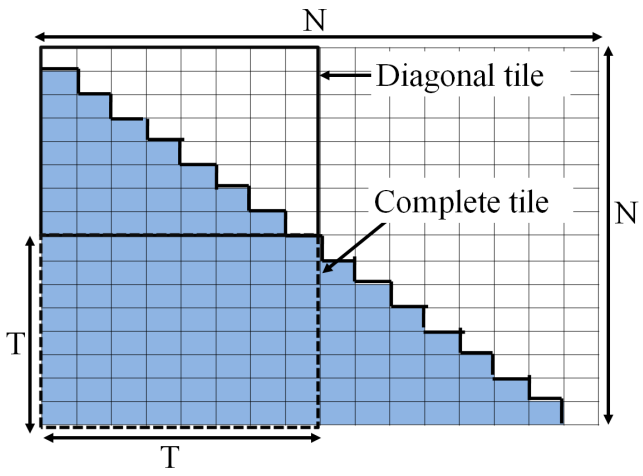


Figure 5.6: Tiling on 16×16 matrix

tiles, where k is the column number ranging from 1 to N/T as shown in Figure 5.6. Thus, the number of diagonal and complete tiles is given by:

$$\#diag\_tiles(N,T) = N/T \tag{5.14}$$

$$\#complete\_tiles(N,T) = \sum_{k=1}^{N/T} (N/T - k) \tag{5.15}$$

We use a register allocation technique where the reuse of data in the register is maximized. To nullify a column of a complete tile, one row of the diagonal tile is required. The remaining vector registers are used for accessing the tile. Within the tile $(R-2)$ rows can be retained for operations throughout all columns while $(T-(R-2))$ rows are sequentially accessed for each column. The memory access count corresponding to a complete tile thus becomes

$$acc\_complete\_tile(R,T) = 2 \times ((R-2) + (T \times (1+T-(R-2)))) \tag{5.16}$$

where a factor of 2 accounts for the reads and writes to memory. In the diagonal tile, $(R-2)$ rows in a tile can be retained for nullifying the entries in a column. The remaining rows $(T-(R-2)-(l-1))$ for a column l of a tile are sequentially accessed (the first $(l-1)$ rows are not required). Thus, the memory accesses per diagonal tile is:

$$acc\_diag\_tile(R,T) = 2 \times ((R-2) + \sum_{l=1}^{T-R} (T-(R-2)-(l-1))) \tag{5.17}$$

Since for each rotation the complete row for both input matrix and Q matrix is to be multiplied

by the rotation matrix, we multiply the memory access count for each tile by $2 \times \lceil N/W \rceil$ assuming N elements in a row of the matrix, of which W are accessed once. The total memory access count for a tiled $N \times N$ matrix, is estimated as:

$$\begin{aligned} &\#mem\_access_{tile}(N,R,T,W) = 2\lceil N/W \rceil \times (\#complete\_tiles(N,T) \times acc\_complete\_tile(R,T) \\ &\quad + \#diag\_tiles(N,T) \times acc\_diag\_tile(R,T)) \\ &\quad = \lceil N/W \rceil \times \left(N^2 \left(1 + \frac{(3-R)}{T} + \frac{(R-2)}{T^2} \right) - N \left(T - R + 4 + \frac{(2R-1)}{T} \right) \right) \end{aligned} \tag{5.18}$$

Given a tile size the memory access count reduces with increasing number of registers. The tiled implementation has lower number of accesses compared to the column and row elimination sequences without tiling. Comparing the memory accesses for different sequences discussed above, we conclude that:

$$\#mem\_access_{opt} < \#mem\_access_{tile} < \#mem\_access_{col} < \#mem\_access_{row}$$

The different operation counts for the tiling transformation are the same as that for the column sequence (Table 5.1). Thus, the proposed strategy outperforms the standard tiling transformation as well along with the conventional column and row elimination sequences.

5.5 SIMD Implementation

Exploiting the SIMD feature of our target architecture, we propose two possible ways of implementing QR decomposition – *Vectorization within a matrix* and *Vectorization across the matrices*. In this section, we discuss these implementation strategies and quantify the memory access and operation counts for the different sequences – conventional column and row elimination sequences, and our proposed sequence for these implementations.

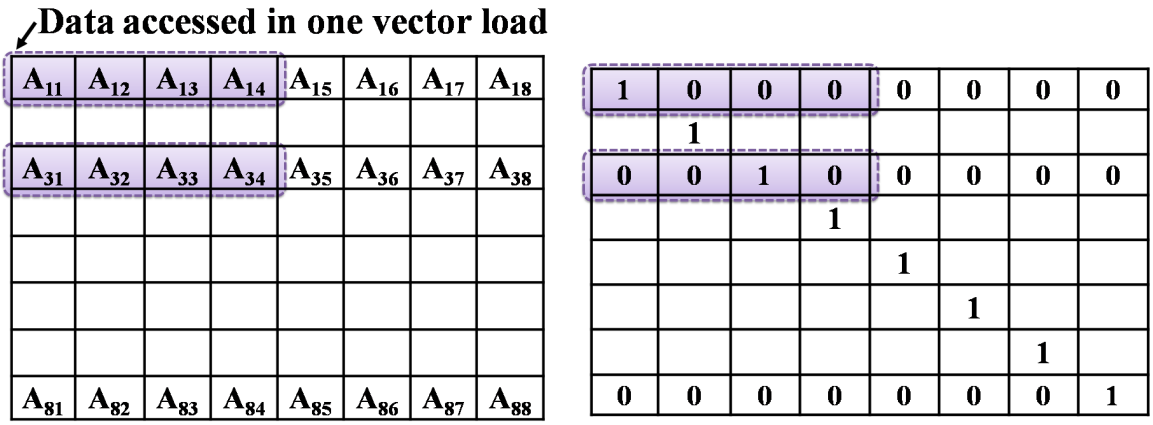
5.5.1 Vectorization within a matrix

Vector registers in the processor architecture are used for loading multiple elements from the rows of a matrix (as shown in Figure 5.7). In this case, the rotation operation can be applied simultaneously on all W elements of a row fetched together. This results in $\lceil \frac{N}{W} \rceil$ Type X_Y operations per rotation (as each row has at least one non-zero entry even in the input diagonal matrix). The distinction between different operation types (Type X_Y, etc.) is not made since the same operation type would have to be performed on all the W elements in a SIMD FU. The

computation count is thus given by:

$$\#computes_{total} = \left\lceil \frac{N}{W} \right\rceil \times \frac{N(N-1)}{2} \tag{5.19}$$

As a result, only reductions in memory accesses due to the ordering of rotation sequences (Equations 5.12, 5.13 and 5.10) contribute to the overall energy savings.



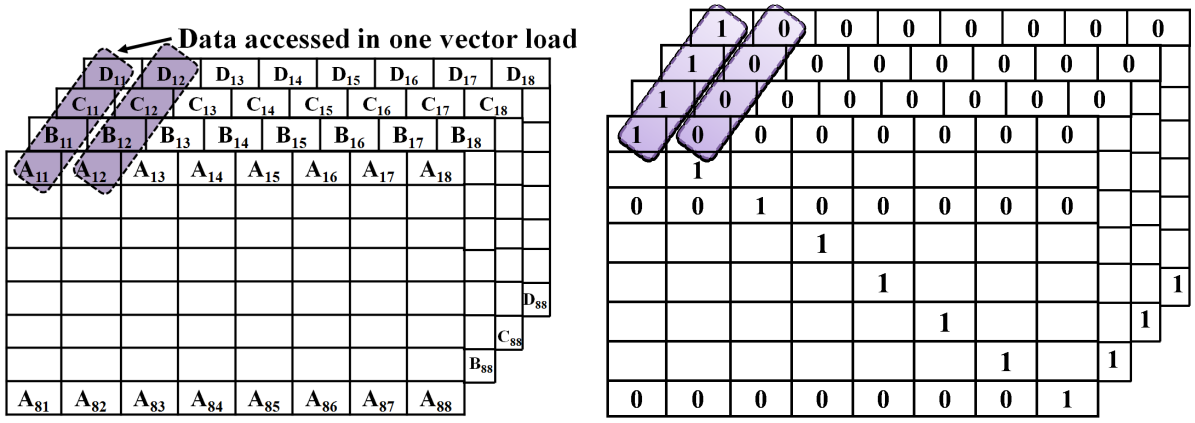
(a) Input matrix (b) Q matrix

Figure 5.7: SIMD within a matrix ($W = 4$)

5.5.2 Vectorization across matrices of same size

In this technique, the vector registers are used to access the elements across the matrices. Elements corresponding to the same position from W (SIMD width) matrices are accessed together (Figure 5.8). This access pattern is efficient while computing rotating matrices as a rotation for all W matrices can be computed in parallel through SIMD operation, since entries corresponding to any two rows (i and j) and k^{th} column for all the matrices are loaded together.

For this implementation, the different types of computations also play a significant role. The grouping of the matrices lead to the same operation type X_Y, X_0, etc., across the vector register (which is not the case when SIMD is employed within a matrix). Since the energy associated with SIMD operations of different types are significantly different, the variations in their count affect the overall energy.



(a) Input matrix (b) Q matrix

Figure 5.8: Exploring SIMD across multiple (W) matrices

5.5.3 Operation and Memory Access counts for different implementations

To quantify the energy savings achieved with different implementation strategies discussed above, consider C matrices of size $N \times N$ for which matrix decomposition is to be performed using conventional column elimination sequence.

- *Vectorization within a matrix* – For this implementation on C matrices, the memory access count is given by

$$\#mem\_acc_{within}(R) = C \times \#mem\_access_{seq}(R)$$

where, $\#mem\_access_{seq}(R)$ is the memory accesses associated with one matrix for any of the sequences (Equations 5.12, 5.13 and 5.10). The corresponding compute count (all of which are of Type X_Y) is

$$\#computes_{within} = C \times \left\lceil \frac{N}{W} \right\rceil \times \frac{N(N-1)}{2}$$

- *Vectorization across the matrices* – For this implementation exploiting SIMD across W matrices, the memory access counts can be obtained by multiplying the Equations 5.12, 5.13 and 5.10 by $\lceil C/W \rceil \times N$ instead of $\lceil N/W \rceil$, as only elements of W matrices out of C matrices can be operated upon in parallel and loads for all N elements in a row have to be performed.

  The corresponding operation counts can be estimated by multiplying the counts in Ta-

ble 5.1 and Equation 5.11 (for proposed sequence) by $\lceil C/W \rceil$ as one operation is now performed across C matrices. The compute count is thus given by

$$\#computes_{across} = \left\lceil \frac{C}{W} \right\rceil \times \{\text{Type X\_Y}_{col} + \text{Type X\_0}_{col} + \text{Type 0\_0}_{col} + \text{Type 0\_1}_{col}\}$$

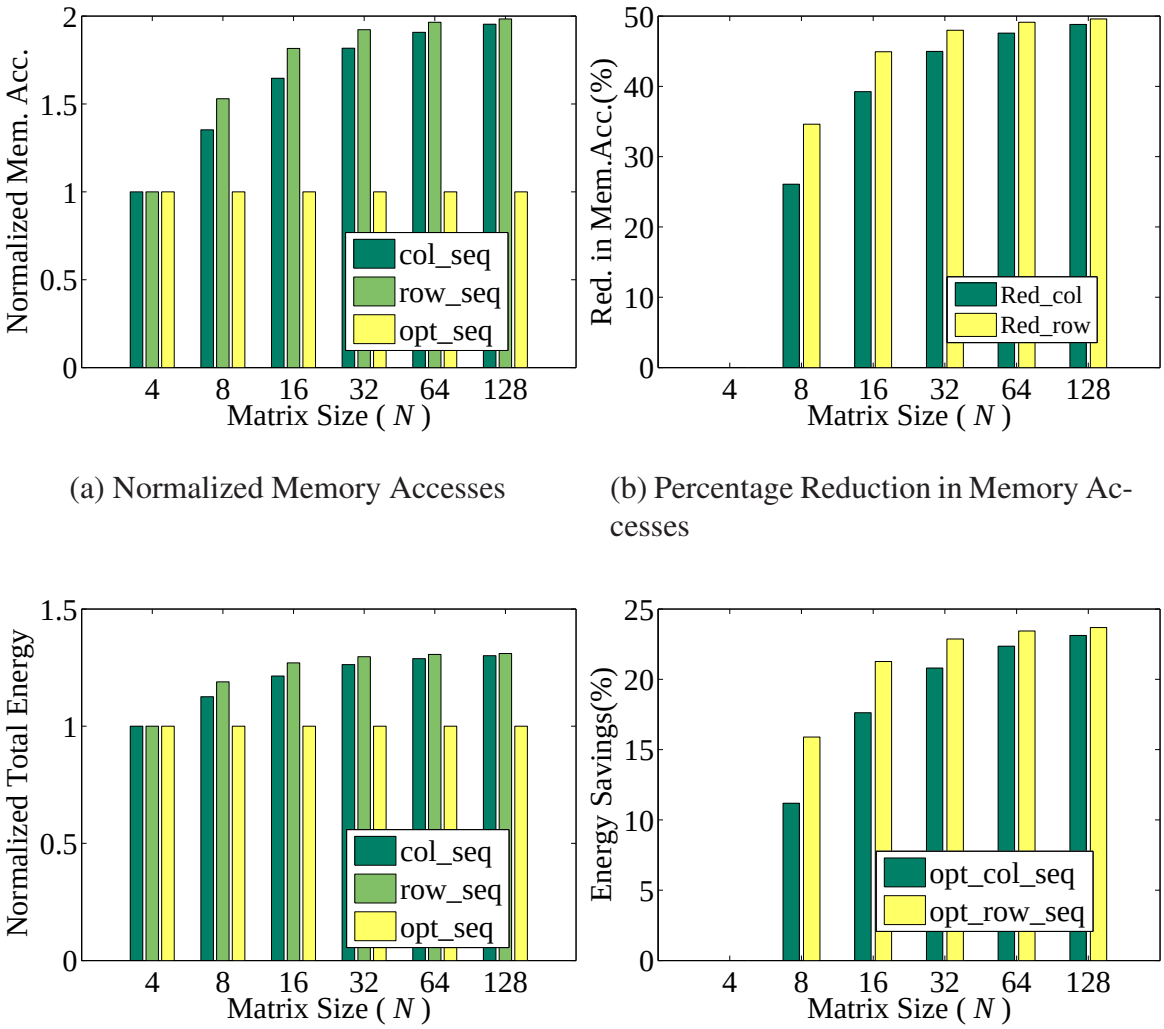
where Type 0_0_{col}, Type X_Y_{col} and Type X_0_{col} denotes the number of Type 0_0, Type X_Y and Type X_0 operations respectively for the column sequencing. Here, the compute counts other than Type X_Y help in reducing the overall energy. Section 5.6 validates the significant energy savings achieved with the proposed strategy for different test cases for different implementation strategies.

5.6 Experimental Results

We evaluated our proposed data flow transformation by comparing the energy dissipation against that of the conventional row and column elimination sequences. We implemented the algorithm using the framework discussed in Section 3.5. Exploiting the SIMD feature of the processor, we consider two different implementations discussed in Section 5.5. For each implementation strategy we compare the proposed sequencing against the conventional sequences.

- *SIMD in a matrix* – Figure 5.9(a) shows a comparison for the memory accesses normalized to the corresponding values for the proposed data flow transformation. Comparing the proposed data flow strategy with the conventional column sequence, the reduction in memory accesses varies from 26.08% for a small matrix of size 8×8 to 48.8% for N = 128. These percentages increase with increasing matrix size. The percentage improvement over the conventional row sequence is higher, with 34.61% for matrix with N = 8 and also increases with increasing N. Figure 5.9(c) compares the total energy for the different rotation sequences normalized against the energy for the proposed rotation sequence. The overall energy savings shown in Figure 5.9(d) are due to the differences in memory accesses only, with compute energy remaining same for all (as discussed in Section 5.5.1). For matrix sizes close to the number of available registers, different rotation sequences result in the same number of memory accesses (as seen for $N = 4$) thereby leading to no energy savings while considering SIMD within a matrix.

- *SIMD across matrices* – Here we consider cases when multiple matrices of the same size have to undergo QR Decomposition. In real time scenarios, such cases occur in MIMO detection kernels of wireless applications where matrix inversion is to be performed on multiple carriers together. The number of matrices that can be operated in parallel can



(a) Normalized Memory Accesses

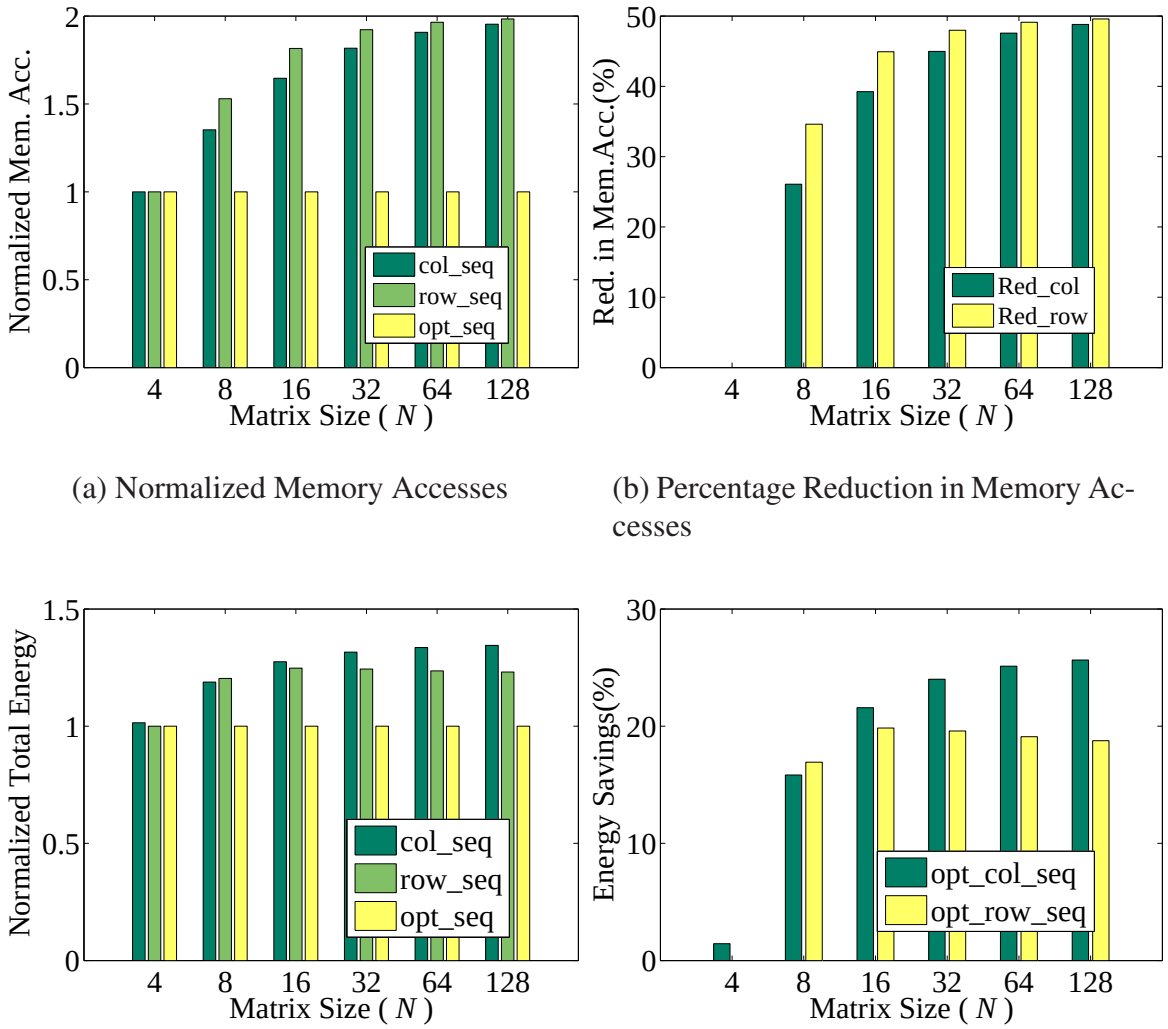
(b) Percentage Reduction in Memory Accesses

(c) Normalized Total Energy (Mem + Compute)

(d) Overall Energy Savings

Figure 5.9: Evaluation of the proposed strategy against the conventional sequences when SIMD is considered within a matrix (R = 3, W = 8)

be upto 1200 as is the case in the LTE standard with 20MHz channel bandwidth. For such cases, we explore how the energy savings can be improved while using the vector registers to access data across the matrices. Figure 5.10 illustrates the improvements obtained with the proposed strategy against the conventional sequences assuming that W matrices are operated together. Here the total energy for row elimination sequence is lower than that for the column sequence (vice-versa when compared to SIMD within a matrix) because of the reduction in compute energy due to greater number of Type 0_0 operations. It is only in the case when the different compute types are the same that the column elimination sequence is better than the row elimination sequence.



(a) Normalized Memory Accesses

(b) Percentage Reduction in Memory Accesses

(c) Normalized Total Energy (Mem + Compute)

(d) Overall Energy Savings

Figure 5.10: Evaluation of the proposed strategy against the conventional sequences when SIMD is considered across multiple (C) (here C=W) matrices (R = 3, W = 8)

Figure 5.11 shows the percentages of each type of operation when a particular sequence is followed. The lower the Type X_Y count and larger the Type 0_0 count, the smaller the total energy. The figure shows that row sequencing is better than the column sequencing in terms of compute energy. We also observe that X_Y count with the proposed data flow increases w.r.t. that for row sequence while there is corresponding decrease in the 0_0 operation type count. Operations of Type 0_1 and X_0 are same irrespective of the rotation sequence. The percentage reduction in memory access counts are similar to those with SIMD within matrix because the number of matrices under consideration is a multiple of W, thereby fully utilizing the vector accesses.

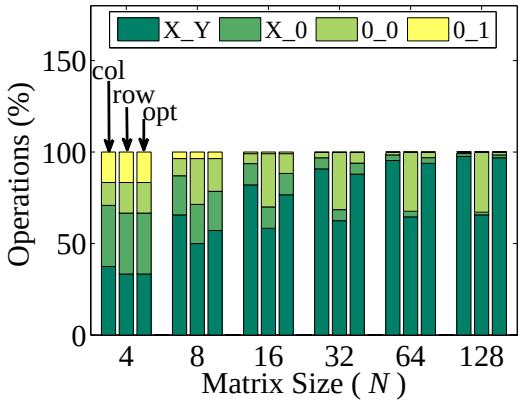


Figure 5.11: Fraction of different types of computations

Thus, with the proposed data flow transformation for QRD there is gain with respect to memory access energy and compute energy when compared with conventional column sequence. Comparison against the row sequence shows that the proposed data flow transformation results in larger gain with respect to memory accesses, though there is some compute energy overhead. However, overall energy savings are reported with respect to row sequence as well (upto 17.03% for $N = 128$) as shown in Figure 5.10(d). These gain percentages increase with increasing number of registers (shown later). A non-uniform variation is observed with increasing matrix sizes, as the gain achieved from reduced memory accesses is reduced because of the overheads associated with growing compute energy. Energy savings of 15.05% for $N = 8$ and 23.46% for $N = 128$ are achieved over the column sequence with the proposed strategy. Comparison of the proposed strategy against the column elimination sequence shows a constant increase in energy savings percentage as there are improvements both in terms of memory accesses and compute cost. An important question still remains. Which technique should be adopted under what circumstance? That is, when should SIMD be done across the matrices and when should the matrices be considered individually. We discuss this next.

When is vectorization across the matrices beneficial?

Figure 5.12 helps in taking the decision for the above matrix decomposition implementations. The plot shows a comparison of the total normalized energy (normalization with respect to average energy dissipation of a 32 bit adder) for the proposed data flow sequence for different number of matrices (C) of dimension 16×16 for SIMD factor $W = 8$. Since for the vectorization across the matrices the energy cost remains the same even if only a part of the vector register handles useful data, the total energy is independent

of the number of matrices. We conclude that SIMD across the matrices is better for the proposed sequencing strategy only if the complete vector register data and operations are useful. If C is not a multiple of W (say $C = k \times W + m$), then kW matrices should undergo matrix decomposition by vectorizing across matrices while the remaining m matrices can be factorized individually.

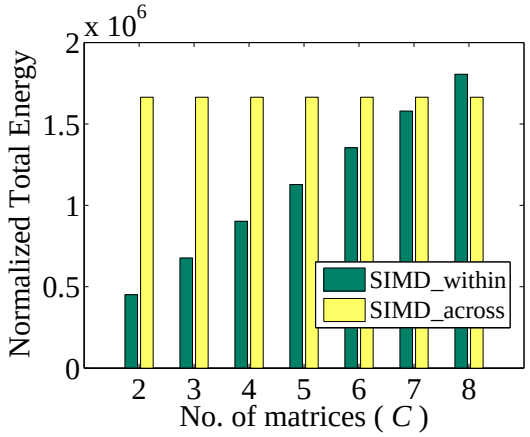
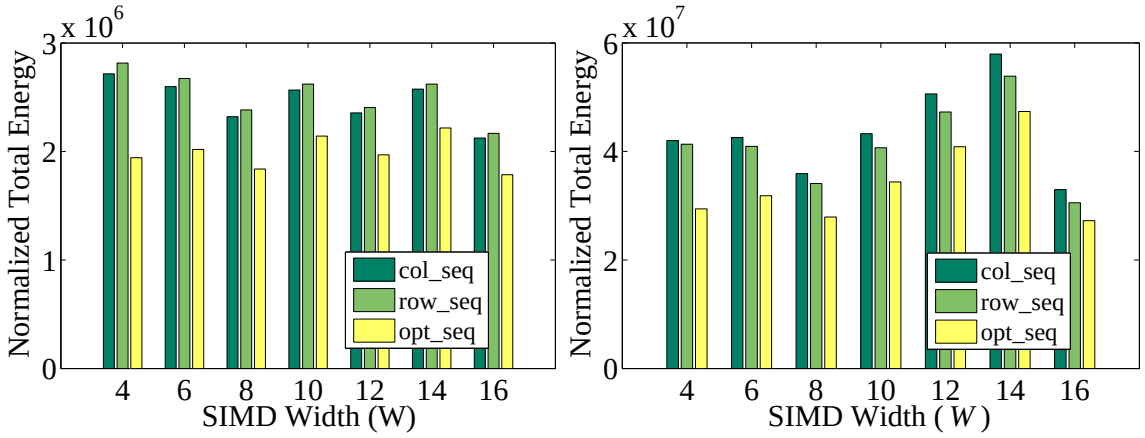


Figure 5.12: Energy variations for SIMD within and across the matrices for QRD of C matrices ($R = 3$, $W = 8$)



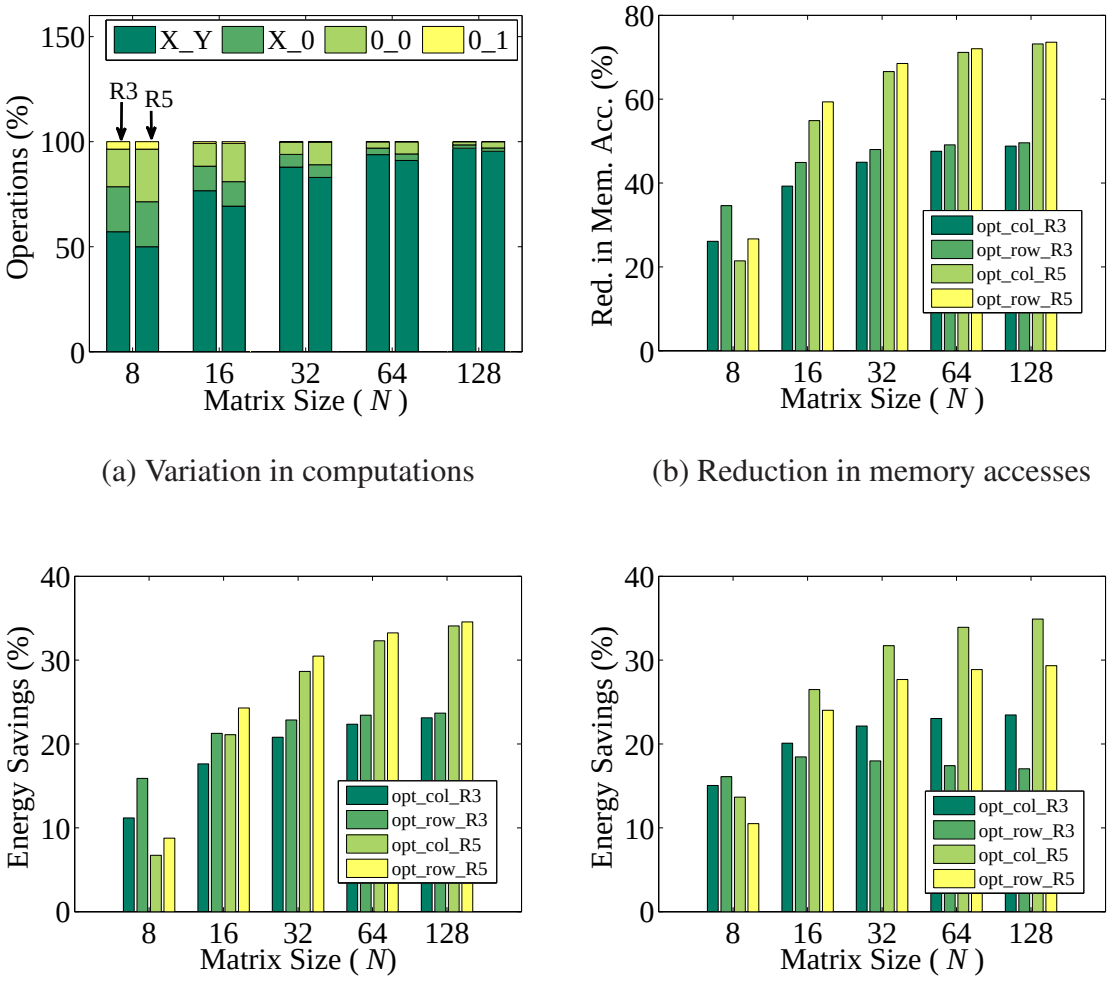
(a) SIMD within a matrix (b) SIMD across M=16 matrices

Figure 5.13: Energy variation with SIMD width on matrix of size 32×32 ($R = 3$)

- *Effect of SIMD Width* – The variation in energy savings with SIMD width is not uniform. For $N \times N$ matrices, widths W that are factors of N result in lower total energy than other widths due to efficient vector utilization (Figure 5.13(a)). With W ranging from 4 to 16 while considering SIMD across the W matrices of size 32×32, the energy savings with

our proposed strategy over the conventional column sequence varies from 30% to 17.5% and from 29% to 11% over the row sequence (Figure 5.13(b)).

- *Effect of number of registers* – On varying the number of registers, our proposed data flow sequence changes while the conventional sequences remain the same. This affects the computation counts for our proposed sequence. Different types of computations are affected differently. Computations of Type X_Y decrease; those of Type 0_0 increase; while those of types X_0 and 0_1 remain the same (Figure 5.14(a)). The percentage



(a) Variation in computations

(b) Reduction in memory accesses

(c) Energy savings for SIMD within a matrix for W matrices

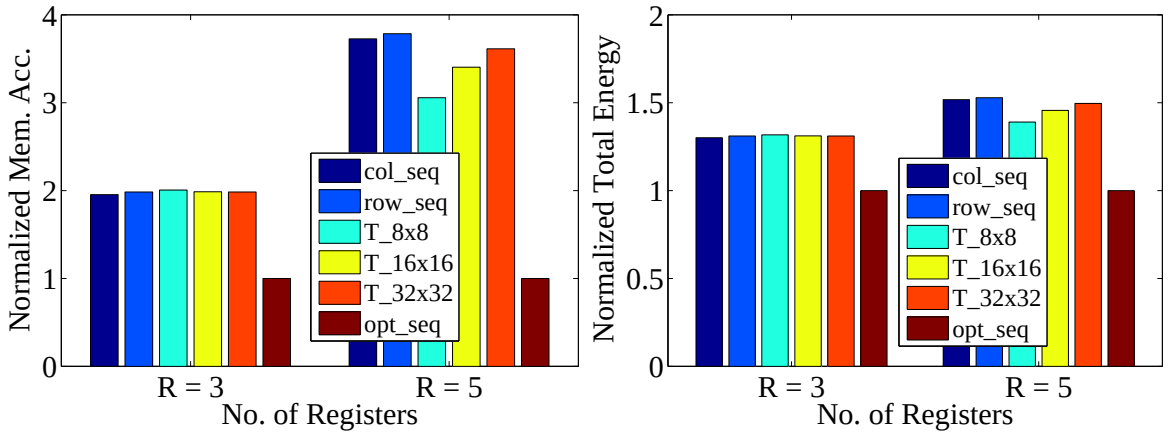
(d) Energy savings for SIMD across W matrices

Figure 5.14: Impact of varying the number of registers ($W = 8$)

reduction in memory accesses varies from 48.8% to 73.2% for $N = 128$ over column sequencing and from 49.5% to 74% over row sequencing when the number of registers is increased from 6 ($R = 3$) to 10 ($R = 5$) (Figure 5.14(b)). While considering SIMD within

a matrix, the overall energy savings for the transformed data flow increases on an average by 5.5% over column sequencing and 4.8% over row sequencing (Figure 5.14(c)). For SIMD across the matrices, these average savings show an increase of 7.4% over column sequencing and 6.7% over row sequencing (Figure 5.14(d)).

- *Tiling transformation* – Figure 5.15 shows a comparison of the different sequencing strategies with the tiling transformation (discussed in Section 5.4.3). Different tile sizes of 8×8 (T_8×8), 16×16 (T_16×16) and 32×32 (T_32×32) are considered for a 128×128 matrix. The variation in memory accesses and thus the total energy is a function of both tile sizes and number of registers. For a fixed tile size, the reduction in memory accesses with respect to conventional sequences increases with increasing number of registers. With 3 vector registers for a 128×128 matrix, the different tile sizes result in similar number of memory accesses. The optimized sequence is about 20% more energy efficient than the corresponding tiled implementations. For the implementation using 5 vector registers, the optimized sequence results in more than 67% reduction in memory accesses over the tiled implementation and is at least 29% more energy efficient than the tiled implementation.



(a) Normalized Memory Accesses

(b) Normalized Total Energy (Mem + Compute)

Figure 5.15: Comparison of the different sequencing strategies with tiling for matrix of size 128×128

5.7 Conclusion

The sequence of rotations plays an important role in determining the computation and memory access energy in the Givens Rotation based matrix decomposition algorithm. We presented a data flow transformation strategy for QRD targeting embedded processor systems with features such as SPMs, SIMD FUs, and vector register files with wide interfaces to both SPM and FUs. We also presented different implementations for QRD exploiting the SIMD feature of the architecture. We observed around 75% reduction in memory accesses with our proposed strategy over the conventional sequences. The effect of variation in computations on energy was observed when SIMD was implemented across the matrices. Experimental results show a significant reduction (up to 36%) in overall energy across different implementations with our proposed strategy. With increasing number of vector registers, the percentage reduction with respect to conventional sequences increases further. We also presented a comparison of the proposed sequencing strategy with the standard tiling transformation. Experimental results show that tiling results in an energy efficient implementation w.r.t. conventional sequences but still consumes higher energy consumption compared to our proposed sequence. Matrix decomposition being widely applicable in the communication and multimedia domain, the results show that our technique can make a significant difference in the energy efficiency of such applications.

Chapter 6

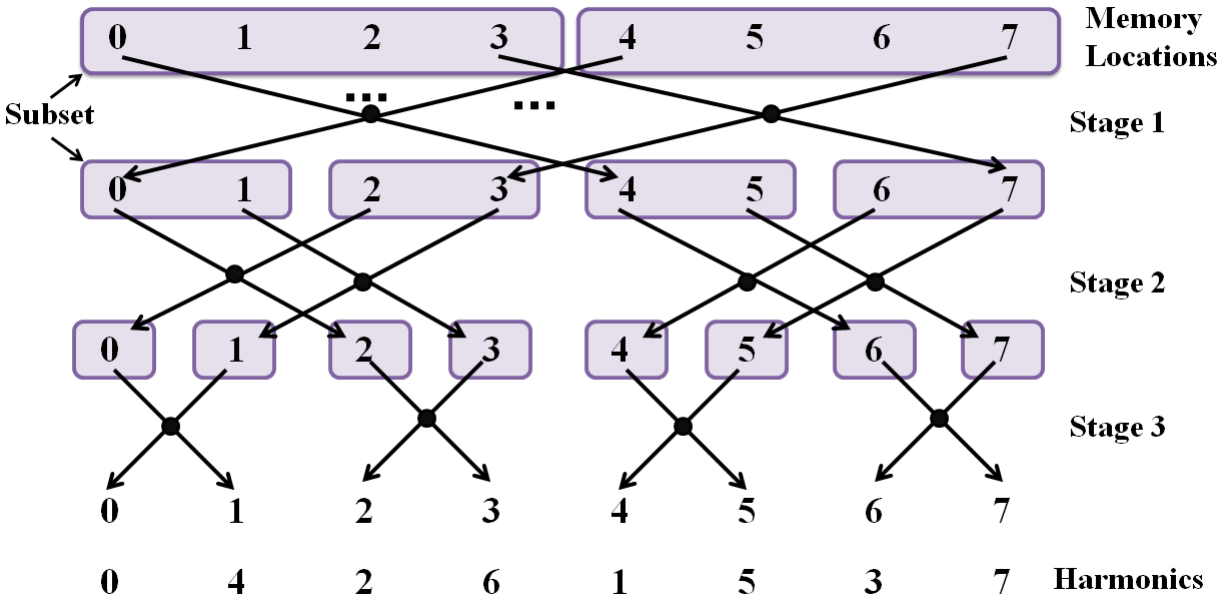
Register File Size Exploration for FFT

Fast Fourier Transform (FFT) is an efficient algorithm for Discrete Fourier Transform (DFT), a commonly used sub-routine in digital signal processing and image processing domains and for computing convolutions over large data points. It transforms the input samples in the time domain to frequency domain to reduce the compute complexity such as the multiplication operation in time domain is transformed to addition in frequency domain. The *radix-2 Cooley-Tukey algorithm* [8] is popularly used for FFT implementations. Being a memory intensive application (as discussed in Section 1.6), the FFT needs to be explored for an energy efficient implementation. In this chapter, we discuss how the overall energy in FFT processing can be reduced by an appropriate Register File (RF) size selection. This exploration is especially relevant when there are a large number of zeros at the input, which occurs very often in our application domain (Section 1.3).

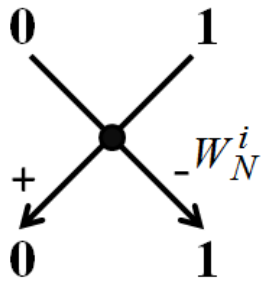
6.1 Introduction

The radix-2 Cooley-Tukey algorithm for an N-length FFT is implemented in log_2N stages. The most popular implementations for radix-2 FFTs are – Decimation-in-Frequency (DIF) and Decimation-in-Time (DIT), differing in the way the data points are available at the input and produced at the output. In DIF implementation, the inputs are available in natural order while the output is in bit reversed order; it is the other way around for DIT.

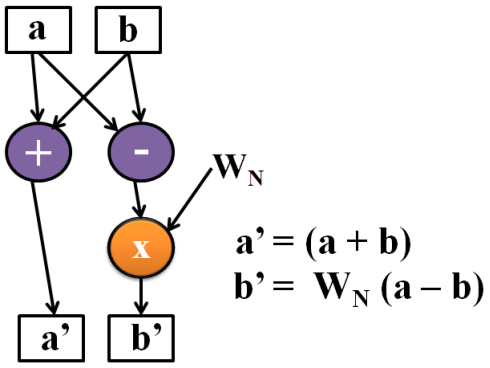
Figure 6.1 illustrates the DIF algorithm. The input at each stage k is divided into $\frac{N}{2^k}$ segments with k varying from 1 for the first stage to log_2N for the last stage. The butterfly operation is applied between the elements at offset $\frac{N}{2^k}$. With a and b as inputs to the butterfly unit (Figure 6.1(b)), we have $(a+b)$, $W_N \times (a-b)$ at the outputs. Hence, we have $\frac{N}{2}$ butterfly units at each stage with each butterfly operation involving two memory reads, two addition/subtraction operations, one multiplication and two memory writes. Thus, at every stage we have:



(a) Flow across the stages



(b) Butterfly unit



(c) DFG for Butterfly operation

Figure 6.1: DIF implementation for radix-2 FFT (with $N = 8$)

- Number of memory reads = N
- Number of memory writes = N
- Number of addition/subtractions = N
- Number of multiplications = $N/2$

On the basis of the number of registers available in the register file, certain stages can be merged by bringing in data from appropriate offsets. This does not affect the compute cost but may lead to reduced memory access count (depending upon the number of sample points) as

the read and write accesses would be across the merged stages. However, increasing the register file size also increases the access energy. Hence, there is a trade-off between the memory access count and the register file size, and thus a need for a strategy for an energy efficient RF size selection.

There are large number of cases in our application domain where the percentage of zero values at the input to FFT is very high. For example, it is around 55% in Chinese broadcasting standard [95] and 62.5% in 3GPP LTE [65]. We presented an illustrative example in Section 1.6 to bring out the relevance of the above exploration problem.

This chapter is organized as follows. Section 6.2 discusses the related research. Section 6.3 introduces the proposed exploration strategy for size selection of the register file comprising either scalar or vector registers. We present the experimental results justifying the need of our strategy in Section 6.4 and finally conclude in Section 6.5.

6.2 Related Work

In this section, we discuss some of the memory energy optimizations proposed by researchers for FFT processing. Chen and Sorensen [96] proposed Brunn's algorithm based strategy for computing DFT of real symmetric data. Using this strategy, if the input is real and symmetric the number of butterflies is reduced to one-fourth. The proposed algorithm has arithmetic complexity of an order similar to split-radix FFT [97, 98] but has a major drawback of the irregularity in output order which can be handled by maintaining a secondary memory unit.

Radhouane et al. [99] presented an algorithm for minimizing the memory requirement for radix-2 FFT using mixed-mode implementation. The approach involves switching the FFT processor between the Decimation-in-Frequency (DIF) and Decimation-in-Time (DIT) modes across the stages, and exploits the property that for DIF the input is in natural order and the output is in bit-reversed order, while DIT takes its input in bit-reversed form and produces the natural order output.

Several schemes to avoid access conflicts for different FFT implementations have been proposed, such as for radix-2 [100], radix-r [101, 102] and mixed-radix [103]. These schemes differ in the complexity of the address generation units. They involve architectural techniques such as the use of multiple banks, number of butterfly architectures, and so on. Memory interleaving is popularly applied to distribute data over multiple banks.

There are cases when the number of data points at the input or the number of useful data points at the output of an N-size FFT is smaller than N. Such cases cause a lot of redundant computations and memory accesses if full FFT is performed. Several pruning algorithms have been proposed by researchers [104, 105, 106, 107, 108] to remove the redundant data memory

accesses and operations involving FFTs on zero-valued inputs, on some undesirable outputs or both partial inputs and outputs. Jiang et al. [109] proposed an algorithm to reduce the twiddle factor accesses. The FFT structure is partitioned such that the computations of butterflies involving a particular twiddle factor are combined to the maximum extent. This reduces the redundant memory accesses of twiddle factors at every stage. None of these schemes addresses the effect or choice of RF size, which is the focus of this work.

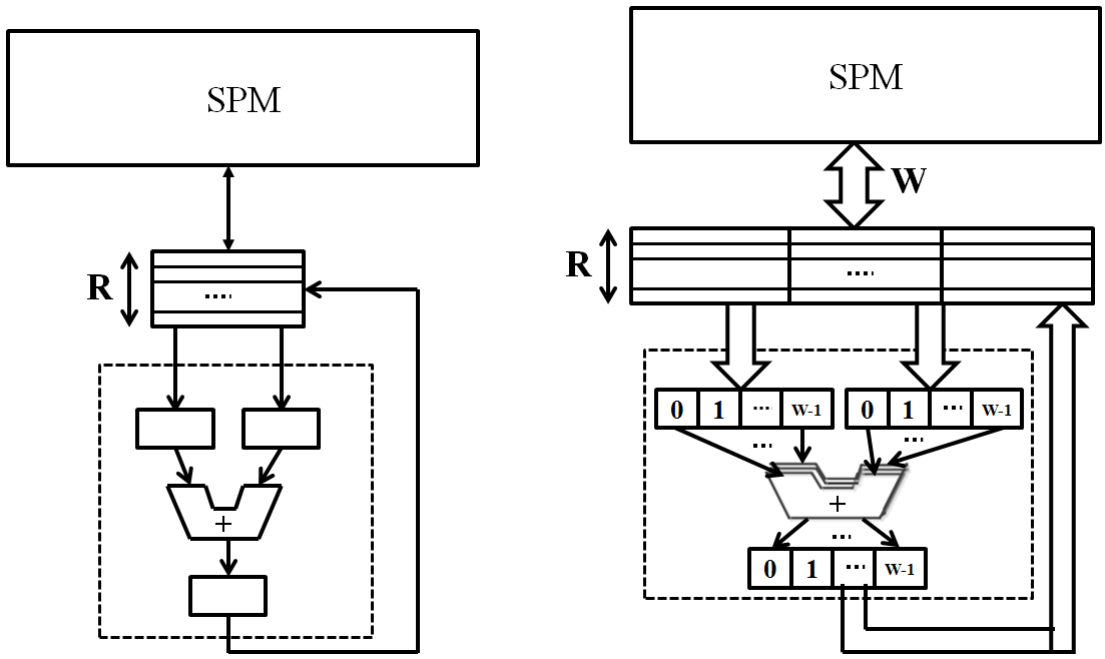
Brockmeyer et al. [110] explored the register re-use possibility over FFTs of smaller sizes. The exploration was carried out for a small number of registers and the conclusion drawn was that the percentage increase in gain decreases with increasing number of registers. An exploration for the energy and performance efficient foreground and background memory sizes for FFT as an example case has been presented [111]. These works do not account for the presence of zero-valued points at the input. Our exploration results show that the energy efficient RF size for FFTs varies over the FFT lengths and the percentage of non-zero values at the input because of the variation in the number of values to be stored. Further, the gain does not always increase with increasing number of registers. For the SIMD architecture under consideration, the RF size is a function of both non-zero values at the input and the SIMD width W. We present a detailed exploration considering the above architectural parameter variations.

6.3 Exploration Strategy

The RF size plays an important role in the overall energy consumption in processing FFT. As motivated in Section 1.6, the number of data points at the input plays an important role in the RF size selection. We propose a strategy based on access count and compute count estimates for deciding an energy optimal RF size for architectures supporting scalar or vector registers (Figure 6.2).

Figure 6.1 illustrates the FFT with $N = 8$. At a stage k, the elements are divided into $\frac{N}{2^k}$ groups. Each of these groups is called a *subset*. For N-length FFT, there are two subsets in Stage 1: $\left(0, 1, \cdots, \frac{N}{2}-1\right), \left(\frac{N}{2}, \frac{N}{2}+1, \cdots, N-1\right)$; each of which is equally sub-divided in Stage 2 and the subsets become: $\left(0, 1, \cdots, \frac{N}{4}-1\right), \left(\frac{N}{4}, \frac{N}{4}+1, \cdots, \frac{N}{2}-1\right), \left(\frac{N}{2}, \frac{N}{2}+1, \cdots, \frac{3N}{4}-1\right)$, and $\left(\frac{3N}{4}, \frac{3N}{4}+1, \cdots, N-1\right)$. This way the number of subsets on Stage 2 doubles, and continues to increase as we move to Stage 3, 4 and so on. The number of subsets at the input in stage k is given by $subsets_k = 2^k$.

The number of subsets at stage k is independent of N and grows by a factor of 2 for the radix-2 FFT implementation. The proposed strategy is applicable provided these non-zero values at the input of the FFT are in contiguous locations. For simplicity, we assume the non-zero inputs to be continuously placed from location 0 in our discussions.



(a) Scalar RF Architecture (b) Vector RF Architecture

Figure 6.2: Architectures with scalar and vector registers

6.3.1 Approach for non-SIMD scalar registers

We first estimate the number of non-zero elements at the output of every stage k for an N point FFT with $log_2 N$ stages and m non-zero points at the input ($m < N$). On moving from stage $(k-1)$ to k, the non-zero terms in a subset makes the corresponding terms in the subset on the other input of the butterfly unit non-zero. For the example shown in Figure 6.3, we have two subsets at the input: (0, 1, $\cdots$, 15) and (16, 17, $\cdots$, 31). With non-zero inputs from position 0 to 3, the butterfly operations (Figure 6.1(c)) in Stage 1, makes the terms at positions 16 to 19 non-zero. The number of non-zero terms across the stages grow by m for every subset pair input to the butterfly units in the previous stage. Thus the number of non-zero points at the output of stage k is given by $m.2^k$. Since the number of points at any stage cannot exceed N, the FFT length, we have the number of non-zero terms at stage k defined as:

$$stage\_terms(N,m,k) = min(N, m.2^k) \tag{6.1}$$

Thus, the effective memory read and write access counts at stage k are given by:

$$mem\_acc_{RD} = stage\_terms(N,m,k-1) \tag{6.2}$$

$$mem\_acc_{WR} = stage\_terms(N,m,k) \tag{6.3}$$

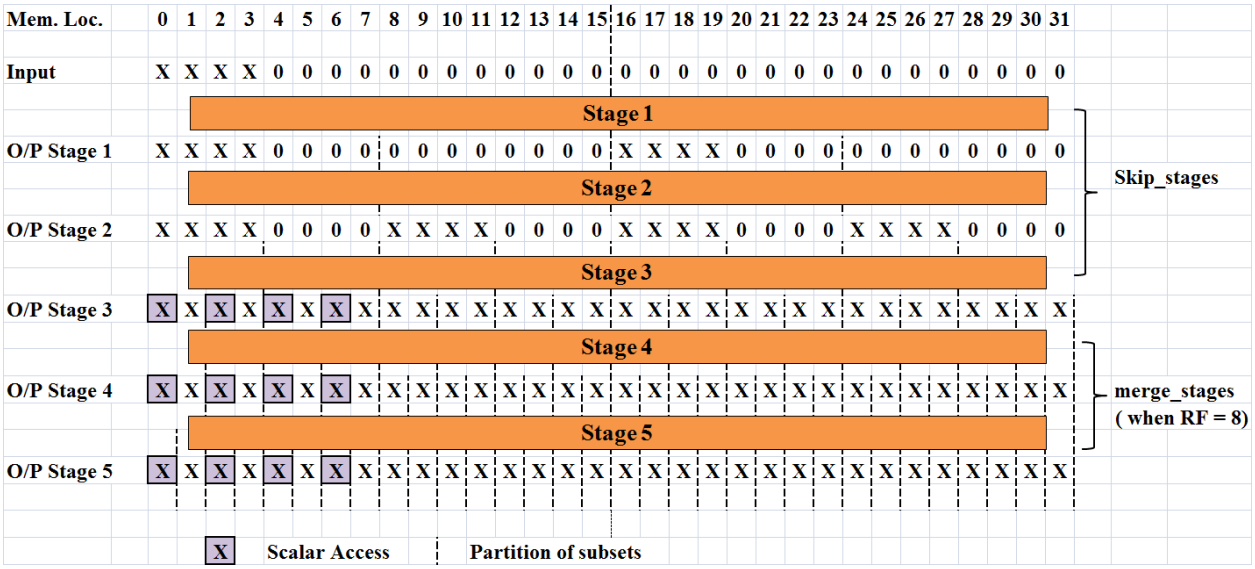


Figure 6.3: Illustration of a 32-point FFT with 4 non-zero terms (represented by X) at the input

The memory read access count at a stage is equal to the number of non-zero elements in the previous stage $(k-1)$, as each element is read once. Similarly, the number of memory write accesses at a stage is equal to the number of non-zero elements in that stage as each element is written once.

Considering the different computation counts at stage k, we have

$$compute_{ADD} = stage\_terms(N,m,k-1) \tag{6.4}$$

$$compute_{MULT} = \frac{stage\_terms(N,m,k-1)}{2} \tag{6.5}$$

since the number of additions/subtractions at a stage equals the number of non-zero inputs to that stage and the number of multiplications is half the number of terms because there is one MULT operation per butterfly unit (Figure 6.1(c)).

Given there are m non-zero elements at the input, computations up to some stages (*skip_stages*) involve only multiplication of the twiddle factors (W_N) with the non-zero inputs. Figure 6.4 illustrates the equivalent DFG. Since the output of the butterfly unit with a zero input, is a function of one input only, the inputs can be propagated until *skip_stages* as shown in Figure 6.5 for $N = 8$ and $m = 2$. *skip_stages* are the stages until which the number of non-zero elements in a subset $\leq$m and is determined as the minimum value of i satisfying

$$\frac{N}{2^i} \leq m \Rightarrow i = \left\lfloor log_2 \frac{N}{m} \right\rfloor$$

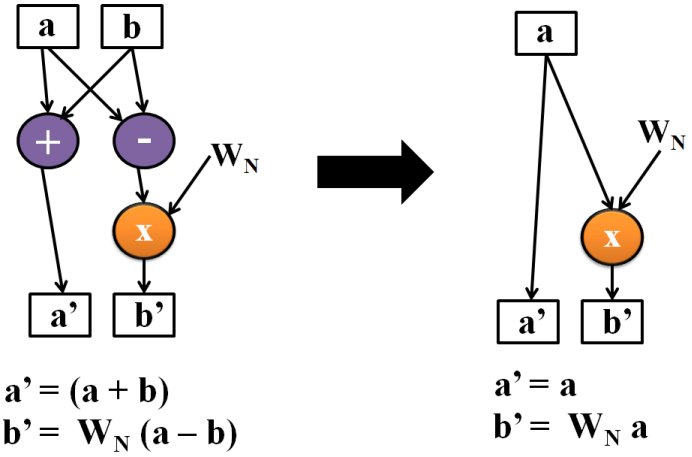


Figure 6.4: Transformed DFG of the butterfly unit (Figure 6.1(c) when $b = 0$)

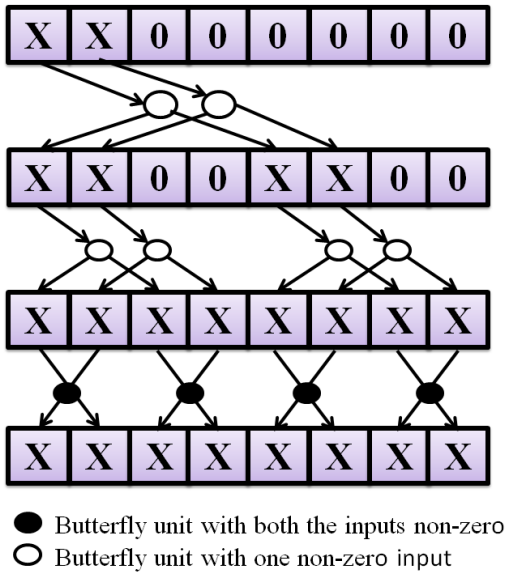


Figure 6.5: Illustration for *skip_stages*

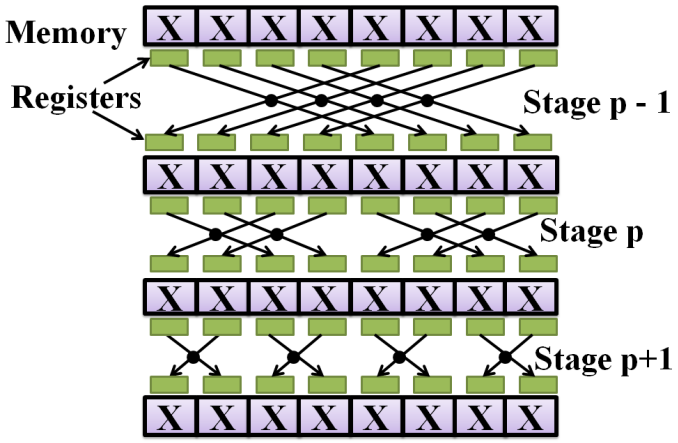
For the above example, *skip_stages* = $\lfloor log_2 \frac{8}{2} \rfloor = 2$. In the third stage, there are 8 subsets with one non-zero element in each (which is less than m), and we have butterfly units with both inputs as non-zero. Thus, computations have to be performed and inputs cannot be simply propagated further.

We now discuss the effect of RF size on the memory access counts and compute counts. Assuming R registers are available at the input of a stage p, memory accesses for a certain number of following stages (*merge_stages*) can be skipped. Figure 6.6(b) shows this merging for 2 stages. This is possible by accessing elements from offsets $\frac{N}{2^p}, \frac{N}{2^{(p+1)}}, \cdots, \frac{N}{2^{(p+n)}}$, where n is given by:

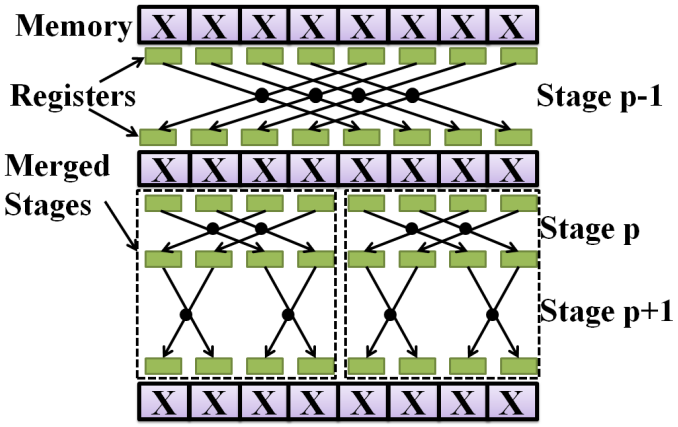
$$2^n \leq R$$

$$\Rightarrow n = \lfloor log_2 R \rfloor$$

as at every stage k, data from offsets ($\frac{N}{2^k}$) is required in two registers. With this data in the RF, n stages can be executed without any intermediate accesses. Green rectangles in the figure indicate register accesses. The skipped row of X's corresponds to skipped memory accesses because data is accessed only from the registers. This process of loading elements at a stage and writing back after n stages continues until ($(p+n) \leq log_2 N$) the last stage is reached.



(a) Access pattern without stage merging



(b) Access pattern with merged stages

Figure 6.6: Illustration for merging stages with $RF = 8 (\Rightarrow R = 4)$

The total number of computations equals the sum of the computations across all stages and is independent of the RF size. The compute count at each stage is estimated by the number of non-zero elements at the inputs of that stage using Equations 6.4 and 6.5. The overall compute counts and memory access counts for N size FFT as a function of R, m and N are estimated as

follows.

Compute counts – Total number of computations, which depend only on the number of terms at the input of each stage, is given by (using Equations 6.4 and 6.5):

$$tot\_compute_{ADD}(N,m,k) = \sum_{k=\lfloor log_2 \frac{N}{m} \rfloor}^{log_2 N-1} stage\_terms(N,m,k) \tag{6.6}$$

$$tot\_compute_{MULT}(N,m,k) = \sum_{k=0}^{log_2 N-1} \frac{stage\_terms(N,m,k)}{2} \tag{6.7}$$

where $tot\_compute_{ADD}$ and $tot\_compute_{MULT}$ represent the total number of additions and multiplications involved in the overall FFT process. For addition, the lower limit of k is $\lfloor log_2 \frac{N}{m} \rfloor$, as up to this stage we have only multiplications due to the presence of a zero input in each butterfly operation (from the simplified DFG of Figure 6.4), as discussed earlier. With the last stage being $log_2 N$, the inputs are the outputs of stage $(log_2 N - 1)$.

Memory Access counts – As discussed earlier, memory read accesses occur at the input, then at stage I ($= \lfloor log_2 \frac{N}{m} \rfloor$), followed by accesses after every $log_2 R$ (=*merge_stages*) stages. Thus,

$$\begin{aligned} tot\_mem\_acc_{RD} = {} & m + stage\_terms(N,m,I) + stage\_terms(N,m,I+log_2 R) \\ & + stage\_terms(N,m,I+2log_2 R) + \cdots + stage\_terms(N,m,I+(t_{max}-1)log_2 R) \end{aligned}$$

$$tot\_mem\_acc_{RD}(N,m,R) = m + \sum_{t=1}^{t_{max}} stage\_terms(N,m,I+(t-1)log_2 R) \tag{6.8}$$

where t_{max} is such that

$$I + (t_{max}-1)log_2 R < log_2 N$$

$$\Rightarrow t_{max} = \left\lfloor \frac{log_2 N + log_2 R - I}{log_2 R} \right\rfloor$$

Similarly, for memory writes we have

$$\begin{aligned} tot\_mem\_acc_{WR} = {} & stage\_terms(N,m,I) + stage\_terms(N,m,I+log_2 R) + stage\_terms(N,m,I+2log_2 R) \\ & + \cdots + stage\_terms(N,m,I+(t_{max}-1)log_2 R) + stage\_terms(N,m,log_2 N) \end{aligned}$$

as the data is first written back to memory at stage I followed by offsets of $log_2 R$ and then in

the final stage. Thus,

$$tot\_mem\_acc_{WR}(N,m,R) = \left[\sum_{t=1}^{t_{max}} stage\_terms(N,m,I+(t-1)log_2R)\right] + stage\_terms(N,m,log_2N) \tag{6.9}$$

The overall energy consumption for FFT processing on a scalar architecture is estimated by multiplying the counts derived above by their relative weights and adding them. Thus we obtain the normalized energy estimates:

$$\begin{aligned} Energy\_est_{scalar} = tot\_mem\_acc_{WR} \times W_{Mem\_WR} + tot\_mem\_acc_{RD} \times W_{Mem\_RD} + \\ tot\_compute_{MULT} \times W_{MULT} + tot\_compute_{ADD} \times W_{ADD} \end{aligned} \tag{6.10}$$

where W_{op} represents the relative weight of operation op and our objective is to find R for a particular choice of (N,m) that leads to minimum energy consumption.

For the example shown in Figure 6.3, we estimate the corresponding memory access counts and the energy efficient size. Let us assume that we have Register File (RF) sizes: 4, 8, and 16. For the scalar registers, the number of memory reads and memory writes at any stage is equal to the number of non-zero points in that stage. With the number of non-zero inputs (m) as 4 in the 32 point (N) FFT, we can propagate the inputs (for *skip_stages*) until Stage 3 $\left(= log_2(\frac{32}{4})\right)$. The number of *skip_stages* is independent of the RF size.

Up to Stage 3, the memory access counts are the same for all the RF sizes and are given by

$$\#mem_{RD} = \#\text{non-zero inputs} = 4$$

$$\#mem_{WR} = \#\text{non-zero points at output of Stage 3} = 32$$

The RF size makes a difference in subsequent stages. We can merge the computation of *merge_stages* $\left(= log_2(\frac{RF}{2})\right)$, assuming equal number of registers are available for inputs and outputs. We have:

- For $RF = 4$, $merge\_stages = 1$, and the memory read and write accesses are done for each of the Stages 4 and 5.

$$\#mem_{RD} = \#\text{non-zero inputs at stage 4 and 5} = 32+32 = 64$$

$$\#mem_{WR} = \#\text{non-zero points at output of stage 4 and 5} = 32+32 = 64$$

Thus, the total number of memory accesses for $RF = 4$ is given by:

$$\#mem_{RD} = 68, \#mem_{WR} = 96$$

- For $RF = 8$, $merge\_stages = 2$, and thus the computations for Stages 4 and 5 can be merged. So for the data brought into the RF at the input of Stage 4, computations corresponding to both the stages will be done and then written back to memory. This reduces the memory accesses across the stages and we have

$$\#mem_{RD} = \text{\#non-zero inputs at Stage } 4 = 32$$

$$\#mem_{WR} = \text{\#non-zero points at output of Stage } 5 = 32$$

  Thus, the total number of memory accesses for $RF = 8$ is given by:

$$\#mem_{RD} = 36, \#mem_{WR} = 64$$

- Similarly, for $RF = 16$ we have $merge\_stages = 3$. Since there are at most 5 stages for 32 point FFT, for the last 2 stages it has memory access counts same as that for $RF = 8$. Thus, the total number of memory accesses for $RF = 16$ is given by:

$$\#mem_{RD} = 36, \#mem_{WR} = 64$$

Since the rate at which memory access energy changes with growing RF size is much smaller than the corresponding change in memory access count (Section 1.6) we conclude that

$$Energy_{RF=8} < Energy_{RF=16} < Energy_{RF=4}$$

Thus, RF size of 8 is the most energy efficient for our example.

6.3.2 Approach for SIMD vector registers

An approach similar to that for scalar registers is followed for vector registers with some variations due to the SIMD feature. With vector registers of width W, W consecutive elements can be loaded in the register as shown in Figure 6.7 for $W = 4$. In FFT, the operations are performed across the subsets as shown in Figure 6.1. For SIMD operations, it is required to have elements from these sets in separate registers. Hence, we estimate the number of memory accesses per subset. For any stage k, with 2^k subsets, the number of elements to be read in a subset is $min\left(m, \frac{N}{2^k}\right)$ as the number of elements in a subset cannot exceed $\frac{N}{2^k}$. Assuming W is the vector register width, the number of memory accesses for a subset in the k^{th} stage is given by:

$$subset\_acc(N,m,k,W) = \left\lceil \frac{min(m, \frac{N}{2^k})}{W} \right\rceil \tag{6.11}$$

We use the ceiling operator so that if the number of non-zero points is not a multiple of SIMD width W, then the additional load fetches the remaining values. We do not exploit the zeroes within one SIMD operation on a single word when all elements are not zero. Thus, the memory access count at stage k is given by:

$$\begin{aligned} stage\_acc(N,m,k,W) &= subsets_k \times subset\_acc(N,m,k,W) \\ &= 2^k \times \left\lceil \frac{min(m, \frac{N}{2^k})}{W} \right\rceil \end{aligned} \tag{6.12}$$

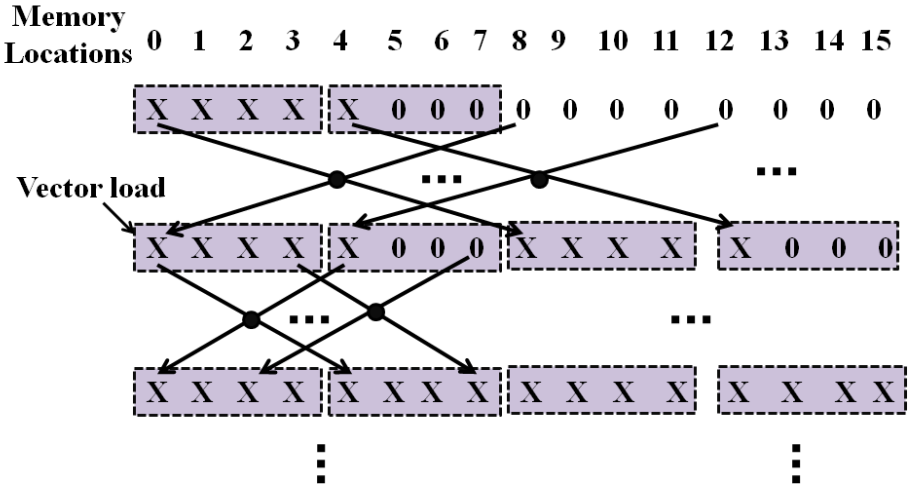


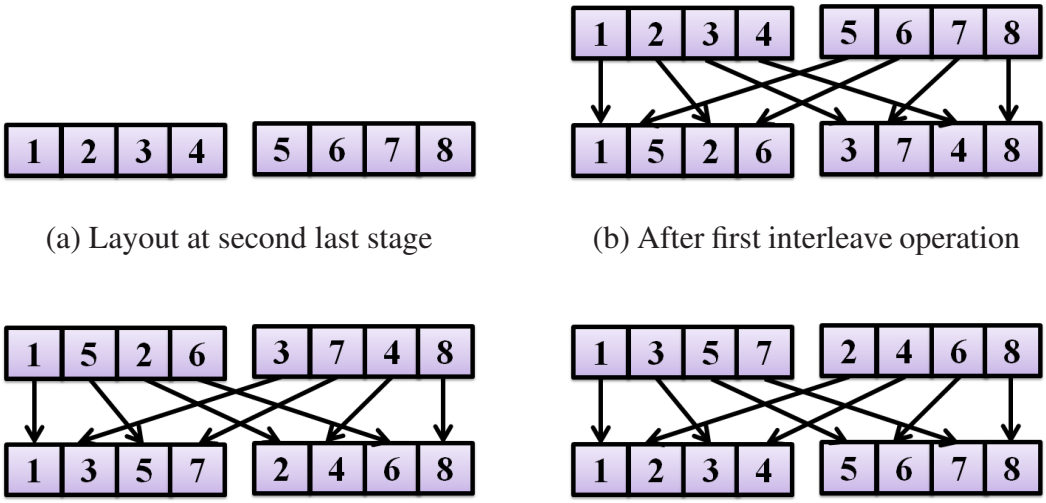
Figure 6.7: Illustration of vector loads for two stages of 16-point FFT ($W = 4$)

Figure 6.7 shows the vector access pattern for two stages of a 16 point FFT. The memory access and compute counts can be estimated using Equations 6.2 – 6.5 with $terms_k$ being replaced by $stage\_acc$ for vector architectures. $stage\_acc$ equals $terms_k$ for scalar architectures ($W = 1$).

Another difference arises when for the last $log_2 W$ stages the elements are in the vector register of width W itself. Now the butterfly operation is to be performed on the elements within the vector register. However the SIMD FUs (Figure 2.9) require operands to be in different registers. To separate them out we use interleave operations (Figure 3.10). For example, the data is in the order as shown in Figure 6.8(a) in the second last stage for $W = 4$. The butterfly operation is performed between elements 1 and 3, 2 and 4, 5 and 7 and so on. Interleaving them results in the layout as shown in Figure 6.8(b). This separates them out into different registers and now SIMD FUs can be used to operate upon them in parallel. For the last stage the butterfly is operated upon the consecutive elements 1 and 2, 3 and 4 and so on. Another interleave operation is carried out to obtain the layout as shown in Figure 6.8(c). After the last stage, to have elements in sequence (which is actually the bit-reversed order output in DIF implementation) we interleave again and obtain the layout as shown in Figure 6.8(d). The number of interleave operations at any stage is equal to the memory access count at that stage, as each interleave

operation requires two registers and two operations, *interleave low* and *interleave high*, per pair of registers. The interleave operation count at stage k is

$$\text{interleave\_ops}(N,m,k,W) = 2 \times \frac{stage\_acc(N,m,k,W)}{2} \tag{6.13}$$



(a) Layout at second last stage

(b) After first interleave operation

(c) Interleave for last stage computation

(d) Interleave before writing back to memory

Figure 6.8: Data re-arrangement for last $log_2 W$ stages while using vector registers with $W = 4$

Each interleave operation is associated with an energy overhead given by:

$$\text{interleave\_ovhd}(W,R) = 2 \times (\text{Reg\_RD}(W,R) + \text{Reg\_WR}(W,R)) \tag{6.14}$$

where Reg_RD and Reg_WR are the read and write energies for the registers as a function of W and R. Thus, the energy overhead per stage due to the interleave operation is

$$\text{Energy\_ovhd}(N,m,k,W) = \text{interleave\_ops}(N,m,k,W) \times \text{interleave\_ovhd}(W,R) \tag{6.15}$$

The overall compute counts and memory access counts for a N size FFT processing on a vector architecture are a function of R, m, N and W and are estimated as follows.

Compute counts – The total number of computations that are a function of only the number of non-zero terms at the input of each stage is given by (using Equations 6.4 and 6.5):

$$tot\_compute\_vec_{ADD}(N,m,k,W) = \sum_{k=\lfloor log_2 \frac{N}{m} \rfloor}^{log_2 N - 1} stage\_acc(N,m,k,W) \tag{6.16}$$

$$tot\_compute\_vec_{MULT}(N,m,k,W) = \sum_{k=0}^{log_2N-1} \frac{stage\_acc(N,m,k,W)}{2} \tag{6.17}$$

where $tot\_compute\_vec_{ADD}$ and $tot\_compute\_vec_{MULT}$ represent the total number of vector addition and multiplication operations. For addition, the lower limit of k is $\lfloor log_2\frac{N}{m} \rfloor$, as up to this stage we have only multiplications as discussed earlier in this section. With the last stage being log_2N, the inputs are the outputs of stage (log_2N-1).

Memory Access counts – Memory read accesses occur at the input, then at stage $I(= \lfloor log_2\frac{N}{m} \rfloor)$, followed by offsets of log_2R as the intermediate stages do not need any additional memory accesses. This is repeated until the last log_2W stages, since here, the elements across which the butterfly operations are performed in subsequent stages are in the registers and are processed as shown in Figure 6.8. With m non-zero elements at the input and vector register width W, $\lceil \frac{m}{W} \rceil$ memory read accesses occur at the input. Thus,

$$tot\_mem\_acc\_vec_{RD}(N,m,R,W) = \left\lceil \frac{m}{W} \right\rceil + \sum_{t=1}^{t_{max}} stage\_acc(N,m,I+(t-1)log_2R,W) \tag{6.18}$$

where t_{max} is obtained from:

$$I+(t_{max}-1)log_2R < log_2N - log_2W$$

$$\Rightarrow t_{max} = \left\lfloor \frac{log_2N - log_2W + log_2R - I}{log_2R} \right\rfloor$$

Similarly, for memory writes we have

$$tot\_mem\_acc\_vec_{WR}(N,m,R) = \left[\sum_{t=1}^{t_{max}} stage\_acc(N,m,I+(t-1)log_2R,W) \right] + stage\_acc(N,m,log_2N,W) \tag{6.19}$$

as the data is first written back to memory at stage I followed by offsets of log_2R and then finally at last stage.

The overall energy consumption for FFT processing on a vector architecture can be estimated by multiplying the counts derived above by the relative weights, which are a function of W,R and adding the interleave operation overhead. The interleaving overhead, $total\_ovhd$, is computed by using Equation 6.15 and is calculated over the stages in which interleaving is done.

$$total\_ovhd = \sum_{x=ll}^{log_2N} Energy\_ovhd(N,m,x,W) \tag{6.20}$$

where $x = ll$ represents the lower bound which is equal to $(log_2N - log_2W - 1)$ as in the last log_2W stages the elements to be read are the non-zero elements in stage $(log_2N - log_2W - 1)$. Thus, we obtain the normalized energy estimates

$$Energy\_est_{vector} = tot\_mem\_acc\_vec_{WR} \times W_{Mem\_WR} + tot\_mem\_acc\_vec_{RD} \times W_{Mem\_RD} + tot\_compute\_vec_{MULT} \times W_{MULT} + tot\_compute\_vec_{ADD} \times W_{ADD} + total\_ovhd \quad (6.21)$$

Here W_{op} represents the relative weight of the operation op and our objective is to find R for a particular choice of (N, m, W) that leads to minimum energy consumption in FFT processing on the vector architecture.

In the next section, we present some experimental results for mappings on both scalar and vector architectures.

6.4 Experimental Results

In our experiments on the proposed optimization strategy for energy efficient FFT, we evaluated the overall energy consumption for varying register file sizes, over different numbers of non-zero inputs (m), for FFT lengths of 2K, 4K, 8K, and 16K. The RF has an interface of width W with the SPM. Our experimentation includes the exploration for different RF sizes (8, 16, 32, 64, and 128) with SIMD widths ranging from $W = 1$ for scalar registers to $W = 32$ for vector registers. The inputs are assumed to be complex and continuous. For example, $m =$ 200, N = 2K represents 200 non-zero elements at the input from position 0 to 199, with all elements from 200 to 2K being zero for an FFT of length 2K. We also show a comparison of the proposed FFT processing strategy with the non-optimized and optimized pruning radix-2 FFT implementations.

The proposed optimizations and analysis are validated on an architecture with a RF with configurable size (Figure 6.2). The processor simulator provides run-time statistics that are used to obtain the memory access counts and compute counts. The HDL models of the processor were synthesized using Cadence RTL compiler for TSMC 40nm standard library. Synopsys Primepower was used to carry out power simulations on the synthesized design. Energy numbers for memory structures such as register file and SPM were derived from a commercial memory compiler. RF sizes and SIMD widths are chosen in powers of 2 as this is a reasonable assumption for energy efficient implementation of radix-2 FFT.

6.4.1 Energy Efficient RF Size Selection

The proposed energy optimization techniques for processing FFTs with large number of zeros at the input makes the selection of RF size an important point of exploration. We present the exploration results for non-SIMD scalar and SIMD vector register file architectures.

Exploration Results for non-SIMD scalar registers

The energy consumption for the non-optimized implementation is independent of the number of non-zero points at the input while it varies for our proposed optimized implementation.

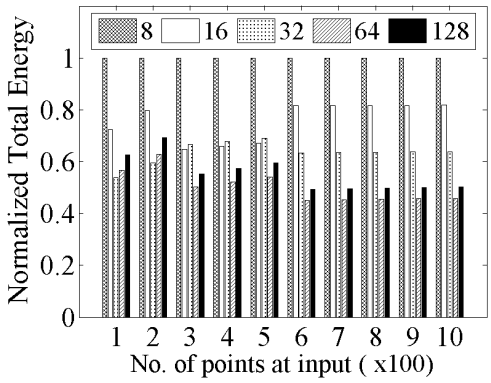
Figure 6.9(a) shows a comparison of the energy values for m non-zero points followed by $(2K-m)$ zero values at the input of 2K length FFT with m ranging from 100 to 1000. From the simulation results for each m with different number of registers in the Register File, we observe that the RF configuration corresponding to minimum energy is not constant over m for the same size FFT.

The variation in total energy over the RF size choices for different FFT lengths (Figure 6.9) is summarized in Table 6.1. *Opt RF* is the RF size corresponding to minimum energy and *% var* gives the percentage variation in energy of the energy optimal RF size with respect to the worst case choice for the particular m. With variation in energy consumption ranging from 40% to 59%, we conclude that the register file size is an important point of exploration for obtaining an energy efficient implementation. Another important observation is that, as the fraction of non-zero points approaches 50%, the gain begins to saturate. This is because of the reduction in the number of redundant memory accesses and computations across the stages with higher number of non-zero inputs.

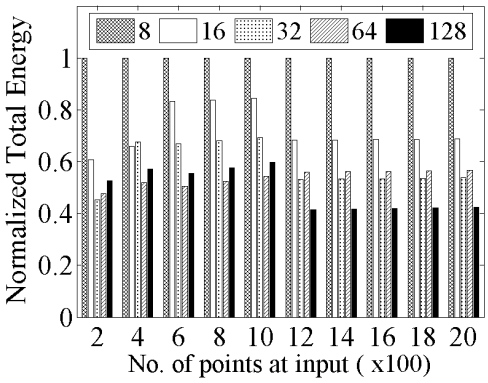
Exploration Results for SIMD vector registers

From the exploration results for scalar registers we conclude that the RF size corresponding to energy efficient implementation is not constant for a given FFT length. It varies with the number of non-zero points at the input. For the register file comprising vector registers another parameter, SIMD width (W), also plays an important role. We present the exploration results for the same test cases with variation in W.

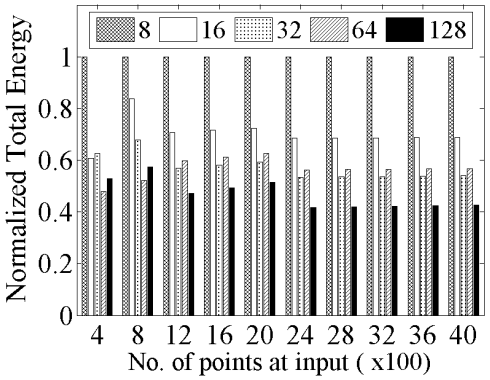
Figure 6.10 shows the total energy variation over the RF size choices for 2K length FFT for SIMD width of registers taken as 4 and 8. Higher SIMD widths are not considered as they result in bringing the entire input data to the RF. Table 6.2 summarizes the exploration results for different RF configurations. For $W = 4$, the reduction in energy consumption with the optimal RF choice varies from 20% to 51% over the range of non-zero inputs, while for $W = 8$, the savings vary from 21% to 58%.



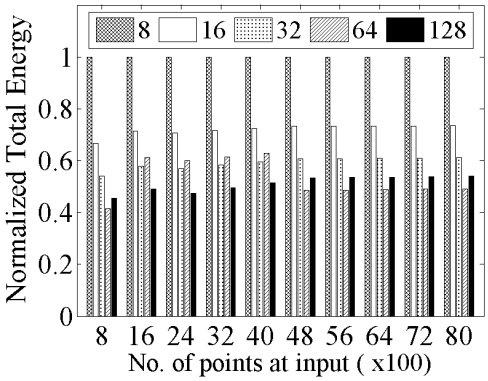
(a) For 2K FFT



(b) For 4K FFT

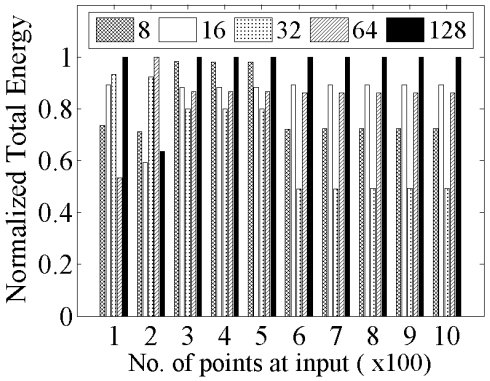


(c) For 8K FFT

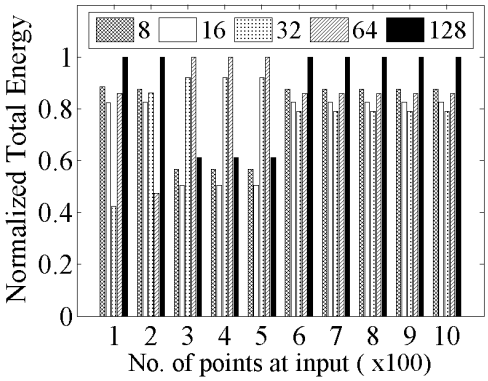


(d) For 16K FFT

Figure 6.9: Total energy variation over RF size varying from 8 to 128 registers ($W = 1$)



(a) $W = 4$



(b) $W = 8$

Figure 6.10: Total energy variation over RF size varying from 8 to 128 registers for 2K FFT

Figure 6.11 shows the total energy variation over the RF size choices for 4K length FFT for SIMD width of registers taken as 4, 8 and 16. SIMD width 32 is not considered as it results

m	opt RF	% var
100	32	46.1
200	32	40.41
300	32	49.78
400	64	47.79
500	64	45.92
600	64	55.01
700	64	54.8
800	64	54.59
900	64	54.37
1000	64	54.16

(a) For 2K FFT

m	opt RF	% var
200	32	54.73
400	64	48.04
600	64	49.52
800	64	47.55
1000	64	45.71
1200	128	58.46
1400	128	58.24
1600	128	58.03
1800	128	57.81
2000	128	57.6

(b) For 4K FFT

m	opt RF	% var
400	64	52.06
800	64	47.79
1200	128	52.73
1600	128	50.63
2000	128	48.65
2400	128	58.19
2800	128	57.97
3200	128	57.76
3600	128	57.54
4000	128	57.33

(c) For 8K FFT

m	opt RF	% var
800	64	58.53
1600	128	50.9
2400	128	52.49
3200	128	50.42
4000	128	48.45
4800	64	51.44
5600	64	51.29
6400	64	51.15
7200	64	51.01
8000	64	50.87

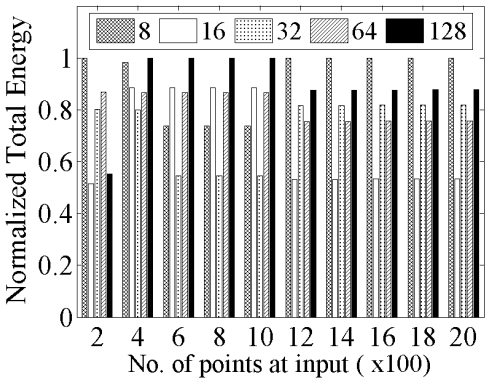
(d) For 16K FFT

Table 6.1: Summary of the exploration results using scalar registers

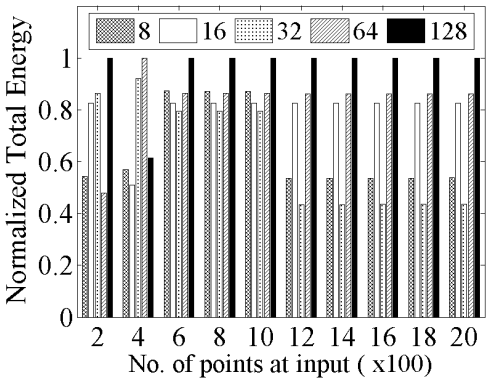
m	W = 4		W =8	
	opt RF	% var	opt RF	% var
100	64	46.62	32	57.61
200	16	40.78	64	52.55
300	32	20.15	16	49.5
400	32	20.13	16	49.46
500	32	20.12	16	49.41
600	32	50.92	32	21.1
700	32	50.83	32	21.09
800	32	50.74	32	21.08
900	32	50.66	32	21.07
1000	32	50.57	32	21.05

Table 6.2: Summary of the exploration results using vector registers for 2K FFT

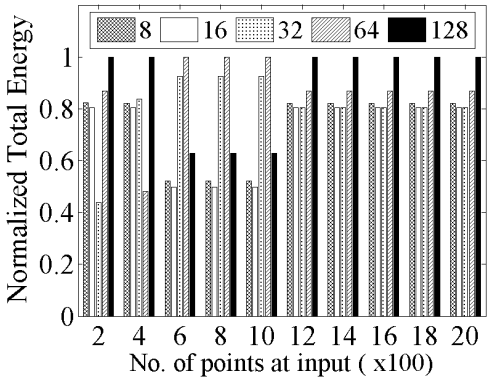
in bringing the entire input data to the RF. Table 6.3 summarizes the exploration results for different RF configurations. For $W = 4$, the reduction in energy consumption with the optimal RF choice varies from 20% to 49%, for $W = 8$, from 20% to 57%, and from 19% to 57% for $W = 16$.



(a) $W = 4$



(b) $W = 8$



(c) $W = 16$

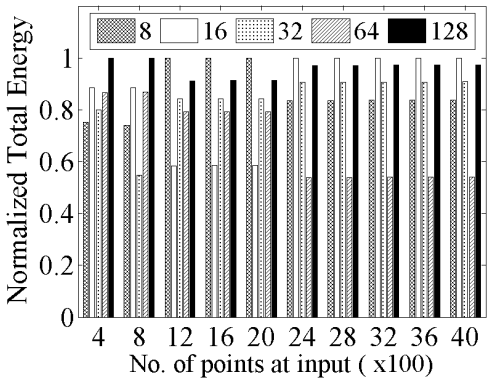
Figure 6.11: Total energy variation over RF size varying from 8 to 128 registers for 4K FFT

For FFT lengths of 8K and above, we consider 4 choices of SIMD widths: 4, 8, 16, and 32. Figure 6.12 shows the total energy variation over the RF size choices for 8K length FFT for different SIMD widths. Table 6.4 summarizes these results. For $W = 4$, the reduction in energy consumption with the optimal RF choice varies from 24% to 47%, for $W = 8$, from 20% to 53%, for $W = 16$, from 19% to 57%, and from 18% to 51% with W as 32.

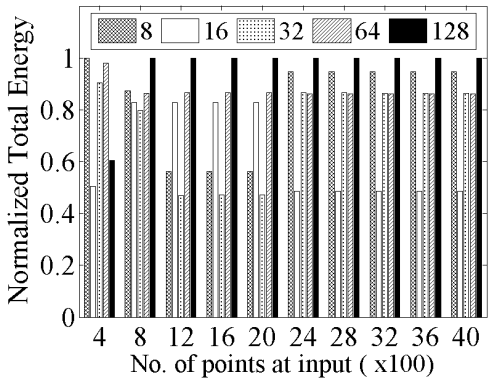
Figure 6.13 shows the total energy variation over the RF size choices for 16K length FFT with SIMD widths ranging from 4 to 32. Table 6.5 summarizes the exploration results for different RF configurations. The variation in energy savings with the optimal RF choice for different SIMD widths is as follows: 26% to 51% for $W = 4$, 48% to 53% for $W = 8$, 19% to 55% for $W = 16$, and 18% to 50% for $W = 32$.

m	W = 4		W = 8		W =16	
	opt RF	% var	opt RF	% var	opt RF	% var
200	16	48.48	64	52.17	32	56.24
400	32	19.99	16	48.99	64	51.86
600	32	45.52	32	20.45	16	50.33
800	32	45.46	32	20.44	16	50.31
1000	32	45.39	32	20.43	16	50.28
1200	16	46.86	32	56.61	32	19.67
1400	16	46.78	32	56.56	32	19.67
1600	16	46.7	32	56.52	32	19.67
1800	16	46.62	32	56.46	32	19.66
2000	16	46.54	32	56.41	32	19.66

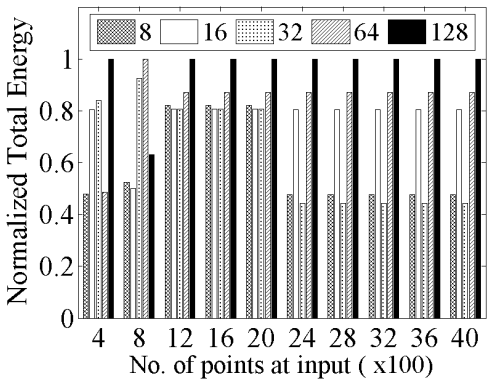
Table 6.3: Summary of the exploration results using vector registers for 4K FFT



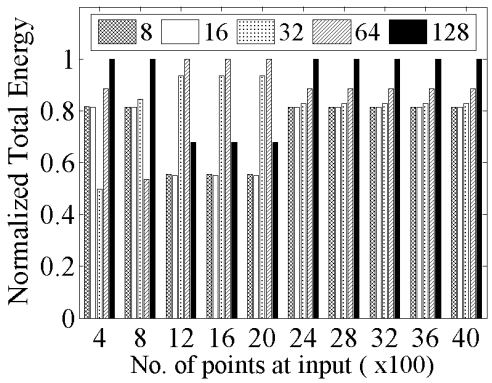
(a) $W = 4$



(b) $W = 8$



(c) $W = 16$

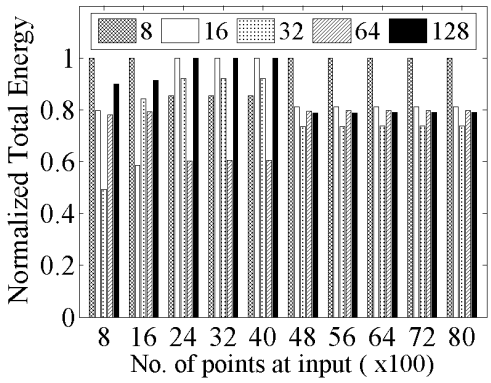


(d) $W = 32$

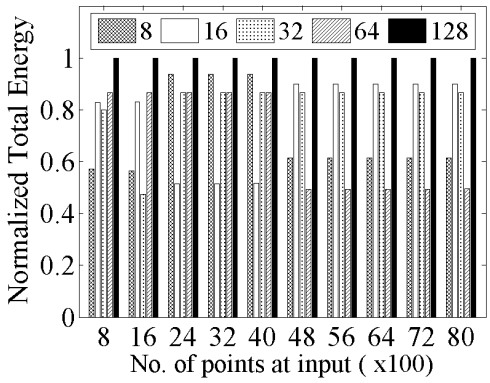
Figure 6.12: Total energy variation over RF size varying from 8 to 128 registers for 8K FFT

m	W = 4		W =8		W =16		W =32	
	opt RF	% var	opt RF	% var	opt RF	% var	opt RF	% var
400	8	24.77	16	49.59	8	52.06	32	50.12
800	32	45.22	32	20.26	16	49.89	64	46.39
1200	16	41.6	32	52.97	16	19.39	16	45.11
1600	16	41.53	32	52.93	16	19.39	16	45.09
2000	16	41.47	32	52.89	16	19.39	16	45.08
2400	64	46.24	16	51.47	32	55.66	8	18.55
2800	64	46.14	16	51.42	32	55.64	8	18.55
3200	64	46.04	16	51.38	32	55.61	8	18.55
3600	64	45.94	16	51.34	32	55.59	8	18.55
4000	64	45.83	16	51.3	32	55.57	8	18.55

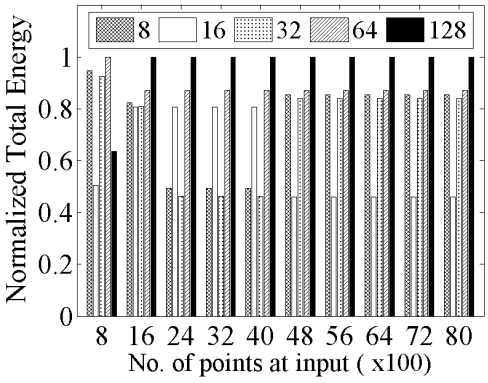
Table 6.4: Summary of the exploration results using vector registers for 8K FFT



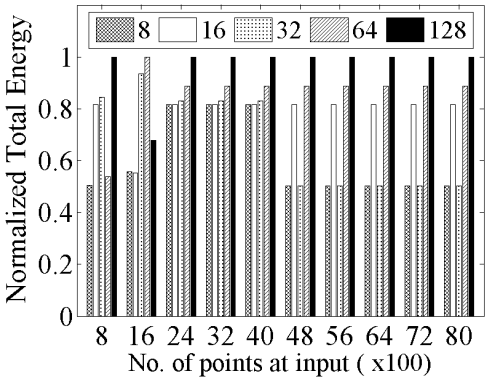
(a) $W = 4$



(b) $W = 8$



(c) $W = 16$



(d) $W = 32$

Figure 6.13: Total energy variation over RF size varying from 8 to 128 registers for 16K FFT

m	W = 4		W =8		W =16		W =32	
	opt RF	% var	opt RF	% var	opt RF	% var	opt RF	% var
800	32	50.62	8	42.83	16	49.47	8	49.55
1600	16	41.36	32	52.51	16	19.24	16	44.82
2400	64	39.69	16	48.48	32	53.74	8	18.45
3200	64	39.62	16	48.44	32	53.71	8	18.45
4000	64	39.53	16	48.41	32	53.69	8	18.45
4800	32	26.38	64	50.72	16	54.07	32	49.78
5600	32	26.33	64	50.68	16	54.05	32	49.77
6400	32	26.29	64	50.63	16	54.03	32	49.76
7200	32	26.24	64	50.58	16	54	32	49.75
8000	32	26.2	64	50.53	16	53.98	32	49.74

Table 6.5: Summary of the exploration results using vector registers for 16K FFT

From the exploration results over the different FFT sizes and different SIMD widths, we conclude that the optimal RF size for a range of non-zero inputs percentage over a given length FFT is similar with almost constant energy savings. These non-zero input percentage ranges are identified as 15% to 30% and 30% and above. For less than 15% non-zero inputs, the behaviour is less predictable. The results show that the energy efficient RF size varies with SIMD Width W also as anticipated in Equation 6.21. For architectures with vector registers it is important to consider not only the number of registers but also the vector register width. With an identical number of registers in the RF, up to 55% variation in the overall energy is observed with changing SIMD width (derived from the actual energy values).

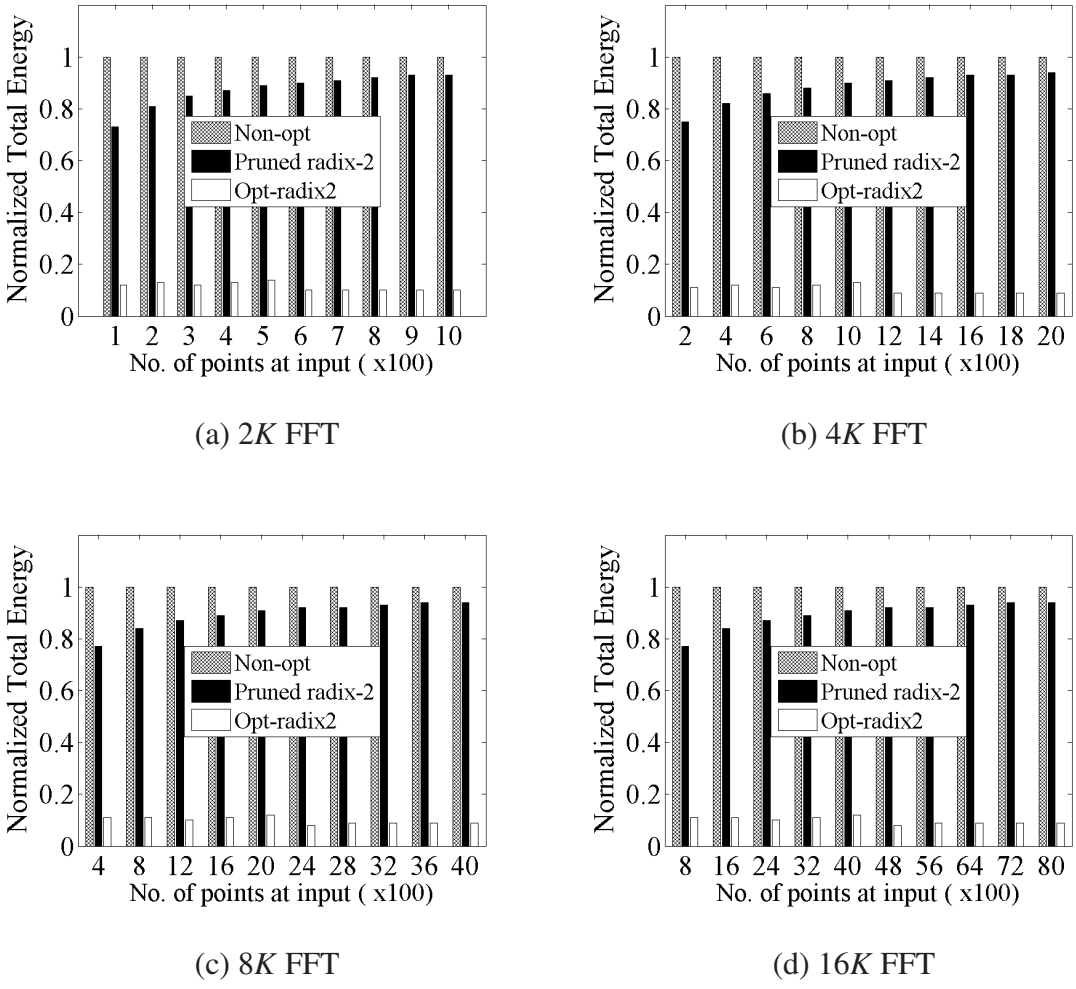
6.4.2 Comparison of Proposed Strategy with Pruned Radix-2 FFT implementation

In this section, we present a comparison of our proposed FFT processing strategy with the non-optimized and pruned radix-2 FFT implementation. The FFT pruning method [106, 108] is popularly used for partial processing of FFTs with a large number of zeros at the input or output.

Effect of Register File Size on different implementation strategies

Register file (RF) size is an important parameter in the compilation to the embedded processors. With increasing RF size, the energy consumption increases for both – the non-optimized implementation and the pruned radix-2 implementation. The energy variation with the RF size is almost the same in both implementations thereby making the gain achieved with the FFT

pruning independent of the RF size. However, the RF size plays an important role for our proposed FFT implementation (as discussed in Section 1.6) and causes non-uniform variation in the energy consumption. We select an energy efficient RF size for comparison against the non-optimized and the pruned radix-2 implementations.



(a) 2*K* FFT

(b) 4*K* FFT

(c) 8*K* FFT

(d) 16*K* FFT

Figure 6.14: Comparison of different implementations for non-SIMD architectures

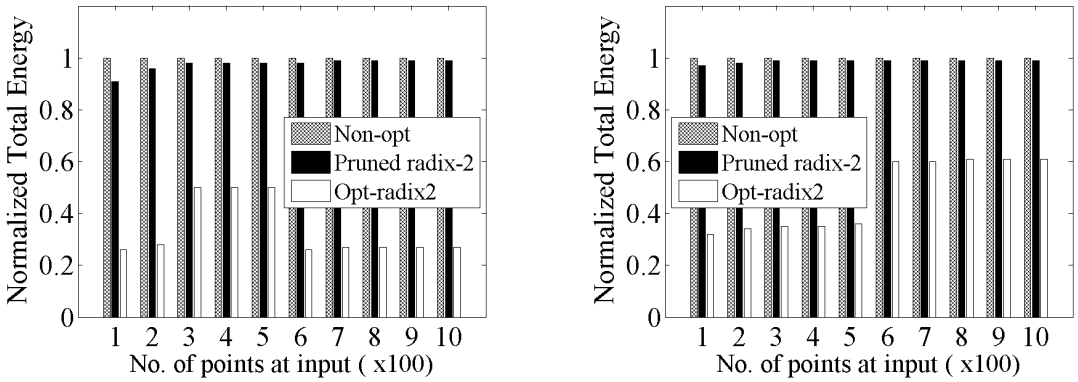
Implementations with non-SIMD scalar registers

Figure 6.14 shows the energy variation across the three different FFT processing strategies for different FFT lengths with large variation in the number of non-zero points at the input. With increasing number of non-zero points at the input, the percentage improvement with the pruning strategy decreases. For the test cases under consideration, this improvement varies from 5% to 27%. On the other hand, this variation is non-uniform for the proposed FFT processing strategy. This is because the energy efficient RF size is not constant for a fixed length FFT or the number

of non-zero points. With the energy efficient RF size selection, we obtain an improvement in the range 87-92% over the non-optimized implementation.

Implementations with SIMD vector registers

For SIMD architectures, the RF size selection is a function not only of the number of registers but also the vector register width. The improvement over the non-optimized implementation decreases here as compared to the high gains in scalar architectures. This is because the vector register implementation leads to redundant memory accesses also which brings down the energy savings that could be achieved with the optimized implementation.



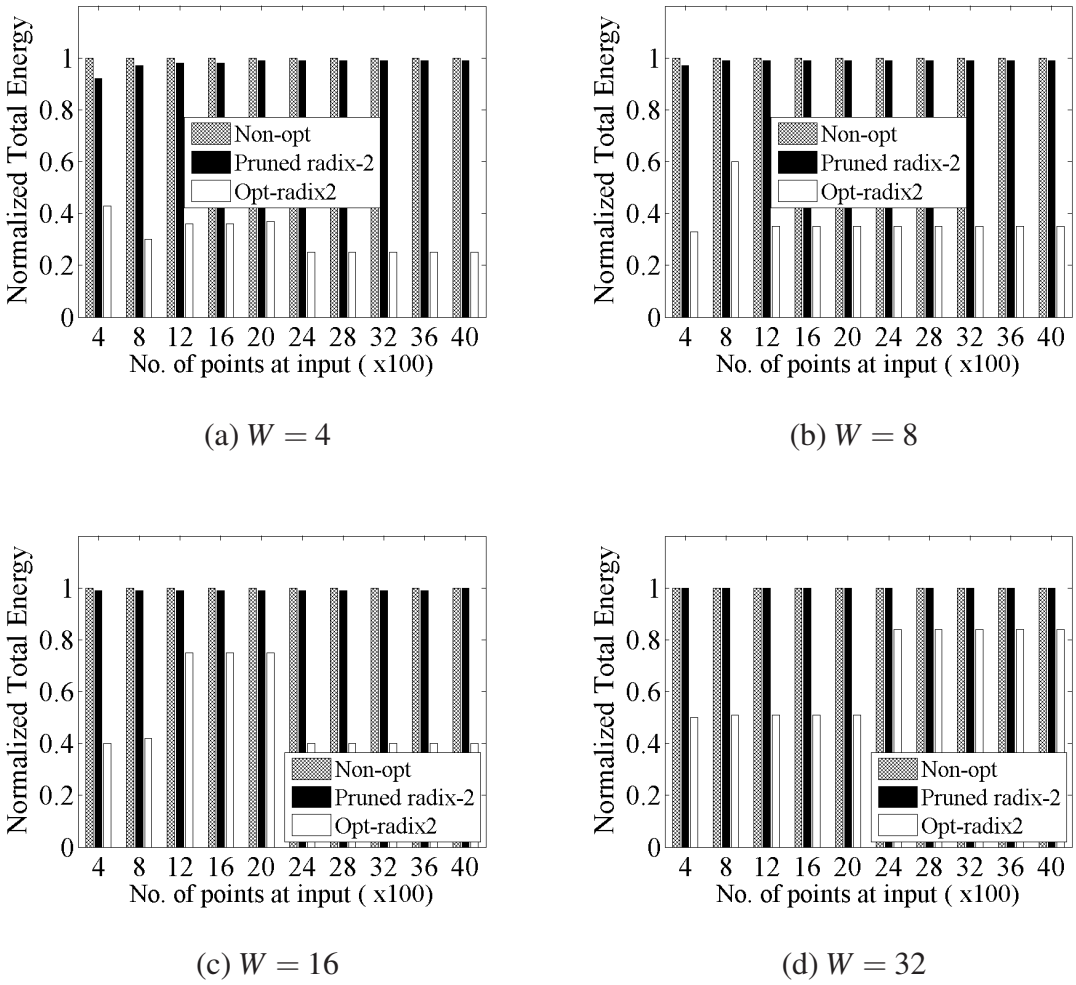
(a) $W = 4$ (b) $W = 8$

Figure 6.15: Comparison of different implementations of 2K point FFT for SIMD architectures

Figure 6.15 shows a comparison of the different FFT implementation strategies on vector architectures for 2K length FFT. With vector register width of 4 ($W = 4$), the pruning radix-2 strategy consumes up to 9% less energy than the non-optimized implementation. The proposed strategy implementation with the energy efficient RF size results in 50-74% improvement over the non-optimized implementation. For $W = 8$, these improvements are up to 3% for pruning radix-2 implementation and 40-68% for our proposed FFT processing strategy.

Figure 6.16 shows a comparison of the different FFT implementation strategies on vector architectures for 8K length FFT. With increasing vector register widths, the improvement with the pruned radix-2 FFT implementation over the non-optimized implementation decreases and is a maximum 8% for $W = 4$. On the other hand, our proposed strategy with the energy efficient RF size results in 57% to 75% improvement for $W = 4$, 40% to 67% for $W = 8$, 25% to 60% for $W = 16$, and 16% to 50% for $W = 32$ over the non-optimized implementation.

Table 6.6 summarizes the comparison results of different FFT implementation strategies. The numbers correspond to the reduction in energy consumption over the non-optimized FFT



(a) $W = 4$ (b) $W = 8$

(c) $W = 16$ (d) $W = 32$

Figure 6.16: Comparison of different implementations of 8K point FFT for SIMD architectures

implementation with the two implementations – Pruned Radix-2 FFT (Pruned) and our proposed optimized implementation (Opt). The energy efficient RF size was selected for the given input configuration for which the different implementations are being compared.

From the comparison of different implementation strategies, we conclude that:

- The pruning radix-2 FFT implementation achieves reasonable improvement over the non-optimized implementation for scalar architectures. This gain reduces with increasing vector register widths and becomes negligible for higher vector widths with relatively larger number of non-zero points at the input ($\leq$50%).

- The gains with our proposed FFT processing strategy are sensitive to the RF size. With the energy efficient RF size selection, improvements ranging from 16% to 92% are obtained over the pruned radix-2 implementation across the explored test cases.

Table 6.6: Comparison of the different FFT implementation strategies for a range of m in N length FFT

SIMD Width (W)	% variation in energy consumption over the non-optimized implementation							
	N = 2K		N =4K		N = 8 K		N = 16K	
	Pruned	Opt	Pruned	Opt	Pruned	Opt	Pruned	Opt
1	≤27%	86% - 90%	≤25%	87% - 91%	≤23%	88% - 92%	≤23%	88% - 92%
4	≤9%	50% - 74%	≤8%	53% - 68%	≤8%	57% - 75%	≤7%	63% - 72%
8	≤3%	40% - 68%	≤3%	40% - 68%	≤3%	40% - 67%	≤3%	57% - 70%
16	-	-	≤1 %	26% - 60%	≤1%	25% - 60%	≤1%	27% - 59%
32	-	-	-	-	0%	16% - 50%	0%	16% - 50%

6.5 Conclusion

The interpolation of m points to $N(=2^k)$ points is a common functionality in signal processing applications. It is implemented by the processing of 2^k point FFT on the input data with m points and zeros appended at the end. Through this work we explored how the Register File (RF) size affects the total energy consumption in such FFT processing. We presented a strategy that selects the energy optimal point based on the processing energy estimates for m data points at the input of N point FFT over a range of RF sizes. We observe that the energy optimal RF size changes with the number of non-zero points at the input and the FFT sizes. The energy optimal points also change with changing vector register widths. Experimental results show a variation of 18.5% to 58.5% in energy consumption across the best and worst choice of RF size in the given range. The test cases have the number of non-zero points at the input ranging from 5% to about 50% for the FFT sizes of 2K, 4K, 8K, and 16K.

For vector register files, our results show that the energy optimal point is a function of both SIMD width W and the number of registers RF. For the same register file size and a fixed length FFT, a variation of up to 50% in energy is observed over the test cases. Thus, an intelligent selection of the RF size can significantly reduce the overall energy consumption. The energy reduction over previous work (the pruned radix-2 implementation) is as high as 92% across the explored test cases. These gains were achieved with the proposed implementation on the corresponding energy efficient RF size. Our proposed strategy is also applicable for Inverse FFT (IFFT) pruning.

In conclusion, using the proposed transformations and architectural exploration we can significantly reduce the data memory energy consumption for applications executing on scratch pad based SIMD architectures.

Chapter 7

Conclusion and Future Work

With modern embedded processors having features such as SIMD Functional Units, vector registers, and wide interfaces to scratch pad memories, the memory sub-system contributes significantly to the overall system energy. An energy efficient implementation on these architectures requires data in the memory to be intelligently organized and accessed, because of the dominant role played by accesses to the wide memory and vector register files.

In this thesis we presented data memory energy optimization strategies targeted at such embedded processor systems, that significantly reduce the memory accesses, thereby reducing the overall implementation energy.

7.1 Summary of Contributions

Summarizing the optimizations and results presented in this thesis:

- We explored LTE, a data intensive wireless standard with SDR. We proposed a data layout strategy through a detailed data dependence analysis of the application. Significant reduction (7-15%) in data memory energy consumption can be achieved without any performance overhead if array data are interleaved. The absence of dependencies across PRBs allows the merging of some kernels in the data processing block, which reduces the overall memory access count by around 15%.

- We presented *Array Interleaving*, a data layout technique to improve memory access energy. Efficient data layouts are important for systems with wide memory interfaces to help reduce memory access energy by reducing the memory access count. Our array interleaving strategy helps reduce memory energy significantly by improving the data locality that helps in loading relevant data sets to the vector registers. The interleaving decisions for different possible sets of candidate arrays are taken based on cost estimates of the benefits

of interleaving as well as the overhead involved in performing the layout changes. Our global analysis is aware of arrays being accessed in different ways in different loops, and generates interleaving decisions that are globally optimal. We also considered the possibility of remapping of the arrays across the loops if the estimated energy reduction is greater than the overhead involved in the layout changes. Our exploration also considers the interleaving granularity as a parameter. Experimental evaluation on several applications in the wireless communication and image processing domain showed a significant reduction (6%–34%) in memory energy consumption.

- We presented a data flow transformation strategy for QRD targeting embedded processor systems with features such as SPMs, SIMD FUs, and vector register files with wide interfaces to both SPM and FUs. We also presented different implementations for QRD exploiting the SIMD feature of the architecture. Up to 75% reduction in memory accesses is obtained with our proposed strategy over the conventional sequences. Experimental results show a significant reduction (up to 36%) in overall energy across different implementations with our proposed strategy. With increasing number of vector registers, the reduction with respect to conventional sequences increases further. We also presented a comparison of the proposed sequencing strategy with the standard tiling transformation. Experimental results show that tiling results in an energy efficient implementation with respect to conventional sequences but still has higher energy consumption compared to our proposed sequence implementation. Matrix decomposition being widely applicable in the communication and multimedia domain, the results show that our technique can make a significant difference in the energy efficiency of such applications.

- We explored the effect of RF size on the overall energy consumption in N point FFT processing with m continuous non-zero data points at the input. Up to 75% variation in the energy for architecture with scalar registers was obtained, which motivated us to extend the exploration for vector architectures. We presented a strategy that selects an energy optimal RF configuration based on the energy estimates for processing N point FFT on m non-zero data points. The energy efficient RF size changes with the number of non-zero points at the input and the FFT sizes. Experimental results show a variation of 18.5% to 58.5% in energy consumption for the best and worst choice of RF size in the given range. The test cases have the number of input points ranging from 5% to about 50% for the FFT sizes of 2K, 4K, 8K, and 16K. For vector register files, the optimal configurations also change with changing vector register widths. For the same register file size and a fixed length FFT, a variation of up to 50% in energy is observed over the test cases on varying SIMD width W.

7.2 Future Directions

The work presented in this thesis could be extended in the following directions in the future.

- The *Array Interleaving* problem can be further explored to address non-manifest loops. This creates an interesting exploration problem as this would lead to distinct schedules that may affect the interleaving decision. Combining the array interleaving transformation with other optimization techniques such as loop transformations is another direction for extension. An appropriate RF configuration selection helps in significantly reducing the overall energy dissipation by reducing the redundant memory accesses and computations.

- In this work our exploration for an energy optimized sequencing strategy is restricted to QR Decomposition algorithm. Many other matrix decomposition algorithms such as LU decomposition and Cholesky decomposition also follow a similar implementation. Extending the applicability of the proposed data flow transformation for energy efficient implementations of these algorithms could make the proposed strategy more generic. Another extension could be to combine the algorithmic improvements proposed in literature with our sequencing strategy so as to the maximize the reduction in data memory energy and improve the performance for the matrix decomposition sub-routine.

- We performed our exploration for FFT with non-zero input points using *radix-2* implementation. Extending the exploration for a generic *radix-r* FFT implementation is a possible future direction. For mixed-radix implementation, the exploration becomes more complex. In our work, we assumed complex data type for the inputs, which remains the same across the stages as well. But what if the inputs are purely real or imaginary? The number of real and complex points at the input also affects the compute complexity and memory accesses across the stages. Extending the exploration strategy to include the effect of data types at the inputs and across the stages is another direction for extension.

Bibliography

[1] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn. A 477mW NoC-based digital baseband for MIMO 4G SDR. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 278–279, February 2010.

[2] Ulrich Ramacher, Wolfgang Raab, J. A. Ulrich Hachmann, Dominik Langen, Jrg Berthold, R. Kramer, A. Schackow, Cyprian Grassmann, Mirko Sauermann, P. Szreder, F. Capar, G. Obradovic, W. Xu, Nico Brls, Kang Lee, Eugene Weber, Ray Kuhn, and John Harrington. Architecture and implementation of a Software-Defined Radio baseband processor. In *International Symposium on Circuits and Systems*, pages 2193–2196. IEEE, 2011.

[3] Vasile Surducan, Mayan Moudgill, Gary Nacer, Emanoil Surducan, Pablo I. Balzola, John Glossner, Stuart Stanley, Meng Yu, and Daniel Iancu. The Sandblaster Software-Defined Radio Platform for Mobile 4G Wireless Communications. *International Journal of Digital Multimedia Broadcasting*, 2009.

[4] Tomoya Suzuki, Hideki Yamada, Toshiyuki Yamagishi, Daisuke Takeda, Koji Horisaki, Toshio Fujisawa, Yasuo Unekawa, Tom Vander Aa, and Liesbet Van der Perre. High-Throughput, Low-Power Software-Defined Radio Using Reconfigurable Processors. *IEEE Micro Magazine*, 31(6):19–28, 2011.

[5] Application Note. LTE PHY Layer Measurement Guide, 2011.

[6] Zheng-Yu Huang and Pei-Yun Tsai. Efficient Implementation of QR Decomposition for Gigabit MIMO-OFDM Systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(10):2531–2542, October 2011.

[7] Lei Ma, K. Dickson, J. McAllister, and J. McCanny. QR Decomposition-Based Matrix Inversion for High Performance Embedded MIMO Receivers. *IEEE Transactions on Signal Processing*, 59(4):1858–1867, April 2011.

[8] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[9] Preeti Ranjan Panda, Alexandru Nicolau, and Nikil Dutt. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[10] Francky Catthoor, Praveen Raghavan, Andy Lambrechts, Murali Jayapala, Angeliki Kritikakou, and Javed Absar. *Ultra-Low Energy Domain-Specific Instruction-Set Processors*. Springer, New York, NY, USA, 2010.

[11] M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu. Compiler-directed scratch pad memory optimization for embedded multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):281–287, March 2004.

[12] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *International Symposium on Hardware/Software Codesign (CODES)*, pages 73–78, May 2002.

[13] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *European Design and Test Conference (ED&TC)*, pages 7–11, 1997.

[14] Doosan Cho, Ilya Issenin, Nikil Dutt, Jonghee W. Yoon, and Yunheung Paek. Software controlled memory layout reorganization for irregular array access patterns. In *International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES)*, pages 179–188, 2007.

[15] M. Kandemir, I. Kadayif, and U. Sezer. Exploiting scratch-pad memory using Presburger formulas. In *International Symposium on System Synthesis (ISSS)*, pages 7–12, October 2001.

[16] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *International Symposium on System Synthesis*, pages 213–218, October 2002.

[17] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 104–109, 2004.

[18] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[19] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[20] TriMedia (mediaprocessor). http://en.wikipedia.org/wiki/TriMedia.

[21] C6000 - High Performance DSP. http://www.ti.com/lsds/ti/dsp/platform/c6000-high-performance/device.page.

[22] S. Rixner, W.J. Dally, B. Khailany, P. Mattson, U.J. Kapasi, and J.D. Owens. Register organization for media processing. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 375–386, January 2000.

[23] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Mapping Applications onto Reconfigurable KressArrays. In *Field Programmable Logic and Applications*, volume 1673 of *Lecture Notes in Computer Science*, pages 385–390. Springer Berlin Heidelberg, 1999.

[24] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.

[25] Carl Ebeling, DarrenC. Cronquist, and Paul Franklin. RaPiD Reconfigurable pipelined datapath. In *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, volume 1142 of *Lecture Notes in Computer Science*, pages 126–135. Springer Berlin Heidelberg, 1996.

[26] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. DRESC: a retargetable compiler for coarse-grained reconfigurable architectures. In *International Conference on Field-Programmable Technology (FPT)*, pages 166–173, 2002.

[27] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proceedings of the Conference on Field Programmable Logic*, volume 2778, pages 61–70. Springer, 2003.

[28] Bingfeng Mei. A Coarse-Grained Reconfigurable Architecture Template and its Compilation Techniques. Technical report, IMEC, K.U. Leuven, 2005.

[29] T. Vander Aa, M. Palkovic, M. Hartmann, P. Raghavan, A. Dejonghe, and L. Van der Perre. A multi-threaded coarse-grained array processor for wireless baseband. In *Symposium on Application Specific Processors (SASP)*, pages 102–107, June 2011.

[30] Namita Sharma, Tom Vander Aa, Prashant Agrawal, Praveen Raghavan, Preeti Ranjan Panda, and Francky Catthoor. Data memory optimization in LTE downlink. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2610–2614, May 2013.

[31] Namita Sharma, Preeti Ranjan Panda, Min Li, Prashant Agrawal, and Francky Catthoor. Energy efficient data flow transformation for Givens Rotation based QR Decomposition. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–4, March 2014.

[32] Luca Benini, Alberto Macii, and Massimo Poncino. Energy-aware Design of Embedded Memories: A Survey of Technologies, Architectures, and Optimization Techniques. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(1):5–32, February 2003.

[33] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *International Symposium on Microarchitecture*, pages 237–248, December 2001.

[34] P. Raghavan, A. Lambrechts, M. Jayapala, F. Catthoor, D. Verkest, and H. Corporaal. Very Wide Register: An Asymmetric Register File Organization for Low Power Embedded Processors. In *Design, Automation Test in Europe Conference Exhibition*, pages 1–6, April 2007.

[35] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis. Energy and performance improvements in microprocessor design using a loop cache. In *International Conference on Computer Design (ICCD)*, pages 378–383, October 1999.

[36] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *International Symposium on Microarchitecture (MICRO)*, pages 184–193, 1997.

[37] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Local memory exploration and optimization in embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 18(1):3–13, 1999.

[38] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 49(2), 1970.

[39] P.R. Panda. Memory bank customization and assignment in behavioral synthesis. In *International Conference on Computer-Aided Design (ICCAD)*, pages 477–481, November 1999.

[40] M. Kandemir. Impact of data transformations on memory bank locality. In *Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 506–511, February 2004.

[41] O. Ozturk and M. Kandemir. Nonuniform banking for reducing memory energy consumption. In *Design, Automation and Test in Europe*, pages 814–819, March 2005.

[42] Preeti Ranjan Panda, Francky Catthoor, Nikil D. Dutt, Koen Danckaert, Erik Brockmeyer, Chidamber Kulkarni, Arnout Vandecappelle, and Per Gunnar Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2):149–206, 2001.

[43] Dattatraya Kulkarni and Michael Stumm. Loop and Data Transformations: A Tutorial. Technical report, June 1993.

[44] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[45] Naraig Manjikian and Tarek S. Abdelrahman. Array data layout for the reduction of cache conflicts. In *International Conference on Parallel and Distributed Computing Systems*, October 1995.

[46] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Memory data organization for improved cache performance in embedded processor applications. *ACM Transactions on Design Automation of Electronic Systems*, 2:384–409, October 1997.

[47] D. Kulkarni and M. Stumm. *Languages, Compilers and Run-time Systems for Scalable Computers*. Kluwer Academic Publishers, 1995.

[48] MichałCierniak and Wei Li. Unifying Data and Control Transformations for Distributed Shared-memory Machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 205–217. ACM, 1995.

[49] Mahmut Kandemir, J. Ramanujam, and Alok Choudhary. Improving Cache Locality by a Combination of Loop and Data Transformations. *IEEE Transactions on Computers*, 48(2):159–167, February 1999.

[50] Victor Delaluz, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, Anand Sivasubramaniam, and Ibrahim Kolcu. Compiler-Directed Array Interleaving for Reducing Energy in Multi-Bank Memories. In *International Conference on VLSI Design*, pages 288–, 2002.

[51] H. S Kim, N. Vijaykrishnan, M. Kandemir, E. Brockmeyer, F. Catthoor, and M.J. Irwin. Estimating influence of data layout optimizations on SDRAM energy consumption. In *International Symposium on Low Power Electronics and Design*, pages 40–43, August 2003.

[52] E. Brockmeyer, M. Miranda, and F. Catthoor. Layer assignment techniques for low energy in multi-layered memory organisations. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 1070–1075, March 2003.

[53] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor, and H. De Man. Cache conscious data layout organization for conflict miss reduction in embedded multimedia applications. *IEEE Transactions on Computers*, 54(1):76–81, January 2005.

[54] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: a first approach. In *Workshop on Embedded Systems for Real-Time Multimedia*, pages 115–120, September 2005.

[55] P.R. Panda, L. Semeria, and G. De Micheli. Cache-efficient memory layout of aggregate data structures. In *International Symposium on System Synthesis*, pages 101–106, September-October 2001.

[56] Gerard J. Foschini, Glen D. Golden, Reinaldo A. Valenzuela, and Peter W. Wolniansky. Simplified processing for high spectral efficiency wireless communication employing multi-element arrays. *IEEE Journal on Selected Areas in Communications*, 17(11):1841–1852, 1999.

[57] Q.H. Spencer, A.L. Swindlehurst, and M. Haardt. Zero-forcing methods for downlink spatial multiplexing in multiuser MIMO channels. *IEEE Transactions on Signal Processing*, 52(2):461–471, February 2004.

[58] Runhua Chen, Zukang Shen, J.G. Andrews, and R.W. Heath. Multimode Transmission for Multiuser MIMO Systems With Block Diagonalization. *IEEE Transactions on Signal Processing*, 56(7):3294–3302, July 2008.

[59] Jie Xu, Ling Qiu, and Chengwen Yu. Improving energy efficiency through multimode transmission in the downlink mimo systems. *EURASIP Journal of Wireless Communication and Networking*, 2011.

[60] S. Schwarz, M. Wrulich, and M. Rupp. Mutual information based calculation of the Precoding Matrix Indicator for 3GPP UMTS/LTE. In *Workshop on Smart Antennas*, pages 52–58, February 2010.

[61] L.A.M. Ruiz de Temino, C. Navarro i Manchon, C. Rom, T.B. Sorensen, and P. Mogensen. Iterative Channel Estimation with Robust Wiener Filtering in LTE Downlink. In *Vehicular Technology Conference*, pages 1–5, September 2008.

[62] Francky Catthoor, Eddy De Greef, and Sven Suytack. *Custom memory management methodology: Exploration of memory organisation for embedded multimedia system design*. Kluwer Academic Publishers, 1998.

[63] Preeti Ranjan Panda, Aviral Shrivastava, B. V. N. Silpa, and Krishnaiah Gummidipudi. *Power-efficient System Design*. Springer, New York, NY, USA, 2010.

[64] P. Ranjan Panda, N.D. Dutt, A. Nicolau, F. Catthoor, A. Vandecappelle, E. Brockmeyer, C. Kulkarni, and E. De Greef. Data memory organization and optimizations in application-specific systems. *IEEE Design Test of Computers*, 18(3):56–68, May 2001.

[65] Technical Specification Group Radio Access Network. 3GPP TS 36.211 V8.9.0 (2009-12). Technical report, 3rd Generation Partnership Project (3GPP), Release 8, 2009.

[66] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter, and Wen mei W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *International Symposium on Computer Architecture*, pages 266–275, May 1991.

[67] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3):682–704, July 2000.

[68] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious Structure Definition. In *Conference on Programming Language Design and Implementation*, pages 13–24. ACM, May 1999.

[69] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array Regrouping and Structure Splitting Using Whole-program Reference Affinity. In *Conference on Programming Language Design and Implementation*, pages 255–266, June 2004.

[70] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4):1–23, January 2013.

[71] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Analytical Computation of Ehrhart Polynomials: Enabling More Compiler Analyses and Optimizations. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 248–258, 2004.

[72] Sven Verdoolaege, Kristof Beyls, Maurice Bruynooghe, and Francky Catthoor. Experiences with Enumeration of Integer Projections of Parametric Polytopes. In *International Conference in Compiler Construction*, pages 91–105, April 2005.

[73] Alexander I. Barvinok. A Polynomial Time Algorithm for Counting Integral Points in Polyhedra when the Dimension is Fixed. *Mathematics of Operations Research*, 19(4):769–779, November 1994.

[74] P.R. Panda and N.D. Dutt. Low-power memory mapping through reducing address bus activity. *IEEE Transactions on VLSI Systems*, 7(3):309–320, September 1999.

[75] Chidamber Kulkarni, Francky Catthoor, and Hugo De Man. Advanced Data Layout Optimization for Multimedia Applications. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 186–193, May 2000.

[76] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. System-scenario-based Design of Dynamic Embedded Systems. *ACM Transactions on Design Automation Electronic Systems*, 14(1):3:1–3:45, January 2009.

[77] Michael Z. Spivey. A Generalized Recurrence for Bell Numbers. *Journal of integer sequences*, 11, November 2008.

[78] Ying Zhao and S. Malik. Exact memory size estimation for array computations without loop unrolling. In *Design Automation Conference*, pages 811–816, June 1999.

[79] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica*, 48(1):37–66, March 2007.

[80] Preeti Ranjan Panda and Nikil D. Dutt. 1995 High Level Synthesis Design repository. In *International Symposium on System Synthesis*, pages 170–174, September 1995.

[81] N. Ahmed, T. Natarajan, and K.R. Rao. Discrete Cosine Transform. *IEEE Transactions on Computers*, C-23(1):90 –93, January 1974.

[82] IEEE 802.11-03/940r4. IEEE P802.11 Wireless LANs*, TGn Channel Models*, May 2004.

[83] Weiping Li and Ya-Qin Zhang. Vector-based signal processing and quantization for image and video compression. *Proceedings of the IEEE*, 83(2):317 –335, February 1995.

[84] John W. Woods. *Subband Image Coding*. Kluwer Academic Publishers, Boston, 1991.

[85] D. Cescato and H. Bolcskei. Algorithms for Interpolation-Based QR Decomposition in MIMO-OFDM Systems. *IEEE Transactions on Signal Processing*, 59(4), April 2011.

[86] A. Maltsev, V. Pestretsov, R. Maslennikov, and A. Khoryaev. Triangular systolic array with reduced latency for QR-decomposition of complex matrices. In *International Symposium on Circuits and Systems*, page 4, May 2006.

[87] Kuang-Hao Lin, R.C. Chang, Chien-Lin Huang, Feng-Chi Chen, and Shih-Chun Lin. Implementation of QR decomposition for MIMO-OFDM detection systems. In *IEEE International Conference on Electronics, Circuits and Systems*, August -September 2008.

[88] Jochen Rust, Frank Ludwig, and Steffen Paul. Low complexity QR-decomposition architecture using the logarithmic number system. In *Design, Automation Test in Europe Conference Exhibition*, pages 97–102, March 2013.

[89] Yin-Tsung Hwang and Wei-Da Chen. A low complexity complex QR factorization design for signal detection in MIMO OFDM systems. In *IEEE International Symposium on Circuits and Systems*, pages 932–935, May 2008.

[90] C.K. Singh, Sushma Honnavara Prasad, and P.T. Balsara. VLSI Architecture for Matrix Inversion using Modified Gram-Schmidt based QR Decomposition. In *International Conference on VLSI Design*, pages 836–841, January 2007.

[91] Alan George, Joseph W Liu, and Ng Esmond. *Row-Ordering Schemes for Sparse Givens Transformations*. Elsevier, 1984.

[92] Min-Woo Lee, Ji-Hwan Yoon, and Jongsun Park. High-speed tournament givens rotation-based QR Decomposition Architecture for MIMO Receiver. In *IEEE International Symposium on Circuits and Systems*, pages 21–24, May 2012.

[93] Marc Hofmann and Erricos John Kontoghiorghes. Pipeline Givens sequences for computing the QR decomposition on a EREW PRAM. *Parallel Computing*, 32(3):222–230, January 2006.

[94] N. Park, Bo Hong, and V.K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 14(7):640–654, July 2003.

[95] Ji-nan Leng, Lei Xie, Hui-fang Chen, and Kuang Wang. A novel 3780-point fft processor scheme for the time domain synchronous ofdm system. *Journal of Zhejiang University SCIENCE C*, 12(12):1021–1030, 2011.

[96] J. Chen and H. Sorensen. An efficient FFT algorithm for real-symmetric data. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 5, pages 17–20, March 1992.

[97] P. Duhamel. Implementation of "Split-radix" FFT algorithms for complex, real, and real-symmetric data. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34(2):285–295, April 1986.

[98] H.V. Sorensen, M. Heideman, and C.S. Burrus. On computing the split-radix FFT. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34(1):152–156, February 1986.

[99] R. Radhouane, P. Liu, and C. Modlin. Minimizing the memory requirement for continuous flow FFT implementation: continuous flow mixed mode FFT (CFMM-FFT). In *IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 1, pages 116–119, May 2000.

[100] David T. Harper. Block, multistride vector, and FFT accesses in parallel memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):43–51, January 1991.

[101] H. Sorokin and J. Takala. Conflict-free parallel access scheme for mixed-radix FFT supporting I/O permutations. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1709–1712, May 2011.

[102] J.H. Takala, T.S. Jarvinen, and H.T. Sorokin. Conflict-free parallel memory access scheme for FFT processors. In *International Symposium on Circuits and Systems*, volume 4, pages 524–527, May 2003.

[103] D. Reisis and N. Vlassopoulos. Conflict-Free Parallel Memory Accessing Techniques for FFT Architectures. *IEEE Transactions on Circuits and Systems I*, 55(11):3438–3447, December 2008.

[104] J. Markel. FFT pruning. *IEEE Transactions on Audio and Electroacoustics*, 19(4):305–311, December 1971.

[105] D.P. Skinner. Pruning the decimation in-time FFT algorithm. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 24(2):193–194, April 1976.

[106] T.V. Sreenivas and P. Rao. FFT algorithm for both input and output pruning. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 27(3):291–292, June 1979.

[107] R.G. Alves, P. L. Osorio, and M.N.S. Swamy. General FFT pruning algorithm. In *IEEE Midwest Symposium on Circuits and Systems*, volume 3, pages 1192–1195, 2000.

[108] Linkai Wang, Xiaofang Zhou, G.E. Sobelman, and Ran Liu. Generic Mixed-Radix FFT Pruning. *IEEE Signal Processing Letters*, 19(3):167–170, March 2012.

[109] Yingtao Jiang, Ting Zhou, Yiyan Tang, and Yuke Wang. Twiddle-factor-based FFT algorithm with reduced memory access. In *International Parallel and Distributed Processing Symposium (IPDPS)*, page 8, April 2002.

[110] E. Brockmeyer, C. Ghez, J. D'Eer, F. Catthoor, and H. De Man. Parametrizable behavioral IP module for a data-localized low-power FFT. In *IEEE Workshop on Signal Processing Systems*, pages 635–644, October 1999.

[111] Mohamed Mostafa Sabry Aly. *Multi-Level Optimization Methodologies For Thermally Reliable Multi-Core Architectures*. PhD thesis, EPFL, Lausanne, Switzerland, 2013.

Publications

1. *Array Interleaving – An Energy Efficient Data Layout Transformation*, Namita Sharma, Preeti Ranjan Panda, Francky Catthoor, Praveen Raghavan, Tom Vander Aa, *Accepted for publication in ACM Transactions on Design Automation of Electronic Systems (TODAES).*

2. *Power Optimization Techniques for DDR3 SDRAM*, Preeti Ranjan Panda, Vishal Patel, Praxal Shah, Namita Sharma, Vaidyanathan Srinivasan, *VLSI Design & Embedded Systems Conference (VLSID) 2015*.

3. *High Level Energy Modeling of Controller Logic in Data Caches*, Preeti Ranjan Panda, Sourav Roy, Srikanth Chandrasekaran, Namita Sharma, Jasleen Kaur, Sarath Kumar Kandalam and Nagaraj N. , *Great Lakes Symposium on VLSI (GLSVLSI) 2014*.

4. *Energy Efficient Data Flow Transformation for Givens Rotation Based QR Decomposition*, Namita Sharma, Preeti Ranjan Panda, Min Li, Prashant Agrawal, and Francky Catthoor, *Design, Automation and Test in Europe (DATE) 2014*.

5. *Energy Optimization in Android Applications through Wakelock Placement*, Faisal Alam, Preeti Ranjan Panda, Nikhil Tripathi, Namita Sharma, and Sanjiv Narayan, *Design, Automation and Test in Europe (DATE) 2014*.

6. *Array Scalarization in High Level Synthesis*, Preeti Ranjan Panda, Namita Sharma, Arun Kumar Pilania, Gummidipudi Krishnaiah, Sreenivas Subramoney, and Ashok Jagannathan, *Asia and South Pacific Design Automation Conferenc (ASP-DAC) 2014*.

7. *Data Memory Optimization in LTE Downlink*, Namita Sharma, Tom Vander Aa, Prashant Agrawal, Praveen Raghavan, Preeti Ranjan Panda, and Francky Catthoor, *International Conference on Acoustics, Speech, and Signal Processing (ICASSP) 2013*.

8. *Early Exploration for Platform Architecture Instantiation with Multi-mode Application Partitioning*, Prashant Agrawal, Praveen Raghavan, Matthias Hartmann, Namita Sharma,

Liesbet Van der Perre, and Francky Catthoor, *Design Automation Conference (DAC) 2013*. Awarded the HiPEAC paper award.

9. *Energy Aware Task Scheduling for Soft Real Time Systems using an Analytical Approach for Energy Estimation*, Namita Sharma, Vineet Sahula, and C.P. Ravikumar, *International Journal of Advanced Studies in Computers, Science and Engineering (IJASCSE) Dec 2012*.

10. *Two-Dimensional Analytical Device Modeling of a Novel Multi-Layered Four-Gate MOSFETA Novel Design*, N.Sharma, R.S.Gupta, J.Bansal, R.Chaujar, and M.Gupta, *Recent Advances in Microwave Theory and Applications (MICROWAVE) 2008*.

11. *Two-Dimensional Analytical Sub-threshold model of double gate MOSFET with gate stack*, J.Bansal, N.Sharma, R.Chaujar, M.Gupta, and R.S.Gupta, *Recent Advances in Microwave Theory and Applications (MICROWAVE) 2008*.

12. *Analytical Modeling of Multi-Layered dielectric four Gate MOSFET for improved short channel effects*, N.Sharma, J.Bansal, S.P.Kumar, R.Chaujar, M.Gupta, and R.S. Gupta, *Mathematical Techniques: Emerging Paradigms for Electronics and IT Industries (MATEIT) 2008*.

Biography

Namita Sharma holds a Bachelors (2003-06) and Masters degree (2006-08) in Electronics from University of Delhi, South Campus, India. She received Masters degree in VLSI Design (2008-2010) from Malaviya National Institute of Technology, Jaipur, India.

She joined the Department of Computer Science and Engineering, IIT Delhi as a PhD student in July 2010. Her research interests include Embedded Systems Design, Low Power Design, Modeling and Estimation, High Level Synthesis and Device Modeling.