Efficient High-level Synthesis Implementation of massive MIMO Processing on RFSoC

SIJIA CHENG

MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Efficient High-level Synthesis Implementation of massive MIMO Processing on RFSoC

Sijia Cheng
`si4168ch-s@student.lu.se`

Department of Electrical and Information Technology
Lund University

Supervisor: Liang Liu, Steffen Malkowsky

Examiner: Erik Larsson

March 21, 2022

© 2022
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Massive multiple-input multiple-output (MIMO) refers to a wireless access technology that equips base station (BS) with hundreds to thousands of antennas to serve tens of user equipment (UE) in the same time-frequency resource. These extensive antennas improve spectral and energy efficiency, but the detection algorithms tend to be more complex with operations, multiplications, and inversions on larger size matrix.

The traditional register transfer level (RTL) design process is time-consuming and risks starting over if the proposed architecture does not meet the requirements. High-level synthesis (HLS) addresses this issue by employing a higher level of abstraction and providing an error-less path to generate the RTL code from user-defined architecture. However, more attention is needed during implementation as coding at a too high level might deteriorate the design quality, leading to area overhead and down the throughput.

In this thesis, an efficient HLS implementation of massive MIMO processing is demonstrated and optimized for higher throughput and less area occupation. The design is written in C++ and synthesized by Mentor Catapult HLS. Firstly, the baseline implementation with all default settings is synthesized and simulated, and then loop and memory optimization is applied. The result shows that correct coding style and well-designed constraints improve the performance to a large extent.

Popular Science Summary

To meet the demanding user expectation on network capability in the 5G era, engineers adopted a fundamentally new approach to communicate between users and their base station. Compared with the multiple-input multiple-output (MIMO) technology employed in 4G, the new solution utilizes much more antennas in the base station, and that's why it's named "massive MIMO".

Why does the number of antennas on the base station side increase from 4G to 5G? The designer needs to enhance the receiving antenna power for more reliable transmission. You may think of increasing transmitted power. But it is regrettable that due to technical limitations and related regulations, designers cannot increase the transmitting power infinitely; also, the antenna gain is limited by current technology. You may also suggest placing the transmitter and receiver closer. Mobile communication carriers won't want to do this because it will cost more money to build new base stations. Thanks to talented engineers who came up with the "beamforming" concept. It is a solution that adaptively adjusts the radiation graph of the antenna array according to a specific scene. Metaphorically speaking, a signal antenna transmission is like an electric bulb that lights up the whole room, while beamforming is like a flashlight where the light can be intelligently converged to the target location. And also, the number of flashlights can be constructed according to the number of targets. The more antennas in a communication system, the more obvious beamforming can play.

However, this improvement does not come without a price. For the hardware engineer, more base station antennas mean more register-level operations. Hence, the frequently used register transfer level (RTL) language programming is time-consuming and complex. This issue is solved with the help of high-level synthesis (HLS) that can transform the C++ code to RTL code. In this thesis, a massive MIMO processing system is implemented with HLS and further optimized to have a faster and smaller design.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today is the era of wireless communication. To satisfy the increased demand for reliable performance, high data rate, low latency, massive multiple-input multiple-output (MIMO) is one of the most promising solutions, which features hundreds to thousands of antennas incorporated with the base station. By utilizing a high number of antennas to multiplex messages for several devices on each time-frequency resource, massive MIMO can focus the radiated energy and minimize the inter-cell interference [1], consequently providing extra capacity and wireless reliability systems.

Due to the increased number of antennas, uplink massive MIMO detection algorithms tend to be more complex with operations, multiplications, and inversions on larger size matrix. The deployment of the base station requires significant architecture change, which is timing consuming and resource costly due to the low-level characteristics of the widely used hardware description language (HDL). Fast prototyping helps to some extent, where it allows automatically generating the microarchitecture by setting synthesis directives. High-level synthesis (HLS) allows the implementation of a complex system from a higher level of abstraction, lowering the barrier between hardware design and software design. The development time is reduced significantly without making low-level design decisions.

While HLS dramatically reduces the design productivity gap, non-negligible problems arise. Coding at a too high level might deteriorate the design quality, leading to significant area overhead and bringing down the throughput. How to program the source code and determine the synthesis constraints largely influence the final implementation performance. This project suggests some rules for coding, for example, merging loops together. In the meanwhile, it updates the constraints to optimize the microarchitecture of an uplink MIMO detection system. The final implementation has 517 times larger throughput than the baseline for processing a data vector.

Chapter 2

Background

2.1 Massive MIMO

Massive MIMO makes a clean break with other antenna systems by using an enormous excess transmitter and receiver antennas and spatial multiplexing. It promises significant gains that can accommodate more users at higher data rates with better reliability while consuming less power [2]. With the term of massive, a massive MIMO system implies the utilization of hundreds to thousands of base station antennas simultaneously serving many tens of user devices in the same time-frequency resource. These extensive antennas provide extra degrees of freedom in the spatial domain, thus significantly increasing the signal-to-noise ratio, reliability and coverage [3].

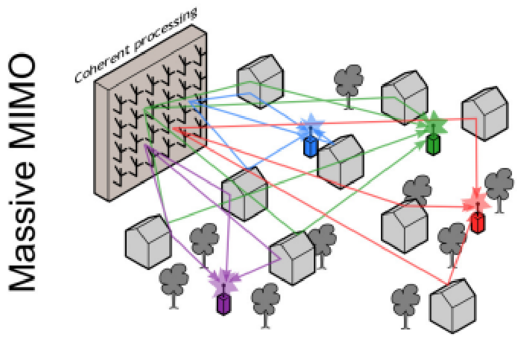


Figure 2.1: Massive MIMO. Artwork by Ove Edfors [4]

The transmission between the base station and UEs is shown in Figure 2.1 multiple independent data streams are transmitted simultaneously. In the uplink transmission, the streams generated from UEs are received by the base station, while the direction is reversed in downlink transmission. Information bits to be sent are first encoded and mapped to a constellation, then modulated before sending out by antennas. During the transmission, the electromagnetic waves experience various effects, like attenuation and phase shift [4]. The stream received is formed by a sequence of OFDM symbols in most cases, where the OFDM uses the (fast) discrete Fourier transform to decompose a frequency selective channel into many orthogonal parallel channels called subcarriers; see Figure 2.2 [5]. To avoid the inter symbol interface (ISI) generated by multi-path channel, cyclic prefix is formed by replicating part of the back of orthogonal frequency division multiplexing(OFDM) symbol to the front. For each subcarrier during uplink transmitting, the baseband data received by base station is the UE data with channel modulation information in this frequency region.

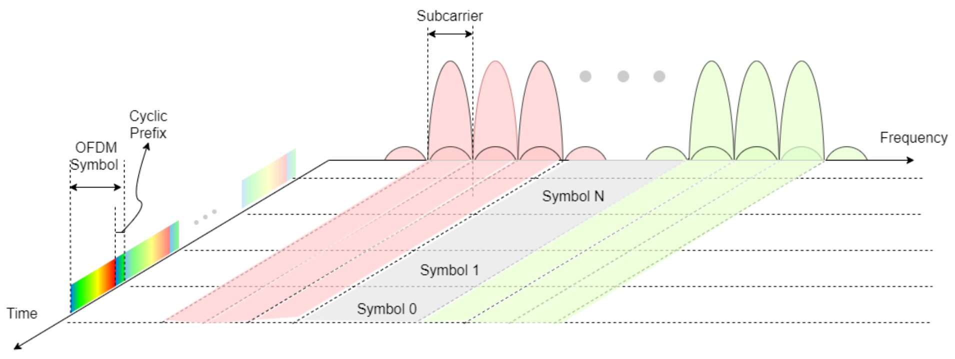


Figure 2.2: Time-frequency domain view of OFDM symbols

2.2 Uplink Detection Algorithm

Massive MIMO relies on spatial multiplexing, where multiple antennas at both the UEs and base station are used to carry multiple data streams simultaneously within the same frequency band [6]. To convert the base station received data vectors back to UE data, the base station has to know what the transmission channel is for every user station and every subcarrier before the data symbol transmission. A time-division duplexing (TDD) protocol shown in Figure 2.3 is suggested to address this issue. During one subframe slot, the channel is assumed to be constant. At the beginning of every subframe, UL estimation is performed during the uplink pilot symbol by sending out known pilots, then use it to estimate the channel modulation effect in the following data symbols [7].

Assume M antennas are employed in the base station(BS) and serve K UEs. The channel modulation effect is represented by a complex $M \times K$ matrix $\mathbf{H}$. Consider the pilot is transmitting in subcarrier0, the information received by BS antennas is :

$$\mathbf{y}_0 = \mathbf{H}_0 \mathbf{S}_{p_0} + \mathbf{n}_0 \tag{2.1}$$

In Equation 2.1, $\mathbf{y}_0$ is the complex $M \times 1$ antenna received vector, and $\mathbf{S}_{p_0}$ is the pilot data transmitting in K UEs, which is a complex $K \times 1$ transmit signal

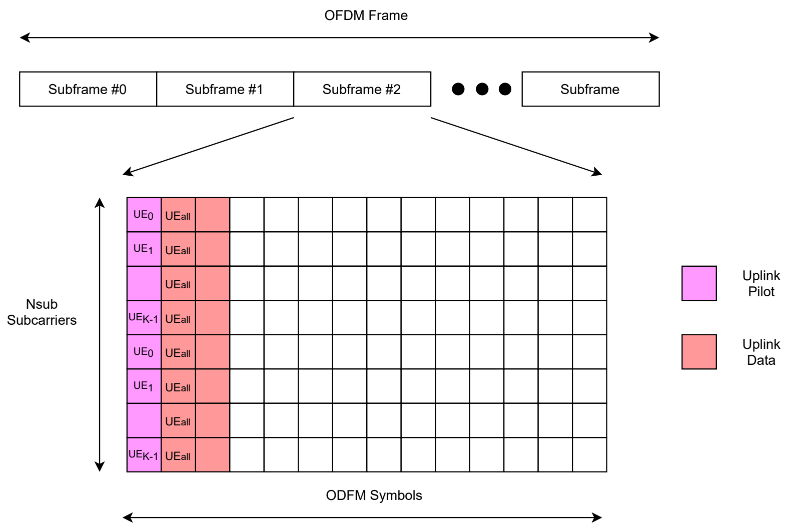


Figure 2.3: OFDM frame structure

vector. $\mathbf{H}_0$ corresponds to the channel gain in subcarrier 0, and $\mathbf{n}_0$ is the noise in subcarrier0 channel.

$$\mathbf{H}_0 = [\mathbf{h^0}, \mathbf{h^1}, \mathbf{h}^{\cdots}, \mathbf{h^{k-1}}] \tag{2.2}$$

$\mathbf{h^0}$ is the channel gain for user 0 in subcarrier 0, $\mathbf{h^1}$ is that for user 1 and so on. Different subscript corresponds to different subcarriers, while different superscript corresponds to different UEs.

Each UE transmits pilot on K-th subcarrier with the first UE starting at subcarrier 0, the second at subcarrier 1, etc, overall utilizing a full OFDM symbol [4]. The channel gain for every K subcarriers for same UE are assumed to be identical. With this assumption, K subcarriers share the same $\mathbf{H}_0$ matrix, which saves the operation numbers and also the memory locations, and the overall performance does not deteriorate by a lot.

Only UE^0 is sending out data in subcarrier0, thus, only the first element is non zero in $\mathbf{S}_{p_0}$.

$$\mathbf{S}_{p_0} = \begin{bmatrix} p_0 \\ 0 \\ \dots \\ 0 \end{bmatrix}_{Kx1} \tag{2.3}$$

And the received y adapts to

$$\mathbf{y}_0 = \mathbf{H}_0\mathbf{S}_{p_0} = \mathbf{h}^0\mathbf{S}_{p_0} \tag{2.4}$$

If the magnitude of $\mathbf{p}_0$ is 1, then $\mathbf{h^0}$ equals to $\mathbf{y_0}$ multiples with conjugation of $\mathbf{p}_0$.

$$\mathbf{h^0} = \frac{\mathbf{y_0}}{p_0} \xrightarrow{|p_0=1|} = \mathbf{y_0} p_0^{'} \tag{2.5}$$

Same treatment is applied to remaining K-1 subcarriers' vector to form a **H** matrix. After got the channel estimation matrix **H**, channel pre-processing manipulates **H** to find the detection matrix. The obtained **H** is further inversed to obtain the detection matrix, so that every received data vector multiplies with one over **H** matrix. A zero forcing detector is employed to calculate the pseudo-inverse of **H** matrix as shown in equation below, and the result is named as detection matrix $\mathbf{W}_{det}$.

$$\mathbf{H}^{\dagger} = (\mathbf{H}^H \mathbf{H})^{-1} \mathbf{H}^H = \mathbf{W}_{det} \tag{2.6}$$

For the received data vector, all UEs are transmitting on all subcarriers, and the channel is assumed to be noiseless.

$$\mathbf{y} = \mathbf{H}\mathbf{S}_d \tag{2.7}$$

The detection matrix $\mathbf{W}_{det}$ is used to extract the transmitted data back based on the received data vector $\mathbf{y}$. The estimated UE data $\hat{\mathbf{S}}_d$ is calculated in Equation 2.8, and equals to original data $\mathbf{S}_d$ with noiseless channel ($\mathbf{n} = 0$ in Equation 2.1)

$$\hat{\mathbf{S}}_d = \mathbf{W}_{det}\mathbf{y} = (\mathbf{H}^H \mathbf{H})^{-1} \mathbf{H}^H \mathbf{y} = (\mathbf{H}^H \mathbf{H})^{-1} (\mathbf{H}^H \mathbf{H}) \mathbf{S}_d = \mathbf{S}_d \tag{2.8}$$

2.3 Challenge in Implementing Uplink MIMO Detection

Since massive MIMO forces the use of more antennas on the base station side, there are many issues to be addressed to make the system realistic.

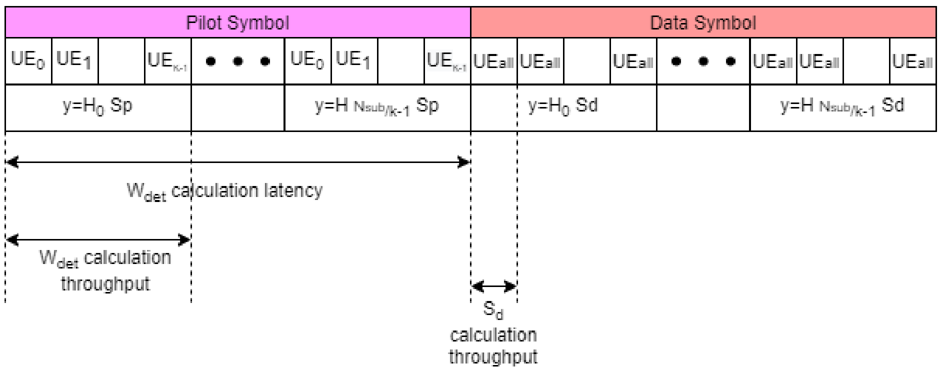

Pilot Symbol
Data Symbol
UE0 UE1 UEK-1 • • • UE0 UE1 UEK-1
UEall UEall UEall • • • UEall UEall UEall
y=H0 Sp
y=H Nsub/K-1 Sp
y=H0 Sd
y=H Nsub/K-1 Sd
Wdet calculation latency
Wdet calculation throughput
Sd calculation throughput

Figure 2.4: Latency and throughput requirements

2.3.1 Throughput Requirement

To avoid data accumulation in the anterior block, the uplink MIMO detection implementation is supposed to have comparable processing speed with vector receiving. The throughput requirement shown in Figure 2.4 depends on the number of subcarriers employed and symbol duration time. For example, the throughput requirement is 36 MSample/s if the system parameters in Table 2.1 is selected and 600 subcarriers are employed. The OFDM symbols are arriving at the speed of 60 kSample/s and each symbol is carrying 600 vectors.

Table 2.1: Example system parameters

Parameters	Value
Bandwidth	50 MHz
Sampling frequency	61.44 MSample/s
Subcarrier spacing	60 kHz
OFDM symbol length	16.7 μs

2.3.2 Latency Requirement

Because of the TDD feature, the system forwards a requirement for processing latency. The detection matrix of a particular subcarrier needs to be calculated by the received pilot vector before its corresponding data vector is transmitted to the base station. Usually, the latency requirement is approximate to one symbol duration as shown in Figure 2.4. If the system parameter is set to values in Table 2.1, the latency requirement is supposed to be 16.7 μs.

2.3.3 Design Flexibility

In addition to the time constraints, the challenge comes from ensuring the flexibility of actual implementation. For example, the number of antennas may change in later use, and this design has to work smoothly at that time. It is not just changing the number of antennas, but some other parameters need to be updated to match the changes, such as, unrolling factor, pipelining factor, memory size. Therefore, in a flexible design, these parameters are the ones that should be modified automatically.

2.4 Different Implementation Methods

Nowadays, most projects start with functional specifications. Usually, an executable model is created with high-level languages, for example, SystemC. Once validated, this behavioral model will be further developed to actual hardware implementation with specific architecture. Different architecture leads to various consequences on throughput, latency, and hardware utilization. While the functionality defines 'what' the design implements, the architecture determines 'how' the design implements it.

2.4.1 Hardware Description Languages

Using hardware description language(HDL) to implement often requires longer development. This is because the programmer needs to make all low-level decisions manually, such as explicitly connecting wires between modules and specifying cycle-by-cycle hardware behavior. After the final optimal architecture is determined, the designer can start coding on register level in the forms of Verilog or VHDL programming and experiencing writing bugs and debugging back and forth. To validate the functionality, an testbench written in HDL is employed once the design source code is determined. Due to the low level feature of HDL, the designer can have a general idea of timing performance and hardware utilization during programming. As a result, the final implementation can meet all requirements in most cases.

2.4.2 High Level Synthesis

The source code for HLS is written in high level language, like C++, that is easy to program for most of designers. After the executable model is proven to be working, the HLS tool develops corresponding RTL code from abstract specifications with user-defined architecture through an error-less path. Therefore, the verification is usually executed before RTL generation, with a high level testbench. The time it consumes in programming and HDL generation is negligible compared to directly coding with HDL. These advantages of HLS promotes the development of new tools, such as Vivado HLS (Xilinx), Catapult C (Mentor Graphics), and Intel OpenCL SDK (Intel) [8].

However, the consequence of letting the software do most of the work is that the designer loses fine control over the resulting hardware. It's hardly to get the same level of optimization and precision with HLS as direct HDL implementation. Too many designer find the translated hardware result is either missing the timing constraint or too large to fit in chips or boards, or even both. The quality of synthesized design is dependent on the coding style and chosen architecture, where finding the ones that lead to satisfying result can be difficult.

Chapter 3

System Architecture

This chapter discusses different system architectures for further implementation. The basic architecture is illustrated with necessary execution steps by Figure 3.1 with two things under-determined. Firstly, before writing channel information to memory, what degree of completion should be for the pre-processing block. Secondly, how to partition different steps into blocks.

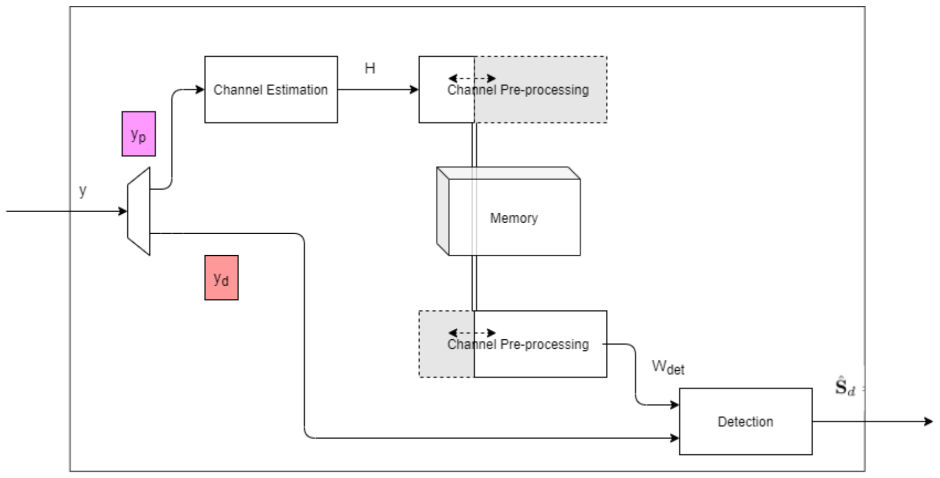


Figure 3.1: Basic architecture and all executions

3.1 Real-time Computing v.s. Keeping in Memory

As described in Section 2.1, massive MIMO separates different signals by time division multiplexing, therefore, a memory block is necessary to keep the calculated channel modulation effect. The problem arises during selecting what to be stored. Theoretically, all the data sets holding same information as channel estimation matrix **H** can be candidates, and they differ in required calculation complexity and memory occupation.

By taking a closer look at Equation 2.6 for channel pre-processing, the first option is storing $\mathbf{H}$ matrix directly, and do the pseudo-inversion in real time as receiving data vectors. One other candidate for storing is the detection matrix $\mathbf{W}_{det}$, and the channel pre-processing is performed right after got $\mathbf{H}$ matrix. The third choice is storing the intermediate calculation result: $(\mathbf{H}^H\mathbf{H})^{-1}$and $\mathbf{H}^H$.

The massive MIMO algorithm for channel pre-processing and detection is rewritten in Equation 3.1 and 3.2, respectively.

$$\mathbf{W}_{det} = \mathbf{H}^{\dagger} = (\mathbf{H}^H\mathbf{H})^{-1}\mathbf{H}^H \tag{3.1}$$

$$\hat{\mathbf{s}} = (\mathbf{H}^H\mathbf{H})^{-1}\mathbf{H}^H\mathbf{y} \tag{3.2}$$

This process can be broken into four steps:

1. Multiplication of $K \times M$ matrix $\mathbf{H}^H$ with $M \times K$ matrix $\mathbf{H}$.
2. Inversion of a $K \times K$ size matrix.
3. Multiplication of $K \times K$ matrix $(\mathbf{H}^H\mathbf{H})^{-1}$ with $K \times M$ matrix $\mathbf{H}^H$.
4. Multiplication of $K \times M$ matrix $(\mathbf{H}^H\mathbf{H})^{-1}\mathbf{H}^H$ with $M \times 1$vector $\mathbf{y}$.

The third and forth steps exchange if the intermediate matrix are stored:

$3^{(2)}$. Multiplication of $K \times M$ matrix $\mathbf{H}^H$ with $M \times 1$vector $\mathbf{y}$.
$4^{(2)}$. Multiplication of $K \times K$ matrix $(\mathbf{H}^H\mathbf{H})^{-1}$ with $K \times 1$vector $\mathbf{H}^H\mathbf{y}$.

Based on the operation complexity for every step provided in Table 3.1, the

Table 3.1: Operation complexity for every processing step

	Step1	Step2	Step3	Step4	Step3$^{(2)}$	Step4$^{(2)}$
Complexity	$\mathcal{O}(MK^2)$	$\mathcal{O}(K^3)$	$\mathcal{O}(MK^2)$	$\mathcal{O}(MK)$	$\mathcal{O}(MK)$	$\mathcal{O}(KK)$

corresponding operation numbers for one matrix set and the needed memory size are concluded as Table 3.2.

The number of antennas at base station (M) is always larger than that of UEs

Table 3.2: Required operation numbers and memory occupation for three architectures

To be stored	Memory size	Operation number
$\mathbf{H}$	MK	$\mathcal{O}(K^3 + 2MK^2 + MK)$
$\mathbf{W_{det}}$	MK	$\mathcal{O}(K^3 + 2MK^2 + MK)$
$(\mathbf{H}^H\mathbf{H})^{-1}$, $\mathbf{H}^H$	$K(M+K)$	$\mathcal{O}(K^3 + MK^2 + MK + KK)$

(K), thus, the overall complexity for storing two matrix is smaller than storing one. However, this architecture is not adopted in this thesis work because of it's extra memory usage. Storing $\mathbf{H}$ or $\mathbf{W}_{det}$ will behave same in uplink detection implementation. In this work, $\mathbf{W}_{det}$ is stored in memory for reusing it in downlink pre-coding. In detail, the matrix to pre-code the downlink data transmitted to UEs is just the transpose of matrix $\mathbf{W}_{det}$.

3.2 Optimizing Block Partition

Since all sub-blocks are running with parallelism implicitly, it is better to split the whole design into separate blocks for higher throughput. However, overly fine splitting granularity will deteriorate the design performance, as data sharing between blocks will be the bottleneck.

The first partition choice is integrating all process in one large block, which simply bypasses the data sharing problem. If the throughput can be satisfied by carefully controlling the microarchitecture, the coding process will be much easier. However, a realistic requirement mentioned in Section 2.3.1 is 36MHz, which implies an new output should be available every 5 clock cycles if the system is running at a frequency of 200MHz. Achieving that high throughput costs large amount of optimizing effort, and that's why a finer partition is expected.

The execution for pilot and data symbol is unrelated and without data transmission, so that it's better to not keep them together. The channel estimation and pre-processing after receiving a pilot vector are implemented in one sub-block to avoid transmission of matrix $\mathbf{H}$. Another drawback for separating them is duplicated hardware utilization, as each block needs memory resource to keep local array for $\mathbf{H}$. The detection process after receiving a data vector is another block, as only memory reading and matrix multiplication is performed. It's meaningless to split it finer.

One more block is added for distinguishing the input vectors. After the preceding OFDM demodulation block, the input vector arrives in the subcarrier order. Then, the flow control block will demultiplex the vectors to different symbols. Channel estimation and pre-processing block is responsible for processing pilot symbol's vector, while detection block carries out manipulation for data symbol's vector.In this thesis work, the uplink MIMO detection system is divided into three main calculation blocks and one memory resource as shown in Figure 3.2.

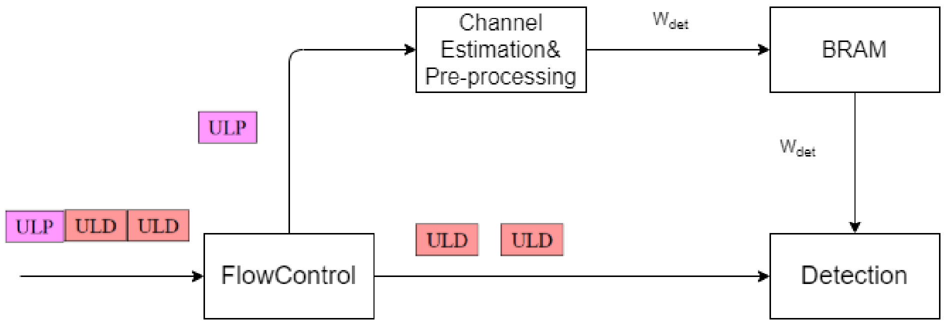


Figure 3.2: Block Diagram

3.3 Scheduling

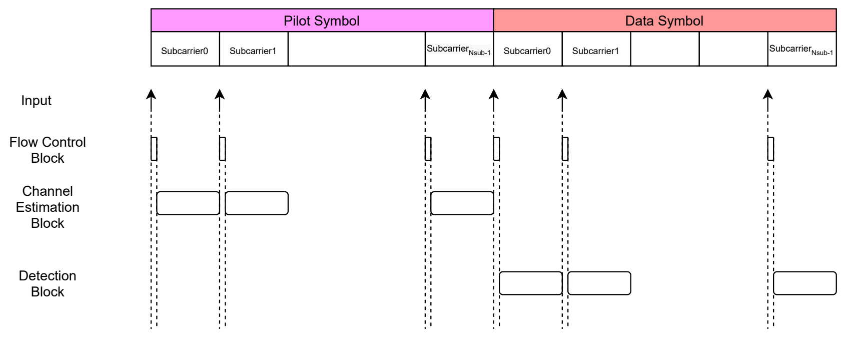


Figure 3.3: Ideal schedule

Proper scheduling of processing blocks is necessary when combining them. The prior FFT executor transforms the analog signals received by BS antennas to data in the frequency domain to be read in the order of subcarrier. An ideal schedule is shown in Figure 3.3 that the throughput is high enough to process all subcarriers during one symbol without pipelining. After every input vector is read into the system, the flow control block transmits it to channel estimation&pre-processing or detection block based on its symbol type.

Chapter 4

Catapult Synthesis Flow

Catapult HLS from Mentor Graphics is used for high-level synthesis in this thesis work. This section goes through the Catapult design flow to conclude some guidelines on the macro-architecture of uplink processing implementation by explaining how the tool transforms C++ code to RTL implementation ready for simulation and gate-level synthesis. All tasks are listed in the Task Bar (shown in the blue background in Figure 4.1), where each of them corresponds to particular stages in the Catapult workflow.

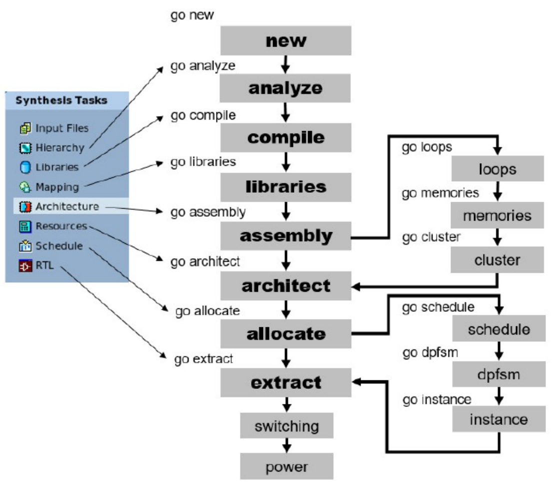


Figure 4.1: Catapult design flow

4.1 Compiling the Design

After correct setting the working directory and path of Catapult integrated tools, the next step is compile the design into Catapult. The following list outlines the steps in this process:

1. Adding source files. (Go new)

2. Analyzing the code. (Go analyze)
3. Compiling the design. (Go compile)

4.1.1 Go New

User imports source code files to the design in this step, and the tool will implicitly add their included header files. The high level language testbench is also added to verify the functionality.

4.1.2 Go Analyze

The hierarchy task will analyze the files imported to the current project and find functions candidates for hierarchical blocks in C++. There are three settings to identify these functions: Top, Block, or Inline. One and only one function is supposed to be designated as Top, and this function is the one that will be called in the testbench. The Block designation leads to a sub-block under the top block. Different sub-blocks can run in parallel to split the top block into stages and pipelined. The inline setting results in executions inside the hierarchy blocks, which will be executed sequentially if the sub-block is not explicitly pipelined.

C++ is a sequential execution language and does not naturally support hierarchical structures. However, Catapult enables the transforming from a flat design into a hierarchical one with the help of *hls_ design_ pragma* added before the function name.

4.1.3 Go Compile

This command compiles the design to generate the synthesis internal format database, which keeps the project's current state and can be used to restore it.

4.2 Building the Architecture

To determine the micro-architecture for every processing entity in the design, the following tasks are supposed to be executed:

1. Specifying the libraries. (Go libraries)
2. Adding clock and reset signal and assembling design. (Go assembly)
3. Determining the design architecture. (Go architect)

4.2.1 Go Libraries

By telling Catapult what RTL synthesis tool, vendor, and technology will be used to characterize the design, the timing and area estimations for hardware implementation components are appended to the design. Scheduling and performance evaluation in future steps are from these values.

4.2.2 Go Assembly

This call assembles the design from bottom to top and adds clock and reset signal to all parts if the input files are written in C++. It verifies if the connections between blocks are correct and realizes the interface for every sub-block and the interconnections in the top block. The data transfer between sub-blocks or between top-block and testbench is realized by *ac_channel*. It allows Catapult to properly synchronize the data between blocks by being synthesized to a FIFO pipe. After the design is successfully assembled, Catapult suggested architecture constraints are listed in GUI, and users can evaluate and modify them in GUI or with directives.

4.2.3 Go Architecture

The architecture building process has three steps: loop transformations, memory mapping, and cluster pattern characterizing, respectively. All the loops are left rolled by default, and the user can design the micro-architecture by unrolling, pipelining, and merging them. The arrays in source code will be mapped into RAMs/ROMs or split into registers if the number of elements is smaller than **MEM_MAP_THRESHOLD** value. For *go cluster* command, Catapult goes through the design and searches pre-defined cluster patterns, such as adder trees, multiply-add, and squares. These data path operators will be clustered together to reduce the inefficiencies with fine-grained scheduling. However, clustering does not help much in FPGA design due to the dedicated MAC and adder resources. These three steps set architectural constraints, and *go architecture* command applies them to the design.

4.3 Hardware Resource Allocation

Once functions, loops, and arrays have been determined by hierarchy constraints, loop controlling, and memory architecture, respectively, Catapult can start the real synthesis process, including scheduling operations, generating data path finite state machine(dpfsm), and then instance binding.

During scheduling, Catapult converts a series of operations to data flow graphics (DFG) according to the sequential order and data dependency. Then it maps the behaviors in DFG onto states in the dpfsm using estimates for delays. After that, Catapult starts assigning specified hardware instances to all operations. The actual timing cannot be calculated until this step completes. If timing violations are identified, new instances will be generated as needed without changing the fsm from the scheduler. After all synthesis steps are completed, Catapult cleans up the design and writes the netlist file.

Chapter 5

HLS Implementation and Optimization

This thesis work aims to implement the uplink massive MIMO detection algorithms effectively with HLS in Catapult. The primary objective is to have a functional C++ source code for MIMO uplink processing and a corresponding testbench to check it. After ensuring the code's correctness, the next step is baselining the architecture to provide a basis for further optimization.

5.1 Baseline Implementation

5.1.1 Source Code Formulation

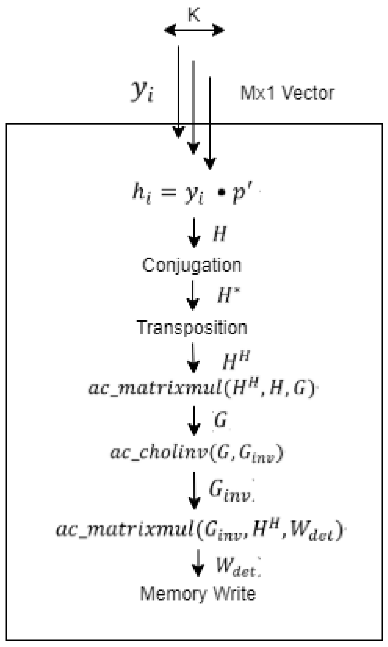


Figure 5.1: Channel Estimation&Pre-processing Block Implementation

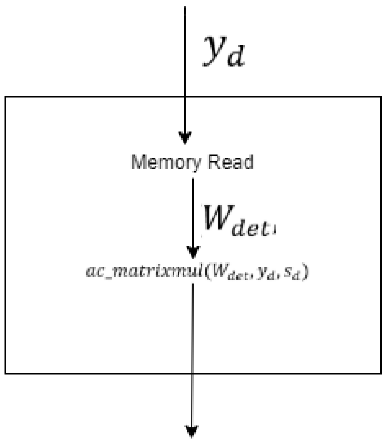


Figure 5.2: Detection Block Implementation

The flow control block splits the input vector stream into two paths. If it belongs to a pilot symbol, channel estimation&pre-processing block performs the calculation flow as shown in Figure 5.1. The first channel estimation part is active for every pilot vector while the remaining channel pre-processing part is only active for every K vectors. The matrix multiplication and Cholesky inversion function are provided in libraries. If a data vector is received, then the process shown in Figure 5.2 will be executed by detection block.

To verify the functionality for the high-level implementation, a testbench written

Table 5.1: Selected parameters for implementation.

Parameters	Value
K	4
M	32
$\mathbf{N}_{sub}$	600

in C++ can be employed. By generating random channel information in MATLAB, the testbench can know what symbols should be received by the base station for a selected series of UE data. And these symbols are then sent into top design in testbench and compared if the output generated is identical to the designated UE data. The parameters selected are listed in Table 5.1.

The source testbench supplies input vectors to the uplink MIMO processing design and captures the output vector for the expected correct numeric values to ensure functional correctness. After that, the Catapult synthesized model is verified by re-running the same tests for the source code and verifying the RTL simulation matches the source behavior. As the source code is written in C++, the testbench executes as untimed code. The SCVerify can generate wrappers, synchronization signals, and make-files to compile the source design. After the C++ model is proven to be functional, these outputs will be used to self-check the synthesized model by comparing equivalence between the C++ and RTL for test vectors provided in the testbench.

5.1.2 Baseline Implementation Result

The clock frequency for Catapult synthesizing is 200MHz. After simulating Catapult generated RTL result, the timing performance is concluded as follows:

• Latency for processing not the k_{th} received pilot vector: 810 ns
• Latency for processing the k_{th} received pilot vector($\mathbf{W}_{det}$ matrix calculation included): 41 620 ns
• Average throughput for received pilot vector: $\frac{K}{(K-1)*810\,\mathrm{ns}+41\,620\,\mathrm{ns}} = 22.7\,\mathrm{kSample/s}$
• Latency for processing not the k_{th} received data vector: 3875 ns
• Latency for processing the k_{th} received data vector (memory reading included): 5815 ns
• Average throughput for received data vector: $\frac{K}{(K-1)*3875\,\mathrm{ns}+5815\,\mathrm{ns}} = 57.3\,\mathrm{kSample/s}$

Table 5.2: Baseline implementation utilization report

Resource	Utilization	Available	Utilization(%)
LUT	8860	274080	3.23
FF	11860	548160	2.16
BRAM	30	912	3.29
DSP	32	2520	1.27

To get a view of hardware resource utilization, the Catapult generated RTL file is then synthesized and implemented using the VIVADO 2020.1 with the default options. The hardware utilization report is listed in Table 5.2, and the corresponding occupation percentage of total resource is attached if employing xczu9eg-ffvb1156-1-e RFSoc board. It's obvious that the processing speed is far from enough while there are still many unemployed hardware entities. Hence, kinds of optimization methods are applied to trade resources for timing performance.

5.2 Loop Optimization

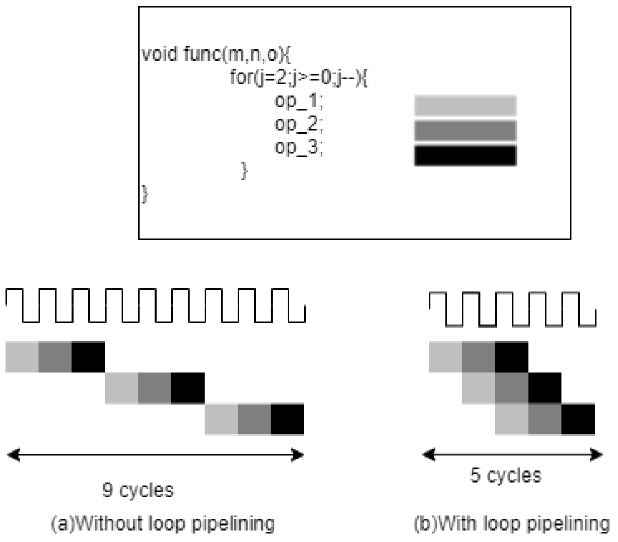
Controlling how Catapult will carry out the loops in hardware implementation is an essential step in Catapult. The functionality specifications determine the number of loop iterations, but there are still three dimensions that the user can control to optimize the design.

5.2.1 Loop Pipelining

In timing behavior, a new loop iteration only starts after the previous one has finished. However, by enabling loop pipelining, the new iteration can be executed certain clock cycles after the last one started instead of waiting it to be finished. The number of clock cycles taken before starting a new iteration is named initialization interval (II). In other words, loop pipelining means that adjacent iterations of the loop overlap and run concurrently, and multiple hardware are active to execute different stages of the loop-body [9].

Before illustrating the benefits of pipelining a loop, two criteria for evaluating the timing performance of a loop are introduced first. They are latency and throughput. The latency refers to the time between the input is fed into the design, and the corresponding output is ready for reading, while the throughput means the rate of output production.

Take the code piece in Figure 5.3 as an example; the latency for a single it-



```
void func(m,n,o){
        for(j=2;j>=0;j--){
            op_1;
            op_2;
            op_3;
        }
}
```

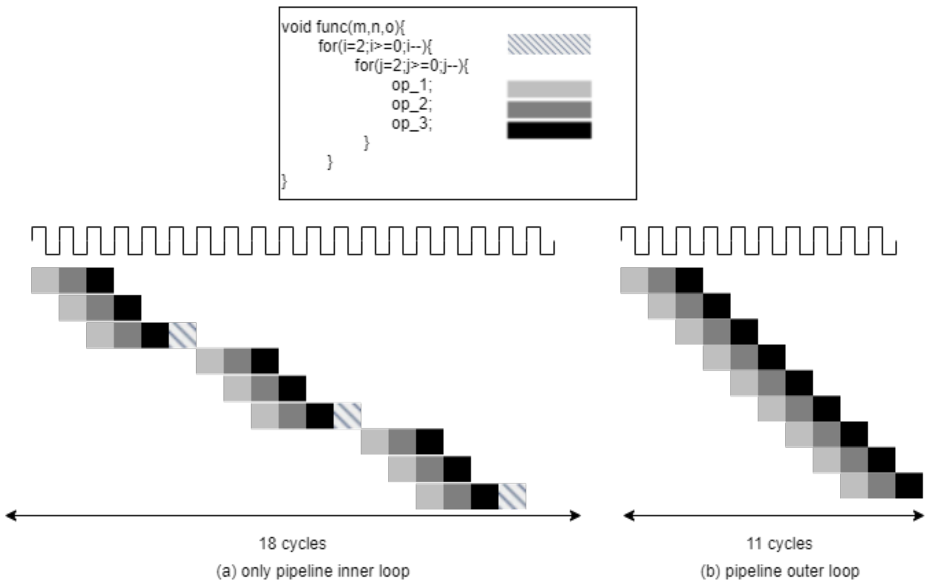
Figure 5.3: Loop execution: without \ with loop pipelining

eration is three cycles, and the whole loop iterates three times. Assume that each operation takes one clock cycle, and the color block indicates that the corresponding hardware is under execution. The second iteration can only start in the fourth cycle without loop pipelining, and the last one starts in the seventh cycle. The loop has not finished until the ninth cycle, which implies the loop latency is nine cycles. If loop pipelining is applied with II=1, causes the second iteration initiation occurs in the second cycle, and the last iteration starts in the third cycle. In total, all the operations are completed after five cycles; thus, the latency is reduced by four compared with the unpipelined design. This example demonstrates how loop pipelining improves timing performance without introducing replicated hardware resources.

Pipeline loops have several problems associated with dependency, and they are generally called hazards. The most common hazard in this design is data hazards, and it occurs when the current operation requires the result of a preceding instruction, but that data is still under calculation [10]. Data hazards between iterations may cause subsequent schedule steps to fail. In the example demonstrated above, if op_1 needs the value from op_3 of the last iteration, scheduling pipelined loops will be impossible because op_1 is carried out even one cycle before op_3 from the last iteration being performed. This conflict can be solved by removing data dependencies or increasing the initialization interval.

If a loop exists inside the body of another loop, this structure is called a nested loop. Pipelining nested loops need more attention in high-level C++ synthesis. It is a good practice to start from pipelining the innermost loops and proceed towards outer loops, because of the more and more heavy extra control logic.

As shown in Figure 5.4, if nested loops are pipelined together, the loops will



```
void func(m,n,o){
      for(i=2;i>=0;i--){
            for(j=2;j>=0;j--){
                  op_1;
                  op_2;
                  op_3;
            }
      }
}
```

Figure 5.4: Loop execution: pipeline inner loop \ outer loop for nested loop

be flattened into a signal loop, where its number of iterations equals the actual execution times of the inner loop body, and the II constraint will be applied to the flattened loop. The overhead state for controlling outer loop iteration is omitted (indicated by diagonal stripes) and replaced by complex control logic incorporated in flattened loop body.

5.2.2 Loop Unrolling

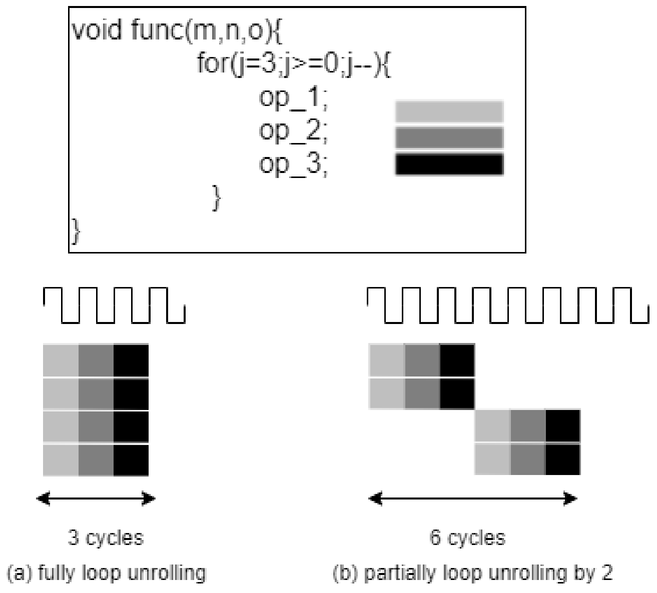


Figure 5.5: Loop execution: fully unrolled loop \ partially unrolled loop

By partially or completely performing multiple loop iterations in parallel, loop unrolling can achieve speedy implementation. Assume no data dependency between iterations so that every iteration can be executed from the start. In Figure 5.5(a), the hardware is replicated four times (iteration number) to execute all iterations simultaneously to unroll the loop fully. In Figure 5.5(b), the loop is unrolled by a factor of 2. As a result, the hardware is duplicated by 2, and the iteration number is halved.

Data dependencies between loop iterations also deteriorate the degree of timing optimization gained from loop unrolling. If one iteration requires the operation result from a previous iteration, it cannot be scheduled until that operation is completed, which leads to wasted hardware.

Unrolling the innermost loop replicates the inner loop body and leaves the outer loop unchanged for nested loops. Unrolling the outer loop without unrolling the inner loop results in the inner loop being replicated as many times as the outer loop is unrolled.

5.2.3 Loop Merging

For a rolled loop, each iteration takes at least one clock cycle when scheduling, because there is an implied ‘wait until clock’ for the loop body [11]. So, it’s common to merge loops into one to save iterations. Compared with the code on

```
Loop1: for (int i=0; i<4; i++) {             Merged_Loop: for (int i=0; i<4; i++) {
            op_1;                                          op_1;
            op_2;                                          op_2;
             }                                             op_3;
Loop2: for (int i=0; i<4; i++) {                           op_4;
            op_3;                                           }
            op_4;
             }
```

Figure 5.6: Code for loops with \ without merging

the left side of Figure 5.6, the two loops are merged into one, as shown on the right side, which leads to less iteration number.

5.2.4 Loop Optimized Implementation and Result

There is no loop in the flow control block. Based on Catapult generated schedule, the FSM for this block is formed by two states without feedback beyond state registers; as a result, to accomplish a higher throughput, the main function is pipelined with II=1.

The channel estimation block calls for special attention, which contains plenty of inline loops. As mentioned before, inline functions inside the block are executed sequentially if the main block is not pipelined. Pipelining the main function of a complex block is the last option since it leads to longer synthesis time, and the synthesis may fail because of the feedback path in the schedule. Hence, the latency for every loop should be as minimum as possible. However, fully unrolling a loop is also not encouraged. In addition to the larger area that will inevitably lead to, it also has the potential to deteriorate the overall timing performance. The reason is that if the loop is conditionally executed and it's fully rolled, then the scheduler will include its data path in the main function, and the condition is only checked for writing outputs after finishing the execution. Even if the condition is set to be false, the states are wasted anyway. Partially unrolling helps here since the iteration condition needs to be checked before starting a loop execution, while the outer condition will also be integrated into the judgment. Some principles are followed during optimization: the innermost loop for nested loop is fully unrolled, and the iteration number is kept small for higher throughput; the loops are pipelined if there are no data dependency violations. By manually restructuring the loops in the source code, the compiler can merge the conjugation and transposition loop to execute their loop body parallel.

For the detection block, only the outermost loop is kept rolled, and the main function is pipelined with II=1 to achieve a higher throughout during detection.

Loop operations like unrolling and pipelining are accompanied by memory design, as the function must access multiple addresses in the same cycle. So, the word length increases for corresponding memory to provide enough data resources for the port connection.

• Latency for processing not the k_{th} received pilot vector: 30 ns
• Latency for processing the k_{th} received pilot vector($\mathbf{W}_{det}$ matrix calculation included): 525 ns
• Average throughput for received pilot vector: $\frac{K}{(K-1)*30\,\mathrm{ns}+525\,\mathrm{ns}} = 6.5\,\mathrm{MSample/s}$
• Latency for processing not the k_{th} received data vector: 30 ns
• Latency for processing the k_{th} received data vector (memory reading included): 45 ns
• Average throughput for received data vector: $\frac{K}{(K-1)*30\,\mathrm{ns}+45\,\mathrm{ns}} = 29.6\,\mathrm{MSample/s}$

Compared with baseline implementation, the design after loop optimization has

Table 5.3: Loop optimized implementation utilization report

Resource	Utilization	Utilization(%)
LUT	45169	16.48
FF	33126	6.04
BRAM	37.5	4.11
DSP	1201	47.66

a better timing performance with the expense of higher hardware resource utilization. The area overhead comes from the complex control logic, extra registers and multiplexers from pipelining, and resource replication from unrolling [12].

5.3 Memory Optimization

The arrays in C++ code will be transformed to memories during synthesis, and in this step, Catapult specify the type of memory resource, such as RAM/ROM or registers. Different memory architectures are suitable for different data accessing patterns.

Take a 2 rows 3 columns matrix as an example, and assume it can be stored in a single memory. The address of each element is indexed sequentially by rows as shown in Figure 5.7(a). Only one entry can be accessed in every clock cycle without memory optimization. However, when the loop is unrolled or pipelined, multiple elements must be read or written simultaneously. There are three ways to increase the number of accessible elements [13].

Interleaving memory is the process of rearranging sequential data storage into many non-contiguous storage blocks. Reading from original continuous locations takes advantage of this design since the required data locations are distributed in

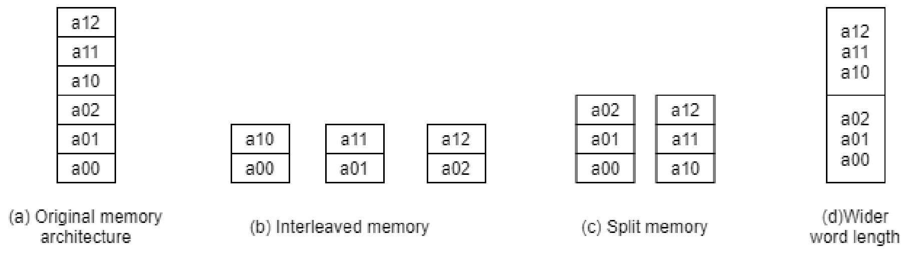


Figure 5.7: Different memory architectures

different memory banks. As shown in Figure 5.7(b), every row is physically spread out in distinct memories, and each memory has its ports for operations. Hence, a full row of a $\mathbf{M}\times\mathbf{K}$ matrix can be reached at the same time if the memory is interleaved by $\mathbf{K}$.

Splitting memory will divide a large memory into several blocks with a specified block size. The original continuous data locations are still kept in the same or neighboring memory bank. As shown in Figure 5.7(c), the original memory is split with block size 3, so that each memory bank holds one row of the matrix, and the elements in the same column is distributed in different banks so that they can be accessed at the same time. If the matrix size $\mathbf{M}\times\mathbf{K}$, then the memory should be split into $\mathbf{M}$ parts to allow simultaneous reading or writing of a column.

The memory can also be developed by changing the word length of elements stored in one location. In other words, the adjacent data can be merged and read or written together. In Figure 5.7(d), the number of bits for a word is increased by 3 times, so all data in one row can be stored as one word. Splitting the memory into blocks and interleaving it are different ways to split a memory resource into several new resources, which can be accessed individually. The way how elements are distributed among new resources differs. Reorganizing the memory, instead, still results in a single memory.

5.3.1 Memory Optimized Result

The timing performance is largely improved by properly designing the memory architecture since all needed data can be accessed simultaneously.

• Latency for processing not the k_{th} received pilot vector: 30 ns
• Latency for processing the k_{th} received pilot vector($\mathbf{W}_{det}$ matrix calculation included): 455 ns
• Average throughput for received pilot vector: $\frac{K}{(K-1)*30\,\text{ns}+455\,\text{ns}} = 7.34\,\text{MSample/s}$
• Latency for processing not the k_{th} received data vector: 30 ns
• Latency for processing the k_{th} received data vector (memory reading included): 45 ns

- Average throughput for received data vector: $\frac{K}{(K-1)*30\,\text{ns}+45\,\text{ns}} = 29.6\,\text{MSample/s}$

It's easy to understand the increment in BRAM resource, since either interleaving or splitting creates more memory blocks. Optimized memory architecture largely lower the usage of DSP resource. The reason why more DSP is utilized in previous design is more multiplexers occupy more LUT resource, and force some calculations are executed by DSP block.

Table 5.4: Loop and memory optimized implementation utilization report

Resource	Utilization	Utilization(%)
LUT	26425	20.58
FF	40142	7.32
BRAM	79	8.66
DSP	857	34.01

5.4 Optimization Summary

Table 5.5: Average throughput summary table

	Pilot vector	Data vector
Baseline	22.7 kSample/s	57.3 kSample/s
Loop optimized	6.5 MSample/s	29.6 MSample/s
Memory optimized	7.34 MSample/s	29.6 MSample/s

Table 5.6: Hardware resource utilization percentage summary table

	LUT	FF	BRAM	DSP
Baseline	3.23	2.16	3.29	1.27
Loop optimized	16.48	6.04	4.11	47.66
Memory optimized	20.58	7.32	8.66	34.01

By comparing the timing performance in Table 5.5 and hardware cost in Table 5.6 for baseline and loop optimized implementations,it is easy to draw the conclusion that employing many hardware in parallel largely improves the timing performance, creating higher overall throughout. Furthermore, memory optimization helps reducing the hardware utilization and accelerating the system, because of the well designed data-path.

Chapter 6

Conclusion and Future Work

6.1 Discussion and Conclusion

Catapult synthesized RTL result performs slightly worse than expected, the first bottleneck is memory control transfer between blocks.

> Theoretically, the throughput when the input vector belongs to the data symbol should be four since the main function of the detection block is pipelined with II=1, implying all the loops inside pipelined. Both of the loops have iteration number equal to 4. If no memory reading operation is needed for the currently received data vector, every iteration in the multiply loop can calculate one entry of an output vector, and the output vector will be available every four clock cycles. The RTL simulation takes more cycles than expected because of the shared memory. To ensure consistent reading and writing rates, extra logic circuit blocks the memory from the same type of operation after one memory read or memory write, prohibiting channel estimation and detection block from accessing memory simultaneously. The main function takes at least two clock cycles for the channel estimation block; after two clock cycles, this block release memory control, and the memory is available for detection block. That's where the two extra cycles come from. If I was coding in VHDL, the memory access operation could be carried out randomly at will. Catapult wants to ensure the same read/write rate to avoid struct accumulation in C++ simulation, but hardware loses flexibility.

Catapult's lack of flexibility is reflected in the default sequential execution schedule inside of blocks, too.

> Pipelining the block main function will automatically pipeline all loops; however, the data taken into main loop iteration is usually different from the data for the inline loop body. For example, a main function call deals with one vector, and a loop inside it only copes with one entry of this vector. In this situation, ideally, the initialization interval for the main loop should be the iteration number of the loop inside. However, this cannot be set independently in Catapult. Also, if the user wants to pipeline the

inline blocks inside of a CCS_BLOCK to be executed concurrently instead of sequentially, the only solution is to split them into blocks. As a result, complicated interconnections and data sharing problems arise.

The problems above can be treated as the by-product of high abstract level design. The scheduling and hardware allocation is replaced by software, then the loss of detail control is a matter of course. Despite this, high-level synthesis accelerates the whole design flow overall. Trying a new architecture is simple by just editing new constraints in GUI, and then a new solution is branched out so that the user can compare their performance to choose easily. Also, the automatic synthesis path avoids error introduction.

6.2 Further Work

If a more optimized implementation is wanted, designer can carry out some potential improvements in further work.

1. Data dependency in the Cholesky function prevents the loops in the channel estimation block from pipelining and limits the speed acceleration of unrolling. It's better to look into the function body and re-edit to remove the feedback path as much as possible.

2. Splitting the channel estimation block into finer sub-blocks can pipeline the sequential execution in the current design, but it leads to more programming time for complicated data sharing.

3. The further design can try to get rid of *ac_ channel*, and switch to use *ac_ sync* and *ac_ shared* to build and control a shared memory channel. This can simplify the design process, but the RTL file produced by Catapult may function differently from the C code.

References

[1] Federico Boccardi, Robert W. Heath, Angel Lozano, Thomas L. Marzetta, and Petar Popovski. Five disruptive technology directions for 5G. *IEEE Communications Magazine*, 52(2):74–80, 2014. doi=10.1109/MCOM.2014.6736746.

[2] 5G massive MIMO testbed: From theory to reality. `https://www.ni.com/sv-se/innovations/white-papers/14/5G-massive-MIMO-testbed--from-theory-to-reality--.html`.

[3] Erik G. Larsson, Ove Edfors, Fredrik Tufvesson, and Thomas L. Marzetta. Massive MIMO for next generation wireless systems. *IEEE Communications Magazine*, 52(2):186–195, 2014.

[4] Steffen Malkowsky. *Massive MIMO: Prototyping, Proof-of-Concept and Implementation*. PhD thesis, Lund University, 2019.

[5] Thomas L. Marzetta, Erik G. Larsson, Hong Yang, and Hien Quoc Ngo. *Models and Preliminaries*, page 19–44. Cambridge University Press, 2016.

[6] Xiang Liu. Chapter 3 - Challenges and opportunities in future high-capacity optical transmission systems. In John Zyskind and Atul Srivastava, editors, *Optically Amplified WDM Networks*, pages 47–82. Academic Press, Oxford, 2011.

[7] Steffen Malkowsky, Joao Vieira, Karl Nieman, Nikhil Kundargi, Ian Wong, Viktor Öwall, Ove Edfors, Fredrik Tufvesson, and Liang Liu. Implementation of low-latency signal processing and data shuffling for TDD massive MIMO systems. In *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 260–265, 2016.

[8] Roberto Millón, Emmanuel Frati, and Enzo Rucci. A comparative study between HLS and HDL on SoC for image processing applications. *Elektron*, 4(2):100–106, Dec 2020.

[9] Xilinx. *Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)*.

[10] M.S. Schmalz. Organization of computer systems: Pipelining. `https://www.cise.ufl.edu/~mssz/CompOrg/CDA-pipe.html`.

[11] Mentor. *Catapult® Synthesis User and Reference Manual.*

[12] G. Georgiou and G. Theodoridis. Studying the impacts of loop unrolling and pipeline in the fpga design of the simon and roadrunner lightweght ciphers. In *2021 10th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–6, 2021.

[13] Thomas Bollaert Mike Fingeroff. *High-Level Synthesis Blue Book.* Mentor Graphics Corporation.

Printed by Tryckeriet i E-huset, Lund 2022

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2022-860
http://www.eit.lth.se