POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

# On the Design of Multi-directional Systolic Arrays for Band and Generic Matrix-Matrix Multiplications

Tesi di Laurea Magistrale in
Electronics Engineering - Ingegneria Elettronica

Author: **Leonel Gouveia Ergin**

Student ID: 966275
Advisor: Prof. Christian Pilato
Co-advisor: Stephanie Soldavini
Academic Year: 2021-2022

# Abstract

For the most part of the 20th century, computers have benefited from transistor scaling in order to support exponential performance improvement. As computer-chip features get smaller, ever larger proportions of chips must be turned off during operation due to power budget limitations. This new obstacle calls for a shift in paradigm in computer architecture. As a consequence of this, we have moved from an era of single-core design in the 20th century, through an era of homogeneous multi-core design in the beginning of the 21st century to the now ever-expanding trend of heterogeneous multi-core architectures with custom accelerators. In this thesis, we explore multi-directional systolic accelerators for Band and Generic matrix-matrix multiplications (BMMM and GMMM). Starting from a systolic design introduced in the 1970's by Kung and Leiserson, we conceptualized changing the direction of some data paths in order to achieve more than one operation. We then implemented the design in Verilog and the necessary memory management hardware in `C++`, using HLS tools to compile it to hardware. To link the RTL and HLS designs together, we developed a hybrid design workflow using Xilinx tools. For comparison, we also implemented an equivalent fully-HLS kernel. Architecturally, our design achieves 20x performance improvement for many streamed 16x16 GMMM operations and a 610x performance improvement in large BMMM matrices with a band size of 31, while using 30x more DSP's than our best HLS counterpart. Using our own fully-parametric data management hardware, we have achieved performance parity for GMMM and a 23x performance improvement for BMMM. Our benchmarks were performed on a Alveo U280 Data Center Card containing a Virtex Ultrascale+ FPGA and High Bandwidth Memory.

**Keywords:** Heterogeneous Computer Architecture, Systolic Design, Xilinx Blackbox Hybrid RTL-HLS Design, Matrix Multiplications.

# Abstract in lingua italiana

Per la maggior parte del secolo XX, i computer hanno beneficiato della scalabilità dei transistor per sostenere un miglioramento esponenziale delle prestazioni. Man mano che le caratteristiche dei chip dei computer diventano più piccole, una percentuale sempre maggiore di chip deve essere spenta durante il funzionamento a causa dei limiti del budget energetico. Questo nuovo ostacolo richiede un cambiamento di paradigma nell'architettura dei computer. Di conseguenza, siamo passati da un'era di design single-core nel secolo XX, a un'era di design multi-core omogeneo all'inizio del secolo XXI, fino alla tendenza in continua espansione delle architetture multi-core eterogenee con acceleratori personalizzati. In questa tesi esploriamo acceleratori sistolici multidirezionali per le moltiplicazioni matriciali a banda e generiche (BMMM e GMMM). Partendo da un design sistolico introdotto negli anni 70 da Kung e Leiserson, abbiamo concettualizzato il cambio di direzione di alcuni percorsi di dati per ottenere più di un'operazione. Abbiamo quindi implementato il progetto in Verilog e l'hardware necessario per la gestione della memoria in `C++`, utilizzando gli strumenti HLS per la compilazione in hardware. Per collegare insieme i progetti RTL e HLS, abbiamo sviluppato un workflow di design ibrido utilizzando gli strumenti Xilinx. A titolo di confronto, abbiamo anche implementato un kernel equivalente completamente HLS. Dal punto di vista architetturale, il nostro progetto ottiene un miglioramento delle prestazioni di 20x per molte operazioni GMMM 16x16 in streaming e un miglioramento delle prestazioni di 610x in matrici BMMM di grandi dimensioni con una dimensione di banda di 31, pur utilizzando un numero di DSP 30x superiore rispetto alla nostra migliore controparte HLS. Utilizzando il nostro hardware di gestione dei dati completamente parametrico, abbiamo ottenuto la parità di prestazioni per GMMM e un miglioramento delle prestazioni di 23x per BMMM. I nostri benchmark sono stati eseguiti su una scheda Data Center Alveo U280 contenente un FPGA Virtex Ultrascale+ e una memoria ad alta larghezza di banda (HBM).

**Parole chiave:** Architettura di computer eterogenea, design sistolico, design ibrido RTL-HLS Xilinx, moltiplicazioni matriciali.

# Contents

# 1 | Introduction

## 1.1. Moore's Laws and Dark Silicon

Ever since we moved on from tubes to transistors, and then to more specifically to CMOS, we have been tracking many performance metrics of processors. These include transistor count, computational capacity and computational efficiency, to name a few. Gordon Moore, in his 1965 paper [13] noted an empirical trend which stated: "*The complexity [of integrated circuits] for minimum component costs has increased at a rate of roughly a factor of two per year*", where *complexity* referred to the number of components, not just the number of transistors. This statement was not meant to be technological, but more economical, noting a trend in cost of components over time. Later, in 1975, Moore corrected his observation with a doubling of complexity every two years [12], as opposed to one in his original paper. The famous statement "*computing performance doubles every 18 months*", often referred to as *Moore's law* is one that Moore himself never actually made, but has held its course quite accurately from 1975 to 2009. In fact, as is explained in [8], Moore's law is, in a sense, a benchmark of innovation to which engineers have tried to stick. This phenomenon renders the law a sort of *self-fulfilling prophecy*.[20]

Until the early 2000's, the increase of computing performance was largely obtained from the miniaturisation of transistors. Although this solution has has been the source of progress in the first era of computing, CMOS scaling is unfortunately *ceasing to provide the fruits it once did*, according to [20]. The rules of thumb regarding classical CMOS scaling are called Dennardian scaling. They state roughly that the power density of chips stays constant with miniaturisation of transistors. Meaning that by making chips smaller, we can achieve the same computational performance with a smaller amount of power. This is, of course, because in this regime, power is only proportional to the area of the circuit. We have benefited from decreasing the threshold voltages and operating voltages to obtain quadratic improvements to the energy efficiency each consequent generation in the Dennardian scaling regime.

Sadly, it has been more recently studied that, as features become smaller and smaller,

Dennardian scaling breaks down and leaves way to leakage-limited scaling. Indeed, we cannot continue to reduce the threshold voltages without dramatically increasing the leakage currents. This new scaling regime is often referred to as Post-Dennardian scaling. In this regime, the power density is no longer constant but begins to scale quadratically with process generation. Table 1.1 summarizes both regimes.

Table 1.1: Table summarising Dennardian and Post-Dennardian scaling, from [20]. $S$ represents ratios between minimum feature sizes of successive process generations. In Post-Dennardian scaling, the voltages no longer scale quadratically, causing the final power densities to increase quadratically.

| Transistor Property | Dennardian | Post-Dennardian |
|---|---|---|
| Transistor Count (Quantity) | $S^2$ | $S^2$ |
| Frequency | $S$ | $S$ |
| Capacity | $1/S$ | $1/S$ |
| $V_{dd}^2$ | $1/S^2$ | 1 |
| Power density $= QFCV^2$ | 1 | $S^2$ |

Due to Post-Dennardian scaling, the power densities in circuits are increasing each generation, forcing larger and larger portions of the circuit to be shut off or partially shut off to adhere to global power limitations. These portions are called Dark Silicon and Dim Silicon respectively[1]. A common misconception about Dark Silicon is that the term "dark" refers to areas of silicon that are unused or useless, or can not be used. Instead, it means that at any point, locally (spatially and temporally), on average, there must be a certain amount of the chip that must be dark in order not to exceed the thermal design power (TDP). During the days of non-dark design, a lot of circuits were already intrinsically dark-silicon friendly in the way that they were designed to be occasionally used. This is the case for some SIMD units of the x86 architecture and last level caches, which should only be occasionally accessed in normal operation.

Early model predictions state that Dark Silicon as a percentage of total area doubles every generation [7]. Following this trend would result in over 90% of chip area needing to be powered-off by 2020. State-of-the-art process generation ASIC designers must be fully aware of these limitations and take special care when crafting their designs.

Researchers from [7] note that the classical model predicts that 22nm process node chips would suffer from over 50% of Dark Silicon constraints, which has not been observed in practise. It is thus accepted that classical Dark Silicon predictions are a "worst-case" scenario and that these need to be revised in order to have more realistic constraints. In

[1]The term Dark Silicon seems to be more widely accepted and used than Dim Silicon in the literature.

[7], using their own prediction technique, they have observed that overall, for 16nm, 11nm and 8nm processes, we should more realistically expect as much as 20%, 30% and 40% of Dark Silicon respectively.

## 1.2. Techniques for Dark Silicon Taming

Even with more optimistic amounts of Bright Silicon, state-of-the-art commercial processors are currently being manufactured on 5nm process nodes[2], which is well into Dark Silicon territory. Designers must thus take extra care to take this into consideration when making chips. In the following section we will introduce some techniques that designers can employ to embrace Dark Silicon and use it to their advantage.

### 1.2.1. Shrinking

The most simple way to take advantage of the Dark Silicon era is to not take advantage of it at all. This can be achieved by simply shrinking existing devices to hope to get ahead with economical advantages of smaller chip sizes. Note that when fully in Dark Silicon territory, the increase of power density balances with the decrease of area and results in a chip which has similar overall power usage. The only remaining advantage is thus the possible savings from a smaller area. The chips that experience the most shrinkage will be the ones that cannot benefit considerably from the other techniques. Shrinking chips has its very obvious limit. At some point, the cost of silicon becomes negligible to the cost of the packaging process, testing, marketing, sales, support etc. When this point is reached, there is no more economical improvement to be made by shrinking.

A consequence of this is that, if a company relies on shrinking instead of making use of the Dark Silicon for other improvements, it risks to fall behind its competitors and the limited amount of price reduction offered by shrinking the chip compared to the potential of offering a larger, dark-optimised, chip will render the product noncompetitive in the market. Thus, the scenario of shrinking will probably only occur if there is no practical use for Dark Silicon.

### 1.2.2. Dimming

If we accept that we should populate the additional area, we must consider *Dim Silicon* as a possible solution. This term refers to general-purpose logic that is underclocked or used

[2]For example, Apple M2 SoC's or Huawei's HiSilicon Kirin 9000 models, the latter implementing ARM's new architecture called DynamIQ, the successor of the big.LITTLE architecture

infrequently. Multiple options are available to us when considering *Dim Silicon* designs.

The first consideration is employing a Near-Treshold Voltage (NTV) design. NTV design goes beyond Dynamic Voltage and Frequency Scaling (DVFS) by having specialised circuits that operate in this different, voltage-starved region. High-parallelizable workloads might benefit from running on more low-performance NTV cores than fewer high performance regular cores. NTV-design, however low in power budget, is unfortunately more susceptible to failure through process variability [4]. We can expect the overall fabrication yield to be affected by this.

Another technique for Dim Silicon design is to implement larger caches. Because of the power-hungriness of off-chip data accesses, many designers have proposed to consume most Dark Silicon expansion area with larger caches. This is specially advantageous for miss-intensive workloads. This philosophy is however less productive as larger on-chip memories become more common, acting almost like a last-level cache.

When discussing ideas for use of Dark Silicon, one could argue that filling dark area with reconfigurable logic might be a good idea. This idea stems from the "build now, design later" mindset. Considering that bit-level FPGA's are usually very power hungry since they need to power many interconnects and long wires, the obvious compromise is to use *Coarse Grain Reconfigurable Arrays* (CGRA). CGRA's are often useful in order to consume space to steer data through different, more optimised paths in order to achieve different operations. These paths are considered dim because of their occasional use, rendering them dim in time. CGRA's are nothing new in the grand scheme of computer architecture research but new paradigms often make designers reconsider old designs with fresh eyes.

Another temporal dimness technique is to use *Computational Sprinting* whenever needed. This technique consists of exceeding the thermal budget for short periods of time in order to achieve short bursts of high-performance, relying on thermal capacitance to keep the design within operational temperatures[3]. This technique is part of a more general approach called Dynamic Frequency and Voltage Scaling.

### 1.2.3. Material Change

All the considerations made in previous paragraphs are derived from our CMOS "addiction". Just like moving from vacuum tubes to integrated circuits opened up many opportunities and ideas, moving from CMOS to another process might be the solution to

[3]Intel's Turbo Boost technology relies on this approach

keep increasing performance. The Dark Silicon problem stems from the intrinsic, physical properties of MOSFET device physics. A few examples of innovations in material changes are Tunnel Field Effect Transistors [2], Nano-Electro-Mechanical switches [16], or even silicon photonics [9, 17] for interconnect purposes. For now, these design are still occupying the tables and blackboards of many research laboratories leaving us to remain in the current computing era, the Dark Silicon era.

### 1.2.4. Specialized Circuits

With Dark Silicon, the paradigms of computer architecture need to change if we want to keep up with Moore's "Target". "*Where once we spent silicon area to buy performance we must now spend silicon area to buy energy efficiency*" states Michael B. Taylor in [20]. Comparatively speaking, with each new generation, additional area becomes exponentially cheaper. A way to use this *cheap* area is to employ specialized co-processors, often called accelerators. The aim of these co-processors is to achieve specific operations either much faster[4] or much more efficiently, by using techniques discussed earlier, like NTV cores or simply using much more efficient architectures which, for instance, limit redundant transfers of memory. Execution would be passed around processors and run always on the most suited (all things considered) core. Unused logic would be shut-off when not in use to save global thermal budget. This strategy is already showing its fruits in modern architectures, where not only multiple general-purpose heterogeneous cores are employed but also specialized heterogeneous cores[5], whose function is restricted. Many researchers consider that in the future designs will employ more specialized cores than general-purpose ones. An instance of such an architecture can be found on [6].

An obvious obstacle to the trend of specialization is the elimination of the classical division of labour and expertise between software and hardware designers. The more specialized cores a system has, the more tailoring its software must undergo in order to function adequately. The proliferation of specialized hardware is harshly handicapped by the predicament of learning how to take advantage of it. Not only is it a bigger challenge for programmers to make use of specific hardware for specific tasks, but it is also a much bigger undertaking to write code that can run on many different architectures, each having different sets of accelerators. This is one of the reasons why, in the consumer market, only Apple has had success in moving away from the x86 architecture for its newest computers. They are one of the only market players with a large enough influence and following that

[4] in terms of number of cycles, taking advantage of speed to improve the energy efficiency

[5] Again, a famous example is the new Apple M2 chip, which employs 4 high-performance cores, 4 low-performance cores, a Graphics Processing Unit, Specialized Neural Network Hardware and Image Signal Processor.

they have the power to motivate software manufacturers to adapt and rewrite software suited to their architecture change[6]. Of course, we always live under the fantasy that the days of low-level programming are close to over, but in order to make this new ecosystem viable one of the biggest hurdles is to design compilers which will offload the appropriate workloads to the appropriate accelerators, in turn freeing the programmers from this burden. finally, the last hurdle to moving from a software-centered computing paradigm to a hardware-centered one, is the possibility of specialized hardware becoming obsolete with the update of data standards, giving an intrinsic expiration date to the hardware.

## 1.3. Design of Heterogeneous Accelerators

Given the importance of adapting to new paradigms of computing in the Dark Silicon Era, it is essential to discuss details of accelerator core design.

### 1.3.1. Manual Design

In order to manually design accelerators, we must start from top-down, large-scale architectural ideas. The building on the other hand occurs from the bottom-up, module by module, logic by logic, using any of the modern Hardware Description Languages (HDL) to implement the design. Manual design allows the most amount of granular control over the design but of course requires a large amount of expensive, hardware design engineering hands and hours. Given the cost of validating and verifying hardware blocks, manual design may be accelerated using pre-built and pre-validated blocks called IP blocks. Indeed, there is no use in reinventing the wheel when many components of modern systems are common.

### 1.3.2. Automatic and Semi-Automatic Design

A recent trend that is gaining weight year over year is the use of computers to automatically or semi-automatically design hardware from high-level specifications, using one of many high-level programming languages[7] to describe the desired functions.

Similarly to how assembly programming has become nearly obsolete thanks to the advances in compiler technology, many hope that in the future, High-Level Synthesis (HLS) design will be so effective that manually designing accelerators may be considered a waste

[6]They also facilitated the transition by writing their own just-in-time compiler (more precisely, dynamic binary translator) to run x86 software on their ARM-based chips. They marketed this product with the name *Rosetta* and the successor *Rosetta 2*.

[7]typically: C / C++ / SystemC

of time and money.

Semi-automatic design tries to capture the best of both worlds by having fine-grain control of crucial sections of the hardware but leaving the more mundane and simple tasks to be generated by computers, from high-level specifications.

## 1.4. Contributions of this Thesis

During the course of this thesis, we have noticed that computer architects have been considering systolic systems for a long time. We have however also noticed that, despite most of them working on the basis of processing elements which are often very similar if not identical, there seems to have been a lack of interest in studying multi-directional systolic arrays. The systems that we consider implement fixed arrays of processing elements and achieve different operations by rerouting the connections between them. As a working prototype, we have implemented a multi-directional array capable of two distinct operations, depending on the routing and marginally sized peripheral systems.

For this purpose, we have also put in place a complete workflow allowing us to stitch certain families of RTL kernels into HLS systems using tool kits developed by Xilinx. Our working prototype can be used as an example for researchers, designers and students who might want to interface custom RTL kernels with complete systems using RTL-HLS hybrid design.

## 1.5. Summary

In this chapter, we have explored the phenomenon of Dark Silicon and its implications on modern hardware design paradigms.

Before presenting the rest of our thesis, it is important that the reader familiarises theirself to many background concepts which will form the backbone of this thesis. The following section will encompass many of these concepts. The reader is however expected to have some additional background knowledge in modern mathematics, common programming and hardware description languages and computer architectures.

# 2 | Background

## 2.1. Tensors

### 2.1.1. Definition

In modern mathematical, computing and engineering applications, a tensor is a structure that organize data along many dimensions, which are commonly called orders. The modern tensor notation uses subscripts to identify elements of a tensor.

**Example** *Let the variable $a_{ijk}$ denote a third order tensor with subscripts $i$, $j$ and $k$ and size $9 \times 9 \times 9$. The element $a_{111}$ denotes the first element in every dimension. The element $a_{999}$ denotes the last element along every dimension. The element $a_{135}$ denotes an element which is first along the $i$'th dimension, third along the $j$'th dimension and fifth along the $k$'th dimension.*

### 2.1.2. Tensor Algebra

**Addition** Tensor addition is straightforward. Adding two tensors together corresponds to adding each corresponding element together.

**Example** *Let $a$ and $b$ be two distinct third order tensors. We can write:*

$$c = a + b \triangleq a_{ijk} + b_{ijk} = c_{ijk} \quad \forall\ i,\ j, and\ k$$

**Contraction** Tensor contraction is the most interesting of tensor operators. We say that it contracts two tensors into another tensor. The size of the contracted dimensions between both input tensors must match in order for the contraction to be a valid operation.

**Example** *Let $a$ be a third order tensor and $b$ be a second order tensor. Let the $k$'th dimension of tensors $a$ and $b$ be of the same size. We can write:*

$$c_{ijm} = \sum_k a_{ijk} b_{km} \quad \forall\ i,\ j,\ \text{and}\ m$$

Note that under Einstein's notation for tensor contractions, we can omit the summation symbol for ease of reading.

$$c_{ijm} = a_{ijk} b_{km} \triangleq \sum_k a_{ijk} b_{km} \quad \forall\ i,\ j,\ \text{and}\ m$$

**Familiar special cases** Note that matrices and vectors can be generalized as first and second order tensors respectively. Thus, matrix-vector and matrix-matrix multiplications are simply special cases of tensor contractions with low order tensors.

### 2.1.3. Tensor Computations

#### Tensor Addition

Tensor addition is usually not a concern because it can be fully parallelized and it benefits linearly from additional computing elements. For every doubling of computing elements, there should be a doubling of speed in tensor addition considering infinite memory bandwidth. In terms of memory access, each memory element must only be accessed once. This makes tensor addition a cheap and straightforward operation.

#### Memory Access Requirements of Matrix-Matrix Multiplications

When computing a matrix multiplication of the type $C = AB$ ($c_{ij} = a_{ik} b_{kj}$), when we compute for instance $c_{11}$ we need to access the first row of $a$ as well as the first column of $b$. So in other words we need to access $a_{1x}$ and $b_{x1}$. The problem is that every one of these elements needs to be accessed many times and at different times. If we do the computation naively, i.e. we compute every element of $c_{ij}$ in order, starting from $c_{11}$, $c_{12}$, $c_{13}$... then $c_{21}$, $c_{22}, c_{23}$ and so on, we will have to access the $a_{1x}$ vector $n$ times, given $c$ is of size $n \times n$. Note that this is true for every single row and column. As matrix sizes grow, it becomes increasingly impossible to store every necessary vector in cache. Accessing external memory and scrapping the same cached elements multiple times becomes unavoidable, leading to increased stalling of the processor and a drastic reduction of speed. In modern artificial intelligence systems, the biggest bottleneck is the memory access requirements.[19, 24]

## Modern Parallel Algorithms for Matrix-Matrix Multiplications

**Naive divide and conquer** This algorithm takes a N-by-N matrix and transforms it into 8 N/2-by-N/2 matrix multiplications and 4 matrix additions.

$$\left(\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right) * \left(\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right) = \left(\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right)$$

The complexity that this algorithm achieves is $O(n^3)$ [3], which is the same as the Naive implementation.

**Strassen** This algorithm takes the previous divide and conquer and optimises it in order to save one matrix multiplication with the expense of additional matrix additions. The complete Strassen algorithm can be seen below:

$$\left(\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right) * \left(\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right) = \left(\begin{array}{c|c} p_5+p_4-p_2+p_6 & p_1+p_2 \\ \hline p3+p_4 & p_1+p_5-p_3-p_7 \end{array}\right)$$

$$\text{with} \quad \begin{cases} p_1 = a(f-h) \\ p_2 = (a+b)h \\ p_3 = (c+d)e \\ p_4 = d(g-e) \\ p_5 = (a+d)(e+h) \\ p_6 = (b-d)(g+h) \\ p_7 = (a-c)(e+f) \end{cases}$$

The Strassen algorithm achieves a theoretical complexity of $O(n^{2.81})$ [3]. However, this performance is far from begin trivial to achieve because of real world problems like need for inter-core communication and load balancing.

***Communication optimised Parallel Algorithm for Strassen's matrix multiplication (CAPS)*** is a state-of-the-art algorithm that ensures minimal communication needs among processors and outperforms every previous naive and Strassen-like implementation before it. The details of the implementation are out of the scope of this thesis and can be consulted in [3].

### Large Matrix Division Scheme

For matrices larger than a given hardware multiplier size, the operation must be partitioned into manageable blocks. Luckily, this can be done quite easily with a linear algebra based algorithm.

$$C_{ij} = \sum_k A_{ik} B_{kj} \quad \text{for each} \quad i, j \quad \text{with} \quad X_{yz} \quad \text{a N-by-N matrix}$$

We can observe that a large number of matrix sums are required but luckily they can easily be sped up with a very simple SIMD[1] adder array or more realistically computed using the host computer's SIMD hardware.

Special care must be taken for the side portions of large matrices because they might not be of size N-by-N. To solve this, we use zero-padded matrices to have complete operations. Luckily, this is very straightforward. We show an example of this operation in Figure 2.1, note that $\left(\begin{smallmatrix}1&2\\4&5\end{smallmatrix}\right)\left(\begin{smallmatrix}1&2\\4&5\end{smallmatrix}\right) + \left(\begin{smallmatrix}3&0\\6&0\end{smallmatrix}\right)\left(\begin{smallmatrix}7&8\\0&0\end{smallmatrix}\right) = \left(\begin{smallmatrix}30&36\\66&81\end{smallmatrix}\right)$.

$$\left[\begin{array}{cc:cc} 1&2&3&0\\4&5&6&0\\\hdashline 7&8&9&0\\0&0&0&0\end{array}\right] \cdot \left[\begin{array}{cc:cc} 1&2&3&0\\4&5&6&0\\\hdashline 7&8&9&0\\0&0&0&0\end{array}\right] = \left[\begin{array}{cc:cc} 30&36&42&0\\66&81&96&0\\\hdashline 102&126&150&0\\0&0&0&0\end{array}\right]$$

Figure 2.1: Example of a 3x3 matrix multiplication as part of a 4x4 zero-padded matrix divided into $N = 2$-sided slices

## 2.2. Modern Programming Languages

In their simplest form, modern programming languages are a set of high-level instructions made to be readable and writable by humans. They are intended to be compiled into binary machine code and ran on general-purpose processors. There are several possible levels of abstraction in programming languages. Some languages, like C, are intended to be relatively close to the hardware, whereas others are intended to be abstractions with a specific purpose, like Matlab, which is a high-level language with a focus on matrix operations. Some languages are intended to be of very high-level and decouple almost completely the concept of programming from the concept of underlying hardware, like Python. Popular features of very high-level languages include but are not limited to:

[1]Single Instruction Multiple Data. This refers to hardware which is capable of executing a single instruction on an array of input data.

- Packaging common operations into open-source or commercial libraries.
- Syntax to describe parallel instructions using many forms of threading mechanisms.
- Object-oriented abstractions.
- Variable type malleability.
- Syntax to offload compatible computations to auxiliary computing chips, most popularly GPU's.

## 2.3. Hardware Description Languages

Hardware Description Languages (HDL), although visually similar to programming languages, are completely different in nature. They are used to describe combinational, latched and sequential logic circuits in a high-level, human-readable way. This family of languages describe the circuits in a level of complexity called Register-Transfer Level (RTL). In this level, we break down the circuits into banks of clocked registers with combinational logic between them. The most popular HDL are Verilog, VHDL and SystemVerilog. Knowledge in one of these three transfers quite elegantly to the others. VHDL is the most strongly typed language between them. It favours thoroughness over ease of reading. On the other end of the scale, SystemVerilog is the most weakly typed of the group, and includes many Object-Oriented features.

## 2.4. High-Level Synthesis

High-Level Synthesis (HLS) tools are intended to accelerate the design of hardware. These tools compile sets of instructions written in high-level modern programming languages (usually C-based), into hardware which yields equivalent results, usually presented in one of the main HDL's. Ideally, the equivalent circuits can achieve faster performances than a normal processor executing these tasks due to the freedom of implementing more ALU's, increasing spatial parallelism and optimising the data paths between ALU's and local memories, attempting to ensure fast, unobstructed access to data during execution. There are many commercial HLS tools in use today, each offering their own strengths and weaknesses. Some are intended only for research purposes while others are intended for real-world deployment. Figure 2.2 presents an overview of these tools.

| Status | Compiler | Owner | License | Input | Output | Year | Domain | TestBench | FP | FixP |
|---|---|---|---|---|---|---|---|---|---|---|
| In Use | eXCite | Y Explorations | Commercial | C | VHDL/Verilog | 2001 | All | Yes | No | Yes |
| | CoDeve-loper | Impulse Accelerated | Commercial | Impulse-C | VHDL Verilog | 2003 | Image Streaming | Yes | Yes | No |
| | Catapult-C | Calypto Design Systems | Commercial | C/C++ SystemC | VHDL/Verilog SystemC | 2004 | All | Yes | No | Yes |
| | Cynthesizer | FORTE | Commercial | SystemC | Verilog | 2004 | All | Yes | Yes | Yes |
| | Bluespec | BlueSpec Inc. | Commercial | BSV | SystemVerilog | 2007 | All | No | No | No |
| | CHC | Altium | Commercial | C subset | VHDL/Verilog | 2008 | All | No | Yes | Yes |
| | CtoS | Cadence | Commercial | SystemC TLM/C++ | Verilog SystemC | 2008 | All | Only cycle accurate | No | Yes |
| | DK Design Suite | Mentor Graphics | Commercial | Handel-C | VHDL Verilog | 2009 | Streaming | No | No | Yes |
| | GAUT | U. Bretagne | Academic | C/C++ | VHDL | 2010 | DSP | Yes | No | Yes |
| | MaxCompiler | Maxeler | Commercial | MaxJ | RTL | 2010 | DataFlow | No | Yes | No |
| | ROCCC | Jacquard Comp. | Commercial | C subset | VHDL | 2010 | Streaming | No | Yes | No |
| | Synphony C | Synopsys | Commercial | C/C++ | VHDL/Verilog SystemC | 2010 | All | Yes | No | Yes |
| | Cyber-WorkBench | NEC | Commercial | BDL | VHDL Verilog | 2011 | All | Cycle/ Formal | Yes | Yes |
| | LegUp | U. Toronto | Academic | C | Verilog | 2011 | All | Yes | Yes | No |
| | Bambu | PoliMi | Academic | C | Verilog | 2012 | All | Yes | Yes | No |
| | DWARV | TU. Delft | Academic | C subset | VHDL | 2012 | All | Yes | Yes | Yes |
| | VivadoHLS | Xilinx | Commercial | C/C++ SystemC | VHDL/Verilog SystemC | 2013 | All | Yes | Yes | Yes |
| N/A | Trident | Los Alamos NL | Academic | C subset | VHDL | 2007 | Scientific | No | Yes | No |
| | CHiMPS | U. Washington | Academic | C | VHDL | 2008 | All | No | No | No |
| | Kiwi | U. Cambridge | Academic | C# | Verilog | 2008 | .NET | No | No | No |
| | gcc2verilog [45] | U. Korea | Academic | C | Verilog | 2011 | All | No | No | No |
| | HercuLeS | Ajax Compiler | Commercial | C/NAC | VHDL | 2012 | All | Yes | Yes | Yes |
| Abandoned | Napa-C | Sarnoff Corp. | Academic | C subset | VHDL/Verilog | 1998 | Loop | No | No | No |
| | DEFACTO | U. South Cailf. | Academic | C | RTL | 1999 | DSE | No | No | No |
| | Garp | U. Berkeley | Academic | C subset | bitstream | 2000 | Loop | No | No | No |
| | MATCH | U. Northwest | Academic | MATLAB | VHDL | 2000 | Image | No | No | No |
| | PipeRench | U.Carnegie M. | Academic | DIL | bitstream | 2000 | Stream | No | No | No |
| | SeaCucumber | U. Brigham Y. | Academic | Java | EDIF | 2002 | All | No | Yes | Yes |
| | SA-C | U. Colorado | Academic | SA-C | VHDL | 2003 | Image | No | No | No |
| | SPARK | U. Cal. Irvine | Academic | C | VHDL | 2003 | Control | No | No | No |
| | AccelDSP | Xilinx | Commercial | MATLAB | VHDL/Verilog | 2006 | DSP | Yes | Yes | Yes |
| | C2H | Altera | Commercial | C | VHDL/Verilog | 2006 | All | No | No | No |
| | CtoVerilog | U. Haifa | Academic | C | Verilog | 2008 | All | No | No | No |

Figure 2.2: Overview of High-level synthesis tools, from [14]. Note that due to the age of this overview, some tools have already been discontinued or renamed for marketing purposes. An example for this is VivadoHLS, which is now called VitisHLS

## 2.5. From RTL to Hardware: The FPGA Synthesis Flow

The process of placing RTL designs written in HDL, onto an FPGA happens in multiple distinct steps. We will briefly go over these.

### 2.5.1. Synthesis

Synthesis is the first step in the implementation chain of processes. During synthesis, the first thing that happens is syntax checking. The compiler will read the source files and verify that they comply syntactically with the chosen language and that all the statements are recognized and valid. The synthesis tool will then begin translating the RTL representation of the circuit into a netlist. This step is target-agnostic, meaning it

will be common whatever the target FPGA is.

### 2.5.2. Implementation: Optimisation and Mapping

The first step in implementation is called Optimisation. The implementation tool will try to optimize the circuit. Many techniques are employed during this process. One of the simpler techniques is the elimination of redundant logic. If many elements in the circuit produce the same logical output, they will be simplified into a single element.

When a netlist is optimized, the following step is to map it onto the target hardware. This process will transform the combinational functions in the netlist into boolean-equivalent versions which can fit onto the target FPGA's physical hardware. In this step, the arithmetic functions could be mapped to dedicated DSP hardware logic and the generic boolean functions are typically fitted onto lookup tables (LUT's). This step varies depending on the target platform, because different targets contain different hardware building blocks.

### 2.5.3. Implementation: Place and Route

When the design has been mapped onto the target hardware, it is then placed. This step creates a correspondence between the target-aware representation of the functions -created in the previous step- with their physical location in the FPGA fabric.

When all the functions have been placed, the final step in the process is to route the signals between all the physical hardware blocks.

During this process it is common to work in iterations. When a frequency target is set but the post-routing analysis determines that some nodes violate the timing targets, the Place and Route process can be "ripped up", replaced and rerouted to check if the new configuration will be better. Due to the complexity of the optimisation problem that needs to be solved in order to achieve an optimal configuration after place and route, this method of replacing and rerouting follows some heuristic techniques to try to reach an acceptable solution. This process induces quite a lot of variability in the final frequency when the target is set much higher than the expected frequency of the circuit.

## 2.6. Systolicism

Systolicism is a VLSI design technique introduced in the end of the 1970's. To introduce the topic, we have included a quote from Prof. H. T. Kung, often considered one of the fathers of systolicism.

> *In a systolic system, data flows from the computer memory in a rythmic fashion, passing through many processing elements before it returns to memory, much as blood circulates to and from the heart. The system works like an automobile assembly line where different people work on the same car at different times and many cars are assembled simultaneously. An assembly line is always linear, however, and systolic systems are sometimes two-dimensional. They can be rectangular, triangular, or hexagonal to make use of higher degrees of parallelism. Moreover, to implement a variety of computations, data flow in a systolic system may be at multiple speeds in multiple directions-both inputs and (partial) results flow, whereas only results flow in classical pipelined systems. Generally speaking, a systolic system is easy to implement because of its regularity and easy to reconfigure (to meet various outside constraints) because of its modularity.*
>
> (H.T. Kung in [11])

Although his goals for studying these systems were different than ours, we see again an interesting case of "looking at old design with fresh eyes". Kung explains that the main goal of his systolic systems is to resolve bottlenecks, whereas we would like to harness their power because we have "free" silicon real estate, we hope to achieve better energy efficiency by exploiting the opportunities of Dark Silicon using systolic techniques.

Systolic systems are made up of regular, often uniformly clocked, locally interconnected arrays of processing elements. The mindset behind them is to fetch data, only once, in a rhythmic fashion and let it ripple through the processing elements in order to achieve a computational result. The concept of systolic systems thrives on laying out an algorithm in space as opposed to in time. Figure 2.3 shows an example of the systolic mindset.

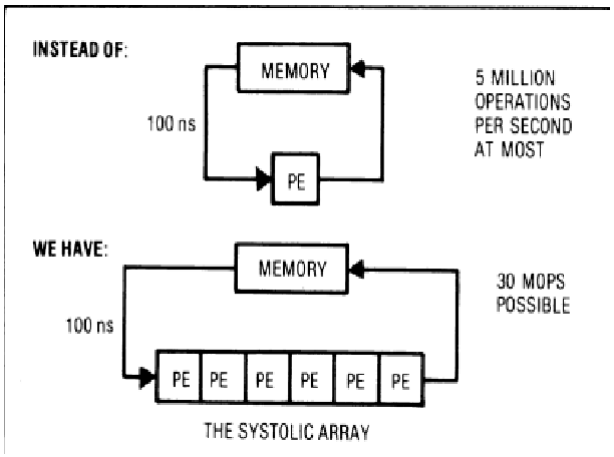


Figure 2.3: Basic principle of a systolic system, from [11]

Another way we can think of systolic arrays is as a block of memory intertwined with ALU elements. In fact, if we consider an array of 100 processing elements, each with 3 registers of 32 bits, it amounts to $1.2kB$ of memory. Their local interconnection patterns usually allow to clock them more easily in custom ASIC's since the results and data must only travel to physically adjacent blocks every cycle, making the system an efficient pipeline.

In FPGA's on the other hand, this property is not always true, since the physical placement is often not decided by the designer, but rather by the Place and Route algorithm and hardware constraints.

Many common problems have been studied using systolic mindsets and the main ones are documented in [11]. After many years of dormancy, systolicism is starting to see a comeback with the recent explosion of popularity of AI applications, which exploit matrix multiplications intensively. This operation is very well suited for systolic approaches.

## 2.7. State-of-the-Art Systolic Systems

### 2.7.1. Google TPU

The Google TPU is a Deep Neural Network (DNN) accelerator which runs inference faster and more efficiently than modern CPU's and GPU's. It implements a 256x256 MAC systolic array in order to save energy on memory accesses. It supports 8-bit integers (INT8) and 16-bit Brain floating point (BF16) data types. Some cost-performance metrics can be found in [15], and the documentation describing the product can be found in [5].

### 2.7.2. Amazon AWS Inferentia

The AWS Inferentia is a custom chip built from the ground up by AWS to accelerate machine learning inference workloads. From Amazon's press releases [1], we can learn that each Inferentia chip contains 4 NeuronCores, each of which implements a high-performance matrix multiplication systolic array which supports 8-bit integers (INT8), 16-bit IEEE floating point (FP16), and 16-bit Brain floating point (BF16) matrix multiplications.

## 2.8. Systolic Arrays for Matrix Multiplications

In order to try to combat the excessive memory accesses of a naive implementation of matrix multiplications, we will resort to studying systolic implementations. Instead of mapping an algorithm temporally, systolic implementations try to map algorithms spatially. This means that instead of sequentially performing operations and writing back to memory the intermediate results, a systolic implementation will access the necessary data as seldom as possible and make it flow through the systolic array such as to minimize memory access requirements. The systolic implementations we will study in-depth are made up of macro-blocks called *Processing Elements*. Complex operations are broken down into much simpler blocks and achieved by interconnecting these blocks. Usually, when the blocks are immutable, we call the system systolic.

### 2.8.1. Kung and Leiserson Designs

#### Kung and Leiserson Processing Element

The Kung and Leiserson Processing Element (KLPE) features three inputs (A, B and C) and three outputs (A, B, and C'). As the names suggest, the A and B inputs do not undergo any operations and are registered to the A and B outputs. The $C'$ output is defined by the operation $C' = C + A \times B$ and is of course also registered. A visual representation of the KLPE can be found in Figure 2.4. In other words, the KLPE is a simple register in two directions and a MAC in the third. It employs 1 multiplier, 1 adder and 3 registers.

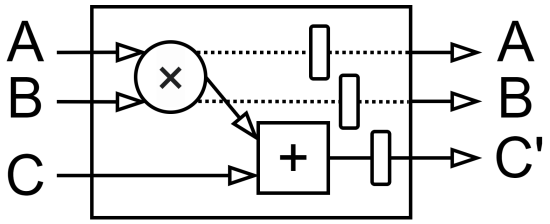


Figure 2.4: Innards of the KLPE, essentially a fully registered Multiply-Accumulate (MAC) block with input data pass-through.

## Kung and Leiserson Matrix-Vector Multiplication

A systolic implementation of a matrix-vector product ($c_i = a_{ij}b_j$) can be achieved with an array of KLPE's linearly interconnected as in Figure 2.6. In this representation, the KLPE's feature the first register banks going from left to right. The second register banks flow from top to bottom. Notice how the top-to-bottom elements are discarded directly after a single use, this is visible on the diagram from the lack of arrows exiting the bottom of the KLPE's. Finally, the MAC registers flow from right to left and exit into a buffer where the final result vector is stored. The elements of $a_{ij}$ are arranged starting with the element $a_{11}$ and proceeding diagonally though the matrix. As seen in Figure 2.5:

$$a_{ij} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{vmatrix} 0 & 0 & a_{33} & 0 & 0 \\ 0 & a_{23} & 0 & a_{32} & 0 \\ a_{13} & 0 & a_{22} & 0 & a_{31} \\ 0 & a_{12} & 0 & a_{21} & 0 \\ 0 & 0 & a_{11} & 0 & 0 \end{vmatrix}$$

Figure 2.5: Disposition of an input matrix when used in a K&L Matrix-Vector Multiplication

This structure will be fed through the system row by row, starting from the bottom row (a row of zeros and the element $a_{11}$). The $b_j$ vector on the other hand is fed element by element at the rate of one element every two cycles. An appropriate amount of leading zeros must be fed before the bottom row of the diagonal $a_{ij}$ structure in order to synchronize the element $a_{11}$ with the element $b_1$. In our example, the appropriate amount of zero rows is two. A representation of the full implementation can be seen in Figure 2.6.

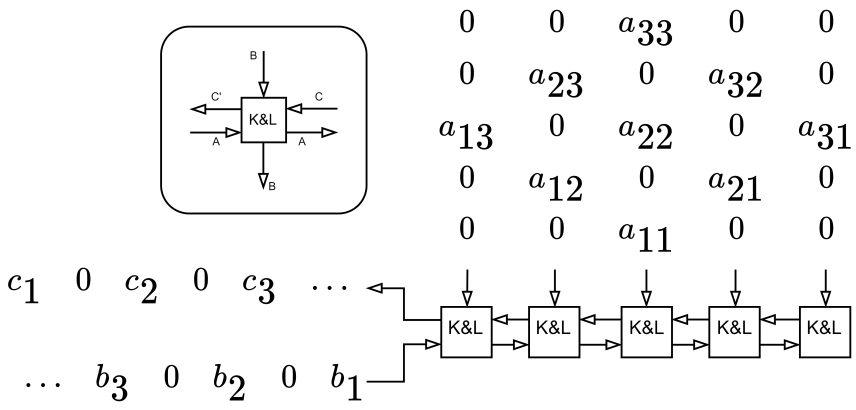


Figure 2.6: Kung and Leiserson linear systolic array for matrix-vector multiplications

Using this implementation, we can imagine a strategy to implement a matrix-matrix multiplication by considering that a matrix-matrix multiplication is nothing more than $n$ independent matrix-vector multiplications.

$$a_{ij}b_{jk} = \left[ \; a_{ij}b_{j1} \;\middle|\; a_{ij}b_{j2} \;\middle|\; \dots \;\middle|\; a_{ij}b_{jn} \; \right]$$

A machine that implements this strategy can be seen in Figure 2.7. Of course, using this strategy requires that every batch of $n$ columns of matrix $B$ must load the entire A matrix again. One could argue that a system to loop back the values of the A matrix could be considered in order to save on global memory accesses. The viability of such a system must be carefully considered since it requires to implement a large amount of registers.

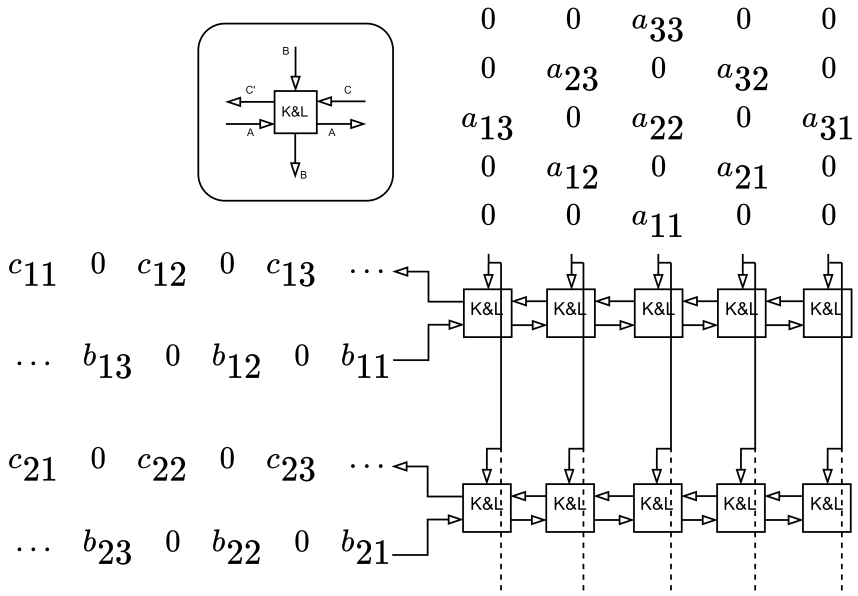


Figure 2.7: Kung and Leiserson-like multilinear array for batches of matrix-vector multiplications

### 2.8.2. Kung and Leiserson Band Matrix-Matrix Multiplier

**Band Matrix-Matrix Multiplications (BMMM)** Let A be a matrix with $p$ non-zero elements in the first line and $q$ non-zero elements in the first row, and such that no non-zero elements can be found outside the diagonal band dictated by these $p$ and $q$ elements. Such a matrix looks as follows:

$$
\begin{array}{l} \\ q \left\{ \right. \end{array}
\overbrace{\phantom{a_{11}\ a_{12}\ a_{13}}}^{p}
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & 0 & \dots & & 0 \\
a_{21} & a_{22} & a_{23} & a_{24} & 0 & \dots & 0 \\
0 & a_{32} & a_{33} & a_{34} & a_{35} & 0 & \\
\vdots & 0 & a_{43} & a_{44} & a_{45} & & \\
 & \vdots & & \ddots & \ddots & \ddots & \\
0 & 0 & & & & &
\end{bmatrix}
$$

This matrix is called a band matrix with width $p$ and height $q$, because the number of non-zero elements in the first row and column respectively are $p$ and $q$.

We define a matrix B which features the same structure as matrix A only transposed:

$$
\overset{q}{\overbrace{\phantom{xxxxxx}}}
\qquad
p\left\{\vphantom{\begin{matrix}1\\1\\1\\1\\1\\1\\1\end{matrix}}\right.
\begin{bmatrix}
b_{11} & b_{12} & 0 & \dots & & 0 \\
b_{21} & b_{22} & b_{23} & 0 & \dots & 0 \\
b_{31} & b_{32} & b_{33} & b_{34} & 0 & \\
0 & b_{42} & b_{43} & b_{44} & b_{45} & \\
\vdots & \vdots & & \ddots & \ddots & \ddots \\
0 & 0 & & & &
\end{bmatrix}
$$

When matrices A and B are multiplied together, they result in a band matrix C, which has width and height $w = p + q - 1$. We call $w$ the band width. Note that $w$ is also the largest number of horizontal or vertical consecutive non-zero elements in matrices A and B. The complete operation is displayed below:

$$
\begin{matrix}
\overset{p}{\overbrace{\phantom{xxxxx}}} \\
q\left\{\vphantom{\begin{matrix}1\\1\\1\\1\\1\\1\end{matrix}}\right.
\end{matrix}
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & 0 & \dots & 0 \\
a_{21} & a_{22} & a_{23} & a_{24} & 0 & \dots 0 \\
0 & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\
\vdots & 0 & a_{43} & a_{44} & a_{45} & \\
& \vdots & & \ddots & \ddots & \ddots \\
0 & 0 & & & &
\end{bmatrix}
\cdot
\begin{matrix}
\overset{q}{\overbrace{\phantom{xxxxx}}} \\
p\left\{\vphantom{\begin{matrix}1\\1\\1\\1\\1\\1\end{matrix}}\right.
\end{matrix}
\begin{bmatrix}
b_{11} & b_{12} & 0 & \dots & 0 \\
b_{21} & b_{22} & b_{23} & 0 & \dots 0 \\
b_{31} & b_{32} & b_{33} & b_{34} & 0 \\
0 & b_{42} & b_{43} & b_{44} & b_{45} \\
\vdots & \vdots & & \ddots & \ddots & \ddots \\
0 & 0 & & & &
\end{bmatrix}
=
\begin{matrix}
\overset{w}{\overbrace{\phantom{xxxxxx}}} \\
w\left\{\vphantom{\begin{matrix}1\\1\\1\\1\\1\\1\\1\end{matrix}}\right.
\end{matrix}
\begin{bmatrix}
c_{11} & c_{12} & c_{13} & c_{14} & \dots & & 0 \\
c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & \dots & 0 \\
c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} & \dots 0 \\
c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} & c_{47} \\
\vdots & \vdots & & \ddots & \ddots & \ddots & \\
0 & 0 & & & & &
\end{bmatrix}
$$

The multiplication of two band matrices with identical band width $w$ can be accelerated both in software, by limiting the multiplication to non-zero elements (as in Section 4.6) and also in hardware, as we will see in the following section.

## Systolic Array Design for BMMM

By extending the idea of interconnecting KLPE's to two dimensions, we can study a multiplier introduced by Kung and Leiserson in [10]. This implementation considers the multiplication of two band matrices with equal band width ($w$). It allows us to pipeline the multiplication of any size of matrices, as long as their band width is smaller or equal than the lateral size of the array. In other words, to multiply two compatible matrices of band width $w$, you need a quadratic[2] array of $w \times w$ interconnected KLPE's. In Figure 2.8, we can see the corresponding array design.

[2]In [10], this kind of array is called *Hexagonal* because of the triliniear motion of the data through the array. We prefer to refer to the array as quadratic because we consider the input data streams as the leading forces of the design and the diagonal interconnections as auxiliary enablers for the algorithm. Of course, we are referring to the same design and all our drawings are adapted accordingly.

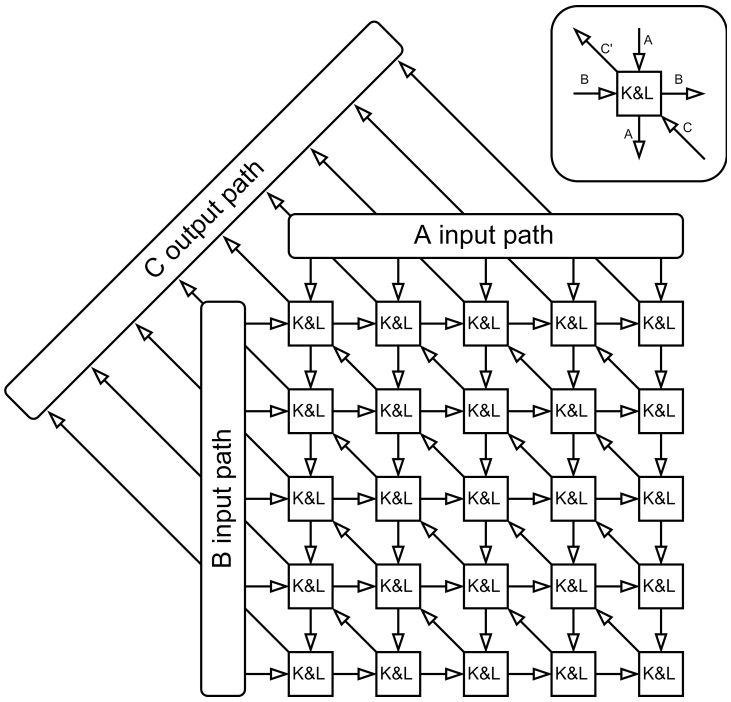


Figure 2.8: K&L systolic array for BMMM

## Input Data Patterning

The input data patterns used for the K&L BMMM are quite complex, they can be best explained using a timing diagram. To help the reader, we have put together such a diagram and displayed it in Figure 2.9. Each row must be sent element by element, with each element being delayed by one cycle. Consecutive lines will be spaced in time by 2 empty cycles. This results in elements from multiple lines being dispatched in the same cycles. In our example in 2.9, since we are dealing with a band width[3] of 5, we can have elements from at most two consecutive lines being dispatched. This can be calculated with the formula $\lfloor \frac{w}{3} \rfloor + 1$. The overlapping of multiple rows can be seen in cycle #3 where elements $a_{12}$ and $a_{31}$ are dispatched in the same cycle despite belonging to separate rows in the matrix.

[3]and hence an array of $5^2 = 25$ KLPE's

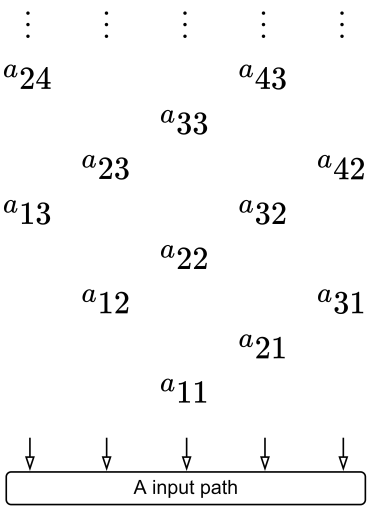


Figure 2.9: Data patterning example for the input paths of the K&L systolic array for BMMM, seen in Figure 2.8

## Output Data Patterning

The output data pattern can be seen in Figure 2.10. The center element is the first to exit the array. Starting from the center element, every consecutive element exits the array from the next processing elements left and right of the previous cycle. This pattern repeats every 3 cycles. Similarly to the input pattern, the formula to figure out how many lines can overlap in the same cycle is $\lfloor \frac{w}{3} \rfloor + 1$.

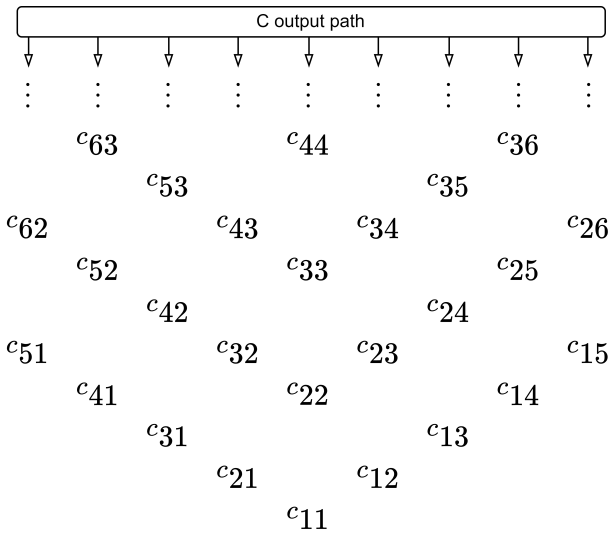


Figure 2.10: Data patterning example for the output path of the K&L systolic array for BMMM, seen in Figure 2.8

### 2.8.3. Generic Matrix-Matrix Multiplication (GMMM)

Although we can multiply generic matrices with a BMMM systolic array by carefully considering the full generic matrix as the band elements of a larger matrix (see Figure 2.11), we will study an array which is tailored to the GMMM operation, achieving more throughput[4] but using the same amount of computing hardware.

$$\begin{bmatrix} \boxed{\begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix}} & 0 & \dots & 0 \\ 0 & \times & \times & \times & & \\ \vdots & \vdots & & & \ddots & \\ 0 & 0 & & & & \times \end{bmatrix}$$

Figure 2.11: Example of a 3x3 matrix represented as part of a band of a larger matrix. A 5x5 K&L band systolic array would be needed to accommodate this operation

#### Systolic Array Design for GMMM

Figure 2.12 displays an array which can achieve GMMM. We begin by noting that it is comprised of KLPE's and delay blocks, labeled $D$ in the diagram. The input vectors flow top to bottom and left to right and the MAC's flow diagonally from top-left to bottom-right.

[4] We will achieve $N$ elements per cycle per input and output instead of $\frac{2N-1}{3}$ elements per cycle. For large $N$, this equates to an increase of throughput of 50%.

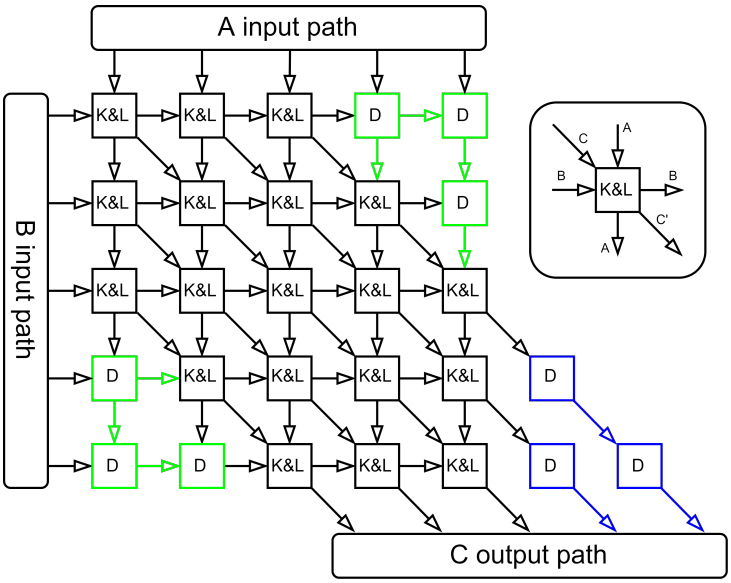


Figure 2.12: Systolic array for GMMM

A naive version of our design employs the same amount of KLPE's as the BMMM. We notice that we can actually spare some KLPE's, since these elements are expensive. We can replace the green top-right and bottom-left triangles of KLPE's with simple 2-way delay blocks. Doing so, the amount of remaining KLPE's that are actually necessary is calculated below:

Let $N$ be the lateral size of the matrix you want to multiply:

$$\begin{aligned} \#_{KLPE,total} &= (2N-1)^2 \\ \#_{D-blocks} &= 2\frac{N(N-1)}{2} \\ \#_{KLPE,final} &= \#_{KLPE,total} - \#_{D-blocks} \\ &= (2N-1)^2 - N(N-1) \\ &= 3N^2 - 3N + 1 \end{aligned}$$

## Input and Output Data Patterning

In order for this design to work and produce the expected GMMM output, The input data rows must be presented to the array row by row. Each consecutive row must be skewed by one element every cycle. The A matrix must be sent row by row, whereas the B matrix must be sent column by column. The output triangle of one-dimensional delay

blocks[5] allow us to retrieve the output matrix with a similar pattern, meaning row by row, or column by column. We can choose which of these output types we want by which side we place the output delay blocks. In Figure 2.12, the input and output data will respectively enter and exit the array in the pattern shown in Figure 2.13.

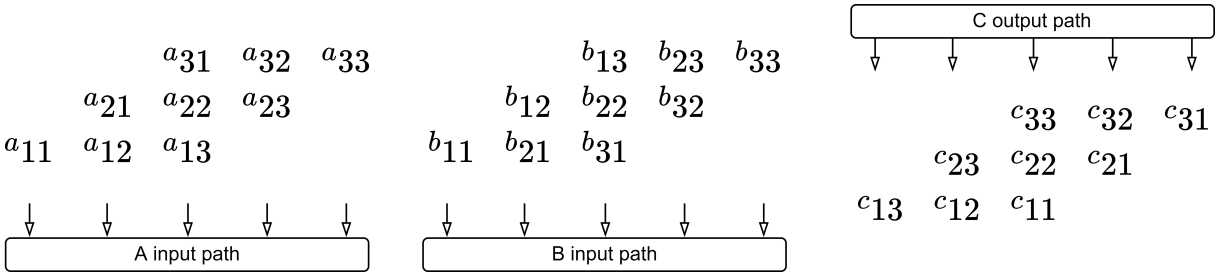


Figure 2.13: Pattern for data delivery for the systolic array of Figure 2.12

## 2.9. Summary

In this chapter, we have gone over all the background concepts forming the backbone of our thesis. We began with a recap of tensor algebra, the followed with an introduction of systolic systems and some of their implementations focused on tensor algebra. We have also very briefly introduced the modern FPGA synthesis flow.

In the following chapter, we will present the kernel upon which we have decided to base our thesis and go inspect its RTL description.

[5]Essentially shift registers

# 3 | RTL Design of a Parametric Multi-directional Systolic Kernel

Noticing the similarities between the BMMM and GMMM systems presented in the previous Chapter, we have decided to implement a Unified Matrix-Matrix Multiplier (UMMM) as the study subject of our thesis. It will be fully parametric in array size and data width. The heart of our UMMM consists of a quadratic systolic array of KLPE's with inputs flowing left-to-right and top-to-bottom. The diagonal paths of our design can be rerouted to flow downwards or upwards, behaving either like the BMMM or GMMM array. Doing so, it reuses expensive KLPE's for both operations. The final design is presented in Figure 3.1. In addition to rerouting the systolic array, each operation will also employ bespoke peripheral hardware to deliver data according to their respective data pattern needs.

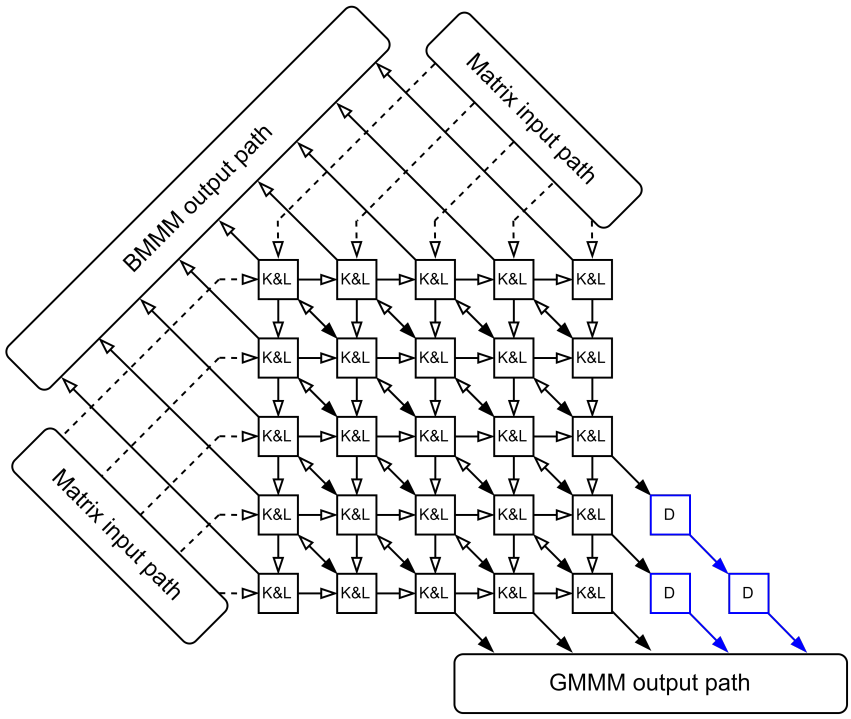


Figure 3.1: Systolic Unified Matrix-Matrix Multiplication Core (UMMM)

In this section, we will display and comment our RTL implementation of the unified array. For reference, the entire system hierarchy lies in Figure 4.3. Consulting it can be useful to the reader in order to visualize the hierarchy at a macro, top-down scale before reading the following sections, which go through the design in a bottom-up order.

## 3.1. KLPE

The KLPE is an elementary block of the system and should be designed as simply as possible. This module is fully parametrized in data width. It implements synchronous reset and enable signals in order to be able to cycle through the computation one cycle at a time. This stalling mechanism will be implemented for every consecutive block of the system. This is required in case we need to pause operation because the peripheral data systems are not able to keep up. We have limited our design to using only integer unsigned data. If we wanted to accommodate different data types, this block is the only one whose functionally would need to change.

Listing 1: Verilog description of the KLPE, equivalent to Figure 2.4. This code snippet is a portion of Listing 54.

```
11    wire [2*DATA_WIDTH-1:0] temp = A_in*B_in; //manually define the operation
12    always @ (posedge clk)
13      begin
14        if (reset) begin
15          A_out <= 0;
16          B_out <= 0;
17          C_out <= 0;
18        end
19        else if (array_en) begin
20          A_out <= A_in;
21          B_out <= B_in;
22          C_out <= C_in + temp[DATA_WIDTH-1:0]; //manually define the operation
23          end
24      end
25  endmodule // KLPE
```

## 3.2. Unified Array Core

The unified array core consists of an implementation of Figure 3.1. This module is fully parametrized in data width and array size. The `MAT_SIZE` parameter consists of the size of the largest GMMM operation we want to accommodate. If `MAT_SIZE` is 3, the Verilog code will generate the same array as Figure 3.1. The reader must keep in mind that when `MAT_SIZE` is set to $N$, the largest band width ($w$) it can accommodate during BMMM is $2N-1$. However, the size of the matrix that can be computed using this technique is infinite, as long as its band fits in the array. For a `MAT_SIZE`$= N$, the resulting array will feature $(2N-1)^2$ KLPE's.

This module showcases several interesting particularities. The first of which is a shortcoming of the Verilog language. It does not support arrays of signals as module inputs and outputs. This forces us to flatten and unflatten signals at the inputs and outputs of the modules in order to keep the design fully parametric. Although this happens in many places throughout the design, we will not mention it again as it is a recurring feature. An example of module unflattening can be seen in Listing 2.

Listing 2: Signal unflattening in Verilog. This code snippet is a portion of Listing 55

```
36    generate
37      for (i=0;i<(2*MAT_SIZE-1);i=i+1) begin
38        assign A[i] = A_flattened[DATA_WIDTH*i+(DATA_WIDTH-1):DATA_WIDTH*i];
39        assign B[i] = B_flattened[DATA_WIDTH*i+(DATA_WIDTH-1):DATA_WIDTH*i];
40      end
41    endgenerate
```

In Listing 3, we can see our implementation of the unified array core. Note the presence of the `opmode` signal, for example on line 71. It controls whether we are using the array for GMMM or BMMM. Our unified array core not only generates the array of KLPE's, but it also links them all together according to `opmode`. All of this is achieved in a fully parametrized fashion. In order to do so, the Verilog language obliges us to instantiate arrays of wires for the top-down, left to right and bi-diagonal flows. Connecting to these arrays correctly in the `generate-endgenerate` environment is the most fastidious part.

Listing 3: Verilog description of the unified array core. This code snippet is a portion of Listing 55

```
58      wire [DATA_WIDTH-1:0] w_hor [0:(2*MAT_SIZE-1)-1][0:(2*MAT_SIZE-1)-2];
59      wire [DATA_WIDTH-1:0] w_ver [0:(2*MAT_SIZE-1)-1][0:(2*MAT_SIZE-1)-2];
60    //diagonal wires :
61    //some of these wires will not be used but are declared for simplicity of thought
62      wire [DATA_WIDTH-1:0] w_diag [0:(2*MAT_SIZE-1)-1][0:(2*MAT_SIZE-1)-1];
63
64    generate
65      for (j=0; j<(2*MAT_SIZE-1); j=j+1)
66      begin : j_loop
67        for (i=0; i<(2*MAT_SIZE-1); i=i+1)
68        begin : i_loop
69          wire [DATA_WIDTH-1:0] A_in = j==0 ? A[i] : w_ver[i][j-1];
70          wire [DATA_WIDTH-1:0] B_in = i==0 ? B[j] : w_hor[j][i-1];
71          wire [DATA_WIDTH-1:0] C_in = opmode ? ((i==(2*MAT_SIZE-1)-1 || j==(2*MAT_SIZE-1)-1) ? 0 :
↪  w_diag[i+1][j+1]) : ((i==0 || j==0) ? 0 : w_diag[i-1][j-1]);
72
73          wire [DATA_WIDTH-1:0] A_out;
74          wire [DATA_WIDTH-1:0] B_out;
75          wire [DATA_WIDTH-1:0] C_out;
76
77          if (j<(2*MAT_SIZE-1)-1) assign w_ver[i][j] = A_out ;
78          if (i<(2*MAT_SIZE-1)-1) assign w_hor[j][i] = B_out ;
79
80          assign w_diag[i][j] = C_out ;
81
82          if (j>0 && i>0) begin end
83          else if (i==0) assign C_array_out_ver[(2*MAT_SIZE-1)-j-1] = C_out; // needs checkup for
↪  off-by-1
84          else if (j==0) assign C_array_out_hor[i-1] = C_out;//because the top vector is shifted by 1
85
86          if (j<2*MAT_SIZE-2 && i<2*MAT_SIZE-2) begin end
87          else if (j==2*MAT_SIZE-2 && i >= MAT_SIZE-1) assign C_array_out[(3*MAT_SIZE-3) - i] = C_out;
88          else if (j >= MAT_SIZE-1 && j < 2*MAT_SIZE-2) assign C_array_out[j - MAT_SIZE+1] = C_out;
89
90
91
92          KLPE2 #(.DATA_WIDTH(DATA_WIDTH)) pe (
93          .clk(clk),
94          .array_en(array_en),
```

```
95          .reset(reset),
96          .A_in(A_in),
97          .B_in(B_in),
98          .C_in(C_in),
99          .A_out(A_out),
100         .B_out(B_out),
101         .C_out(C_out));
102       end
103     end
104   endgenerate
```

Lastly, we also parametrically generate the output triangle of delay blocks for GMMM. This is done in a similar way to the generation of the main array.

Listing 4: Verilog description of the output triangle of delay blocks for GMMM, visible in Figure 3.1. This code snippet is a portion of Listing 55

```
123 wire [DATA_WIDTH-1:0] w_delays [0:(MAT_SIZE>=3? MAT_SIZE-3:0)][0:(MAT_SIZE>=3? MAT_SIZE-3:0)];
124
125 generate
126   for (i=0; i<MAT_SIZE-1; i=i+1)
127     begin : i_loop
128     for (j=0; j<=i; j=j+1)
129       begin : j_loop
130
131        wire [DATA_WIDTH-1:0] A_in = j==0 ? C_array_out[i] : w_delays[i-1][j-1];
132        wire [DATA_WIDTH-1:0] A_out;
133
134       if (i<MAT_SIZE-2) assign w_delays[i][j] = A_out;
135       else assign C_generic[MAT_SIZE-2-j] = A_out;
136
137       D1D #(.DATA_WIDTH(DATA_WIDTH)) dblock (
138         .clk(clk),
139         .reset(reset),
140         .array_en(array_en),
141         .A_in(A_in),
142         .A_out(A_out));
143     end
144   end
145 endgenerate
```

## 3.3. Peripherals: Generic Matrix-Matrix Multiplication

### 3.3.1. Generic Input Peripheral Device

The sequence of data dispatching into the GMMM kernel is very straightforward. Each input matrix sends one line of data per cycle and the peripherals must only ensure to steer this data to the correct inputs. More precisely, each consecutive data line must be shifted by one to keep up with the kernel operation. In Figure 3.2 we can see a representation of the `datasteerer` working.

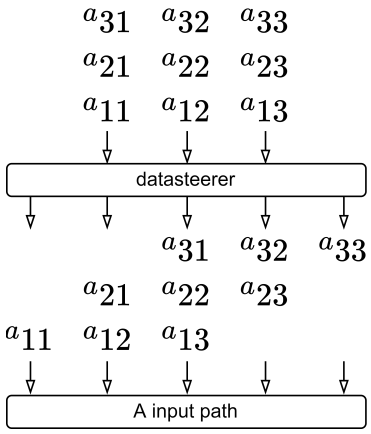


Figure 3.2: Visualisation of the workings of the `datasteerer`

The following Verilog module describes the `datasteerer`. It achieves the operation by padding the input with zeros and rotating the vector. The complete operation is made with flattened vectors.

Listing 5: Verilog description of the `datasteerer`, equivalent to Figure 3.2. This code snippet is a portion of Listing 56.

```
13 module datasteerer #(parameter MAT_SIZE=3, DATA_WIDTH = 64)(
14   input clk,
15   input [(`CLOG2(MAT_SIZE))-1:0] counter,
16   input [MAT_SIZE*DATA_WIDTH-1:0] data_in,
17   output [(2*MAT_SIZE-1)*DATA_WIDTH-1:0] data_out);
18     wire [(MAT_SIZE-1)*DATA_WIDTH-1:0] zeropadding = 0;
19   assign data_out = {zeropadding,data_in} << DATA_WIDTH*counter;
20 endmodule // datasteerer
```

### 3.3.2. Generic Output Peripheral Device

The following block is called the `datacollector`. It ensures the inverse operation to the `datasteerer` and is used at the output of the array for GMMM. We can see a representation of the workings of the `datacollector` in Figure 3.3.

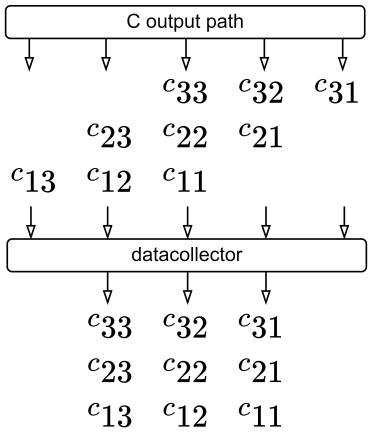


Figure 3.3: Visualisation of the workings of the `datacollector`

Its Verilog description is very similar to the `datasteerer`.

Listing 6: Verilog description of the `datacollector`, equivalent to Figure 3.3. This code snippet is a portion of Listing 56.

```
22 module datacollector #(parameter MAT_SIZE=3,DATA_WIDTH = 64)(
23   input clk,
24   input [(`CLOG2(MAT_SIZE))-1:0] counter,
25   input [(2*MAT_SIZE-1)*DATA_WIDTH-1:0] data_in,
26   output [MAT_SIZE*DATA_WIDTH-1:0] data_out
27   );
28     assign data_out =  data_in >> DATA_WIDTH*(MAT_SIZE-1-counter);
29 endmodule // datacollector
```

### 3.3.3. GMMM Peripheral Devices Assembly

For the GMMM, the last step is to put all the peripherals in a module. For proper functioning of our system we must ensure that the counter dictating the steering amount is appropriately delayed in time between the input `datasteerers` and output `datacollector`. A schematic of the GMMM assembly can be seen in Figure 3.4.

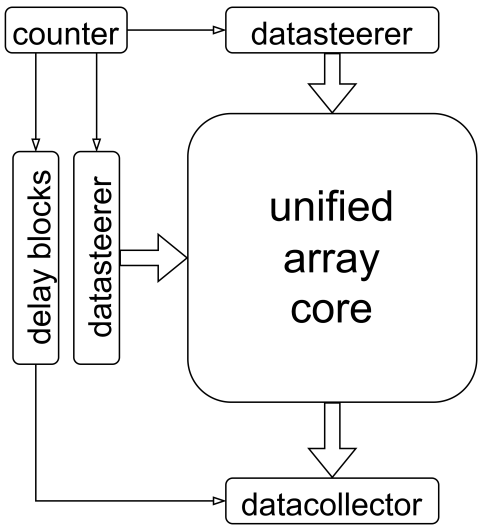


Figure 3.4: Unified array core surrounded by the necessary peripherals to enable GMMM.

In order to instantiate the appropriate amount of delay blocks to ensure proper operation, we have used a custom parametric shift register. For this particular application, a much simpler Verilog description could be written. We decided to write it in this way because we had already designed the delay blocks for the GMMM array and decided to reuse them. Of course, the final generated hardware is not affected by these choices.

Listing 7: Verilog description of our parametric shift register, implementing the chain of delay blocks visible in Figure 3.4. This code snippet is a portion of Listing 56.

```
66   genvar i;
67     wire [(`CLOG2(MAT_SIZE))-1:0] counter_intermediate [2*MAT_SIZE-2:0];
68   generate
69     for(i=0;i<2*MAT_SIZE-1;i=i+1) begin
70
71       wire [(`CLOG2(MAT_SIZE))-1:0] cnt_in,cnt_out;
72       assign cnt_in = i==0? counter_in : counter_intermediate[i-1];
73       assign counter_intermediate[i] = cnt_out;
74
75       D1D #(.DATA_WIDTH(`CLOG2(MAT_SIZE))) dblock (
76         .array_en(array_en),
77         .clk(clk),
78         .reset(0),
79         .A_in(cnt_in),
80         .A_out(cnt_out));
81     end
82 endgenerate
```

## 3.4. Peripherals: Band Matrix-Matrix Multiplication

### 3.4.1. Custom Input And Output Formats

By taking advantage of the freedom of hardware-software co-design, we have decided to organise the input matrices in a custom format. This will be extremely practical for simplification of the hardware as well as for keeping our data in a very efficient format.

#### Custom Input Format

For the input matrices, we have packed the bands into a rectangular matrix structure with width $w = p + q - 1$ and with height equal to the original matrix size. This allows for a very efficient packing of the input band matrix and additionally allows for line by line reading and memory dispatching to the multiplier's input buffer.

The packing procedure is to take each row of the non-zero band and stack them vertically in the rectangular matrix. The resulting matrix will be of the same height as the original band matrix but the width will be only $w = p + q - 1$. Of course the first and last lines need to be respectively pre- and post-padded with zeros to follow the procedure. We can see an example of this format below, with a width ($p$) of 3 and height ($q$) of 2 resulting in a band width ($w$) of 4.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} & 0 & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} & a_{46} & 0 \\ 0 & 0 & 0 & a_{54} & a_{55} & a_{56} & a_{57} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} & a_{67} \\ 0 & 0 & 0 & 0 & 0 & a_{76} & a_{77} \end{bmatrix} \rightarrow \begin{bmatrix} 0 & a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} & a_{46} \\ a_{54} & a_{55} & a_{56} & a_{57} \\ a_{65} & a_{66} & a_{67} & 0 \\ a_{76} & a_{77} & 0 & 0 \end{bmatrix}$$

Figure 3.5: Example of a rectangular representation of a band matrix with $p = 3$, $q = 2$, $w = 4$ and an height of 7

For this custom input format, the rigorous index relocation function from normal indices to rectangular indices is:

$$\begin{cases} I_r(i,j) = i \\ J_r(i,j) = q + j - i \end{cases}$$

The function can of course be inverted to find the following expression:

$$\begin{cases} i(I_r, J_r) = I_r \\ j(I_r, J_r) = I_r + J_r - q \end{cases}$$

The relocation function can also be combined with a transposition[1]. This yields:

$$\begin{cases} I_r(i, j) = j \\ J_r(i, j) = p + i - j \end{cases}$$

## Custom Input Format: Packing Efficiency

The number of zeros in the top left corner is simply found using the triangular number formula:

$$Z_{TL}(q) = \frac{q(q-1)}{2}$$

The number of zeros we can find in the bottom right corner is:

$$Z_{BR}(p) = \frac{p(p-1)}{2}$$

The packing efficiency is thus:

$$\begin{aligned} \eta(p, q, M) &= \frac{M(p+q-1) - \frac{q(q-1)}{2} - \frac{p(p-1)}{2}}{M(p+q-1)} \\ &= 1 - \frac{q(q-1) + p(p-1)}{2M(p+q-1)} \\ &= 1 - \frac{q^2 - q + p^2 - p}{2M(p+q-1)} \\ &= 1 - \frac{q^2 + p^2 - (p+q-1) - 1}{2M(p+q-1)} \\ &= 1 - \frac{1}{2M}\left(\frac{p^2+q^2-1}{p+q-1} - 1\right) \end{aligned}$$

Although drawing conclusions from this formula is not straightforward, we can immediately see that the efficiency approaches 100% as $M$, the original matrix lateral size, becomes bigger, which is hardly surprising since the starting and ending zeros become

[1] This will be used in the design considered in Section 4.6

negligible compared to the overall matrix size. We can also see that the formula is symmetrical in $p$ and $q$. If we calculate an example with $p = 5$, $q = 6$ (such that $w = 10$) and $M = 1000$, we get a packing efficiency of 99.75%. For comparison the original matrix, without a sparse implementation has a packing efficiency of 1%.

## Custom Output Format

The custom output format has been chosen according to the output data pattern seen in Figure 2.10. Each row of the arrangement corresponds to a $\Gamma$ shape in the original output band matrix. The width of the resulting matrix is thus $2w - 1$ and the height is the original matrix size. An illustration of this format can be seen in Figure 3.6.

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & 0 & 0 & 0 \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & 0 & 0 \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} & 0 \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} & c_{47} \\ 0 & c_{52} & c_{53} & c_{54} & c_{55} & c_{56} & c_{57} \\ 0 & 0 & c_{63} & c_{64} & c_{65} & c_{66} & c_{67} \\ 0 & 0 & 0 & c_{74} & c_{75} & c_{76} & c_{77} \end{bmatrix} \rightarrow \begin{bmatrix} c_{41} & c_{31} & c_{21} & c_{11} & c_{12} & c_{13} & c_{14} \\ c_{52} & c_{42} & c_{32} & c_{22} & c_{23} & c_{24} & c_{25} \\ c_{63} & c_{53} & c_{43} & c_{33} & c_{34} & c_{35} & c_{36} \\ c_{74} & c_{64} & c_{54} & c_{44} & c_{45} & c_{46} & c_{47} \\ 0 & c_{75} & c_{65} & c_{55} & c_{56} & c_{57} & 0 \\ 0 & 0 & c_{76} & c_{66} & c_{67} & 0 & 0 \\ 0 & 0 & 0 & c_{77} & 0 & 0 & 0 \end{bmatrix}$$

Figure 3.6: Example of an output rectangular representation of a band matrix with $w = 4$ and an height of 7

The rigorous function to find rectangular indices, although less trivial, is:

$$\begin{cases} I_r(i,j) = \begin{cases} i & \text{if} \quad j \geq i \\ j & \text{if} \quad j \leq i \end{cases} \\ \\ J_r(i,j) = w - i + j \end{cases}$$

Its reciprocal is:

$$\begin{cases} i(I_r, J_r) & = \begin{cases} I_r & \text{if} \quad J_r \geq w \\ w + I_r - J_r & \text{if} \quad J_r \leq w \end{cases} \\ \\ j(I_r, J_r) & = \begin{cases} J_r - w + I_r & \text{if} \quad J_r \geq w \\ I_r & \text{if} \quad J_r \leq w \end{cases} \end{cases}$$

These formulas can be used to access the data of the resulting matrix without having to rewrite the matrix in another shape in memory.

## Custom Output Format: Packing Efficiency

Keeping the data in this format in memory can be very space efficient. The waste in memory zero trailing elements is easily calculated to be two triangles of side $w-1$, thus:

$$Z(w) = \frac{2(w-1)(w-1+1)}{2} = w(w-1)$$

The packing efficiency of this output format is:

$$\begin{aligned}\eta(r = \frac{w}{M}, w) &= \frac{(2w-1)M - w(w-1)}{(2w-1)M} \\ &= 1 - \frac{w(w-1)}{(2w-1)M} \\ &= 1 - r\frac{w-1}{2w-1}\end{aligned}$$

with the $\frac{w-1}{2w-1}$ term being at most 0.5 for large values of $w$, we can safely say that the efficiency is above $1 - \frac{w}{2M}$ or 100% minus the half ratio of the band size to the matrix size. For a matrix size of 1000 and a band size of 10, we get an efficiency of 99.53%. For comparison, the packing efficiency of this matrix written in a non-sparse fashion is 1.89%.

## Band Input Peripheral Device

The band input peripheral device takes in rows in the custom format and dispatches the elements according to the pattern seen in Figure 2.9. Thanks to the custom input format, no data steering is needed and we must only time the dispatching of the data vectors. In order to do so, out solution employs internal buffers to have access to multiple rows of data simultaneously. A visualisation of the band input device functionality is represented in Figure 3.7.

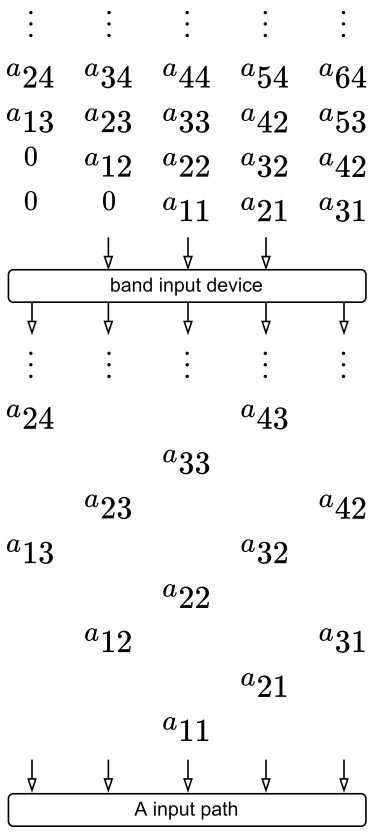


Figure 3.7: Visualisation of the workings of the band input device

The amount of rows of data which are simultaneously needed, and thus the size of the buffer we must implement can be calculated using the following formula:

$$\#_{entries} = \left\lfloor \frac{w}{3} \right\rfloor + 1$$

This formula was implemented as a precompiler macro. An example of this can be seen in Listing 57.

Table 3.1 showcases the beginning of the operation of the band input devices. We will use it as a visual support to understand the inner workings of the band input device.

Table 3.1: Cycle-by-cycle operation of the band input devices. It shows the logic we employ to ensure adequate data patterns for BMMM.

| Example signals in time | | | | | |
|---|---|---|---|---|---|
| cycle | binary pattern | dispatch pattern | write pointer | data buffer | output |
| 0 | 00000 | 00000 | 0 | 0: $0\ 0\ a_{11}\ a_{21}\ a_{31}$ | $0\ 0\ 0\ 0\ 0$ |
| 1 | 10000 | 00000 | 1 | 1: $\times\ \times\ \times\ \times\ \times$ | $0\ 0\ 0\ 0\ 0$ |
| 2 | 01000 | 10000 | 1 | | $0\ 0\ 0\ 0\ 0$ |
| 3 | 00100 | 11000 | 1 | 0: $0\ 0\ a_{11}\ a_{21}\ a_{31}$ | $0\ 0\ a_{11}\ 0\ 0$ |
| 4 | 10010 | 11100 | 0 | 1: $0\ a_{12}\ a_{22}\ a_{32}\ a_{42}$ | $0\ 0\ 0\ a_{21}\ 0$ |
| 5 | 01001 | 01110 | 0 | | $0\ a_{12}\ 0\ 0\ a_{31}$ |
| 6 | 00100 | 00111 | 0 | 0: $a_{13}\ a_{23}\ a_{33}\ a_{43}\ a_{53}$ | $0\ 0\ a_{22}\ 0\ 0$ |
| 7 | 10010 | 00011 | 1 | 1: $0\ a_{12}\ a_{22}\ a_{32}\ a_{42}$ | $a_{13}\ 0\ 0\ a_{32}\ 0$ |
| 8 | 01001 | 10001 | 1 | | $0\ a_{23}\ 0\ 0\ a_{42}$ |
| 9 | 00100 | 11000 | 1 | 0: $a_{13}\ a_{23}\ a_{33}\ a_{43}\ a_{53}$ | $0\ 0\ a_{33}\ 0\ 0$ |
| 10 | 10010 | 11100 | 0 | 1: $a_{24}\ a_{34}\ a_{44}\ a_{54}\ a_{64}$ | $a_{24}\ 0\ 0\ a_{43}\ 0$ |
| 11 | 01001 | 01110 | 0 | | $0\ a_{34}\ 0\ 0\ a_{53}$ |
| 12 | 00100 | 00111 | 0 | 0: $a_{35}\ a_{45}\ a_{55}\ a_{65}\ a_{75}$ | $0\ 0\ a_{44}\ 0\ 0$ |
| 13 | 10010 | 00011 | 1 | 1: $a_{24}\ a_{34}\ a_{44}\ a_{54}\ a_{64}$ | $a_{35}\ 0\ 0\ a_{54}\ 0$ |
| 14 | 01001 | 10001 | 1 | | $0\ a_{45}\ 0\ 0\ a_{64}$ |

For better understanding, we have described below the workings of the most important signals in the band input device.

- `binary_pattern`: This signal is positional. It initiates a *one* in the first position every 3 cycles. These *ones* then ripple rightwards towards the end of the vector. The position of these *ones* determine which element in the buffer address should be dispatched.

Listing 8: Verilog description of the `binary_pattern` signal behaviour. This code snippet is a portion of Listing 57.

```
27  always @(posedge clk) begin
28    for(k=0;k<BAND_SIZE;k=k+1) begin
29      if(k==0) begin
30        if(reset) binary_pattern[k] <= 0;
31        else if (array_en) binary_pattern[k] <= (tricounter==0) ? 1'b1 : 1'b0;
32      end
33      else begin
34        if (reset) binary_pattern[k] <= 0;
35        else if(array_en) binary_pattern[k] <= (tricounter == 2'b11) ? 1'b0 : binary_pattern[k-1];
36      end
37    end
38  end
```

- `dispatch_pattern`: This signal is also positional and is not binary, although because of the small size of the example in Table 3.1 it may appear so. Each position increments by 1 when the binary pattern was a 1 at the same position. This signal serves to know from which buffer address the next value should be dispatched.

Listing 9: Verilog description of the `dispatch_pattern` signal behaviour. This code snippet is a portion of Listing 57

```
50  always @(posedge clk) begin
51    if(reset) begin
52      for(k=0;k<BAND_SIZE;k=k+1) begin
53        dispatch_pattern[k]<= 0;//reset everything
54      end
55    end
56    else if(array_en) begin
57      for(k=0;k<BAND_SIZE;k=k+1) begin
58        dispatch_pattern[k] <= (binary_pattern[k]==1) ? (dispatch_pattern[k] + 1) %
    ↪  (`GET_BUF_SIZE(BAND_SIZE)) : dispatch_pattern[k];
59      end
60    end
61  end
```

- `write_pointer`: This signal determines which address in the buffer the next input row of values should be written to every 3 cycles. Every time a vector is written into the data buffer this value will be incremented.

Listing 10: Verilog description of the `write_pointer` signal behaviour. This code snippet is a portion of Listing 57

```
41  always @(posedge clk)
42    if (reset)
43        write_pointer<=0;
44    else if(array_en)
45      if(binary_pattern[0]==1)
46        write_pointer <= (write_pointer+1) % (`GET_BUF_SIZE(BAND_SIZE));
```

- `data_buffer`: This buffer holds the vectors that need to be appropriately dispatched.

Listing 11: Verilog description of the `data_buffer` signal behaviour. This code snippet is a portion of Listing 57

```
72  always @(posedge clk) begin
73    if (reset) begin
74      for(k=0;k<BAND_SIZE;k=k+1) begin
75        for(l=0;l<(`GET_BUF_SIZE(BAND_SIZE));l=l+1) begin//not generate block but loop block
76          databuffer[l][k] <= 0; // reset everything
77        end
78      end
79    end
80    else if(array_en)
81      if (tricounter == 2'b00)
82        for(k=0;k<BAND_SIZE;k=k+1) begin
83          databuffer[write_pointer][k] <=  IN_flattened[k*DATA_WIDTH +: DATA_WIDTH];
84        end
85  end
```

- `output`: The output is determined using a combination of the other signals.

Listing 12: Verilog description of the `output` signal behaviour. This code snippet is a portion of Listing 57

```
66  generate
67    for(i=0;i<BAND_SIZE;i=i+1) begin
68      assign OUT_flattened[i*DATA_WIDTH  +: DATA_WIDTH] = (binary_pattern[i] == 1) ?
    ↪ databuffer[dispatch_pattern[i]][i] : 0;
69    end
70  endgenerate
```

### 3.4.2. Band Output Peripheral Device

The band output device takes the output vectors dispatched as previously seen in Figure 2.10. It receives the individually dispatched values and packs them into compact vectors. Only when these vectors are full, are they dispatched. A visualisation of the band output device functionality is represented of Figure 3.8.

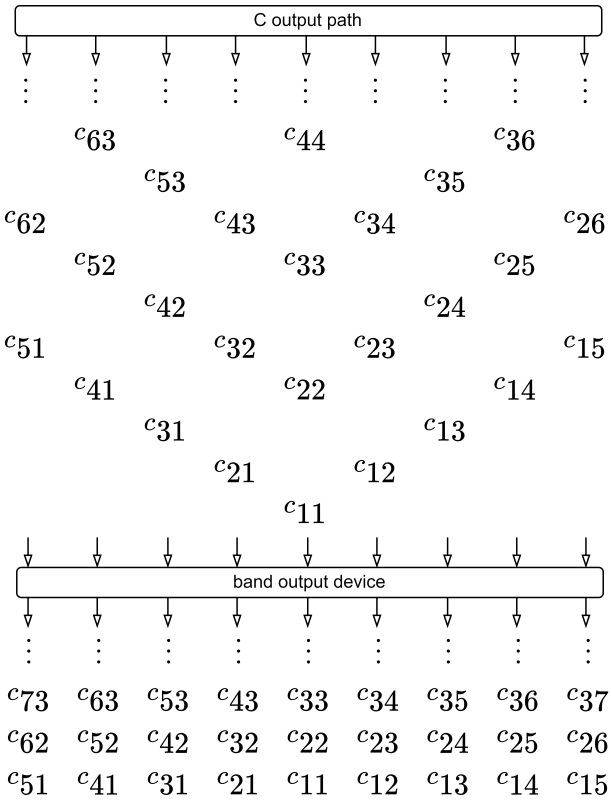


Figure 3.8: Visualisation of the workings of the band output device

Due to the similarities between this block and the band input device, we will only explain its functionality using Verilog code snippets.

The necessary signals are the following:

- `current_line`: This signal says which vector is the next to be dispatched.

Listing 13: Verilog description of the `current_line` signal behaviour. This code snippet is a portion of Listing 58

```
23  always @(posedge clk) begin
24    if(reset) current_line <= 0;
25    else if(array_en) if(binary_pattern[0]==1) current_line <= write_pointer[0];
26  end
```

- `binary_pattern`: This signal contains the pattern with which values should be written into the data buffer. This signal begins with a 1 in the center every three cycles which ripples outward in both directions.

Listing 14: Verilog description of the `binary_pattern` signal behaviour. This code snippet is a portion of Listing 58

```
28   always @(posedge clk) begin
29     for(k=0;k<=2*BAND_SIZE-2;k=k+1) begin
30       if(reset) binary_pattern[k] <= 0;
31       else if (array_en) begin
32         if(k==BAND_SIZE-1)      binary_pattern[k] <= (tricounter == 0) ? 1'b1 : 1'b0;
33         else if(k<BAND_SIZE-1)  binary_pattern[k] <= (tricounter == 2'b11) ? 1'b0 :
↪  binary_pattern[k+1];
34         else                    binary_pattern[k] <= (tricounter == 2'b11) ? 1'b0 :
↪  binary_pattern[k-1];
35       end
36     end
37   end
```

- `write_pointer`: This signals indicates which data buffer address should be written to, for each position.

Listing 15: Verilog description of the `write_pointer` signal behaviour. This code snippet is a portion of Listing 58

```
39   always @(posedge clk) begin
40     for(k=0;k<=2*BAND_SIZE-2;k=k+1) begin
41       if (reset) write_pointer[k] <= 0;
42       else if(array_en) begin
43         if(binary_pattern[k] == 1) begin
44           if(write_pointer[k] == (`GET_BUF_SIZE(BAND_SIZE))-1) write_pointer[k] <= 0;
45           else write_pointer[k] <= write_pointer[k] + 1;
46         end
47       end
48     end
49   end
```

- `data_buffer`: This is the data buffer which contains all temporary vectors until they are complete and ready to be dispatched. The buffer depth is the same as with the input device.

Listing 16: Verilog description of the `data_buffer` signal behaviour. This code snippet is a portion of Listing 58

```
51  always @(posedge clk) begin
52    for(k=0;k<=2*BAND_SIZE-2;k=k+1) begin
53      for(l=0;l<=(`GET_BUF_SIZE(BAND_SIZE))-1;l=l+1) begin
54        if (reset) databuffer[l][k] <= 0;
55        else if (array_en)
56          if(binary_pattern[k]==1)
57            if(write_pointer[k]==l)
58              databuffer[l][k] <= IN_flattened[k*DATA_WIDTH +: DATA_WIDTH];
59      end
60    end
61  end
```

- `output`: The output is simply made of vectors of numbers in the custom format described previously.

Listing 17: Verilog description of the `output` signal behaviour. This code snippet is a portion of Listing 58

```
64  generate
65    for(i=0;i<=2*BAND_SIZE-2;i=i+1)
66          assign OUT_flattened[(i+1)*DATA_WIDTH-1:i*DATA_WIDTH] = databuffer[current_line][i];
67  endgenerate
```

### 3.4.3. BMMM Peripheral Devices Assembly

The `band peripherals` module puts together two input devices and one output device. The whole band assembly is dictated by a single `tricounter`, which is a two-bit counter that increments upwards from `0b00` to `0b10` then cycles back to `0b00`. For correct operation, the output device must be controlled with a delayed version of the input `tricounter` regardless of the size of the array.

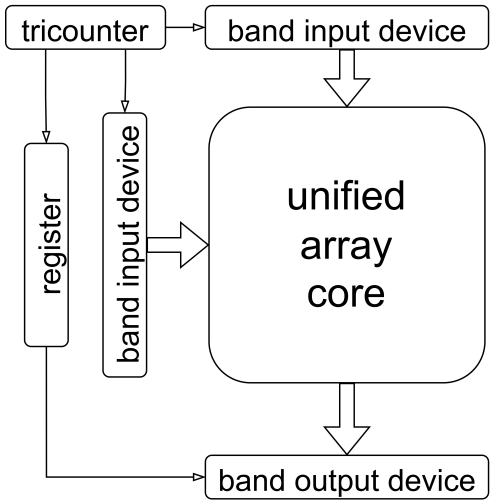


Figure 3.9: Unified array core surrounded by the necessary peripherals to enable BMMM

Note that because the `tricounter` is reset to `0b00`, the delayed version must be reset to the value that comes prior to `0b00`. In our case this value is `0b10` (or 2).

Listing 18: Verilog description of the implementation of the `delayed_tricounter`. This code snippet is a portion of Listing 59

```
53      reg [1:0] tricounter_delayed;
54
55     always @(posedge clk) begin
56       if (reset) tricounter_delayed <= 2;
57       else if(array_en) tricounter_delayed <= tricounter;
58     end
```

### 3.4.4. Unified Array

The only thing left to do to complete our unified array kernel is to put all the modules together. A final overview of the kernel can be seen in Figure 3.10. We can see the addition of some multiplexers controlled by the `opmode` signal, which chooses whether we are want to execute a BMMM or a GMMM.

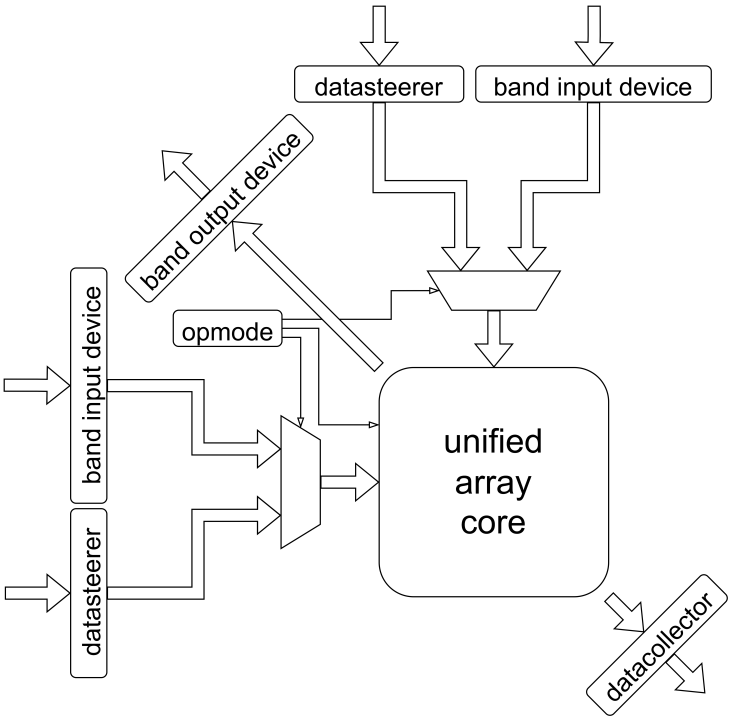


Figure 3.10: Unified array, assembling the unified array core and all its necessary peripherals for both BMMM and GMMM.

The description of this module is straightforward and features no notable particularities. This module can be consulted in Listing 59.

## 3.5. Summary

In this chapter, we have shown how we implemented our UMMM in RTL. We have also presented our motivation for using custom data formats for the BMMM operation.

In the next chapter, we will see how we integrated our kernel into a system. This will be achieved by writing a Matrix Multiplication Controller (MMC) which serves as the controller for our kernel and implements the necessary interfaces with the HLS hardware, which in turn will deal with data communication to and from the host machine.

This integration is enabled by Xilinx's blackbox design capabilities, which we will present in Section 4.4.

# 4 | Integration of the RTL Kernel into a System

## 4.1. Hardware: Alveo U280 Data Center Card

### 4.1.1. General Specifications

The Xilinx® Alveo™ U280 Data Center accelerator cards are Peripheral Component Interconnect express (PCIe®) Gen3 x16 compliant and Gen4 x8 compatible cards featuring the Xilinx 16 nm UltraScale+™ technology. The Alveo U280 card offers 8 GB of HBM2 at 460 GB/s bandwidth to provide high-performance, adaptable acceleration for memory-bound, compute-intensive applications including database, analytics, and machine learning inference.

### 4.1.2. FPGA

From [21], the Xilinx Alveo U280 accelerator card is a custom-built UltraScale+ FPGA that runs optimally (and exclusively) on the Alveo architecture. The Alveo U280 card features the XCU280 FPGA, which uses Xilinx stacked silicon interconnect (SSI) technology to deliver breakthrough FPGA capacity, bandwidth, and power efficiency. This technology allows for increased density by combining multiple super logic regions (SLRs). The XCU280 comprises three SLRs with the bottom SLR (SLR0) integrating an HBM controller to interface with the adjacent 8 GB HBM2 memory.

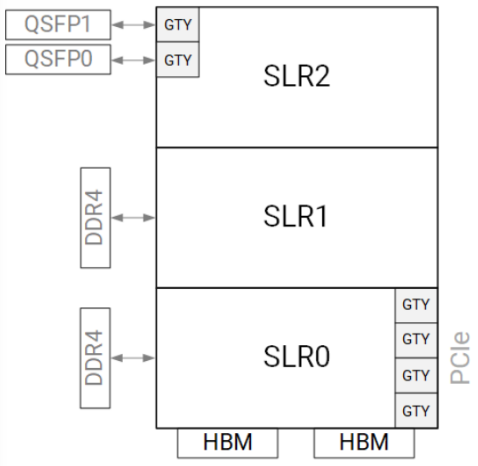


Figure 4.1: Floorplan of the XCU280 FPGA, from [21]

## 4.2. Tools

### 4.2.1. ModelSim

ModelSim is an RTL simulation software. It was designed by *Mentor Graphics* and is now distributed by many companies, such as Intel or Siemens, under slightly different releases. This software has the particularity of being barebones but excelling in reliability and speed. The free version (accessible to students) is limited in simulation size. It can also be used as the RTL simulator for larger software suites such as Xilinx Vivado or Intel Quartus. We have used this software as a RTL simulator in the early stages of designing and testing our unified array.

### 4.2.2. Xilinx Vivado

Vivado is an FPGA development tool which aims to unify the RTL simulation with synthesis and implementation into a single software. Naturally, this software is intended as a complete workflow environment for RTL development on Xilinx hardware. We have used this software to finalize our RTL design and as a separate support tool for Vitis.

### 4.2.3. Xilinx Vitis

Vitis is a unified software platform designed to be the complete work environment for developers designing hardware accelerators. It includes the HLS compiler, hardware linker, bitstream generator and host program development environment for systems including an FPGA and a host computer. Vitis englobes Vivado and uses it to synthesize, implement

and write the FPGA bitstream. We need to use Vitis in order to develop and run our hybrid RTL-HLS systolic array.

## 4.3. Vitis Design Workflow

Our workflow has been primarily based on the Linux Terminal version of Vitis. Vitis allows us to package many types of sources into a single hardware kernel. The different options for doing so are featured in Figure 4.2.

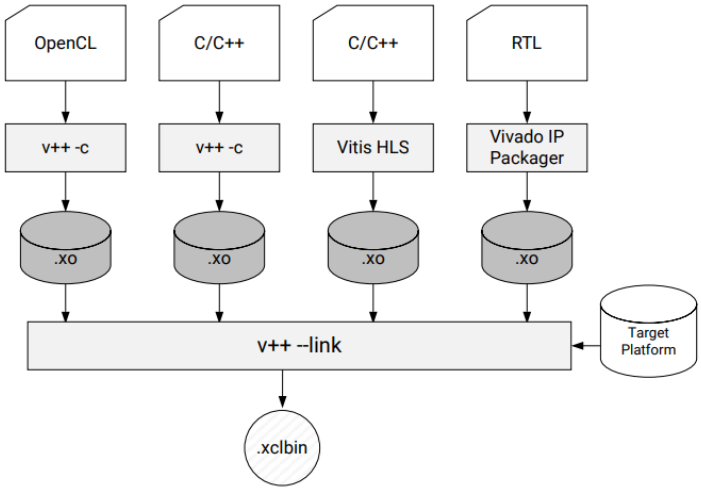


Figure 4.2: Vitis device build process, from [22]

### 4.3.1. Main Vitis Commands

Building a Vitis system requires three main steps, each having its own command. In the following section we will go through the building process.

#### Vitis Compile: `v++ -c`

The Vitis compile (-c) command builds all the source files into Xilinx Object (.xo) files. This process includes but is not limited to:

- Compiling the source `C/C++` files into their HDL-equivalent through Xilinx's high-level synthesis process.
- Checking that the interfaces between RTL and HLS blocks are valid when using RTL blackboxes.

#### Vitis Link: `v++ -l`

The Vitis link command takes the resulting `.xo` files and links them all together into a cohesive system. Depending on the target options, different kinds of files will be created and different processes will be launched.

#### Vitis Compile Host Code: `g++`

The Vitis Host Code compiler takes the host `C++` code into the appropriate machine code for the host computer.

### 4.3.2. Target Options

Vitis target options must be included when compiling and linking, using the `-t` tag.

#### Software Emulation (`-t sw_emu`)

This option allows us to compile everything into a fully-software version of our algorithm. This is particularly useful for fully HLS systems because it allows us to quickly and efficiently debug and check the validity of our `C++` algorithms. The compile time is the shortest among targets and is thus adequate for iterative design. It is also useful for hybrid HLS-RTL kernel workflows to create golden results to which the hardware results can later be compared.

#### Hardware Emulation (`-t hw_emu`)

This option allows us to compile the hardware into a netlist and normally compiles the software. It also compiles an emulation-equivalent version of the data transfer hardware between the host and the FPGA. The emulation is then run on an RTL simulation environment. This allows us to check the validity of our handmade RTL kernels and their interaction with the HLS portions of the device. Any internal signals can be probed post-linking, which makes the debugging workflow convenient and efficient. The compile time for hardware emulation is approximately one order of magnitude longer than for software emulation.

#### Hardware (`-t hw`)

This option allows us to run synthesis and implementation for our desired FPGA. This then lets us run the actual hardware in real life and in real time. The compile time for hardware takes the longest and should not be used for iterative design. For a similar

design, this option's compile time is an order of magnitude longer than for hardware emulation.

### 4.3.3. Debugging

All three Vitis flow steps must receive the debugging directive (`-g or -debug`). For the compile step, some debugging features will be activated in order to produce a debuggable design. For the linking step, the design will be implemented into the appropriate version of the design, including debugging capabilities. For the host code compilation, the directive is a simple `g++` debugging toggle.

#### Software Emulation Debugging

Software emulation debugging allows us to use the Linux console to print out any intermediate values of variables in order to check their validity and compliance to the design, much like we would in regular computer program debugging. This is true even for parts of the design which are intended to be compiled into hardware later on.

In order to keep compilable HLS functions, we must surround them with a precompiler macro conditional statement. This will ensure that when we are compiling the design into hardware, these sections of code will be ignored. An example of this can be seen in Listing 19.

Listing 19: Example of debugging code intended for `sw_emu`. This code snippet is a portion of Listing 64.

```
440 #ifndef __SYNTHESIS__
441     std::cout << "line_accu_bit: " << line_accu_bit << std::endl << std::flush;
442     std::cout << "chunk_accu_bit: " << chunk_accu_bit << std::endl << std::flush;
443 #endif
```

#### Hardware Emulation Debugging

Hardware emulation debugging allows us to use the Vivado simulator to run our design and probe any signals in our kernel and examine them cycle by cycle. This is extremely useful in order to debug and verify RTL designs. After adding the directives presented above, setting up hardware emulation debugging requires an additional step.

In order to open a Vivado session with a live waveform viewer, we must add the following code into a `xrt.ini` file before compiling the host code:

```
[Emulation]
debug_mode=gui
```

This will ensure that the Vivado simulation GUI is opened when launching the kernel, allowing us to probe any signals. When the GUI launches, the user must pause the simulation and probe the signals they need. This must be done every time the user relaunches the simulation. The user can also speed up this process by saving a waveform configuration file (`.wcfg`) and opening it manually every time the simulation starts.

#### Hardware debugging

Hardware debugging consists of implementing additional hardware into our FPGA fabric whose role is to record and relay specific signals' timeline traces back to the host computer. This hardware is often referred to as an Integrated Logic Analyzer (ILA). During our thesis, we were unable to include an ILA into our workflow. Luckily, relying on hardware emulation and synthesis reports has proven to be sufficient for debugging our systems.

### 4.3.4. Makefile

In order to package all our building necessities into concise commands, we will use a `Makefile`. This `Makefile` will call `v++ -c`, `v++ -l`, and `g++` with all the appropriate arguments and make the appropriate path managements in order to cleanly build our kernels. Our `Makefile` is heavily inspired by the work of S. Soldavini in [18].

## 4.4. Vitis Blackbox Design

Vitis allows us to replace a function in the source `C++` code with our own version of the circuit, essentially stitching a handmade kernel into an otherwise computer-generated RTL system. We will use this functionality to transform our RTL kernel into a Vitis blackbox, allowing us to implant it into a complete system. In the following section we will go through the procedure to set up a blackbox using Vitis.

### 4.4.1. Blackbox Signals

Vitis blackboxes use simple block-level control protocols to interact with the rest of the system. The two protocols which are supported for use in blackboxes are `ap_ctrl_hs` and `ap_ctrl_chain`. The prior is intended for single execution kernels and the latter is intended for pipelined execution kernels. These protocols use a handful of signals to com-

municate between the blackbox top module and the HLS peripherals. The documentation describing the blueprint for these protocols features in [23]. We will mainly focus on the `ap_ctrl_chain` protocol, which features the following signals:

- `ap_start`: This signal is an input to the blackbox. It asserts to the blackbox that it can begin operation. For non-pipelined designs, this signal is also intended as an indicator that the data is ready to be read on the inputs of the blackbox.
- `ap_idle`: This signal is an output of the blackbox. It signals to the wrapper if the blackbox is idle or not. It must be asserted low immediately and asynchronously when the `ap_start` signal is asserted high and it must be asserted high synchronously one cycle after the `ap_done` signal is asserted high.
- `ap_ready`: This signal is an output of the blackbox. It signals to the wrapper that the blackbox is ready to accept new data. For pipelined designs this signal can be permanently pulled high, because the data timing management is handled by FIFO interfaces. We will discuss this further in Section 4.4.2.
- `ap_done`: This signal is an output of the blackbox. It indicates to the wrapper that the operation is done.
- `ap_return`: This signal is an output of the blackbox. It indicates to the wrapper that the data is valid on the return line for simple designs. We will not need this signal and did not implement it because the data timing management will be handled by FIFO interfaces. We will discuss this further in Section 4.4.2.
- `ap_continue`: This signal is an input to the blackbox. It asserts to the blackbox that the next block in the kernel chain is ready to receive more data. We will not need this signal and did not implement it because the data timing management is handled by FIFO interfaces. We will discuss this further in Section 4.4.2.

### 4.4.2. Implementing FIFO's

Since our kernel's operation falls under the pipeline category, we must interface with input and output FIFO's. If we implement the following signals, the Vitis compiler will build a pipelined design.

- `input_empty_n`: This signal is an input to our blackbox. It is active-low and indicates if there is any data left in the FIFO to read. In other words, it indicates if the FIFO is empty. The blackbox must check if this signal is high before reading the associated input.
- `input_re`: This signal is an output to our blackbox. It must be asserted high when the blackbox is reading the data at the interface and signals to the input FIFO that the data has been read, can be discarded and should be replaced with the next data in the FIFO. The suffix `_re` refers to the common *read enable* terminology.
- `output_full_n`: This signal is an input to our blackbox. It is active-low and indicates if it is possible to write to the output FIFO. In other words, in indicates if the FIFO is full. The blackbox must check if this signal is high before attempting to write any data.
- `output_we`: This signal is an output to our blackbox. It must be asserted high when the blackbox is presenting data at the interface. It signals to the output FIFO that the data is valid on the interface, should be read and should be queued. The abbreviation `_we` refers to the common *write enable* terminology.

On the `C++` side of the design, these FIFO's present themselves as `C++` streams. The HLS `stream` library must be included into the HLS source code using `#include "hls_stream.h"`. A stream can then be instantiated using `hls::stream<data_type> stream_name;`. Data can be pushed into the FIFO with `stream_name << data;` and can be pulled from the FIFO with `stream_name >> data;`.

### 4.4.3. Linking RTL and HLS

In order for the HLS compiler to properly map the RTL module IO with its high-level `C++` representation of the kernel, we must fill a `JSON` file which defines all the correspondences between `C++` functions and RTL signals. The first part of the `JSON` file deals with source code file paths.

It is important to note that your blackbox function **must** have a corresponding `C++` function signature in your `C++` source code or header. If you are working with a purely RTL kernel, you must define a `C++` dummy function, with the appropriate function arguments. The body of the function can stay empty, because it will be replaced with its RTL equivalent during compilation of the kernel.

Listing 20: JSON declaration of the source file input paths. This code snippet is a portion of Listing 60.

```
{
    "c_function_name"     : "mmc",
    "rtl_top_module_name" : "mmc",
    "c_files" : [{
                  "c_file" : "/.../mmc.cpp",
                  "cflag" : ""
                }],
    "rtl_files" : [
                    "/.../mmc.v",
                    "/.../unified_array.v",
                    "/.../datasteering.v",
                    "/.../KLPE.v",
                    "/.../band_peripherals.v",
                    "/.../band_input_device.v",
                    "/.../band_output_select_and_route.v"
                  ],
```

We must then map each C++ function argument with its corresponding RTL top module inputs and outputs. Note that these entries will have different fields depending on the type of interface desired. In our case we have only implemented FIFO's for the inputs and outputs so we must define the FIFO_empty_flag, FIFO_read_enable, and FIFO_data_read_in ports.

Listing 21: JSON example declaration of the mapping between RTL signals and their C++ counterparts. This code snippet is a portion of Listing 60

```
    "c_parameters" : [
                        {
                          "c_name" : "opmode_stream",
                          "c_port_direction" : "in",
                          "rtl_ports" : {
                                        "FIFO_empty_flag" : "opmode_empty_n",
                                        "FIFO_read_enable" : "opmode_re",
                                        "FIFO_data_read_in" : "opmode"
                                        }
                        },
                        ...
                    ],
```

We must then define the RTL common signals, which we earlier referred to as block-level control protocol signals. They do not have C++ counterparts because they will be

automatically implemented by the high-level synthesis compiler. Note the addition of the purely RTL signals `ap_clk`, `ap_rst`, and `ap_ce`, which are self-explanatory.

Listing 22: `JSON` declaration of the control signals. This code snippet is a portion of Listing 60

```
99    "rtl_common_signal" : {
100                           "module_clock"                    : "ap_clk",
101                           "module_reset"                    : "ap_rst",
102                           "module_clock_enable"             : "ap_ce",
103                           "ap_ctrl_chain_protocol_idle"     : "ap_idle",
104                           "ap_ctrl_chain_protocol_start"    : "ap_start",
105                           "ap_ctrl_chain_protocol_ready"    : "ap_ready",
106                           "ap_ctrl_chain_protocol_done"     : "ap_done",
107                           "ap_ctrl_chain_protocol_continue" : "ap_continue"
108                         },
```

The next section allows the user to input data about the kernel. Functionally, no part of this section is useful for us. In designs which do not feature input or output FIFO's, the `latency` field dictates for how long the `ap_ce` signal remains high after the last `ap_start` signal goes high. The `Initiation Interval` (`II`) field indicates the minimal interval between consecutive launches of our kernel. Since our design features FIFO's, we did not notice any influence from these fields. The Xilinx documentation [22, 23] is unclear about this field's use under these circumstances. We will thus set them to dummy values.

The last section allows for data about the RTL IP to be manually input. This fields have also been filled with dummy values because these values are not fixed throughout our designs and do not have any functional purpose.

Listing 23: `JSON` declaration of performance and resource usage data. This code snippet is a portion of Listing 60

```
109   "rtl_performance" : {
110                         "latency" : "0",
111                         "II"      : "1"
112                       },
113   "rtl_resource_usage" : {
114                           "FF"   :   "0",
115                           "LUT"  :   "0",
116                           "BRAM" :   "0",
117                           "URAM" :   "0",
118                           "DSP"  :   "1"
119                          }
```

### 4.4.4. Compiling a Blackbox

When the `JSON` and the source files have been prepared, we must instruct the compiler to include the blackbox in the design. This can be done by running the Tcl command: `add_files -blackbox path.json` before HLS begins. Realistically, this can be done by putting this instruction into a `.tcl` file and running the `v++ -c` command with the following option: `-hls.pre_tcl=pre.tcl`.

### 4.4.5. Blackbox Design Notes and Complications

**RTL C function prototype** Since the Vitis workflow is primarily designed for fully HLS designs, a kernel function prototype must **always** be defined. Thus, when designing our RTL kernel, in order to make a valid blackbox interface, we had to include a dummy function with the same inputs and outputs as our RTL function. Although the exact specifications of such a function are still unclear to us, we think that it is good practice to at least read the inputs and put some data on the output, for good measure. We have also found that reading and writing some data in the dummy function has been quite useful in order to be able to use the `sw_emu` feature to quickly debug the HLS peripherals. An even better approach is to always write a software version of your kernel instead of a dummy one, if it is possible.

**Properly defining functions in a HLS source code** When writing functions in an HLS source code, their prototype must always feature in the header file and must be surrounded by an `extern C{}` construct. If this rule is not respected, the compilation of the source `C` code will fail and the error messages are not obvious to decode.

**Unusual behaviour of the Vitis Verilog parsing during HLS compilation** When defining a Verilog top module with global parameters, we advise you to use our example as a starting point. Any other configuration of line breaks and spaces in the lines surrounding the `module` keyword will result in a failure to compile and one of the following error messages to be displayed:

```
1    ERROR: [v++ 200-653] Can not find blackbox json port 'xxx' in the port list of RTL top module 'xxx'
2    ERROR: [v++ 200-654] Cannot find blackbox RTL port 'module' in the json file
```

To be more precise, a space must be included between the module name and the # symbol and a line break must be used after the parameter declaration section. This information

is not featured in any Xilinx documentation and has been discovered through trial and error. The configuration we came up with which works is the following:

Listing 24: Working syntax for the header of a Verilog top module including global parameters. Note the space between `module` and the `#` character. Note also the line break after the global parameters and before the declaration of the inputs and outputs. This code snippet is a portion of Listing 61

```
29 module mmc #(parameter MAT_SIZE=4, DATA_WIDTH=8)
30       (input ap_clk, ap_rst, ap_ce, ap_start, ap_continue,
```

## 4.5. Hybrid HLS-RTL Implementation

In this section we will go through the last blocks necessary to implement a fully working system. We will first go through the top RTL module which must implement the aforementioned `ap_ctrl_chain` protocol and synchronizes the data management with the kernel control signals of the unified array. We will then discuss the HLS blocks we have implemented to bridge the RTL kernel with the host computer. The final block diagram of the full RTL-HLS hybrid system is featured in Figure 4.3.

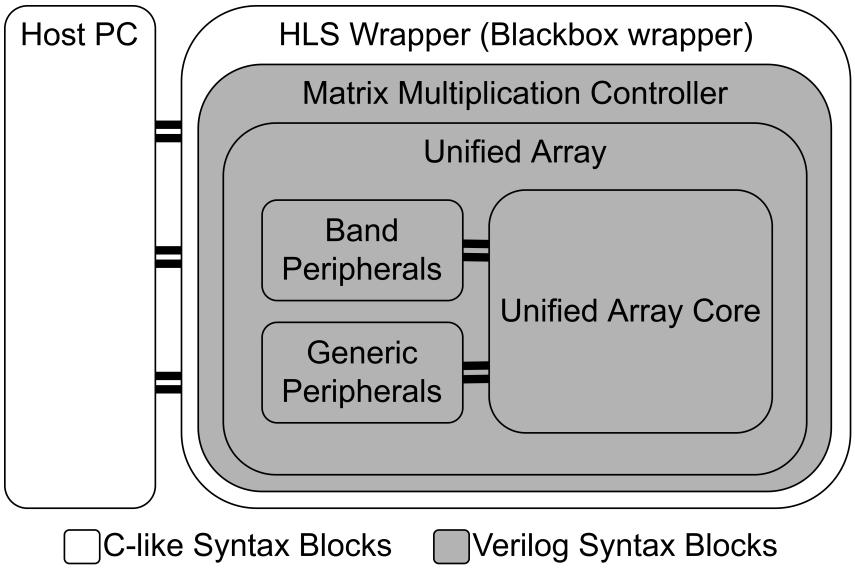


Figure 4.3: Block diagram of the entire system

### 4.5.1. Matrix Multiplication Controller

The Matrix Multiplication Controller (MMC) deals not only with reading and writing data from and to the input and output FIFO's but also with synchronizing the kernel's operating counter and enable signals with the availability of data. The MMC is also the interface with the surrounding HLS portions of the design and thus must implement the `ap_ctrl_chain` protocol. It is implemented as a complex Finite State Machine (FSM).

Our FSM features 8 main states, a main counter and some states also employ sub-states. The FSM macro-diagram can be found in Figure 4.4.

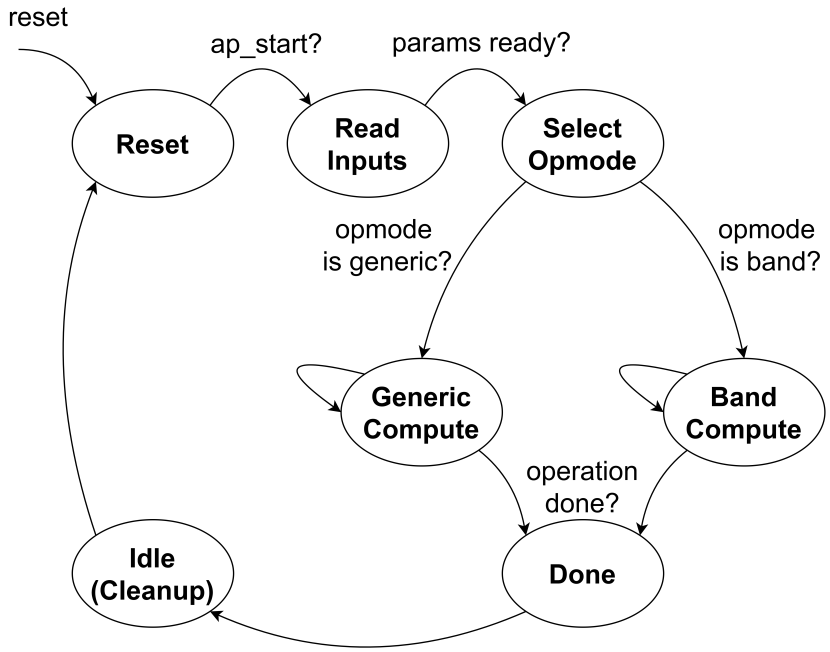


Figure 4.4: FSM implemented in the MMC

The Verilog FSM style we used divides the state transitions into a clocked `always` block and the next state logic into a separate combinational `always` block. This structure can be seen in Listing 25. Note that in addition to our state transitions and counter transitions, the clocked `always` block also features two other signals. These are part of a speedup trick we employed to increase our maximum clock speed and will be discussed later. Also note the comment on line 174. When using a segmented FSM style, all the registers which are not assigned in the clocked `always` block must be given a default value in the combinational `always` block, **outside** of the `case` statement. This will ensure that latches are not inferred in the design. Failure to prevent latches will result in hardware which does **not** guarantee behaviour equivalence with hardware emulation.

Listing 25: Verilog template for our top-level finite state machine. This code snippet is a portion of Listing 61

```
154 always @ (posedge ap_clk)
155 if (ap_rst) begin
156   current_state <= s_reset;
157   counter <= 0;
158   tricounter <= 0;
159
160   band_duration <= 1;
161   start_band_validity <= 1;
162 end
163 else if (ap_ce) begin
164   current_state <= next_state;
165   counter <= next_counter;
166   tricounter <= next_tricounter;
167
168   band_duration <= next_band_duration;
169   start_band_validity <= next_start_band_validity;
170 end
171
172 always @(*)
173 begin
174 // HERE GO THE DEFAULT ASSIGNMENTS TO PREVENT LATCHING
175   case(current_state)
176 // HERE GOES THE NEXT STATE LOGIC
177   endcase
178 end
```

In the following part, we will go over each of our states and describe what they do.

### FSM: Reset

- Resets all the registers to a known state
- Moves to the next state when the `ap_start` signal is asserted

Listing 26: Verilog description of the `reset` state. This code snippet is a portion of Listing 61

```
199     s_reset: begin
200        // reset all the output registers in the reset state
201       a_band_re     = 0;
202       b_band_re     = 0;
203       a_gen_re      = 0;
204       b_gen_re      = 0;
205       opmode_re     = 0;
206       size_re       = 0;
207       band_type_re  = 0;
208       c_band_we     = 0;
```

```
209       c_gen_we      = 0;
210
211       array_en_reg  = 0;
212       inputs_zero   = 0;
213
214       next_state = s_reset;
215       next_counter = 0;
216       next_tricounter = 0;
217
218       if (ap_start) begin
219         next_state = s_readparams;
220       end
221     end
```

### FSM: Read Parameters

- Waits until the parameter FIFO's are all populated, then moves to the next state.

Listing 27: Verilog description of the `read parameters` state. This code snippet is a portion of Listing 61

```
223     s_readparams: begin
224       opmode_re     = 0;
225       size_re       = 0;
226       band_type_re  = 0;
227       next_state = s_readparams;
228       if (opmode_empty_n && band_type_empty_n && size_empty_n) begin
229         next_state = s_selectopmode;
230       end
231     end
```

### FSM: Select Opmode

- Computes and stores some constants which involve multiplications of the input parameters with other constants. This is an unplanned addition of functionality to this state which helped resolve some timing bottlenecks and allowed us to increase the clock speed of our kernel.
- Moves either to the generic compute state, or to the band compute state, depending on the value read from the `opmode` parameter.

Listing 28: Verilog description of the `select opmode` state. This code snippet is a portion of Listing 61

```
233     s_selectopmode: begin
234
235       next_start_band_validity = `START_BAND_VALIDITY; // make the actual calculation here
236       next_band_duration = `BAND_DURATION;
237
238       case(opmode)
239         mode_gen: begin
240           next_state = s_gencompute;
241           next_counter = 0;
242           next_tricounter = 0;
243         end
244
245         mode_band: begin
246           next_state = s_bandcompute;
247           next_counter = 0;
248           next_tricounter = 0;
249         end
250
251         default: begin
252           next_state = s_done;
253         end
254       endcase
255     end
```

### FSM: Generic Compute

- Deals with timing, array enabling, data reading and writing in order to enable the GMMM operation. This state works in three phases. First, it must cycle through the computation while reading the input data. Then, it must cycle through the computation without taking in more inputs. Lastly, it must cycle through the computation while only writing out data. While working, this state always checks if the inputs are ready to be read and the output is ready to be written to. If any condition is missing, the entire operation is stalled and waits for the problems to be resolved. Note that some modifications will be made to this state in 4.5.4.
- Moves to the `done` state when the operation is finished. This is determined by comparing the counter with the theoretical value the counter should have when the operation is complete. This value is computed from the input parameters.

Listing 29: Verilog description of the `generic compute` state. This code snippet is a portion of Listing 61

```
267     s_gencompute: begin
268       // go into this state with counter = 0 and substate = 0;
269       //cleanup the re signals from previous state
270       //for generic operation, reading and writing are separate in time,
271       //thus we do not need to check if the inputs and the outputs are both free to enable the register
272
273     //-------deal with inputs ----------/
274       a_gen_re  = 0;
275       b_gen_re  = 0;
276       next_counter  =  counter;
277       array_en_reg = 0;
278       inputs_zero = 0;
279       c_gen_we = 0;
280
281       case(gen_substate)
282         ss_0: begin //read inputs
283           if (a_gen_empty_n && b_gen_empty_n) begin // overwrite previous statements in necessary
284             a_gen_re  = 1;
285             b_gen_re  = 1;
286             next_counter = counter + 1;
287             array_en_reg = 1;
288           end
289         end
290
291         ss_1: begin //dead cycling
292           a_gen_re = 0;
293           b_gen_re = 0;
294           next_counter = counter + 1;
295           array_en_reg = 1; //corresponding 0 is up top
296           inputs_zero = 1;
297         end
298
299         ss_2: begin //write outputs
300           if (c_gen_full_n) begin
301             c_gen_we = 1;
302             next_counter = counter + 1;
303             array_en_reg = 1;
304           end
305         end
306       endcase
307
308       //------mark end of operation-------/
309       next_state = current_state;
310       if(end_of_gen_op) begin
311         next_state = s_done;
312         next_counter = 0; //reset the counter, you never know
313       end
314     end
```

### FSM: Band Compute

- Deals with timing, array enabling and data reading and writing. The band operation data needs are more complex and result in a more complicated state. We first note that we have employed a structure of substates which are dictated by the combination of the `must_read` and `must_write` flags. These flags are set according to the parameters computed in the `select_opmode` state and the value of `tricounter`. The different substates are: read only, read and write, write only, cycle without reading or writing. In order to properly complete the operation we also have a `inputs_zero` signal which sets all inputs to zero when no more data is to be sent to the array. This is necessary because of the direction of flow of the data. At the end of the operation, if we do not set all the inputs to zero when cycling the operation, the non-zero leftovers in the FIFO ports will contaminate the calculations.
- Moves to the `done` state when the operation is finished. This is determined by comparing the counter with the theoretical value the counter should have when the operation is complete. This value is computed from the input parameters.

Listing 30: Verilog description of the `band compute` state. This code snippet is a portion of Listing 61

```
325     s_bandcompute: begin // reachable now. The design has to be smart.
326       inputs_zero = 0;
327       array_en_reg = 0;
328       next_counter = counter;
329       next_tricounter = tricounter;
330       must_read = 0;
331       must_write = 0;
332
333       a_band_re = 0;
334       b_band_re = 0;
335       c_band_we = 0;
336       if (counter >= band_duration) begin
337         inputs_zero = 1;
338       end
339       if(tricounter == 2'b00 && counter < band_duration) begin // if the counter is below the band
    ↪  duration then we must read
340         must_read = 1;
341       end
342
343       if (counter >= start_band_validity && tricounter == 2'b00) begin
344         must_write = 1;
345       end // then we must write
346
347
348       //put a case statement here
349       case(band_substate)
350         ss_0: begin// must only read
351           if(a_band_empty_n && b_band_empty_n) begin //read and cycle
```

```
352             a_band_re = 1;
353             b_band_re = 1;
354             next_counter = counter + 1;
355             next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
356             array_en_reg = 1;
357           end //else jsut wait
358         end
359
360         ss_1: begin// must read and write
361           if(a_band_empty_n && b_band_empty_n && c_band_full_n) begin
362             a_band_re = 1;
363             b_band_re = 1;
364             c_band_we = 1;
365             next_counter = counter + 1;
366             next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
367             array_en_reg = 1;
368           end // else just wait
369         end
370
371         ss_2: begin // must only write
372           if(c_band_full_n) begin
373             c_band_we = 1;
374             next_counter = counter + 1;
375             next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
376             array_en_reg = 1;
377           end
378         end
379
380         ss_3: begin //must do nothing but still cycle
381           next_counter = counter + 1;
382           next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
383           array_en_reg = 1;
384         end
385
386         default: begin // must do nothing, same as ss_3
387           next_counter = counter + 1;
388           next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
389           array_en_reg = 1;
390         end
391       endcase
392
393
394       next_state = s_bandcompute;
395       if (end_of_band_op) next_state = s_done;
396     end
```

### FSM: Done

- Empties the parameter FIFO's
- Is used to set the blackbox output `ap_done` signal
- Moves to the `idle` state immediately

Listing 31: Verilog description of the `generic compute` state. This code snippet is a portion of Listing 61

```
398     s_done: begin // put all the output we to 0;
399       opmode_re     = 1;
400       size_re       = 1;
401       band_type_re  = 1;
402       c_gen_we = 0;
403       c_band_we = 0;
404       next_state = s_idle;
405     end
```

**FSM: Idle**

- Provides cleanup for the FIFO's which require it
- Is used to set the blackbox output `ap_idle`
- Moves to the `reset` state immediately

Listing 32: Verilog description of the `idle` state. This code snippet is a portion of Listing 61

```
407     s_idle: begin
408       opmode_re     = 0;
409       size_re       = 0;
410       band_type_re  = 0;
411       next_state = s_reset;
412     end
```

### 4.5.2. HLS Wrapper

The HLS wrapper is an HLS block which will wrap around our kernel and deal with memory transfers to and from the Host computer. It is organised into three distinct functions and is structurally inspired by the work of S. Soldavini in [18]. This structure is also the one suggested by Xilinx in [22].

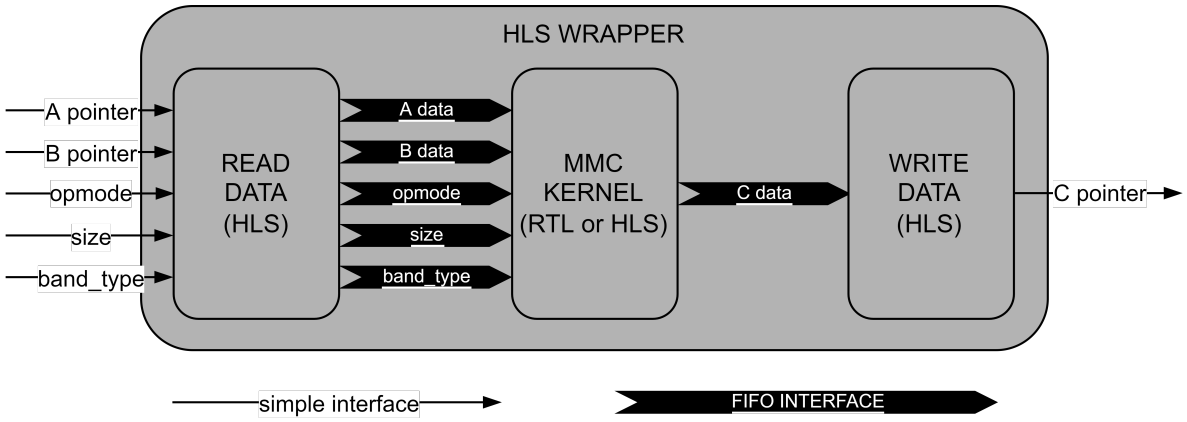


Figure 4.5: Internal structure of the HLS wrapper

## Read Data Function

The read function is tasked with receiving chunks of data from the High Bandwith Memory (HBM) and packing them into the correct line width to feed to the kernel. This is of course because our kernel needs its data to be delivered line by line. An astute reader might question why we did not use simple interfaces instead of FIFO's for the parameters, since they do not require to be streamed. This is simply due to a limitation from Vitis blackboxes. A blackbox is not allowed to have simple interfaces and FIFO's at the same time. Since our design inherently uses FIFO's for the streaming of input and output data, we had to move every input and output to FIFO's, including the one-time parameters.

**Naive Implementation** Our naive implementation achieves the reading of the HBM chunks by taking data-wide pieces from the chunks and putting them into a data-wide FIFO. Once the data is in a data-wide FIFO, we then pull the necessary amount of data and fill a line-wide FIFO with it. This line-wide FIFO is the one from which our blackbox can read data.

Listing 33: Portion of the naive (almost) fully parametric data delivery function, showcasing the GMMM

```
41        for(unsigned int i=0;i<num_transfers;i++){
42            chunk_A = in_A[i];
43            chunk_B = in_B[i];
44            for(unsigned int j=0;j<data_per_transfer && j<elems_left;j++){ // this will fill the fifo
    ↪  with the data
45                A_data_fifo << chunk_A.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
46                B_data_fifo << chunk_B.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
47            }
48            elems_left -= data_per_transfer;
49        }
```

```
50          for(unsigned int i=0;i<size;i++){ // for each line, pull out as many elements as you need,
51          //concatenate them and put them into the line FIFO (connected to the blackbox)
52              gen_in_t A_line;
53              data_t A_data;
54              gen_in_t B_line;
55              data_t B_data;
56              for(unsigned int j=0;j<size;j++){ // this will fill the fifo with the data
57                  A_data_fifo >> A_data;
58                  A_line.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH) = A_data.range();
59                  B_data_fifo >> B_data;
60                  B_line.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH) = B_data.range();
61              }
62              A_gen_line << A_line;
63              B_gen_line << B_line;
64          }
```

Although simple in nature, this technique has its obvious drawbacks, the first being it restricts the data bit-width to be a power of two so that it properly fits in the chunks. It also does not work if the data width is larger than the size of the chunk.

The ultimate drawback this design is its inherent slowness. We will illustrate this statement with an example. In order to fill a 16-element wide line of data, it must pull from the data-FIFO 16 times. We can thus expect that this will take 16 cycles if an element is pulled each cycle. This could result in it being a severe bottleneck since it only allows the kernel to run 1 out of every 16 cycles. This equates to a 94% stall rate, for such a configuration. It also becomes slower with the increase of the size of the input lines, which is counterproductive. Figure 4.6 displays this algorithm.

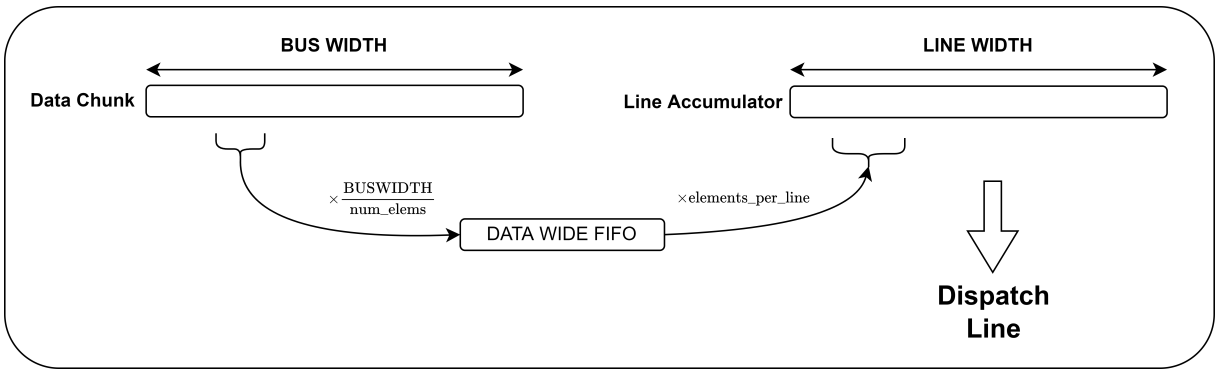


Figure 4.6: Diagram of the naive read data algorithm

**Optimised implementation** Our optimised Implementation achieves much better performances in theory by filling the line-wide FIFO directly. We use pointers to keep track of our position in the chunks and in our temporary line. Each cycle we pull as many bits

as we can, being either limited by the bits left in the chunk or by the bits needed to fill a line of data. Whenever a line is ready it is dispatched to the kernel. As well as working with any size of data, not just powers of two, this algorithm also allows for line widths which are larger than the chunk width, taking more than one chunk to entirely fill up a line. Figure 4.7 displays this algorithm and Listing 34 shows our implementation.

Listing 34: Portion of the optimised fully parametric data delivery function, showcasing the GMMM

```
40          gen_in_t A_line_accu;
41          gen_in_t B_line_accu;
42
43          for(unsigned int i=0;i<num_transfers;i++){
44
45              chunk_A = in_A[i];
46              chunk_B = in_B[i];
47              chunk_accu_bit = 0;
48
49              while(chunk_accu_bit < BUSWIDTH){ //while the chunk still has data to pull
50                  bits_to_be_pulled = min(BUSWIDTH-chunk_accu_bit,GEN_IN_SIZE-line_accu_bit); //determine
    ↪  if we are line-limited or chunk-limited
51                  A_line_accu.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit) =
    ↪  chunk_A.range(chunk_accu_bit + bits_to_be_pulled - 1,chunk_accu_bit);
52                  B_line_accu.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit) =
    ↪  chunk_B.range(chunk_accu_bit + bits_to_be_pulled - 1,chunk_accu_bit);
53                  chunk_accu_bit += bits_to_be_pulled;
54                  line_accu_bit += bits_to_be_pulled;
```

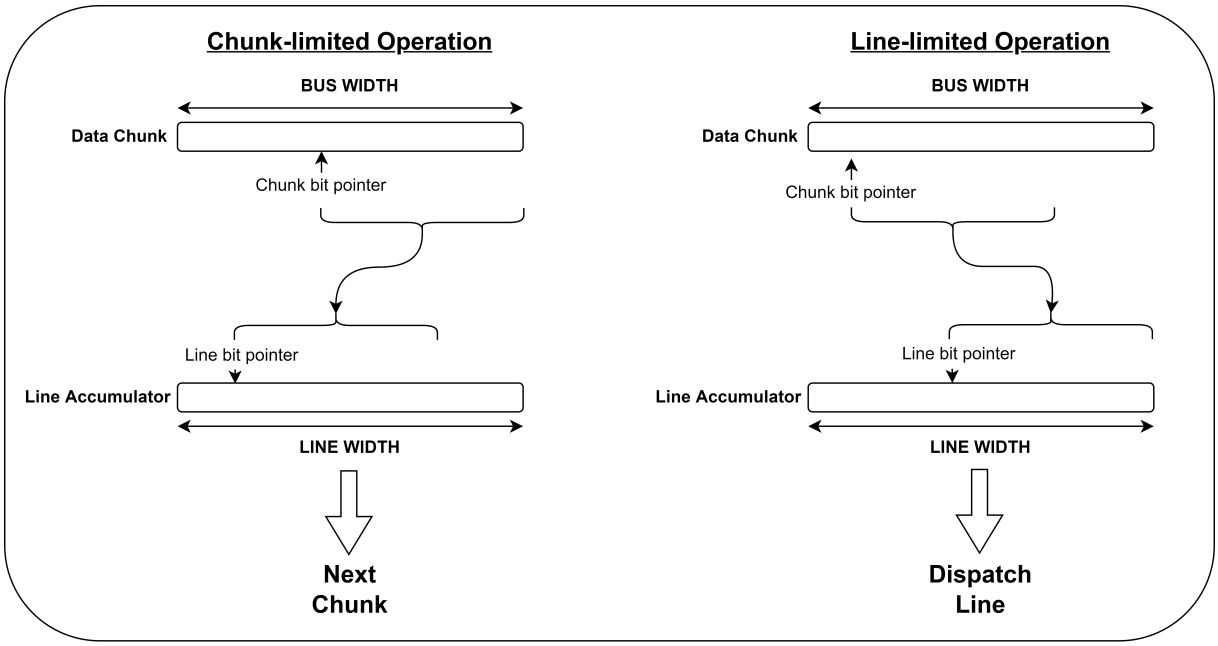


Figure 4.7: Diagram of the optimised read data algorithm

### Write Data Function

The write function is tasked with receiving the output data lines from the kernel and packaging them into HBM-compatible chunks of data. It is designed exactly like the read function and thus will not be further discussed. The writing functions in their entirety can be consulted in Listing 64.

## 4.5.3. Host Code

Our host code is based on the `OpenCL` 1.2 API. In the following sections, we will describe the steps to build up the Host code.

### Platform and Devices

We must go through all Platforms to find the Xilinx Platform and then find our device.

Listing 35: Host code portion showcasing the Platform and Devices. This code snippet is a portion of Listing 63

```
85      std::vector<cl::Device> devices;
86      cl::Device device;
87      cl_int err;
88      cl::Context context;
89      cl::CommandQueue q;
90      cl::Kernel krnl_matrix_mult;
91      cl::Program program;
92      std::vector<cl::Platform> platforms;
93      bool found_device = false;
94
95      //traversing all Platforms To find Xilinx Platform and targeted
96      //Device in Xilinx Platform
97      cl::Platform::get(&platforms);
98      for(size_t i = 0; (i < platforms.size() ) & (found_device == false) ;i++){
99          cl::Platform platform = platforms[i];
100         std::string platformName = platform.getInfo<CL_PLATFORM_NAME>();
101         if ( platformName == "Xilinx"){
102             devices.clear();
103             platform.getDevices(CL_DEVICE_TYPE_ACCELERATOR, &devices);
104             if (devices.size()){
105                     device = devices[0];
106                     found_device = true;
107                     break;
108             }
109         }
110     }
111     if (found_device == false){
112        std::cout << "Error: Unable to find Target Device "
113            << device.getInfo<CL_DEVICE_NAME>() << std::endl;
114        return EXIT_FAILURE; }
```

## Context and Command Queues

The next step is to set up the context and the command queues.

Listing 36: Host code portion showcasing the Context and Command Queues creation. This code snippet is a portion of Listing 63

```
118     OCL_CHECK(err, context = cl::Context(device, NULL, NULL, NULL, &err));
119     OCL_CHECK(err, q = cl::CommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &err));
```

## Load the Binary and Program the FPGA

The next step is to find the device binary, load it into memory and use it to program the FPGA.

Listing 37: Host code portion showcasing binary reading and FPGA programming. This code snippet is a portion of Listing 63

```
121     std::cout << "INFO: Reading " << xclbinFilename << std::endl;
122     FILE* fp;
123     if ((fp = fopen(xclbinFilename.c_str(), "r")) == nullptr) {
124         printf("ERROR: %s xclbin not available please build\n", xclbinFilename.c_str());
125         exit(EXIT_FAILURE);
126     }
127 
128     // Load xclbin
129     std::cout << "Loading: '" << xclbinFilename << "'\n";
130     std::ifstream bin_file(xclbinFilename, std::ifstream::binary);
131     bin_file.seekg (0, bin_file.end);
132     unsigned nb = bin_file.tellg();
133     std::cout << "number of program bytes: " << nb << std::endl;
134     bin_file.seekg (0, bin_file.beg);
135     char *buf = new char [nb];
136     bin_file.read(buf, nb);
137 
138 
139     std::cout << "Creating Program from binary file..." << std::endl;
140     // Creating Program from Binary File
141     cl::Program::Binaries bins;
142     bins.push_back({buf,nb});
143     devices.resize(1);
144     std::cout << "OK" << std::endl << "Programming Device...";
145     OCL_CHECK(err, program = cl::Program(context, devices, bins, NULL, &err));
```

## Kernel Object Creation and Buffer Allocation

The next step is to create a kernel object from the program. We then allocate memory on the device. We set up two buffers for the inputs and one buffer for the output with the appropriate space.

Listing 38: Host code portion showcasing the Context and Command Queues creation. This code snippet is a portion of Listing 63

```
147     std::cout << "OK" << std::endl << "Calling Kernel...";
148     // This call will get the kernel object from program. A kernel is an
149     // OpenCL function that is executed on the FPGA.
150     OCL_CHECK(err, krnl_matrix_mult = cl::Kernel(program,"hls_wrapper", &err));
151     std::cout <<  "OK" << std::endl << "Allocating memory...";
152     // These commands will allocate memory on the Device. The cl::Buffer objects can
153     // be used to reference the memory locations on the device.
154     OCL_CHECK(err, cl::Buffer buffer_A(context, CL_MEM_READ_ONLY, gen_matrix_size_in_bytes, NULL, &err));
155     OCL_CHECK(err, cl::Buffer buffer_B(context, CL_MEM_READ_ONLY, gen_matrix_size_in_bytes, NULL, &err));
156     OCL_CHECK(err, cl::Buffer buffer_C(context, CL_MEM_WRITE_ONLY, gen_matrix_size_in_bytes, NULL,
↪   &err));
```

## Setting the Kernel Arguments

The next step is to set the kernel arguments. They must be set in the same order as the order in which they are in the top level function of our C++ kernel. We will set first our input and output buffers, then our kernel parameters.

Listing 39: Host code portion showcasing the Kernel Arguments setup. This code snippet is a portion of Listing 63

```
162     OCL_CHECK(err, err = krnl_matrix_mult.setArg(0,buffer_A));
163     OCL_CHECK(err, err = krnl_matrix_mult.setArg(1,buffer_B));
164     OCL_CHECK(err, err = krnl_matrix_mult.setArg(2,buffer_C));
165     OCL_CHECK(err, err = krnl_matrix_mult.setArg(3,size));
166     OCL_CHECK(err, err = krnl_matrix_mult.setArg(4,opmode));
167     OCL_CHECK(err, err = krnl_matrix_mult.setArg(5,band_type));
```

## Filling the Input Data and Mapping Pointers to Buffers

In the next code portion, we will map C++ pointers to the previously allocated memory buffers. We will then fill the input memories via these pointers with the appropriate data. In this case we will fill each element with consecutive integers.

Listing 40: Host code portion showcasing input data filling and mapping pointers to Buffers. This code snippet is a portion of Listing 63

```
175     data_t *ptr_A;
176     data_t *ptr_B;
177     data_t *ptr_C;
178 
179     OCL_CHECK(err, ptr_A = (data_t*)q.enqueueMapBuffer (buffer_A , CL_TRUE , CL_MAP_WRITE , 0,
    ↪  gen_matrix_size_in_bytes, NULL, NULL, &err));
180     OCL_CHECK(err, ptr_B = (data_t*)q.enqueueMapBuffer (buffer_B , CL_TRUE , CL_MAP_WRITE , 0,
    ↪  gen_matrix_size_in_bytes, NULL, NULL, &err));
181     OCL_CHECK(err, ptr_C = (data_t*)q.enqueueMapBuffer (buffer_C , CL_TRUE , CL_MAP_READ  , 0,
    ↪  gen_matrix_size_in_bytes, NULL, NULL, &err));
182     std::cout << "OK" << std::endl << "Preparing the input data..." << std::flush;
183 
184     //fill in the matrices with relevant numbers here.
185     for(unsigned int k=0;k<size*size;k++){
186             ptr_A[k] = k;
187         ptr_B[k] = k;
188     }
```

## Migrating Data and Launching the Kernel

The next step is to migrate the data from the host computer to the FPGA and launch the kernel.

Listing 41: Host code portion showcasing Data Migration and Kernel Launching. This code snippet is a portion of Listing 63

```
205         OCL_CHECK(err, q.finish());
206         OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_B},0/* 0 means from host*/));
207         OCL_CHECK(err, q.finish());
208         //Launch the Kernel
209         OCL_CHECK(err, err = q.enqueueTask(krnl_matrix_mult));
210         OCL_CHECK(err, q.finish());
211         // The result of the previous kernel execution will need to be retrieved in
212         // order to view the results. This call will transfer the data from FPGA to
213         // source_results vector
214         OCL_CHECK(err, q.enqueueMigrateMemObjects({buffer_C},CL_MIGRATE_MEM_OBJECT_HOST));
215         OCL_CHECK(err, q.finish());
216     }
```

### Releasing the Resources

Finally, after executing the kernel, we must release the resources and clean-up everything.

Listing 42: Host code portion showcasing Data Migration and Kernel Launching. This code snippet is a portion of Listing 63

```
245     OCL_CHECK(err, err = q.enqueueUnmapMemObject(buffer_B , ptr_B));
246     OCL_CHECK(err, err = q.enqueueUnmapMemObject(buffer_C , ptr_C));
247
248     OCL_CHECK(err, err = q.finish());
```

## 4.5.4. RTL Improvement: GMMM Streamability

Although we designed our GMMM core to be streamable, our FSM did not allow it to reach its full potential. Thus we have decided to redesign our FSM in order to accomodate for streamability. The first obstacle for streamability is that the host must be able to communicate to the kernel that it is sending multiple input matrices one after the other which are intended to be multiplied together. Since we had an extra parameter `band_type` which was unused for the GMMM, we decided to reuse it as a signal that indicates how many matrices should be multiplied together. This way we can launch the kernel, execute a certain number of matrix multiplications back-to-back and then stop the execution. The modifications in the kernel are fully contained within the FSM.

We needed to add a local generic counter, which will count from `0` to `MAT_SIZE - 1` then start over at 0. This will ensure that the steering mechanisms are properly fed throughout the numerous operations. The next step is to change the state model to look more like the BMMM. We split the logic into `must_read` and `must_write` signals whose combinations determine the substate. Indeed, now that many operations can be streamed, it is possible that we need to read and write at the same time. We must also modify the HLS wrapper read and write functions to read and write more lines of data according to the `band_type` parameter. These changes can be consulted in Listings 62 and 64.

## 4.5.5. RTL Improvement: Breaking DSP chains

When compiling different sized arrays, we noticed that we began getting problems when trying to compile arrays of `MAT_SIZE` larger than 16. When this was attempted, the implementation would fail and the following error would occur: “Failed to build a DSP

chain shape. The height of the device or SLR DSP column is 110 DSPs. Modify the source so that the chain will fit onto one device or SLR DSP column". This error indicated that somehow, some carry chain exists between our DSP's. This is not normal and does not correspond to our design. We have thus assumed that some optimisation step must be eliminating the boundaries between our processing elements and artificially creating DSP chains. Our solution to this problem has been to encapsulate the processing elements in a Vivado `dont_touch` directive. When this directive is included, the Vivado synthesis tool keeps this entity separate and prevents it from being absorbed into other logic during synthesis optimisation steps.

When this solution was applied, we ran into a new problem. After an unusually long compilation time, the implementation failed with the following error message: "Routing results verification failed due to partially-conflicted nets". While checking the area reports for the placed but not yet routed system, we noticed that the area usage was not high enough to trigger such an error. We concluded that setting our target frequency at 450MHz might be too stringent, making the implementation tool spend all its time trying to reach an unreachable target instead of building a working system. Reducing this target to a more realistic frequency resulted in the successful compilation of a system with `MAT_SIZE` = 32. We also noticed that decreasing the target frequency to values which are closer to realistic values results in less variability in achieved kernel frequency.

## 4.6. Equivalent Kernel HLS Implementation

Our HLS implementation is designed to fit perfectly in place of the RTL implementation. It will feature the same interfaces and works by receiving the lines of data, putting them into a Private Local Memory (PLM) then working on the data when it is structured there. Both the GMMM section and the BMMM section are structured into a triple-for-loop implementation of the matrix multiplication.

### 4.6.1. Generic Matrix-Matrix Multiplication

For our GMMM portion of the code, the resulting implementation is very simple and does not need to be further discussed.

Listing 43: HLS version of the unified matrix multiplication kernel, focus on GMMM.

```
53            for(unsigned int i=0;i<MAT_SIZE;i++){
54                for(unsigned int j=0;j<MAT_SIZE;j++){
55                    index =  i*MAT_SIZE + j;
56                    c_data_array[index]=0; // start the accumulator at 0
```

```
57                  for(unsigned int k=0;k<MAT_SIZE;k++){
58                      index_a = i*MAT_SIZE + k;
59                      index_b = k*MAT_SIZE + j;
60                      c_data_array[index] += a_data_array[index_a]*b_data_array[index_b];
61                  }
62              }
63          }
```

### 4.6.2. Band Matrix-Matrix Multiplication

For our BMMM portion of the code, the first implementation features a few particularities. In Listing 44, note the checks implemented on line 99. These serve to make sure we are not computing any elements outside of the band of the output matrix. The checks on line 102 ensure that only the elements within the input band matrices are considered. When all the preliminary checks have been executed, we can begin the actual computation. This step happens in 3 parts. First, we must calculate where the input data can be found. Then we use these indices to access the data and calculate the contribution to the final result. Finally, we calculate the corresponding output index in rectangular coordinates and write out the data to the output matrix.

Listing 44: HLS version of the unified matrix multiplication kernel, focus on BMMM

```
90          unsigned int index_a;
91          unsigned int index_b;
92          unsigned int index_c;
93
94          const int q = band_type + 1;
95          const int p = in_width - q + 1;
96
97          for(int c_i=0;c_i<size;c_i++){
98              for(int c_j=0;c_j<size;c_j++){
99                  if((c_i-c_j) < in_width && (c_i-c_j) > -in_width){
100                     index_c = get_c_index(c_i,c_j,in_width);
101                     for(int c_k=0;c_k<size;c_k++){
102                         if((c_i-c_k) < q && (c_i-c_k) > -p && (c_k-c_j) < p && (c_k-c_j)> -q){
103                             index_a = get_a_index(c_i,c_k,in_width,p);
104                             index_b = get_a_index(c_j,c_k,in_width,p);
105                             c_data_array[index_c]+=a_data_array[index_a]*b_data_array[index_b];
106                         }
107                     }
108                 }
109             }
110         }
```

### 4.6.3. HLS Improvement: GMMM Streamability

Until this point our HLS design was not able to stream operations, meaning every time the kernel was launched, it could only execute one GMMM. Following the logic put in place by the RTL design, we reused the `band_type` parameter to convey how many operations must be executed. The only necessary change to the kernel was to encompass the entire operation in a for-loop. For each loop, data is pulled into a PLM large enough to fit the input matrix. When an operation is finished, the output data is dispatched and the next operation is also loaded.

Listing 45: Source code of the improved GMMM portion of the HLS kernel. This code snippet is a portion of Listing 64

```
156             const unsigned int in_array_size = MAT_SIZE*MAT_SIZE;
157             const unsigned int out_array_size = MAT_SIZE*MAT_SIZE;
158 
159             data_t a_data_array[in_array_size];
160             data_t b_data_array[in_array_size];
161             data_t c_data_array[out_array_size];
162 
163             for(unsigned int opcount = 0; opcount < band_type; opcount++){
164             #pragma HLS loop_tripcount max=1000
165                 for(unsigned int i=0; i < MAT_SIZE; i++){ //loop through the lines
166                     gen_in_t a_temp;
167                     gen_in_t b_temp;
168                     A_gen_stream >> a_temp;
169                     B_gen_stream >> b_temp;
170                     for(unsigned int j=0; j<MAT_SIZE; j++){//loop through each line and cut it up and
↪  fill the array
171                         index_a = i*MAT_SIZE + j;
172                         index_b = j*MAT_SIZE + i; // in order to match the rtl design
173                         a_data_array[index_a] = a_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
174                         b_data_array[index_b] = b_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
175                     }
176                 }
177 
178                 for(unsigned int i=0;i<MAT_SIZE;i++){
179                     for(unsigned int j=0;j<MAT_SIZE;j++){
180                         index = i*MAT_SIZE + j;
181                         c_data_array[index]=0; // start the accumulator at 0
182                         for(unsigned int k=0;k<MAT_SIZE;k++){
183                             index_a = i*MAT_SIZE + k;
184                             index_b = k*MAT_SIZE + j;
185                             c_data_array[index] += a_data_array[index_a]*b_data_array[index_b];
186                         }
187                     }
188                 }
189                 //put the data back into streams
190                 for(unsigned int i=0; i<MAT_SIZE; i++){ //loop through the lines
191                     gen_out_t c_temp;
192                     for(unsigned int j=0; j<MAT_SIZE; j++){//loop through each line and cut it up and
↪  fill the array
```

```
193                     index = i*MAT_SIZE + j;
194                     c_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH) = c_data_array[index];
195                 }
196                 C_gen_stream << c_temp;
197             }
198         }
```

### 4.6.4. HLS Improvement: GMMM Optimisation

Adding the following pragmas to our PLM which deals with the GMMM operating storage allows the innermost loop to be effectively unrolled and executed much faster. These pragmas partition the memory into multiple memory blocks so that they can be simultaneously accessed. The A array uses a cyclic pattern for memory partitioning because it is expected to be accessed row by row, whereas the B array uses a block pattern because it is expected to be accessed column by column. This change overall has allowed us to obtain an II of 1 for the innermost loop of the GMMM operation.

Listing 46: Source code of the pragmas employed to speed up the GMMM portion of the HLS kernel. This code snippet is a portion of Listing 64

```
162         data_t a_data_array[in_array_size];
163         DO_PRAGMA(HLS array_partition variable=a_data_array type=cyclic factor=MAT_SIZE/2 dim=1)
164         #pragma HLS bind_storage variable=a_data_array type=RAM_2P
165         data_t b_data_array[in_array_size];
166         DO_PRAGMA(HLS array_partition variable=b_data_array type=block factor=MAT_SIZE/2 dim=1)
167         #pragma HLS bind_storage variable=b_data_array type=RAM_2P
```

### 4.6.5. HLS Improvement: BMMM Optimisation

Until now, although functionally correct, our HLS code is very inefficient. At its core it consists of a triple-for-loop that goes through the entire output matrix. This means that for a matrix size of 1000, this operation is designed to go through every $1000^3$ indices, despite most of these indices having no contribution and being expected to result in a 0 in the output, by the very axioms of band matrices.

The solution we employed is much more efficient. Instead of looping through the entire output matrix, we loop through its rectangular representation. As we mentioned in Section 3.4.1, for large matrices, we expect this matrix to be very densely packed and thus only consider index pairs that are expected to be non-zero. This brings down the indices from

the first two for-loops from $N^2$ to $N(2w-1)$. For every one of these indices, we compute the `overlap`. This `overlap` dictates how many elements from the input matrices are overlapping and might thus result in a non-zero contribution. The upper bound of the `overlap` variable is $w$. We loop only on the overlapping data. We then figure out which indices the overlapping data correspond to in the inputs matrices and convert them to rectangular indices. We finally use these rectangular indices to access the input data and compute the contributions to the output matrix. We thus bring the amount of indices considered from $N^3$ to $N(2w-1)w$. When $w$ is much smaller than $N$, we can expect a large increase in speed. These improvements can be seen in Listing 47.

### 4.6.6. HLS improvement: BMMM Streamability

Until now, if we wanted to compute the BMMM, we store the entire rectangular representation of the input and output matrices. This is of course overkill. Exploiting a pattern in locality of data requirements, we noticed that we could fit the entire working set of input data into a $w$-wide, $2w$-long window. Whenever we finish computing a line of output data in rectangular indices, we can afford to discard a line of input data. Since we have a moving window of active operating data, we have decided to implement a circular buffer structure. It works by replacing the obsolete lines of data with the next lines of data. By wrapping around the access indices of the input matrices, we were able to keep the data accesses and actual data in the active memory synchronized.

Listing 47: Source code of the improved BMMM portion of the HLS kernel. This code snippet is a portion of Listing 64

```
204         const int in_width = 2*MAT_SIZE-1;
205         const int out_width = 2*in_width-1;
206 
207         const unsigned int in_array_size = 2*in_width*in_width;
208         const unsigned int out_array_size = out_width;
209 
210         data_t a_data_array[in_array_size];//define arrays with sufficient size
211         data_t b_data_array[in_array_size];
212 
213         int pattern[in_array_size];
214 
215         data_t c_data_array[out_array_size];
216 
217 
218 
219         //simple generic printer
220 
221 
222         for(unsigned int i=0; i<2*in_width && i<size; i++){ //fill the circular buffer initially
223         #pragma HLS loop_tripcount max=1000
```

```
224             band_in_t a_temp;
225             band_in_t b_temp;
226             A_band_stream >> a_temp;
227             B_band_stream >> b_temp;
228             for(unsigned int j=0; j<in_width; j++){//loop through each line and cut it up and fill the
    ↪  array
229                 index = i*in_width + j;
230                 a_data_array[index] = a_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
231                 b_data_array[index] = b_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
232             }
233         }
234 
235             // now that the data is into an array we can easily make the computations with the standard
    ↪  3-loop technique
236             int index_a;
237             int index_b;
238             int index_c;
239 
240             int c_i;
241             int c_j;
242             int c_k;
243 
244             int overlap;
245             int dif;
246             int abs_dif;
247 
248             const int q = band_type + 1;
249             const int p = in_width - q + 1;
250             const int w = in_width;
251 
252             int circ_buffer_offset = 0;
253 
254         #ifndef __SYNTHESIS__
255             std::cout << "q= "<< q << " p= " << p << " w= " << in_width << std::endl;
256         #endif
257 
258             for(unsigned int c_i_rect = 0 ; c_i_rect < size ; c_i_rect++){
259             #pragma HLS loop_tripcount max=1000
260                 if(c_i_rect > in_width && c_i_rect <= size-in_width){
261                     band_in_t a_temp;
262                     band_in_t b_temp;
263 
264                     A_band_stream >> a_temp; // these lines are deadlocking in sw_emu
265                     B_band_stream >> b_temp;
266                     for(unsigned int j=0; j<in_width; j++){
267                         index = circ_buffer_offset*in_width + j;
268                         a_data_array[index] = a_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
269                         b_data_array[index] = b_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
270                     }
271                     circ_buffer_offset = (circ_buffer_offset==2*in_width-1) ? 0 : circ_buffer_offset + 1;
272                 }
273                 for(unsigned int c_j_rect = 0 ; c_j_rect < out_width ; c_j_rect++){
274                     index_c = c_j_rect;
275                     c_data_array[index_c] = 0; //start off every output memory at 0
276                     if(is_in(c_i_rect,c_j_rect,size,in_width,out_width)){
277                         c_i = get_c_i_index(c_i_rect , c_j_rect , w);
278                         c_j = get_c_j_index(c_i_rect , c_j_rect , w);
```

```
279
280                         dif = c_i-c_j;
281                         abs_dif = (dif < 0)? -dif : dif;
282                         overlap = w - abs_dif;
283
284                         for(unsigned int iter=0 ; iter < overlap ; iter++){
285                         DO_PRAGMA(HLS loop_tripcount max=in_width)
286                             c_k = (c_i>c_j) ? c_i + iter - (w-p) : c_j + iter - (w-p);
287                             if(c_k>=0 && c_k<size){
288                                 index_a = get_a_index(c_i, c_k , in_width,p);
289                                 index_b = get_a_index(c_j, c_k , in_width,p);
290
291                                 c_data_array[index_c] += a_data_array[index_a % in_array_size] *
    ↪  b_data_array[index_b % in_array_size];
292                             }
293                         }
294                     }
295                 }
296                 band_out_t c_temp;
297                 for(unsigned int j=0; j<out_width; j++){
298                     c_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH) = c_data_array[j];
299                 }
300                 C_band_stream << c_temp;
301             }
302         }
```

## 4.7. Summary

In this chapter, we have not only gone through the suite of tools and workflows we needed to implement a full system, but we also showed the modules we wrote in Verilog and the functions we wrote in C++ to enable that. We then synthesized the entire system using Vitis. We have also presented our alternative kernel, fully written in C++. Lastly, we discussed the shortcomings of our kernels and how we implemented their improvements.

In the next chapter, we will present and discuss the different results we obtained from implementing and running our kernels.

# 5 | Experiments and Results

## 5.1. Experiment Setup and Data

When our kernels are built, we can make many measurements. We begin by making sure that the kernels we build are functionally correct. For every kernel we have built, its correctness has been rigorously checked. The test inputs have been generated using Matlab programs and their corresponding outputs have been cross-checked.

The next data we can examine is the area utilisation. This information can be found in a report file called `impl_1_full_util_routed.rpt`. Our data will be presented as a percentage of total available resources of the Alveo U280 Ultrascale+ FPGA.

The next data we will examine is the achieved clock frequency of each kernel. The information can be found in a log file in the project called `vivado.log` and will always be presented in MHz.

The following data we can extract from our experiments are run-time data. These data can be extracted only after running the operations.

For most of our kernels, we will measure the time it takes to complete a given operation. This will allow us to compare corresponding kernels together. In order to accomplish this, we must set up some lines in the host code. Our measurements will use the `C++` standard libraries' timing functions. We must instantiate some high resolution time variables and encompass the main data migration and kernel launching functions into a for-loop which we can use to repeat multiple times the same calculation. We insist on pointing out that "running multiple times the same operation" and "streaming multiple operations" is not the same thing. In the first case, we launch the kernel multiple times, in the second case we launch the kernel only once and the operations are streamed through the kernel. For some of our experiments, multiple operations were streamed through our kernel (especially when talking about the GMMM kernel) **and** multiple runs were executed for a single measurement (to filter out variability noise). The additional code necessary for timing can be found in Listing 48.

When executing multiple runs, we have taken the final result and divided it by the amount of times we launched the kernel to get an accurate average time per kernel execution.

Listing 48: Host code snippet from Listing 63 showcasing timing functionality.

```
203     unsigned int loopcount = 1;
204 
205     std::chrono::duration<double> full_time(0);
206     std::chrono::duration<double> kernel_time(0);
207 
208     std::cout << "OK" << std::endl << "Starting " << loopcount << " Operations" << std::flush;
209 
210 
211     auto kernel_start = std::chrono::high_resolution_clock::now();
212 
213     for(unsigned int i=0;i<loopcount;i++){
214         //std::cout << "OK" << std::endl << "Migrating data to the kernel space..." << std::flush;
215 
216         // Data will be migrated to kernel space
217         OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_A},0/* 0 means from host*/));
218         OCL_CHECK(err, q.finish());
219         OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_B},0/* 0 means from host*/));
220         OCL_CHECK(err, q.finish());
221         //std::cout << "OK" << std::endl << "Launching Kernel..." << std::endl << std::flush;
222         //Launch the Kernel
223         OCL_CHECK(err, err = q.enqueueTask(krnl_matrix_mult));
224         OCL_CHECK(err, q.finish());
225         //std::cout << "EXECUTION FINISHED" << std::endl << "Migrating datafrom the kernel space..."<<
    ↪ std::flush;
226         // The result of the previous kernel execution will need to be retrieved in
227         // order to view the results. This call will transfer the data from FPGA to
228         // source_results vector
229         OCL_CHECK(err, q.enqueueMigrateMemObjects({buffer_C},CL_MIGRATE_MEM_OBJECT_HOST));
230         OCL_CHECK(err, q.finish());
231         //std::cout << "OK" << std::endl << std::flush;
232         //std::cout << std::endl;
233     }
234 
235     std::cout << "EXECUTION FINISHED" << std::endl << std::flush;
236 
237     auto kernel_end = std::chrono::high_resolution_clock::now();
238 
239     full_time = std::chrono::duration<double>(kernel_end - kernel_start);
240 
241     kernel_time = full_time / (double) loopcount;
242 
243     std::cout << "time per kernel execution:" << kernel_time.count() << "s" << std::endl << std::flush;
```

## 5.2. Baseline Designs

### 5.2.1. Baseline RTL Kernel

Our baseline for the RTL-infused system yields the area results in Table 5.1. We begin by noting that the RAM requirements seem not to scale with size or data-width of the array. The next interesting note is that the size 16 systems with 8 bit and 16 bit data width appear to use the same DSP blocks, since their utilisation is identical.

Table 5.1: Area results for the implementation of the baseline RTL kernel

| MAT_SIZE | DATA_WIDTH | LUT (%) | REG (%) | DSP (%) | RAM (%) |
|---|---|---|---|---|---|
| 4 | 8 | 10.41 | 7.53 | 0.73 | 11.41 |
| 8 | 8 | 10.88 | 7.79 | 2.68 | 11.41 |
| 16 | 8 | 11.91 | 8.69 | 10.84 | 11.41 |
| 16 | 16 | 12.96 | 9.96 | 10.84 | 11.41 |
| 16 | 32 | 18.07 | 13.53 | 32.11 | 11.41 |

In Table 5.2, we begin by pointing out something unusual. We expected clock frequency to scale with data width, because larger arithmetic operations take longer. However, we did not expect the clock frequency to also scale considerably with array size. Next, if we look at the system with `MAT_SIZE` = 16 and `DATA_WIDTH` = 8, we see that on average, a 16x16 matrix multiplication takes 821$\mu s$. Theoretically, this operation should take 47 cycles. At 182MHz, this equates to 0.26$\mu s$. We presume that the reason we are measuring such a comparatively long time is that launching the kernel is taking up most of this time, since a very small fraction of this time is spent actually computing an answer. This measurement reinforces the streamability improvement discussed in 4.5.4.

Table 5.2: Clock frequency and timing results for the baseline RTL kernel. The GMMM test consisted of 1000 launches of the kernel each achieving one GMMM. The BMMM test consisted of launching the kernel once on a band matrix with lateral size 1000 and band size $w = 2*$`MAT_SIZE`$-1$.

| MAT_SIZE | DATA_WIDTH | Clock Frequency (MHz) | Time per GMMM ($\mu s$) | Time per BMMM ($ms$) |
|---|---|---|---|---|
| 4 | 8 | 231 | 762 | 1.78 |
| 8 | 8 | 229 | 832 | 1.76 |
| 16 | 8 | 182 | 821 | 2.04 |
| 16 | 16 | 181 | 847 | 1.94 |
| 16 | 32 | 95 | 796 | 2.84 |

### 5.2.2. Baseline HLS Kernel

For the area results of the baseline HLS kernel, we can see that the RAM usage increases greatly with array size and data width. This is to be expected and reinforces our streambility improvement considered in 4.6.6. Indeed, this version of the HLS kernel stores the input and output matrices completely into a PLM. Doing so results in the memory usage increasing considerably with both `MAT_SIZE` and `DATA_WIDTH`.

Table 5.3: Area results for the implementation of the baseline HLS kernel

| MAT_SIZE | DATA_WIDTH | LUT (%) | REG (%) | DSP (%) | RAM (%) |
|---|---|---|---|---|---|
| 4 | 8 | 10.40 | 7.53 | 0.24 | 14.04 |
| 8 | 8 | 10.81 | 7.64 | 0.29 | 17.40 |
| 16 | 8 | 11.53 | 7.85 | 0.38 | 24.16 |
| 16 | 16 | 12.14 | 8.23 | 0.38 | 37.23 |
| 16 | 32 | 13.22 | 8.95 | 0.70 | 62.70 |

For the timing and frequency results, we begin by noting that the clock frequencies remain comfortably above 250MHz for the `DATA_WIDTH`= 8 implementations. We can see that no real pattern can be extracted from the three `DATA_WIDTH`= 8 kernels. Their frequency is quite variable since these tests were targeting 450MHz and achieving only up to 300MHz. The reason for this is briefly discussed in 4.5.5.

The next observation is that increasing data size decreases the clock frequency, this is expected because using multipliers and adders with a larger bit-width increases the delay of the critical path of the circuit.

Similarly to the RTL kernel, the results of the GMMM time are somewhat constant and unusually large. Leading us to suspect that they must also be dominated by the launching of the kernel.

Table 5.4: Clock frequency and timing results for the baseline HLS kernel tests. The GMMM test consisted of 1000 launches of the kernel each achieving one GMMM. The BMMM test consisted of launching the kernel once on a band matrix with lateral size 1000 and band size $w = 2*$`MAT_SIZE`$-1$.

| MAT_SIZE | DATA_WIDTH | Clock Frequency (MHz) | Time per GMMM ($\mu s$) | Time per BMMM ($ms$) |
|---|---|---|---|---|
| 4 | 8 | 290 | 676 | 247 |
| 8 | 8 | 263 | 830 | 568 |
| 16 | 8 | 293 | 756 | 1051 |
| 16 | 16 | 262 | 674 | 1171 |
| 16 | 32 | 174 | 778 | 1439 |

### 5.2.3. Comparison of Baseline Kernels

In our area comparison, we can notice that our RTL kernels uses considerably more DSP's than our HLS kernels. RAM usage is much smaller in the RTL kernels since they do not feature any internal PLM's. The amount of LUT's and REG's is larger for the RTL compared to the HLS.

Table 5.5: Area Ratio (RTL/HLS) for the baseline kernels.

| MAT_SIZE | DATA_WIDTH | LUT Ratio | REG Ratio | DSP Ratio | RAM Ratio |
|---|---|---|---|---|---|
| 4 | 8 | 1.00 | 1.00 | 3.04 | 0.81 |
| 8 | 8 | 1.01 | 1.02 | 9.24 | 0.66 |
| 16 | 8 | 1.03 | 1.11 | 28.53 | 0.47 |
| 16 | 16 | 1.07 | 1.21 | 28.53 | 0.31 |
| 16 | 32 | 1.37 | 1.51 | 45.87 | 0.18 |

In our timing comparison, we begin by noting that our RTL kernels are consistently clocked at lower frequencies than our HLS kernels. We also note that the time ratio per GMMM is close to 1. This result would make sense if our assumption that launching the kernel is currently dominating the operation time, is right. Next, we can see that the BMMM is substantially slower in the HLS design than the RTL design. This observation corroborates the optimisation improvements considered in 4.6.5.

Table 5.6: Clock frequency and timing ratio (RTL/HLS) for the unimproved kernels

| MAT_SIZE | DATA_WIDTH | Clock Frequency Ratio | Time per GMMM Ratio | Time per BMMM Ratio |
|---|---|---|---|---|
| 4 | 8 | 0.797 | 1.127 | 0.007 |
| 8 | 8 | 0.871 | 1.002 | 0.003 |
| 16 | 8 | 0.621 | 1.086 | 0.002 |
| 16 | 16 | 0.691 | 1.257 | 0.002 |
| 16 | 32 | 0.546 | 1.023 | 0.002 |

## 5.3. Second Iteration Designs: Comparison

For this experiment we implemented the improvement to the RTL considered in 4.5.4. This optimisation enables the Host to stream as many[1] GMMM operations as it wants with a single launch of the kernel. To the HLS, we have implemented the improvements considered in 4.6.6 and 4.6.5. Similarly to the RTL, these optimisations allow the HLS to stream both GMMM and BMMM operations using a fixed amount of local memory. These optimizations also allow the BMMM implementation to save large amounts of

[1]Limited only by the size of the HBM memory

cycles, in turn making it considerably faster. This test is the first comparison of kernels which is considered completely fair since both kernels feature the same functionality and capabilities.

We begin by observing that all the area metrics are similar, except for the DPS's, of which the RTL employs considerably more. The execution times of both RTL and HLS are very similar. The achieved clock frequencies for the RTL are almost half those for the HLS.

Table 5.7: Comparison of area and performance metrics for optimised kernels with `MAT_SIZE` = 16 and `DATA_WIDTH` = 8. The time for the GMMM is reported as the total time divided by the 1000 streamed operations in order to obtain an average time per operation. The time for BMMM is calculated for one operation of matrix size 1000.

| `MAT_SIZE`=16 `DATA_WIDTH`=8 | LUT (%) | REG (%) | DSP (%) | RAM (%) | Clock Frequency (MHz) | Time GMMM ($\mu s$) | Time BMMM ($ms$) |
|---|---|---|---|---|---|---|---|
| RTL | 11.92 | 8.70 | 10.96 | 11.41 | 148.6 | 10.43 | 15.20 |
| HLS | 11.75 | 7.81 | 0.39 | 12.20 | 287.3 | 9.26 | 15.48 |
| RTL/HLS ratio | 1.014 | 1.114 | 28.10 | 0.935 | 0.517 | 1.126 | 0.982 |

The results from this experiment are surprising. Firstly, from the architecture alone, we would expect our RTL kernel to be much faster than our HLS kernel in both GMMM and BMMM. As we can see by the comparable times, this is not the case. In tandem with this first consideration, we also realise that the operations take a similar time to complete, despite the large difference in clock frequency. In addition, each streamed GMMM operation on the RTL kernel should take approximately 16 cycles. At 148.6MHz, this equates to $108ns$ per generic operation. This is approximately 100 times faster than the measured time.

All of these considerations lead us to speculate that our RTL kernel is not operating at its full potential. We cannot make any conclusion on the potential of the HLS kernel with this data alone. In the following section, we will investigate this.

## 5.4. RTL GMMM Kernel Investigation

In order to attempt to check our hypothesis, we have implemented a counter within our RTL kernel's GMMM section which increments whenever the kernel is ready to receive more data but is obliged to wait instead. For this investigative section, we have focused on the GMMM. In Figure 5.1, we show the results of this experiment.

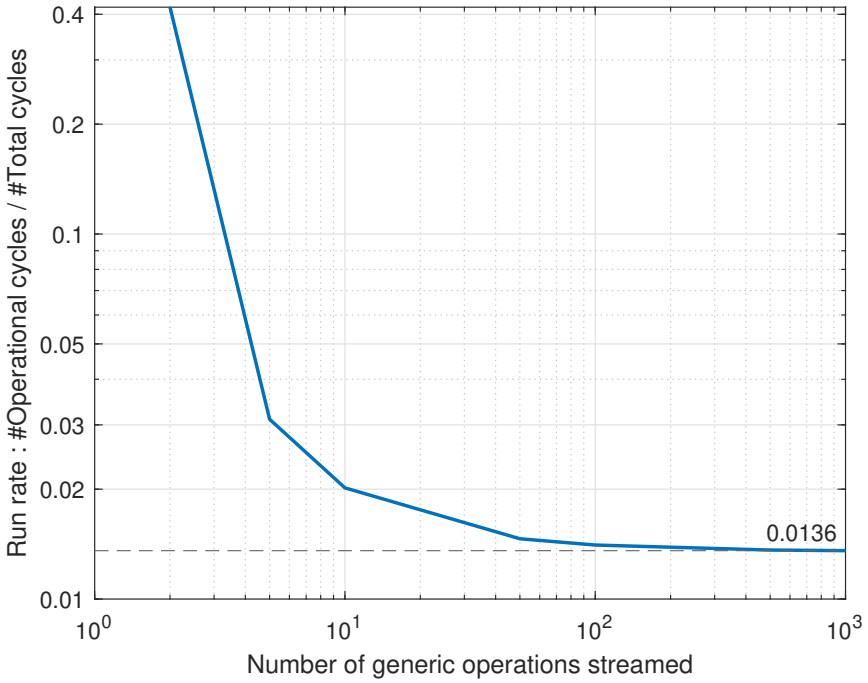


Figure 5.1: Running rate of a RTL kernel's GMMM operation with `MAT_SIZE` = 16 and `DATA_WIDTH` = 8. The running rate has been calculated by dividing the number of working (non-stalled) cycles by the total number of cycles used for the operation.

As the number of streamed operations increases, we can see that we approach a running rate of 0.0136. This signifies that, when 1000 GMMM operations are streamed through our current kernel, 98.64% of its operating time is spent waiting for data. This counter only starts counting the cycles when the first data arrives. This means that launching the kernel is **not** a parameter of this experiment. Judging from these results, we expect that this operation could run 73.5 times faster with appropriate memory delivery systems. This observation seems to corroborate the gap in performance observed in Section 5.3. After the fact, we have found that this is not necessarily the explanation because, as we will see in Section 5.5, 1000 streamed GMMM operations might not be enough to draw solid conclusions.

The next kernel we have implemented employs input and output PLM's into which the data will be loaded before the execution of the kernel. The data is then to be streamed at the rate of 1 line per cycle to and from the array using a simple HLS for-loop with a precompiler `pragma` which ensures the data is dispatched at the rate of 1 line per cycle. The code we employed for this purpose can be seen in Listing 49 and 50. This test will show us if the RTL has the capability of running uninterrupted, at its theoretical speed.

Listing 49: Reading function with a PLM for a MAT_SIZE = 16 and DATA_WIDTH = 8 kernel. Note the pragma at line 369, which tells the Vitis compiler to try to make the loop iteration take 1 cycle. This signals to the compiler to ensure that one line is dispatched to the FIFO every single cycle.

```
339         const unsigned int opcount_max = 1000; // can accomodate a maximum of 2 operations
340         gen_in_t A_PLM[MAT_SIZE*opcount_max];
341         gen_in_t B_PLM[MAT_SIZE*opcount_max];
342 
343         unsigned int index_PLM=0;
344 
345         for(unsigned int i=0;i<num_transfers;i++){
346 
347             chunk_A = in_A[i];
348             chunk_B = in_B[i];
349             chunk_accu_bit = 0;
350 
351             while(chunk_accu_bit < BUSWIDTH && bits_left != 0){ //while the chunk still has data to pull
352                 bits_to_be_pulled = min(BUSWIDTH-chunk_accu_bit,GEN_IN_SIZE-line_accu_bit); //determine
    ↪  if we are line-limited or chunk-limited
353                 A_line_accu.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit) =
    ↪  chunk_A.range(chunk_accu_bit + bits_to_be_pulled - 1,chunk_accu_bit);
354                 B_line_accu.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit) =
    ↪  chunk_B.range(chunk_accu_bit + bits_to_be_pulled - 1,chunk_accu_bit);
355                 chunk_accu_bit += bits_to_be_pulled;
356                 line_accu_bit += bits_to_be_pulled;
357                 bits_left -= bits_to_be_pulled;
358                 if(line_accu_bit == GEN_IN_SIZE){ //dispatch it
359                     A_PLM[index_PLM] = A_line_accu;
360                     B_PLM[index_PLM] = B_line_accu;
361                     index_PLM++;
362                     line_accu_bit=0;
363                 }
364             }
365         }
366         //now the data features in the PLM, we must dipatch it super fast. Do as few conditions and
    ↪  callculations as possible
367         const unsigned int max = MAT_SIZE*opcount_max;
368         for(unsigned int index_PLM=0; index_PLM < band_type*MAT_SIZE; index_PLM++){
369         #pragma HLS pipeline II=1
370             A_gen_line << A_PLM[index_PLM];
371             B_gen_line << B_PLM[index_PLM];
372         }
```

Listing 50: Writing function with a PLM for a `MAT_SIZE` = 16 and `DATA_WIDTH` = 8 kernel's GMMM portion. Note the pragma at line 434, which tells the Vitis compiler to try to make the loop iteration take 1 cycle. This signals to the compiler to ensure that one line is pulled from the FIFO every single cycle.

```
428         const unsigned int opcount_max=1000;
429         gen_out_t Temp_C;
430         const unsigned int max =  MAT_SIZE*opcount_max + 1;
431         gen_out_t C_PLM[max];
432 
433         for(unsigned int index_PLM = 0; index_PLM < band_type*MAT_SIZE + 1; index_PLM++){
434         #pragma HLS pipeline II=1
435             C_gen_line >> C_PLM[index_PLM];
436         }
437 
438         unsigned int in_matrix_size_in_bits = (band_type*size+1)*size*DATA_WIDTH; //one extra line
439         unsigned int bits_left = in_matrix_size_in_bits;
440         const unsigned int num_transfers = (in_matrix_size_in_bits % BUSWIDTH == 0) ?
441             in_matrix_size_in_bits /BUSWIDTH : in_matrix_size_in_bits /BUSWIDTH + 1;
442 
443         for(unsigned int i=0; i<band_type*size + 1; i++){ //for each line, work until the line is empty
444             Temp_C = C_PLM[i];
445             line_accu_bit = 0;
446             while(line_accu_bit < GEN_OUT_SIZE){ //while there is still data in the line to dump into
    ↪ chunk
447                 bits_to_be_pulled = min(BUSWIDTH-chunk_accu_bit,GEN_OUT_SIZE-line_accu_bit);//determine
    ↪ how many bits we can pull
448                 chunk_C.range(chunk_accu_bit + bits_to_be_pulled - 1, chunk_accu_bit)=
    ↪ Temp_C.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit);
449                 chunk_accu_bit += bits_to_be_pulled;
450                 line_accu_bit += bits_to_be_pulled;
451                 bits_left -= bits_to_be_pulled;
452                 if(chunk_accu_bit == BUSWIDTH || bits_left == 0){
453                     out_C[chunk_counter++] = chunk_C;
454                     chunk_accu_bit = 0;
455                 }
456             }
457         }
```

When implementing a PLM large enough to accommodate 1000 operations, while still keeping the stall-counting hardware, we have found that, when running up to 1000 streamed GMMM operations, the kernel **never stalls**. This experiment proves that our kernel is capable of running uninterrupted, if the memory management solutions were adequate. This means that a theoretical improvement of 73.5x is possible to the operating time.

## 5.5. RTL GMMM Analysis using Custom Memory Management Hardware

In this section we will focus once more on the GMMM operation. We have implemented a bespoke memory management system for a kernel of parameters `MAT_SIZE` = 16 and `DATA_WIDTH` = 8. By using the same pragmas as in Listing 49 and 50, we were able to ensure a 1 line per cycle rate of data delivery. This is facilitated because this specific kernel needs 128 bits per cycle per input and produces 128 bits per cycle for its output. Since we have configured our HBM channels to be 256 bits wide, each HBM channel will contain exactly 2 lines of data. The HLS code is thus very simple and the resulting hardware achieves our data rate goals. We have presented this code in Listings 51 and 52.

Listing 51: Custom reading function for a `MAT_SIZE` = 16 and `DATA_WIDTH` = 8 kernel's GMMM portion. Note the pragma at line 338, which tells the Vitis compiler to try to make the loop iteration take 2 cycles. This value has been chosen because each HBM chunk contains 2 lines. This signals to the compiler to ensure that one line is dispatched to the FIFO every single cycle.

```
338         for(unsigned int i=0;i<num_transfers;i++){//for each transfer, dispatch two lines
339         #pragma HLS pipeline II=2
340             chunk_A = in_A[i];
341             chunk_B = in_B[i];
342 
343             A_line_accu.range(127, 0) = chunk_A.range(127, 0);
344             A_gen_line << A_line_accu;
345             A_line_accu.range(127, 0) = chunk_A.range(255,128);
346             A_gen_line << A_line_accu;
347 
348             B_line_accu.range(127, 0) = chunk_B.range(127, 0);
349             B_gen_line << B_line_accu;
350             B_line_accu.range(127, 0) = chunk_B.range(255,128);
351             B_gen_line << B_line_accu;
352         }
```

Listing 52: Custom writing function for a `MAT_SIZE` = 16 and `DATA_WIDTH` = 8 kernel's GMMM portion. Note the pragma at line 421, which tells the Vitis compiler to try to make the loop iteration take 1 cycle. This signals to the compiler to ensure that one line is pulled from the FIFO every single cycle.

```
420         for(unsigned int i=0; i<band_type*size + 1; i++){ //for each line, work until the line is empty
421         #pragma HLS pipeline II=1
422             C_gen_line >> Temp_C;
423             if(i%2==0){
424                 chunk_C.range(127,0)= Temp_C.range(127,0);
425             }
426             else{
427                 chunk_C.range(255,128)= Temp_C.range(127,0);
428                 out_C[chunk_counter++] = chunk_C;
429             }
430         }
```

This time we will use no PLM at all, but will ensure that our kernels can be sustainably fed by checking the HLS compilation logs to see if the desired Iteration Intervals of our loops are achieved. This would mean that the HLS compiler reports that the data management hardware achieves a dispatching rate of 1 line per cycle. For this experiment, we will keep the stall-counting hardware in our processor and use it to make sure that the desired data delivery rate is actually achieved. The kernel we compiled for this experiment has been clocked at 216.4MHz.

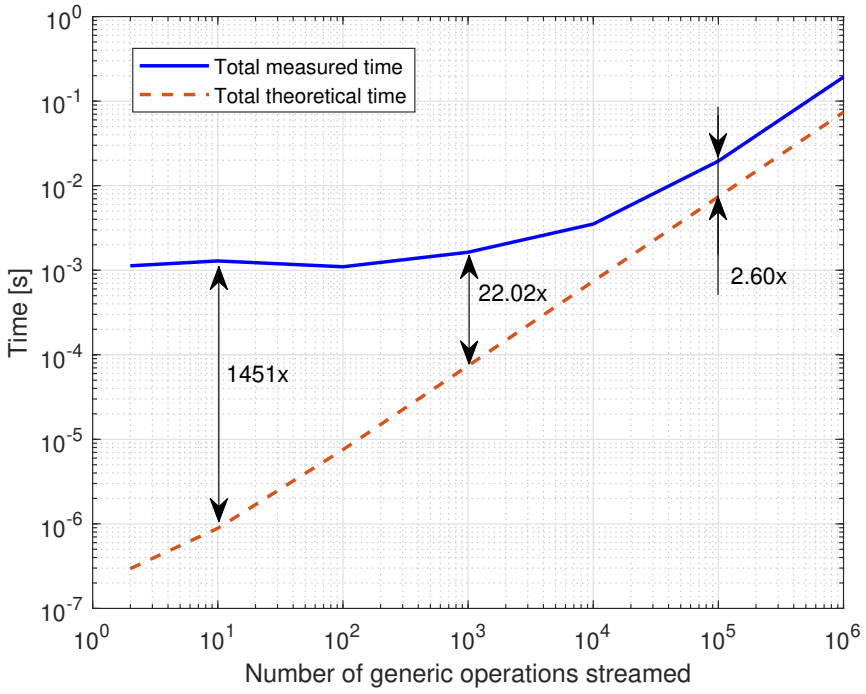


Figure 5.2: Time measurement for a RTL kernel with specifications `MAT_SIZE` = 16 and `DATA_WIDTH` = 8 using custom (non-parametric) memory management. The theoretical time is calculated using the achieved clock frequency and the amount of cycles needed to complete the operation.

The results of our experiment are in Figure 5.2,. The following observations can be made:

- Until 1000 operations, we clearly see that the operating time is dominated by the launch of the kernel. It does not matter how many operations are launched, our kernel will take approximately the same time. This seems to explain the apparent similarity in times we have found earlier in Table 5.7. We previously assumed that this slowness was due to the memory management inadequacy. Instead, we see here that even a kernel with *optimal* memory hardware is bottlenecked by the time needed to migrate the data and launch the kernel.
- At high amounts of streamed operations, launching the kernel becomes absolutely negligible. Thus, we expect that the time per GMMM operation should start to approach the theoretical time. This is however not what we observe. There is a constant performance gap of 2.6x between our measured time and the theoretical achievable time. We remind the reader that throughout this entire experiment, we have been controlling that no stalls are measured.

Noticing this performance gap, we have decided to investigate it further. Our guess is that the HBM interface might have been wrongly configured and could be the source of this slow-down. For the next test, we configured the HBM channel using the recommended settings presented in [23]. We then reran the same test and found the data in Figure 5.3.

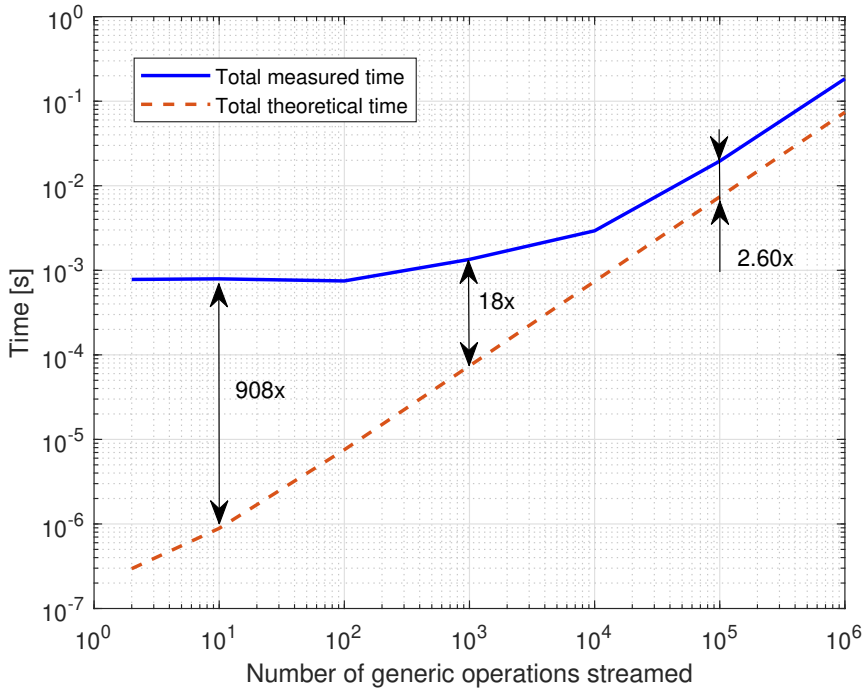


Figure 5.3: Time measurement for a RTL kernel with specifications `MAT_SIZE` = 16 and `DATA_WIDTH` = 8 using custom (non-parametric) memory management.

As we can quickly see, with a large number of operations, we approach the same 2.6x performance gap. We suspect that the mechanism which is slowing down our kernel is probably stalling the entire kernel, including the kernel controller, using the `ap_ce` signal. This would explain why we have not measured any stalls with our integrated stall counter but still observe a performance gap. Measuring this would be possible by including an Integrated Logic Analyzer (ILA) in our design. Unfortunately, as explained in Section 4.3.3, we have not been able to incorporate this hardware into our workflow.

Our next experiment uses the former RTL kernel and compares it with an optimized HLS kernel, featuring the optimizations presented in 4.6.4. They both will run at their maximal achievable speed since we are using the same data management hardware from the previous experiment.

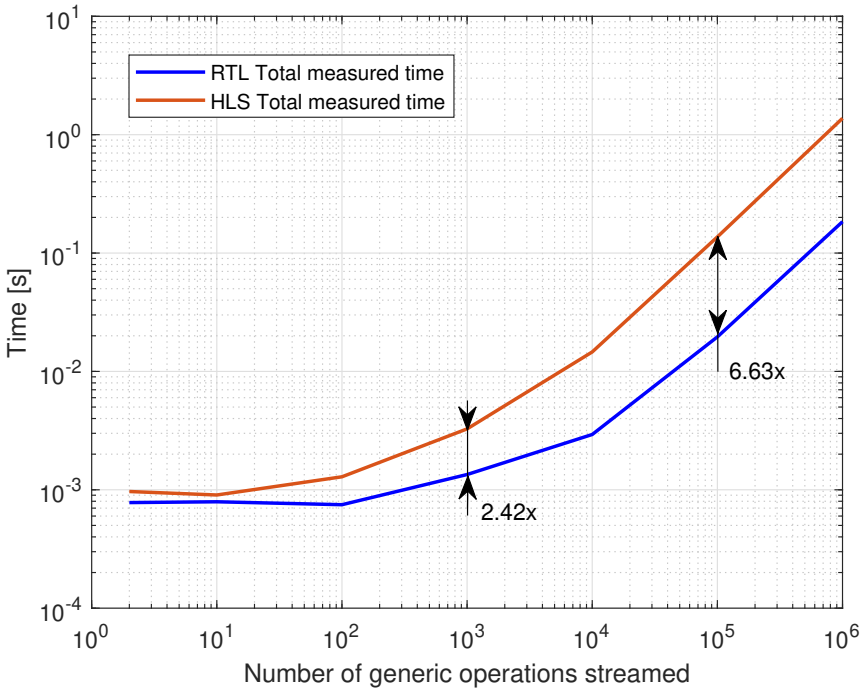


Figure 5.4: Time measurement for a RTL kernel and an HLS kernel, both with specifications `MAT_SIZE` = 16 and `DATA_WIDTH` = 8 using custom (non-parametric) memory management.

From this experiment, the following observations can be made:

- For a low amount of generic operations streamed, virtually no improvement can be obtained by using a systolic kernel in this system because the operation is largely dominated by the actual launch of the kernel. If the target application only runs occasional GMMM operations, implementing a systolic GMMM array compared to

a HLS version is a waste of area and will not improve the speed or latency of the calculation.

- At a high amount of generic operations streamed, our RTL GMMM kernel performs 6.63x faster compared to our HLS GMMM kernel. As we will see in Table 5.8, the RTL kernel employs nearly 30x the amount of DSP's that the HLS kernel employs. We remind that these kernels also include the hardware to enable BMMM, so independant conclusions on the performance/area metrics should not be made.

In Table 5.8, we see that the theoretical cycles per line needed for the HLS is 20 times greater than for the RTL. We would expect that this translates to an increase of performance of 20x, as opposed to 6.63x. It is possible to find the gap in performance. Indeed, if we compound the 6.63x real-world performance gap with the 2.6x gap between the real RTL and the theoretical RTL, we reach 17.24x in performance gap between the real HLS measurement and the theoretical RTL speed. In order to bridge the final gap to 20x, we need to take into account the difference in achieved clock speeds. If we multiply 17.24x with $\frac{250}{216.4}$, we do indeed obtain 19.95x. This consideration shows us that the HLS is running at its full potential but the RTL is being throttled because data requirements might be too stringent for the FPGA we are employing.

Table 5.8: Comparison of area for optimised kernels with `MAT_SIZE` = 16, `DATA_WIDTH` = 8 and custom memory management hardware. The compute cycles have been found from the HLS report for the HLS kernel and theoretically for the RTL kernel.

| `MAT_SIZE` =16 `DATA_WIDTH` =8 | LUT (%) | REG (%) | DSP (%) | RAM (%) | FREQ (MHz) | Compute Cycles |
|---|---|---|---|---|---|---|
| HLS | 11.00 | 7.62 | 0.33 | 11.46 | 250.0 | 20 |
| RTL | 11.58 | 8.53 | 10.83 | 13.05 | 216.4 | 1 |
| RTL/HLS ratio | 1.053 | 1.119 | 32.818 | 1.139 | 0.866 | 1/20 |

## 5.6. Final kernels: Expectations and Measurements

For our final experiments, we have compiled a family of kernels using our fully-parametric data management hardware.

### 5.6.1. Area Analysis

The results of the area utilisation can be found in Table 5.9.

Table 5.9: Area report for our final kernels. Every kernel has `DATA_WIDTH`= 8.

| | `MAT_SIZE` | FREQ (MHz) | LUT (%) | REG (%) | DSP (%) | RAM (%) |
|---|---|---|---|---|---|---|
| RTL | 4 | 242 | 10.32 | 7.45 | 0.75 | 11.41 |
| | 8 | 214 | 10.98 | 7.69 | 2.70 | 11.41 |
| | 16 | 203 | 12.17 | 8.55 | 10.86 | 11.76 |
| | 32 | 153 | 15.14 | 11.81 | 44.19 | 12.80 |
| HLS | 4 | 251 | 10.47 | 7.48 | 0.79 | 11.46 |
| | 8 | 226 | 10.97 | 7.54 | 0.28 | 11.46 |
| | 16 | 199 | 11.83 | 7.63 | 0.37 | 11.81 |
| | 32 | 174 | 12.55 | 7.78 | 0.54 | 13.37 |
| $\frac{\text{RTL}}{\text{HLS}}$ ratio | 4 | 0.97 | 0.99 | 1.00 | 0.95 | 1.00 |
| | 8 | 0.95 | 1.00 | 1.02 | 9.64 | 1.00 |
| | 16 | 1.02 | 1.03 | 1.12 | 29.35 | 1.00 |
| | 32 | 0.88 | 1.21 | 1.52 | 81.83 | 0.96 |

The usage of LUT's, REG's and RAM is very similar between our kernels. The achieved frequencies are also close. The next observations will be made after extracting the DSP usage into a graph.

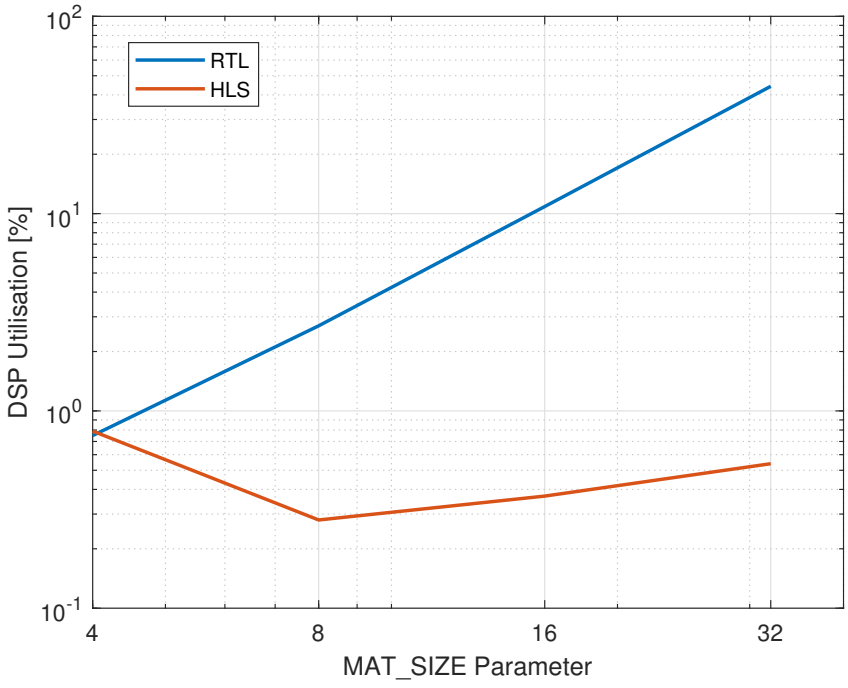


Figure 5.5: DSP usage of our RTL and HLS UMMM kernels.

The first anomaly lays in the smallest HLS kernel. It uses more DSP's than any other HLS block. The next observation is that the RTL's DSP usage quadruples with every doubling of the lateral size. This is expected since this is by design. In fact, function from

`MAT_SIZE` to number of PE's is $(2N-1)^2$, with N being `MAT_SIZE`. The gap in DSP usage between HLS and RTL seems to approximately triple for every doubling of `MAT_SIZE`.

### 5.6.2. Comment on HLS Latency Calculation and Reporting

Before beginning our speed analysis, we will discuss briefly how we use the HLS tool to extract information about loop latency.

For reasons which will become apparent in the following Section, our explanation begins with a showcase of the structure of our write function, which is visible in Listing 53. For our BMMM and GMMM we have used a for-loop within which a while-loop sits. In the outer for-loop we pull a line of data from the FIFO. In the inner while loop, we take bits from this line and append them to the current working HBM chunk of data. Ideally, we would like our outer loop to operate at the same speed as our RTL kernels, in order for them not to be limited. The problem is that this inner while-loop will cycle an unpredictable amount of times from the perspective of the HLS tool. The HLS tool will not report the number of cycles per iteration of the outer for-loop when it contains an unpredictable loop within. A way to force the HLS tool to predict outer loop latencies is to use the `#pragma loop_tripcount min=x max=x` (Seen in Listing 53 at lines 34 and 68) withing the inner loop. This will tell the HLS tool how many times the inner loop is expected to cycle. We must thus calculate how many times we expect the inner while-loop to loop. The results can be found in Table 5.10.

Listing 53: Source code of the write function. This code snippet is a portion of Listing 64

```
1  void write_data_optimised(
2          BUS_TYPE* out_C,
3          const unsigned int size,
4          const bool opmode,
5          const unsigned int band_type,
6          hls::stream<band_out_t >& C_band_line,
7          hls::stream<gen_out_t >&  C_gen_line)
8  {
9  #pragma HLS INLINE OFF
10 #pragma HLS dataflow
11
12     BUS_TYPE chunk_C;
13
14     unsigned int line_accu_bit;
15     unsigned int chunk_accu_bit=0;
16     unsigned int bits_to_be_pulled;
17     unsigned int chunk_counter = 0;
18
19     if (opmode==OPMODE_GEN){
20         gen_out_t Temp_C;
21
```

```
22          unsigned int in_matrix_size_in_bits = band_type*size*size*DATA_WIDTH;
23          unsigned int bits_left = in_matrix_size_in_bits;
24          const unsigned int num_transfers = (in_matrix_size_in_bits % BUSWIDTH == 0) ?
25              in_matrix_size_in_bits /BUSWIDTH : in_matrix_size_in_bits /BUSWIDTH + 1;
26
27          for(unsigned int i=0; i<band_type*size; i++){ // for each line, work until the line is empty
28          #pragma HLS loop_tripcount max=16000
29
30              C_gen_line >> Temp_C;
31              line_accu_bit = 0;
32
33              while(line_accu_bit < GEN_OUT_SIZE){ //while there is still data in the line to dump into
    ↪  chunk
34              #pragma HLS loop_tripcount min=1 max=1
35              #pragma HLS pipeline II=1
36                  bits_to_be_pulled = min(BUSWIDTH-chunk_accu_bit,GEN_OUT_SIZE-line_accu_bit);//determine
    ↪  how many bits we can pull
37                  chunk_C.range(chunk_accu_bit + bits_to_be_pulled - 1, chunk_accu_bit)=
    ↪  Temp_C.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit);
38                  chunk_accu_bit += bits_to_be_pulled;
39                  line_accu_bit += bits_to_be_pulled;
40                  bits_left -= bits_to_be_pulled;
41                  if(chunk_accu_bit == BUSWIDTH || bits_left == 0){
42                      out_C[chunk_counter++] = chunk_C;
43                      chunk_accu_bit = 0;
44                  }
45              }
46          }
47      }
48
49
50      else if(opmode==OPMODE_BAND){
51          band_out_t Temp_C;
52
53          const unsigned int width = 2*(2*MAT_SIZE-1)-1;
54          const unsigned int length = size;
55
56          unsigned int in_matrix_size_in_bits = width*length*DATA_WIDTH;
57          unsigned int bits_left = in_matrix_size_in_bits;
58          const unsigned int num_transfers = (in_matrix_size_in_bits % BUSWIDTH == 0) ?
59              in_matrix_size_in_bits /BUSWIDTH : in_matrix_size_in_bits /BUSWIDTH + 1;
60
61          for(unsigned int i=0; i<length; i++){ // for each line, work until the line is empty
62          #pragma HLS loop_tripcount max=1000
63          #pragma HLS pipeline II=1
64              C_band_line >> Temp_C;
65              line_accu_bit = 0;
66
67              while(line_accu_bit < BAND_OUT_SIZE){ //while there is still data in the line to dump into
    ↪  chunk
68              #pragma HLS loop_tripcount min=3 max=3
69              //#pragma HLS pipeline II=1
70                  bits_to_be_pulled = min(BUSWIDTH-chunk_accu_bit,BAND_OUT_SIZE-line_accu_bit);//determine
    ↪  how many bits we can pull
71                  chunk_C.range(chunk_accu_bit + bits_to_be_pulled - 1, chunk_accu_bit)=
    ↪  Temp_C.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit);
72                  chunk_accu_bit += bits_to_be_pulled;
```

```
73                  line_accu_bit += bits_to_be_pulled;
74                  bits_left -= bits_to_be_pulled;
75                  if(chunk_accu_bit == BUSWIDTH || bits_left == 0){//if the chunk is full or there's no
    ↪  more data to pull
76                      out_C[chunk_counter++] = chunk_C;
77                      chunk_accu_bit = 0;
78                  }
79              }
80          }
81      }
82  }
```

As we can see from Table 5.10, the expected amount of loops are fractional, and in the case of the GMMM they all feature between 0 and 1. In these cases, the tool will still preform the inner loop entirely, thus we always fill in the inner-loop pragmas using the upper bounds of the Chunks per Line metric, named Expected Loops in Table 5.10.

Table 5.10: Expected amount of loops to fill an entire chunk for both GMMM and BMMM and for different `MAT_SIZE`

| | GMMM | | | BMMM | | |
|---|---|---|---|---|---|---|
| `MAT_SIZE` | Bits / Line | Chunks / Line | Expected Loops | Bits / Line | Chunks / Line | Expected Loops |
| 4 | 32 | 0.125 | 1 | 104 | 0.406 | 1 |
| 8 | 64 | 0.250 | 1 | 232 | 0.906 | 1 |
| 16 | 128 | 0.500 | 1 | 488 | 1.906 | 2 |
| 32 | 256 | 1.000 | 1 | 1000 | 3.906 | 4 |

As a result, we can extract the expected amount of cycles for the outer loops from the HLS tool. These can be seen in Table 5.11. Note how despite having always the same amount of expected loops, for some reason the `MAT_SIZE`= 8 and = 16 feature one fewer reported cycle than the `MAT_SIZE`= 4 and = 32. These numbers come from the HLS report and we cannot explain the reason for the small disparity.

Table 5.11: Reported cycles for the write function for BMMM and GMMM.

| | GMMM HLS reports | | BMMM HLS reports | |
|---|---|---|---|---|
| `MAT_SIZE` | Expected Loops | Reported Cycles | Expected Loops | Reported Cycles |
| 4 | 1 | 76 | 1 | 76 |
| 8 | 1 | 75 | 1 | 76 |
| 16 | 1 | 75 | 2 | 150 |
| 32 | 1 | 76 | 4 | 298 |

### 5.6.3. Speed Analysis: GMMM

Our speed analysis begins with the HLS reports, which we have displayed in Table 5.12.

Table 5.12: Latency report of the functions within our kernels for the GMMM operation. All the data presented here is in number of cycles per line. The slowest kernels in the chain are highlighted. The fields marked with an asterisk (*) have gone through some post-processing and do not feature as is in the HLS report. The RTL compute field features the theoretical value. The HLS compute field is an average obtained by dividing the total cycles for a single GMMM operation by the amount of lines produced by that same operation (the corresponding `MAT_SIZE`).

| GMMM `MAT_SIZE` | RTL Read | RTL Compute* | RTL Write | HLS Read | HLS Compute* | HLS Write |
|---|---|---|---|---|---|---|
| 4 | 2 | 1 | **76** | 3 | 3.00 | **76** |
| 8 | 2 | 1 | **75** | 3 | 13.00 | **75** |
| 16 | 2 | 1 | **75** | 2 | 20.00 | **75** |
| 32 | 3 | 1 | **76** | 3 | 36.53 | **76** |

Before moving on, we would like to highligh that the amount of cycles per line for the RTL write function and the RTL compute function differ by a factor 75x. This is probably the reason why, when we measured the stalls for the GMMM portion in Section 5.4, we measured that we could increase the speed by a factor 73.5x.

### 5.6.4. GMMM Predictions and Measurements

In Table 5.12, we can see that both the HLS and RTL kernels are bottlenecked by the writing hardware. We begin by predicting that all the comparable kernels will take the same amount of time to complete, given we account for clock frequency disparities. The timing tests feature in Figure 5.6.

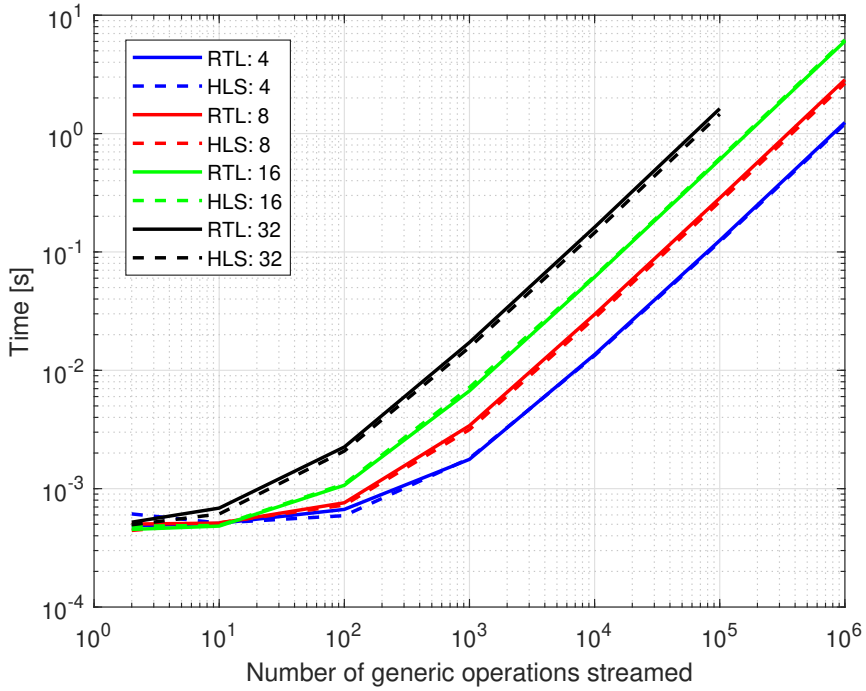


Figure 5.6: Timing results for different amounts of streamed GMMM operations and for every kernel. The kernels with `MAT_SIZE`= 4, 8 and 16 feature measurements up to 1M operations, whereas the `MAT_SIZE`= 32 kernel features data up to 100k operations. This is due to the HBM's 256MB limit for data, which is surpassed when running 1M 32x32, 8-bit calculations. Indeed, this operation would require 1.024GB of HBM memory. Small disparities can be seen between HLS and RTL results. These differences become even smaller when taking the different achieved clock speeds into account.

In Figure 5.6, we can see that our hypothesis is correct. Both the RTL kernel and the HLS kernel take the same time to complete the operation. We can thus say that with it does not matter how fast our kernels are, if they are bottlenecked by data delivery hardware it is not worth to implement expensive systolic kernels, rather than cheap HLS kernels. This statement is hardly surprising, since one of the golden rules of good computer architecture is to match the throughput of data delivery to the computing throughput to achieve maximum performance for a given area. What is interesting about these results is that with a fast (in the order of minutes) HLS compilation, we have accurately predicted the results of this experiment.

In our next experiment we will try to predict our total running time using only the data in our table, using the following formula:

$$\text{Running time} = \text{Number of lines produced} * \frac{\frac{\text{Cycles}}{\text{Line}}}{\text{Achieved Frequency}}$$

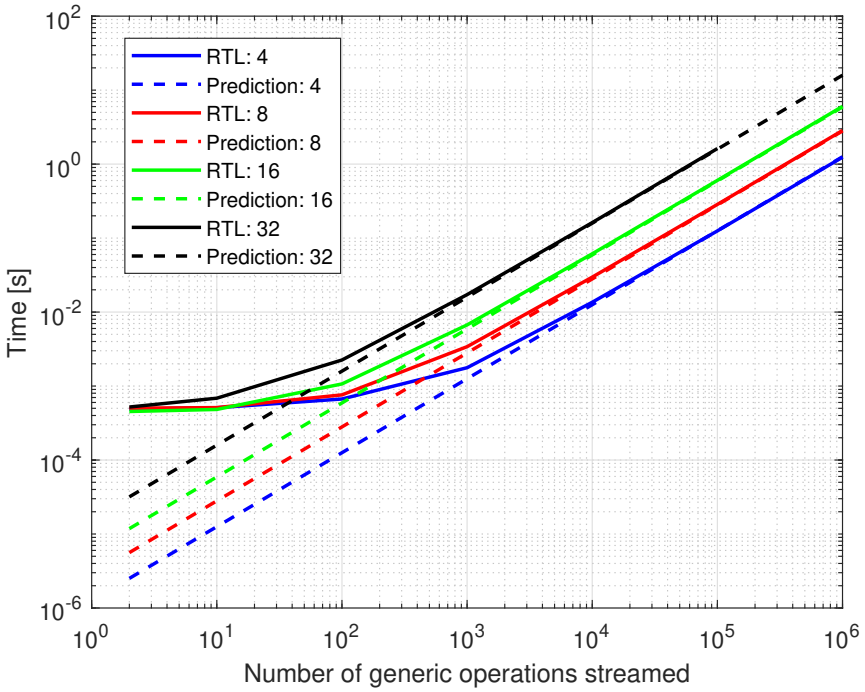


Figure 5.7: Timing results for different amounts of streamed operations for our RTL kernel and the associated theoretical equivalents. Since our HLS and RTL times are so similar, we will only compare with one of them.

In Figure 5.7, we can see that our predictions are very precise. The biggest error we made is on the `MAT_SIZE`= 32 kernel, for which we observed an error of 2%. In Table 5.13, we showcase the quality of our results.

Table 5.13: Comparison of the predicted and real times for 100k streamed GMMM operations

| | Times: 100k GMMM ops. | | |
|---|---|---|---|
| `MAT_SIZE` | Predicted (s) | Real (s) | Ratio (-) |
| 4 | 0.13 | 0.13 | 1.00 |
| 8 | 0.28 | 0.28 | 1.00 |
| 16 | 0.59 | 0.60 | 0.98 |
| 32 | 1.59 | 1.62 | 0.98 |

### 5.6.5. Speed Analysis: BMMM

Our analysis begins once again with the HLS reports in Table 5.14. This time it features the results for the BMMM operation.

Table 5.14: Latency report of the functions within our kernels for the BMMM operation. All the data presented here is in number of cycles per line. The slowest kernels in the chain are highlighted. The RTL compute field features the theoretical value.

| BMMM | RTL | | | HLS | | |
|---|---|---|---|---|---|---|
| MAT_SIZE | Read | Compute* | Write | Read | Compute | Write |
| 4 | 2 | 3 | **76** | 2 | **666** | 76 |
| 8 | 2 | 3 | **76** | 2 | **1591** | 76 |
| 16 | 3 | 3 | **150** | 3 | **3823** | 150 |
| 32 | 3 | 3 | **298** | 2 | **9824** | 298 |

### 5.6.6. BMMM Predictions and Measurements

In Table 5.14, we can see that the write function is still the bottleneck for our RTL implementation. However, the HLS is limited by the computation of the data itself. This is due to the increased complexity of our final BMMM kernel.

Our first prediction is that despite using non-optimal data management hardware, our RTL kernels still run faster than our HLS kernels. In Figure 5.8 we can see the results of the measured times for our kernels.

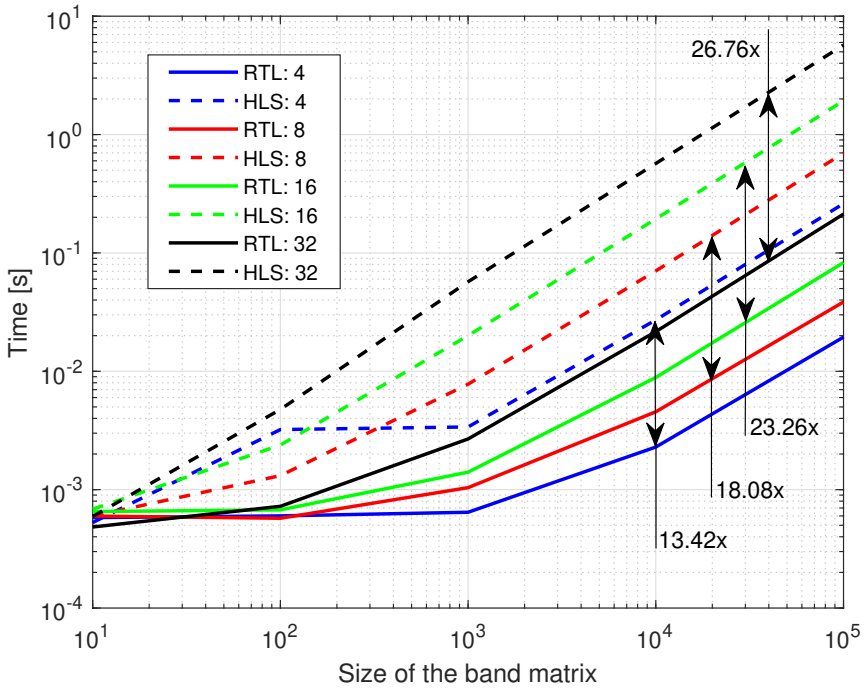


Figure 5.8: Time comparison between our RTL kernels and HLS kernels using fully-parametric memory management running BMMM operations.

In Figure 5.8, we can see first that at low size of the band matrix, all the operations take the same amount of time and the operation time is limited once again by the launching of the kernel. Very quickly after that, we can see that the HLS kernel computation time becomes dominant while the RTL kernel only starts becoming dominant at matrices sized 100 to 1000.

When computing very large matrices, we can indeed see that our RTL kernel runs up to 26.76x faster than our HLS kernel. From the HLS report, we also know that our RTL's kernel potential is limited by the write function. In order to double-check this, we have implemented a version of the `MAT_SIZE` = 16, `DATA_WIDTH` = 8 kernel with a built-in stall counter to see the potential speed increases. The measurements from this experiment feature in Figure 5.9.

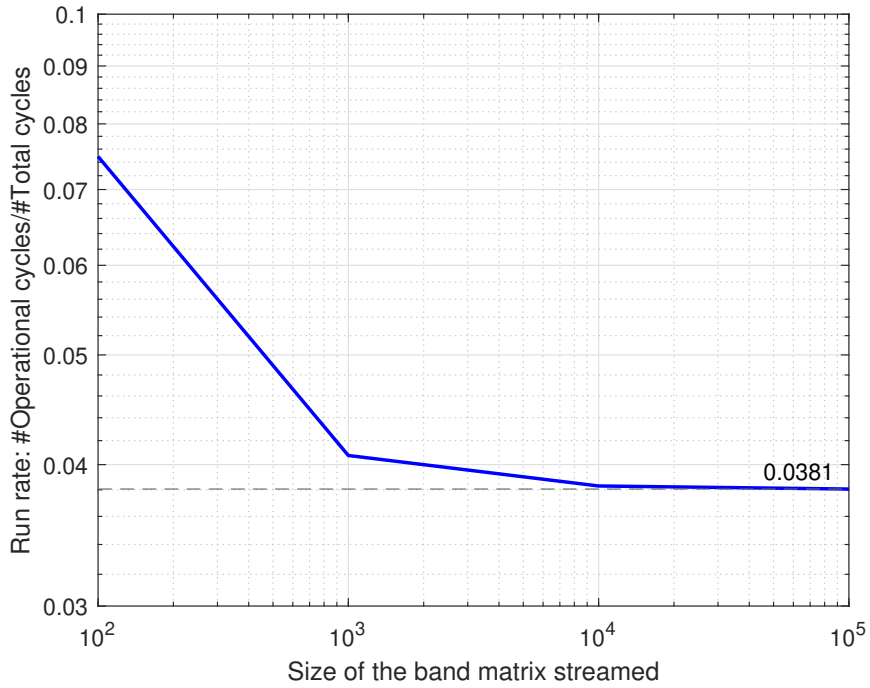


Figure 5.9: Running rate of a RTL kernel's BMMM operation with `MAT_SIZE` = 16 and `DATA_WIDTH` = 8. The running rate has been calculated by dividing the number of working (non-stalled) cycles by the total number of cycles used for the operation.

According to this experiment we can see that our kernel is only operational 3.81% of the time. This means that our theoretical increase in speed is in the order of 26.25x, given an optimal data management hardware. When this number is compounded with the measured speed difference between RTL and HLS for this specific kernel of 23.26x, this would compound to a theoretical performance gap of 610.34x in speed. This consideration is valid because we know that the HLS kernel is limited by its compute hardware and thus would not benefit from any speedup from better data management, contrarily to the RTL kernel.

In the following section we will use the data in our tables to try to predict the working times of our kernels, and then compare those to the actual working time. This time, since our RTL and HLS kernels are bottlenecked for different reasons, we will have to do the discussion twice.

We begin with the RTL kernel discussion.

Table 5.15: Comparison of the predicted and real times for our RTL preforming a BMMM of size 100k

| RTL | Times : Size 100k BMMM | | |
|---|---|---|---|
| | Predicted | Real | Ratio |
| MAT_SIZE | (s) | (s) | (-) |
| 4 | 0.031 | 0.019 | 0.68 |
| 8 | 0.035 | 0.039 | 0.84 |
| 16 | 0.074 | 0.083 | 0.85 |
| 32 | 0.195 | 0.213 | 0.90 |

We can see in Table 5.15, that our predictions achieve the correct order of magnitude. The smallest kernel features an error of 32%, which is far from being negligible. The other three kernels feature an error of less than 16%. In Figure 5.10 we visualize our predictions versus the achieved values.

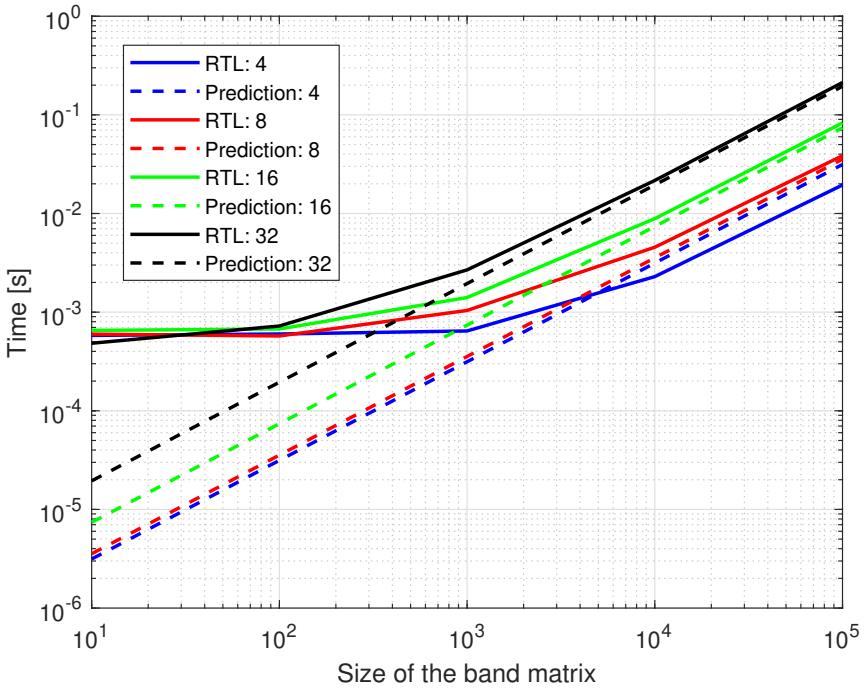


Figure 5.10: Timing results for different sizes of band matrices for our RTL kernel and the associated theoretical predictions.

We can see in Figure 5.10 that our predictions do approach the measurements for large BMMM operations, except for the smallest kernel, whose predictions are quite inaccurate. In our opinion this is because of the number of loops that need to be preformed is more unpredictable. According to Table 5.10, the amount of chunks per line is 0.41, making the inner loop run sometimes once, sometimes twice, but constantly leaving leftovers in

the accumulator. This is different than the GMMM case because the Chunks per line for that operation were simple fractions of 1, meaning the amount of inner loops is constant and predictable. This phenomenon is less apparent in the other three kernels since their Chunk per Line metric more closely approaches the Expected Loops.

Next, we will turn our attention to the HLS kernel's BMMM predictions.

Table 5.16: Comparison of the predicted and real times for our HLS preforming a BMMM of size 100k

| HLS | Times : Size 100k BMMM | | |
|---|---|---|---|
| | Predicted | Real | Ratio |
| MAT_SIZE | (s) | (s) | (-) |
| 4 | 0.266 | 0.261 | 1.02 |
| 8 | 0.703 | 0.697 | 1.01 |
| 16 | 1.917 | 1.929 | 0.99 |
| 32 | 5.643 | 5.699 | 0.99 |

In Table 5.16, we can see that our predictions are incredibly accurate. In our HLS source code for the compute kernel, we also feature many nested loops and thus need to help the HLS tool to estimate the amount of cycles by giving it accurate inner-loop count estimations. For this, we had to estimate the overlap for the inner-most loop in the algorithm. Since the maximal value for this overlap is $2 * \texttt{MAT\_SIZE} - 1$ and the minimum value is 0, we have taken the floored average values of 3, 7, 15 and 31 respectively for our kernels with MAT_SIZE= 4, 8, 16, and 32. These have proven to be extremely good indicators for this loop count since our final results are of high quality. In Figure 5.11, we can see the final comparison between all our predicted times and the measured times.

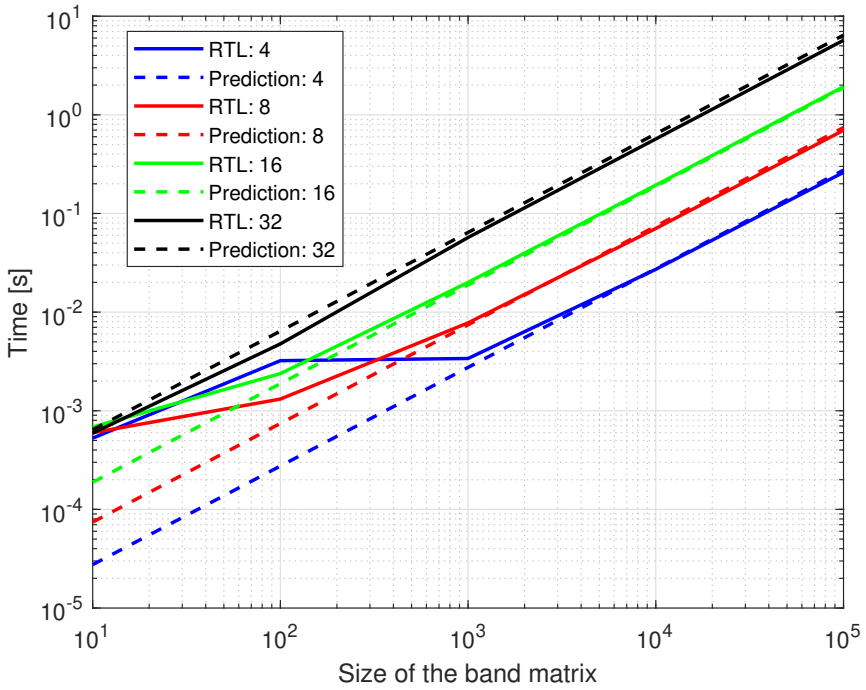


Figure 5.11: Timing results for different sizes of band matrices for our HLS kernel and the associated theoretical predictions.

We see once again that for every kernel size, the estimations begin to be very precise starting from a size 1000 band matrix.

## 5.7. Summary

In this Chapter, we have implemented our kernels and measured their utilisation and achieved times. We have also investigated more closely some specific kernels and implemented stall counters to examine by how much they are being held back from non-optimal memory management hardware. We have found that with our fully-parametric data hardware, our `MAT_SIZE` = 16, `DATA_WIDTH` = 8 RTL GMMM is being slowed down 73.5x from its theoretical speed and our RTL BMMM is being slowed down by 26.2x.

We have also uncovered a 2.6x inexplicable gap in performance between our RTL GMMM operation with custom (ideal) memory management and its theoretical running time.

Lastly, we have compiled a family of kernels utilising our fully-parametric data management functions, and compared our predictions of their running time using only the HLS compilation summaries with their measured running time. We have found that most of our predictions were very accurate and have tried to explain those which were not.

From the data we have gathered, we can say that our current `MAT_SIZE`=16, 8-bit Unified Matrix-Matrix Multiplication RTL kernel is as fast as the HLS version when performing GMMM operations and 23x faster when performing BMMM operations.

We have also proven that architecturally, given optimal memory management, this same configuration utilises approximately 30x more DSP's but can achieve 20x faster GMMM and 610x faster BMMM, given large operations are streamed.

# 6 | Conclusions and Future Developments

During this thesis we have gone through the study case of implementing a Unified Matrix-Matrix Multiplier, capable of preforming Generic Matrix-Matrix Multiplications and Band Matrix-Matrix Multiplications by reusing the same processing elements. We have also implemented functionally equivalent kernels in `C++` using HLS to compile them into hardware. We have then connected these kernels to a host computer using HLS peripherals and developed a complete workflow for Hybrid RTL-HLS design using Xilinx Software.

We have found that for a kernel containing 30x more DSP's, we could theoretically achieve performance increases of 20x for streamed GMMM operations and 610x for streamed BMMM operations, by using a systolic RTL kernel rather than an HLS kernel with optimal memory management hardware. With our memory management hardware, we achieved performance parity in GMMM and a 23.2x improvement in BMMM.

We have also found that with our development board and our methods of communicating between the host computer and the accelerator, implementing expensive systolic hardware is a waste of area if the end application does not require to lump together more than 1000 GMMM operations. On the other hand, for most if not all sizes of BMMM operations, the end application will observe an increase in speed by implementing an RTL systolic kernel rather than an HLS one.

Furthermore, during our tests, we have discovered an unexplained gap of 2.6x in performance between our bottleneck-free kernels, and their theoretical maximum speed, despite the FPGA manufacturer claiming that their hardware is fast enough to implement our architecture at full speed.

In the future, further investigating this 2.6x performance gap is absolutely necessary, in order to understand the limitations of the underlying hardware or what we could do differently in order to achieve the theoretical times.

We have also seen that our fully-parametric data management functions generated in

HLS result in incredibly slow hardware, achieving only a rate of 1 line every 75 cycles for the kernels which expected 1 line per cycle. We believe that much better results can be achieved and more research in this field is crucial.

We do not claim that the systolic architecture we developed is in any way optimal, but we do believe that if a Kung and Leiserson-inspired systolic design is to be implemented for large, streaming BMMM applications, the designers should have very good reasons not to include the GMMM functionality, since it can be implemented with marginal additional hardware.

Some more work must also be done to streamline our Hybrid RTL-HLS workflow, adding fool-proofing for the setup of high-level parameters in multiple files in order to avoid mismatching HLS peripherals with RTL blackboxes intended for different sized kernels, overall decreasing the odds of compiling hardware destined to fail.

Our workflow would also benefit from being able to incorporate Integrated Logic Analyzers into the hardware in order to get a deeper look into what is happening during hardware execution.

Another very welcome feature to add to the workflow would be the automation of kernel compilation, testing, and data logging, allowing for tests to be fully executed without the constant surveillance of the engineer.

Lastly, some more efforts could be made in order to shorten the learning curve for new users willing to make and integrate blackboxes into their own systems. A collection of guidelines in Verilog design and setup tutorials for the many components in a hybrid design would be very convenient.

With this thesis, we only are scratching the surface of multi-directional systolic systems. We believe their potential is immense and we would like to see more research being carried out in this field.

# Bibliography

[1] Amazon. Aws inferentia, high performance machine learning inference chip, custom designed by aws, 2021. URL `https://aws.amazon.com/machine-learning/inferentia/?nc1=h_ls`.

[2] U. E. Avci, D. H. Morris, and I. A. Young. Tunnel field-effect transistors: Prospects and challenges. *IEEE Journal of the Electron Devices Society*, 3(3):88–95, 2015. doi: 10.1109/JEDS.2015.2390591.

[3] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 193–204, 2012.

[4] V. De, S. Vangal, and R. Krishnamurthy. Near threshold voltage (ntv) computing: Computing in the dark silicon era. *IEEE Design & Test*, 34(2):24–30, 2017. doi: 10.1109/MDAT.2016.2573593.

[5] Google. Cloud tpu, 2022. URL `https://cloud.google.com/tpu/docs/system-architecture-tpu-vm`.

[6] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P.-C. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *IEEE Micro*, 31(2):86–95, 2011. doi: 10.1109/MM.2011.18.

[7] J. Henkel, H. Khdr, S. Pagani, and M. Shafique. New trends in dark silicon. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015. doi: 10.1145/2744769.2747938.

[8] J. Koomey, S. Berard, M. Sanchez, and H. Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3):46–54, 2011. doi: 10.1109/MAHC.2010.28.

[9] A. V. Krishnamoorthy, R. Ho, X. Zheng, H. Schwetman, J. Lexau, P. Koka, G. Li,

I. Shubin, and J. E. Cunningham. Computer systems based on silicon photonic interconnects. *Proceedings of the IEEE*, 97(7):1337–1361, 2009. doi: 10.1109/JPROC.2009.2020712.

[10] H. Kung and C. E. Leiserson. Systolic arrays (for vlsi). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for industrial and applied mathematics Philadelphia, PA, USA, 1979.

[11] H.-T. Kung. Why systolic architectures? *Computer*, 15(01):37–46, 1982.

[12] G. E. Moore. Progress in digital integrated electronics [technical literaiture, copyright 1975 ieee. reprinted, with permission. technical digest. international electron devices meeting, ieee, 1975, pp. 11-13.]. *IEEE Solid-State Circuits Society Newsletter*, 11(3): 36–37, 2006. doi: 10.1109/N-SSC.2006.4804410.

[13] G. E. Moore et al. Cramming more components onto integrated circuits, 1965.

[14] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016. doi: 10.1109/TCAD.2015.2513673.

[15] D. Patterson. 50 years of computer architecture: From the mainframe cpu to the domain-specific tpu and the open risc-v instruction set. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 27–31, 2018. doi: 10.1109/ISSCC.2018.8310168.

[16] A. Peschot, C. Qian, and T.-J. King Liu. Nanoelectromechanical switches for low-power digital computing. *Micromachines*, 6(8):1046–1065, 2015.

[17] C. Ramey. Silicon photonics for artificial intelligence acceleration : Hotchips 32. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–26, 2020. doi: 10.1109/HCS49909.2020.9220525.

[18] S. Soldavini, K. F. A. Friebel, M. Tibaldi, G. Hempel, J. Castrillon, and C. Pilato. Automatic creation of high-bandwidth memory architectures from domain-specific languages: The case of computational fluid dynamics, 2022. URL `https://arxiv.org/abs/2203.10850`.

[19] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017. doi: 10.1109/JPROC.2017.2761740.

[20] M. B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *DAC Design Automation Conference 2012*, pages 1131–1136, 2012.

[21] Xilinx. Alveo u280 data center accelerator card data sheet, 2021. URL `https://docs.xilinx.com/r/en-US/ds963-u280`.

[22] Xilinx. Vitis unified software platform documentation: Application acceleration development, 2021. URL `https://docs.xilinx.com/r/2021.1-English/ug1393-vitis-application-acceleration`.

[23] Xilinx. Vitis hls, 2021. URL `https://docs.xilinx.com/r/2021.1-English/ug1399-vitis-hls`.

[24] S. Yu, H. Jiang, S. Huang, X. Peng, and A. Lu. Compute-in-memory chips for deep learning: Recent trends and prospects. *IEEE Circuits and Systems Magazine*, 21(3): 31–56, 2021. doi: 10.1109/MCAS.2021.3092533.

# A | Appendix: Source Code Repository

Listing 54: KLPE.v

```
module KLPE2 #(parameter DATA_WIDTH=64)
  (input [DATA_WIDTH-1:0] A_in,
  input [DATA_WIDTH-1:0] B_in,
  input [DATA_WIDTH-1:0] C_in,
  output reg [DATA_WIDTH-1:0] A_out,
  output reg [DATA_WIDTH-1:0] B_out,
  output reg [DATA_WIDTH-1:0] C_out,
  input array_en,
  input clk,
  input reset);
  wire [2*DATA_WIDTH-1:0] temp = A_in*B_in; //manually define the operation
  always @ (posedge clk)
    begin
      if (reset) begin
        A_out <= 0;
        B_out <= 0;
        C_out <= 0;
      end
      else if (array_en) begin
        A_out <= A_in;
        B_out <= B_in;
        C_out <= C_in + temp[DATA_WIDTH-1:0]; //manually define the operation
        end
    end
endmodule // KLPE

module D1D #(parameter DATA_WIDTH=64)
  (input [DATA_WIDTH-1:0] A_in,
  output reg [DATA_WIDTH-1:0] A_out,
  input array_en,
  input clk,
  input reset);

  always @ (posedge clk)
    if  (reset) A_out <= 0;
    else if (array_en) A_out <= A_in;
endmodule // Delay - 1D
```

Listing 55: unified_array.v

```
`define CLOG2(x) \
  (x <= 2) ? 1 : \
  (x <= 4) ? 2 : \
  (x <= 8) ? 3 : \
  (x <= 16) ? 4 : \
  (x <= 32) ? 5 : \
  (x <= 64) ? 6 : \
  (x <= 128) ? 7 : \
  (x <= 256) ? 8 : \
  (x <= 512) ? 9 : \
  (x <= 1024) ? 10 : 0 //go unrealistically high to cover the basis

module unified_array_core #(parameter MAT_SIZE = 3, DATA_WIDTH=32)
  (
  input clk,
  input array_en,
  input reset,
  input opmode,//0 for generic multiplier, 1 for band multiplier
  input [DATA_WIDTH*(2*MAT_SIZE-1)-1:0] A_flattened,
  input [DATA_WIDTH*(2*MAT_SIZE-1)-1:0] B_flattened,
  output [DATA_WIDTH*(2*(2*MAT_SIZE-1)-1)-1:0] C_band_flattened,
  output [DATA_WIDTH*(2*MAT_SIZE-1)-1:0] C_generic_flattened
  );


      wire [DATA_WIDTH-1:0] A [0:(2*MAT_SIZE-1)-1];
      wire [DATA_WIDTH-1:0] B [0:(2*MAT_SIZE-1)-1];
      wire [DATA_WIDTH-1:0] C_array_out_ver [0:(2*MAT_SIZE-1)-1];
      wire [DATA_WIDTH-1:0] C_array_out_hor [0:(2*MAT_SIZE-1)-2];
      wire [DATA_WIDTH-1:0] C_array_out [0:2*MAT_SIZE-2];
      wire [DATA_WIDTH-1:0] C_generic [0:2*MAT_SIZE-2];//still needs to be flattened and set to
↪  C_flattened
      wire [DATA_WIDTH-1:0] C_band [0:2*(2*MAT_SIZE-1)-2];

    genvar i,j;
    //unflattening
    generate
      for (i=0;i<(2*MAT_SIZE-1);i=i+1) begin
        assign A[i] = A_flattened[DATA_WIDTH*i+(DATA_WIDTH-1):DATA_WIDTH*i];
        assign B[i] = B_flattened[DATA_WIDTH*i+(DATA_WIDTH-1):DATA_WIDTH*i];
      end
    endgenerate

    generate
      for (i=0;i<2*(2*MAT_SIZE-1)-1;i=i+1) begin
        assign C_band_flattened[DATA_WIDTH*i+(DATA_WIDTH-1):DATA_WIDTH*i] = C_band[i];
      end
    endgenerate

    generate
      for (i=0; i<(2*MAT_SIZE-1); i=i+1)
        assign C_generic_flattened[DATA_WIDTH*i+(DATA_WIDTH-1):DATA_WIDTH*i] = C_generic[i];
    endgenerate
```

```
53
54
55
56      //must preemptively declare as many wire arrays as I need
57      //horizontal and vertical wires :
58        wire [DATA_WIDTH-1:0] w_hor [0:(2*MAT_SIZE-1)-1][0:(2*MAT_SIZE-1)-2];
59        wire [DATA_WIDTH-1:0] w_ver [0:(2*MAT_SIZE-1)-1][0:(2*MAT_SIZE-1)-2];
60      //diagonal wires :
61      //some of these wires will not be used but are declared for simplicity of thought
62        wire [DATA_WIDTH-1:0] w_diag [0:(2*MAT_SIZE-1)-1][0:(2*MAT_SIZE-1)-1];
63
64      generate
65        for (j=0; j<(2*MAT_SIZE-1); j=j+1)
66        begin : j_loop
67          for (i=0; i<(2*MAT_SIZE-1); i=i+1)
68          begin : i_loop
69            wire [DATA_WIDTH-1:0] A_in = j==0 ? A[i] : w_ver[i][j-1];
70            wire [DATA_WIDTH-1:0] B_in = i==0 ? B[j] : w_hor[j][i-1];
71            wire [DATA_WIDTH-1:0] C_in = opmode ? ((i==(2*MAT_SIZE-1)-1 || j==(2*MAT_SIZE-1)-1) ? 0 :
↪   w_diag[i+1][j+1]) : ((i==0 || j==0) ? 0 : w_diag[i-1][j-1]);
72
73            wire [DATA_WIDTH-1:0] A_out;
74            wire [DATA_WIDTH-1:0] B_out;
75            wire [DATA_WIDTH-1:0] C_out;
76
77            if (j<(2*MAT_SIZE-1)-1) assign w_ver[i][j] = A_out ;
78            if (i<(2*MAT_SIZE-1)-1) assign w_hor[j][i] = B_out ;
79
80            assign w_diag[i][j] = C_out ;
81
82            if (j>0 && i>0) begin end
83            else if (i==0) assign C_array_out_ver[(2*MAT_SIZE-1)-j-1] = C_out; // needs checkup for
↪   off-by-1
84            else if (j==0) assign C_array_out_hor[i-1] = C_out;//because the top vector is shifted by 1
85
86            if (j<2*MAT_SIZE-2 && i<2*MAT_SIZE-2) begin end
87            else if (j==2*MAT_SIZE-2 && i >= MAT_SIZE-1) assign C_array_out[(3*MAT_SIZE-3) - i] = C_out;
88            else if (j >= MAT_SIZE-1 && j < 2*MAT_SIZE-2) assign C_array_out[j - MAT_SIZE+1] = C_out;
89
90
91
92            KLPE2 #(.DATA_WIDTH(DATA_WIDTH)) pe (
93            .clk(clk),
94            .array_en(array_en),
95            .reset(reset),
96            .A_in(A_in),
97            .B_in(B_in),
98            .C_in(C_in),
99            .A_out(A_out),
100           .B_out(B_out),
101           .C_out(C_out));
102         end
103       end
104     endgenerate
105     //CHECK THIS AS WELL
106
107     generate
```

```
      for (i=0; i<2*(2*MAT_SIZE-1)-1; i=i+1)
      begin
          if (i<(2*MAT_SIZE-1)) assign C_band[i] = C_array_out_ver[i];
          else assign C_band[i] = C_array_out_hor[i-(2*MAT_SIZE-1)];
      end
    endgenerate




  //generate output for generic multiplier
  generate
    for (i=MAT_SIZE-1; i<(2*MAT_SIZE-1); i=i+1) assign C_generic[i] = C_array_out[i];
  endgenerate

  wire [DATA_WIDTH-1:0] w_delays [0:(MAT_SIZE>=3? MAT_SIZE-3:0)][0:(MAT_SIZE>=3? MAT_SIZE-3:0)];

  generate
    for (i=0; i<MAT_SIZE-1; i=i+1)
      begin : i_loop
      for (j=0; j<=i; j=j+1)
        begin : j_loop

         wire [DATA_WIDTH-1:0] A_in = j==0 ? C_array_out[i] : w_delays[i-1][j-1];
         wire [DATA_WIDTH-1:0] A_out;

        if (i<MAT_SIZE-2) assign w_delays[i][j] = A_out;
        else assign C_generic[MAT_SIZE-2-j] = A_out;

        D1D #(.DATA_WIDTH(DATA_WIDTH)) dblock (
          .clk(clk),
          .reset(reset),
          .array_en(array_en),
          .A_in(A_in),
          .A_out(A_out));
      end
    end
  endgenerate

endmodule




module unified_array #(parameter MAT_SIZE = 16, DATA_WIDTH=32)
  (
    input clk,
    input reset,
    input [31:0] counter,//last two bits are used as tricounter when in band operation
    input array_en,
    input opmode, //0 for generic, 1 for band
    input [DATA_WIDTH*(2*MAT_SIZE-1)-1:0] A_band_flattened,
    input [DATA_WIDTH*(2*MAT_SIZE-1)-1:0] B_band_flattened,
    output [DATA_WIDTH*(2*(2*MAT_SIZE-1)-1)-1:0] C_band_flattened,
    input [DATA_WIDTH*MAT_SIZE-1:0] A_generic_flattened,
    input [DATA_WIDTH*MAT_SIZE-1:0] B_generic_flattened,
    output [DATA_WIDTH*MAT_SIZE-1:0] C_generic_flattened
```

```
165   );
166
167 //GENERIC MULTIPLIER PERIPHERALS
168
169   wire [DATA_WIDTH*(2*MAT_SIZE-1)-1:0] A_generic_flattened_steering2array;
170   wire [DATA_WIDTH*(2*MAT_SIZE-1)-1:0] B_generic_flattened_steering2array;
171   wire [DATA_WIDTH*(2*MAT_SIZE-1)-1:0] C_generic_flattened_array2steering;
172
173
174
175 IO_altarray_steering #(.DATA_WIDTH(DATA_WIDTH),
176                              .MAT_SIZE(MAT_SIZE))
177   steerer0 (
178     .clk(clk),
179     .array_en(array_en),
180     .counter_in(counter[(`CLOG2(MAT_SIZE))-1:0]),
181     .flattened_data_in_A(A_generic_flattened),
182     .flattened_data_to_array_A(A_generic_flattened_steering2array),
183     .flattened_data_in_B(B_generic_flattened),
184     .flattened_data_to_array_B(B_generic_flattened_steering2array),
185     .flattened_data_from_array_C(C_generic_flattened_array2steering),
186     .flattened_data_out_C(C_generic_flattened)
187     );
188
189
190
191 //BAND MULTIPLIER PERIPHERALS
192
193   wire [DATA_WIDTH*(2*(2*MAT_SIZE-1)-1)-1:0] array_to_output_device_flattened;
194   wire [DATA_WIDTH*(2*MAT_SIZE-1)-1:0] input_device_A_to_array_flattened;
195   wire [DATA_WIDTH*(2*MAT_SIZE-1)-1:0] input_device_B_to_array_flattened;
196
197 band_peripherals #(.DATA_WIDTH(DATA_WIDTH),.BAND_SIZE(2*MAT_SIZE-1))
198
199   band_peripherals_0 (
200     .clk(clk),
201     .reset(reset),
202     .tricounter(counter[1:0]),
203     .array_en(array_en),
204     .A_flattened(A_band_flattened),
205     .B_flattened(B_band_flattened),
206     .C_flattened(C_band_flattened),
207     .input_device_A_to_array_flattened(input_device_A_to_array_flattened),
208     .input_device_B_to_array_flattened(input_device_B_to_array_flattened),
209     .array_to_output_device_flattened(array_to_output_device_flattened)
210   );
211
212 //UNIFIED ARRAY
213
214   unified_array_core #(.MAT_SIZE(MAT_SIZE), .DATA_WIDTH(DATA_WIDTH))
215     unified_array_0 (
216       .clk(clk),
217       .array_en(array_en), //half implemented /!\
218       .reset(reset), //not yet implemented /!\
219       .opmode(opmode), //0 for generic multiplier, 1 for band multiplier
220       .A_flattened(opmode ? input_device_A_to_array_flattened : A_generic_flattened_steering2array),
221       .B_flattened(opmode ? input_device_B_to_array_flattened : B_generic_flattened_steering2array),
```

```
      .C_band_flattened(array_to_output_device_flattened),
      .C_generic_flattened(C_generic_flattened_array2steering)
      );


endmodule
```

Listing 56: datasteering.v

```
`define CLOG2(x) \
   (x <= 2) ? 1 : \
   (x <= 4) ? 2 : \
   (x <= 8) ? 3 : \
   (x <= 16) ? 4 : \
   (x <= 32) ? 5 : \
   (x <= 64) ? 6 : \
   (x <= 128) ? 7 : \
   (x <= 256) ? 8 : \
   (x <= 512) ? 9 : \
   (x <= 1024) ? 10 : 0 //go unrealistically high to cover the basis

module datasteerer #(parameter MAT_SIZE=3, DATA_WIDTH = 64)(
  input clk,
  input [(`CLOG2(MAT_SIZE))-1:0] counter,
  input [MAT_SIZE*DATA_WIDTH-1:0] data_in,
  output [(2*MAT_SIZE-1)*DATA_WIDTH-1:0] data_out);
    wire [(MAT_SIZE-1)*DATA_WIDTH-1:0] zeropadding = 0;
  assign data_out = {zeropadding,data_in} << DATA_WIDTH*counter;
endmodule // datasteerer

module datacollector #(parameter MAT_SIZE=3,DATA_WIDTH = 64)(
  input clk,
  input [(`CLOG2(MAT_SIZE))-1:0] counter,
  input [(2*MAT_SIZE-1)*DATA_WIDTH-1:0] data_in,
  output [MAT_SIZE*DATA_WIDTH-1:0] data_out
  );
    assign data_out =  data_in >> DATA_WIDTH*(MAT_SIZE-1-counter);
endmodule // datacollector


module IO_altarray_steering #(parameter MAT_SIZE=3,DATA_WIDTH=64)(
  input clk,
  input [(`CLOG2(MAT_SIZE))-1:0] counter_in,
  input array_en,

  input [MAT_SIZE*DATA_WIDTH-1:0] flattened_data_in_A,
  output [(2*MAT_SIZE-1)*DATA_WIDTH-1:0] flattened_data_to_array_A,

  input [MAT_SIZE*DATA_WIDTH-1:0] flattened_data_in_B,
  output [(2*MAT_SIZE-1)*DATA_WIDTH-1:0] flattened_data_to_array_B,

  input [(2*MAT_SIZE-1)*DATA_WIDTH-1:0] flattened_data_from_array_C,
  output [MAT_SIZE*DATA_WIDTH-1:0] flattened_data_out_C
```

```
  );

  datasteerer #(.DATA_WIDTH(DATA_WIDTH),
                .MAT_SIZE(MAT_SIZE))
    datasteererA (
      .clk(clk),
      .counter(counter_in),
      .data_in(flattened_data_in_A),
      .data_out(flattened_data_to_array_A)
      );

  datasteerer #(.DATA_WIDTH(DATA_WIDTH),
                .MAT_SIZE(MAT_SIZE))
    datasteererB (
      .clk(clk),
      .counter(counter_in),
      .data_in(flattened_data_in_B),
      .data_out(flattened_data_to_array_B)
      );


  genvar i;
    wire [(`CLOG2(MAT_SIZE))-1:0] counter_intermediate [2*MAT_SIZE-2:0];
  generate
    for(i=0;i<2*MAT_SIZE-1;i=i+1) begin

      wire [(`CLOG2(MAT_SIZE))-1:0] cnt_in,cnt_out;
      assign cnt_in = i==0? counter_in : counter_intermediate[i-1];
      assign counter_intermediate[i] = cnt_out;

      D1D #(.DATA_WIDTH(`CLOG2(MAT_SIZE))) dblock (
        .array_en(array_en),
        .clk(clk),
        .reset(0),
        .A_in(cnt_in),
        .A_out(cnt_out));
    end
endgenerate


  datacollector #(.DATA_WIDTH(DATA_WIDTH),
                  .MAT_SIZE(MAT_SIZE))
    datacollectorC (
      .clk(clk),
      .counter(counter_intermediate[2*MAT_SIZE-2]),
      .data_in(flattened_data_from_array_C),
      .data_out(flattened_data_out_C)
      );
endmodule
```

Listing 57: band_input_device.v

```
`define GET_BUF_SIZE(x) x/3+1

module band_input_device #(
  parameter DATA_WIDTH=32,
  parameter BAND_SIZE=5)
  (
  input clk,
  input reset,
  input [1:0] tricounter,//figure out how many bits are needed later
  input array_en,
  input [DATA_WIDTH*(BAND_SIZE)-1:0] IN_flattened,
  output [DATA_WIDTH*(BAND_SIZE)-1:0] OUT_flattened
  );
    reg [BAND_SIZE-1:0] binary_pattern;
    reg [9:0] dispatch_pattern [0:BAND_SIZE-1];//figure out how many bits are needed later
    reg [9:0] write_pointer;//figure out how many bits are needed later
    reg [DATA_WIDTH-1:0] databuffer [0:(`GET_BUF_SIZE(BAND_SIZE))-1][0:BAND_SIZE-1];//internal data
↪  buffer


  genvar i,j;
  integer k,l;


  initial binary_pattern=0;
  initial write_pointer=0;

  always @(posedge clk) begin
    for(k=0;k<BAND_SIZE;k=k+1) begin
      if(k==0) begin
        if(reset) binary_pattern[k] <= 0;
        else if (array_en) binary_pattern[k] <= (tricounter==0) ? 1'b1 : 1'b0;
      end
      else begin
        if (reset) binary_pattern[k] <= 0;
        else if(array_en) binary_pattern[k] <= (tricounter == 2'b11) ? 1'b0 : binary_pattern[k-1];
      end
    end
  end


  always @(posedge clk)
    if (reset)
        write_pointer<=0;
    else if(array_en)
      if(binary_pattern[0]==1)
        write_pointer <= (write_pointer+1) % (`GET_BUF_SIZE(BAND_SIZE));



  always @(posedge clk) begin
    if(reset) begin
      for(k=0;k<BAND_SIZE;k=k+1) begin
        dispatch_pattern[k]<= 0;//reset everything
      end
    end
    else if(array_en) begin
```

```
57        for(k=0;k<BAND_SIZE;k=k+1) begin
58          dispatch_pattern[k] <= (binary_pattern[k]==1) ? (dispatch_pattern[k] + 1) %
    ↪  (`GET_BUF_SIZE(BAND_SIZE)) : dispatch_pattern[k];
59        end
60      end
61    end
62
63
64
65
66    generate
67      for(i=0;i<BAND_SIZE;i=i+1) begin
68        assign OUT_flattened[i*DATA_WIDTH  +: DATA_WIDTH] = (binary_pattern[i] == 1) ?
    ↪  databuffer[dispatch_pattern[i]][i] : 0;
69      end
70    endgenerate
71
72    always @(posedge clk) begin
73      if (reset) begin
74        for(k=0;k<BAND_SIZE;k=k+1) begin
75          for(l=0;l<(`GET_BUF_SIZE(BAND_SIZE));l=l+1) begin//not generate block but loop block
76            databuffer[l][k] <= 0; // reset everything
77          end
78        end
79      end
80      else if(array_en)
81        if (tricounter == 2'b00)
82          for(k=0;k<BAND_SIZE;k=k+1) begin
83            databuffer[write_pointer][k] <=  IN_flattened[k*DATA_WIDTH +: DATA_WIDTH];
84          end
85    end
86
87  endmodule
```

Listing 58: band_output_select_and_route.v

```
1   `define GET_BUF_SIZE(x) x/3+1
2
3   module band_output_select_and_route #(
4     parameter DATA_WIDTH=32,
5     parameter BAND_SIZE=5)
6     (
7     input clk,
8     input reset,
9     input [1:0] tricounter,//figure out how many bits are needed later
10    input array_en,
11    input [DATA_WIDTH*(2*BAND_SIZE-1)-1:0] IN_flattened,
12    output wire [DATA_WIDTH*(2*BAND_SIZE-1)-1:0] OUT_flattened
13    );
14      reg [9:0] current_line;//figure out how many bits are needed later
15      reg [2*BAND_SIZE-2:0] binary_pattern;
16      reg [9:0] write_pointer [2*BAND_SIZE-2:0];//figure out how many bits are needed later
17      reg [DATA_WIDTH-1:0] databuffer [0:(`GET_BUF_SIZE(BAND_SIZE))-1][0:2*BAND_SIZE-2];//internal data
    ↪  buffer
```

```
18
19
20    genvar i,j;
21    integer k,l;
22
23    always @(posedge clk) begin
24      if(reset) current_line <= 0;
25      else if(array_en) if(binary_pattern[0]==1) current_line <= write_pointer[0];
26    end
27
28    always @(posedge clk) begin
29      for(k=0;k<=2*BAND_SIZE-2;k=k+1) begin
30        if(reset) binary_pattern[k] <= 0;
31        else if (array_en) begin
32          if(k==BAND_SIZE-1)      binary_pattern[k] <= (tricounter == 0) ? 1'b1 : 1'b0;
33          else if(k<BAND_SIZE-1)  binary_pattern[k] <= (tricounter == 2'b11) ? 1'b0 : binary_pattern[k+1];
34          else                    binary_pattern[k] <= (tricounter == 2'b11) ? 1'b0 : binary_pattern[k-1];
35        end
36      end
37    end
38
39    always @(posedge clk) begin
40      for(k=0;k<=2*BAND_SIZE-2;k=k+1) begin
41        if (reset) write_pointer[k] <= 0;
42        else if(array_en) begin
43          if(binary_pattern[k] == 1) begin
44            if(write_pointer[k] == (`GET_BUF_SIZE(BAND_SIZE))-1) write_pointer[k] <= 0;
45            else write_pointer[k] <= write_pointer[k] + 1;
46          end
47        end
48      end
49    end
50
51    always @(posedge clk) begin
52      for(k=0;k<=2*BAND_SIZE-2;k=k+1) begin
53        for(l=0;l<=(`GET_BUF_SIZE(BAND_SIZE))-1;l=l+1) begin
54          if (reset) databuffer[l][k] <= 0;
55          else if (array_en)
56            if(binary_pattern[k]==1)
57              if(write_pointer[k]==l)
58                databuffer[l][k] <= IN_flattened[k*DATA_WIDTH +: DATA_WIDTH];
59        end
60      end
61    end
62
63
64    generate
65      for(i=0;i<=2*BAND_SIZE-2;i=i+1)
66            assign OUT_flattened[(i+1)*DATA_WIDTH-1:i*DATA_WIDTH] = databuffer[current_line][i];
67    endgenerate
68
69  endmodule
```

Listing 59: band_peripherals.v

```
1
2  //this module encapsulates the band matrix multiplier array with the input and output data interface
3
4  module band_peripherals #(parameter DATA_WIDTH=32, BAND_SIZE=5)
5    (input clk,
6     input reset,
7     input [1:0] tricounter,
8     input array_en,
9     input [DATA_WIDTH*BAND_SIZE-1:0] A_flattened,
10    input [DATA_WIDTH*BAND_SIZE-1:0] B_flattened,
11    output [DATA_WIDTH*(2*BAND_SIZE-1)-1:0] C_flattened,
12    input [DATA_WIDTH*(2*BAND_SIZE-1)-1:0] array_to_output_device_flattened,
13    output [DATA_WIDTH*(BAND_SIZE)-1:0] input_device_A_to_array_flattened,
14    output [DATA_WIDTH*(BAND_SIZE)-1:0] input_device_B_to_array_flattened
15    );
16
17    genvar i;
18
19    //debugging help to read flattened garbage
20   /*
21    wire [DATA_WIDTH-1:0] DEBUG [0:2*BAND_SIZE-2];
22    generate
23    for (i=0; i<2*BAND_SIZE-1; i=i+1)
24      begin
25        assign DEBUG[i] = array_to_output_device_flattened[DATA_WIDTH*i+(DATA_WIDTH-1):DATA_WIDTH*i];
26      end
27    endgenerate
28 */
29
30
31    band_input_device #(.DATA_WIDTH(DATA_WIDTH),.BAND_SIZE(BAND_SIZE)) band_input_device_A
32        (
33        .clk(clk),
34        .reset(reset),
35        .tricounter(tricounter),//figure out how many bits are needed later
36        .array_en(array_en),
37        .IN_flattened(A_flattened),
38        .OUT_flattened(input_device_A_to_array_flattened)
39        );
40
41
42    band_input_device #(.DATA_WIDTH(DATA_WIDTH),.BAND_SIZE(BAND_SIZE)) band_input_device_B
43        (
44        .clk(clk),
45        .reset(reset),
46        .tricounter(tricounter),//figure out how many bits are needed later
47        .array_en(array_en),
48        .IN_flattened(B_flattened),
49        .OUT_flattened(input_device_B_to_array_flattened)
50        );
51
52
53      reg [1:0] tricounter_delayed;
54
55     always @(posedge clk) begin
56       if (reset) tricounter_delayed <= 2;
57       else if(array_en) tricounter_delayed <= tricounter;
```

```
58      end
59
60      band_output_select_and_route #(.DATA_WIDTH(DATA_WIDTH),.BAND_SIZE(BAND_SIZE)) output_device_0
61       (.clk(clk),
62       .reset(reset),
63       .tricounter(tricounter_delayed),
64       .array_en(array_en),
65       .IN_flattened(array_to_output_device_flattened),
66       .OUT_flattened(C_flattened));
67
68
69   endmodule
```

Listing 60: `mmc.json`

```
1   {
2       "c_function_name"     : "mmc",
3       "rtl_top_module_name" : "mmc",
4       "c_files" : [{
5                     "c_file" : "/.../mmc.cpp",
6                     "cflag" : ""
7                   }],
8       "rtl_files" : [
9                       "/.../mmc.v",
10                      "/.../unified_array.v",
11                      "/.../datasteering.v",
12                      "/.../KLPE.v",
13                      "/.../band_peripherals.v",
14                      "/.../band_input_device.v",
15                      "/.../band_output_select_and_route.v"
16                    ],
17      "c_parameters" : [
18                          {
19                            "c_name" : "opmode_stream",
20                            "c_port_direction" : "in",
21                            "rtl_ports" : {
22                                          "FIFO_empty_flag" : "opmode_empty_n",
23                                          "FIFO_read_enable" : "opmode_re",
24                                          "FIFO_data_read_in" : "opmode"
25                                          }
26                          },
27                          {
28                            "c_name" : "size_stream",
29                            "c_port_direction" : "in",
30                            "rtl_ports" : {
31                                          "FIFO_empty_flag" : "size_empty_n",
32                                          "FIFO_read_enable" : "size_re",
33                                          "FIFO_data_read_in" : "size"
34                                          }
35                          },
36                          {
37                            "c_name" : "band_type_stream",
38                            "c_port_direction" : "in",
```

```
                    "rtl_ports" : {
                                "FIFO_empty_flag" : "band_type_empty_n",
                                "FIFO_read_enable" : "band_type_re",
                                "FIFO_data_read_in" : "band_type"
                                }
                  },
                {
                "c_name" : "A_gen_stream",
                "c_port_direction" : "in",
                "rtl_ports" : {
                            "FIFO_empty_flag" : "a_gen_empty_n",
                            "FIFO_read_enable" : "a_gen_re",
                            "FIFO_data_read_in" : "a_gen"
                          }
                },
                {
                "c_name" : "B_gen_stream",
                "c_port_direction" : "in",
                "rtl_ports" : {
                            "FIFO_empty_flag" : "b_gen_empty_n",
                            "FIFO_read_enable" : "b_gen_re",
                            "FIFO_data_read_in" : "b_gen"
                          }
                },
                {
                "c_name" : "A_band_stream",
                "c_port_direction" : "in",
                "rtl_ports" : {
                            "FIFO_empty_flag" : "a_band_empty_n",
                            "FIFO_read_enable" : "a_band_re",
                            "FIFO_data_read_in" : "a_band"
                          }
                },
                {
                  "c_name" : "B_band_stream",
                  "c_port_direction" : "in",
                  "rtl_ports" : {
                              "FIFO_empty_flag" : "b_band_empty_n",
                              "FIFO_read_enable" : "b_band_re",
                              "FIFO_data_read_in" : "b_band"
                            }
                  },
                {
                  "c_name" : "C_band_stream",
                  "c_port_direction" : "out",
                  "rtl_ports" : {
                                "FIFO_full_flag" : "c_band_full_n",
                                "FIFO_write_enable" : "c_band_we",
                                "FIFO_data_write_out" : "c_band"
                              }
                },
                {
                  "c_name" : "C_gen_stream",
                  "c_port_direction" : "out",
                  "rtl_ports" : {
                                "FIFO_full_flag" : "c_gen_full_n",
                                "FIFO_write_enable" : "c_gen_we",
```

```
                                    "FIFO_data_write_out" : "c_gen"
                                  }
                    }],
    "rtl_common_signal" : {
                            "module_clock"                   : "ap_clk",
                            "module_reset"                   : "ap_rst",
                            "module_clock_enable"            : "ap_ce",
                            "ap_ctrl_chain_protocol_idle"    : "ap_idle",
                            "ap_ctrl_chain_protocol_start"   : "ap_start",
                            "ap_ctrl_chain_protocol_ready"   : "ap_ready",
                            "ap_ctrl_chain_protocol_done"    : "ap_done",
                            "ap_ctrl_chain_protocol_continue" : "ap_continue"
                          },
    "rtl_performance" : {
                          "latency" : "0",
                          "II"      : "1"
                        },
    "rtl_resource_usage" : {
                            "FF"   :   "0",
                            "LUT"  :   "0",
                            "BRAM" :   "0",
                            "URAM" :   "0",
                            "DSP"  :   "1"
                           }
    }

```

Listing 61: `mmc.v`

```
//
// Copyright 2021 Xilinx, Inc.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//   http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
//
`timescale 100ps/100ps

`define BAND_IN_SIZE      (DATA_WIDTH*(2*MAT_SIZE-1))
`define GEN_IN_SIZE       (DATA_WIDTH*MAT_SIZE)
`define BAND_OUT_SIZE     (DATA_WIDTH*(2*(2*MAT_SIZE-1)-1))
`define GEN_OUT_SIZE      (DATA_WIDTH*MAT_SIZE)
`define ARRAY_SIZE        (2*MAT_SIZE-1)

`define START_BAND_VALIDITY ((2*MAT_SIZE-1)*4-1-3*band_type)
```

```
25  `define BAND_DURATION (3*size)
26
27  (* use_dsp = "simd" *)
28  (* dont_touch = "true" *)
29  module mmc #(parameter MAT_SIZE=4, DATA_WIDTH=8)
30        (input ap_clk, ap_rst, ap_ce, ap_start, ap_continue,
31        output                          ap_idle, ap_done, ap_ready,
32        input [`GEN_IN_SIZE-1:0]        a_gen,
33        input                           a_gen_empty_n,
34        output  reg                        a_gen_re,
35        input [`GEN_IN_SIZE-1:0]        b_gen,
36        input                           b_gen_empty_n,
37        output  reg                        b_gen_re,
38        input [`BAND_IN_SIZE-1:0]       a_band,
39        input                           a_band_empty_n,
40        output  reg                        a_band_re,
41        input [`BAND_IN_SIZE-1:0]       b_band,
42        input                           b_band_empty_n,
43        output   reg                       b_band_re,
44        input                           opmode,
45        input                           opmode_empty_n,
46        output reg                         opmode_re,
47        input [31:0]                    size,
48        input                           size_empty_n,
49        output  reg                        size_re,
50        input [31:0]                    band_type,
51        input                           band_type_empty_n,
52        output   reg                       band_type_re,
53        output [`BAND_OUT_SIZE-1:0]     c_band,
54        input                           c_band_full_n,
55        output   reg                       c_band_we,
56        output [`GEN_OUT_SIZE-1:0]      c_gen,
57        input                           c_gen_full_n,
58        output  reg                        c_gen_we);
59
60  /*---------declare states and substates as local parameters-----*/
61
62  localparam [4:0] s_reset        = 5'd0;
63  localparam [4:0] s_write        = 5'd1;
64  localparam [4:0] s_done         = 5'd2;
65  localparam [4:0] s_idle         = 5'd3;
66  localparam [4:0] s_readparams   = 5'd4;
67  localparam [4:0] s_readinputs   = 5'd5;
68  localparam [4:0] s_gencompute   = 5'd6;
69  localparam [4:0] s_bandcompute  = 5'd7;
70  localparam [4:0] s_selectopmode  = 5'd8;
71
72  localparam [4:0] ss_0 = 5'd0;
73  localparam [4:0] ss_1 = 5'd1;
74  localparam [4:0] ss_2 = 5'd2;
75  localparam [4:0] ss_3 = 5'd3;
76  localparam [4:0] ss_4 = 5'd4;
77  localparam [4:0] ss_5 = 5'd5;
78
79  localparam mode_gen = 1'b0;
80  localparam mode_band = 1'b1;
81
```

```
82  /*FSM defined by states and a counter*/
83
84    reg [4:0] current_state;
85    reg [4:0] next_state;
86
87    reg [31:0] counter;
88    reg [31:0] next_counter;
89
90    reg [1:0] tricounter;
91    reg [1:0] next_tricounter;
92
93  /* define speedup parameters */
94
95    reg [31:0] start_band_validity;
96    reg [31:0] next_start_band_validity;
97
98    reg [31:0] band_duration;
99    reg [31:0] next_band_duration;
100
101
102 /*--------declare necessary stuff--------*/
103   wire [4:0] gen_substate = (counter < size) ? ss_0 : (counter < (`ARRAY_SIZE))? ss_1 : ss_2;
104   wire end_of_gen_op = (counter == (size + (`ARRAY_SIZE)));
105
106   wire band_output_valid = (counter >= start_band_validity);
107   wire end_of_band_op = counter == (start_band_validity + band_duration);
108
109   reg must_read;
110   reg must_write;
111
112   wire [4:0] band_substate = (must_read && must_write)? ss_1 : (must_read)? ss_0 : (must_write)? ss_2 :
    ↪  ss_3;
113
114 /*assign bb signals*/
115 assign ap_ready = 1;
116 assign ap_idle = ap_start ? 0 : (current_state == s_idle || current_state == s_reset);
117 assign ap_done = current_state == s_done;//(current_state == s_done || current_state == s_idle);
118
119 /*signals related to array and array itself*/
120   reg array_en_reg;
121   reg inputs_zero;
122
123
124   wire [31:0] counter_to_array = opmode ? tricounter : counter;
125   wire array_en = ap_ce ? array_en_reg : 0 ;
126   wire array_rst = ap_rst ? 1 : current_state == s_reset;
127
128   wire [`GEN_IN_SIZE-1:0] a_gen_in = inputs_zero ? 0 : a_gen;
129   wire [`GEN_IN_SIZE-1:0] b_gen_in = inputs_zero ? 0 : b_gen;
130
131   wire [`BAND_IN_SIZE-1:0] a_band_in = inputs_zero ? 0 : a_band;
132   wire [`BAND_IN_SIZE-1:0] b_band_in = inputs_zero ? 0 : b_band;
133
134
135 unified_array #(.MAT_SIZE(MAT_SIZE), .DATA_WIDTH(DATA_WIDTH)) array0
136     (
137       .clk(ap_clk),
```

```
      .reset(array_rst),
      .counter(counter_to_array),//last two bits are used as tricounter when in band operation
      .array_en(array_en),
      .opmode(opmode),        //0 for generic, 1 for band
      .A_band_flattened(a_band_in),
      .B_band_flattened(b_band_in),
      .C_band_flattened(c_band),
      .A_generic_flattened(a_gen_in),
      .B_generic_flattened(b_gen_in),
      .C_generic_flattened(c_gen)
    );



/*---------------FSM STARTS HERE-------------*/
/*-----CLOCKED PART-------------------------*/
always @ (posedge ap_clk)
if (ap_rst) begin
  current_state <= s_reset;
  counter <= 0;
  tricounter <= 0;

  band_duration <= 1;
  start_band_validity <= 1;
end
else if (ap_ce) begin
  current_state <= next_state;
  counter <= next_counter;
  tricounter <= next_tricounter;

  band_duration <= next_band_duration;
  start_band_validity <= next_start_band_validity;
end


always @(*)
begin
  a_band_re     = 0;
  b_band_re     = 0;
  a_gen_re      = 0;
  b_gen_re      = 0;
  opmode_re     = 0;
  size_re       = 0;
  band_type_re  = 0;
  c_band_we     = 0;
  c_gen_we      = 0;

  array_en_reg  = 0;
  inputs_zero   = 0;

  next_state = s_reset;
  next_counter = counter;
  next_tricounter = tricounter;

  must_read = 0;
  must_write = 0;

```

```
195   next_start_band_validity = start_band_validity;
196   next_band_duration = band_duration;
197
198   case(current_state)
199     s_reset: begin
200        // reset all the output registers in the reset state
201       a_band_re     = 0;
202       b_band_re     = 0;
203       a_gen_re      = 0;
204       b_gen_re      = 0;
205       opmode_re     = 0;
206       size_re       = 0;
207       band_type_re  = 0;
208       c_band_we     = 0;
209       c_gen_we      = 0;
210
211       array_en_reg  = 0;
212       inputs_zero   = 0;
213
214       next_state = s_reset;
215       next_counter = 0;
216       next_tricounter = 0;
217
218       if (ap_start) begin
219         next_state = s_readparams;
220       end
221     end
222
223     s_readparams: begin
224       opmode_re     = 0;
225       size_re       = 0;
226       band_type_re  = 0;
227       next_state = s_readparams;
228       if (opmode_empty_n && band_type_empty_n && size_empty_n) begin
229         next_state = s_selectopmode;
230       end
231     end
232
233     s_selectopmode: begin
234
235       next_start_band_validity = `START_BAND_VALIDITY; // make the actual calculation here
236       next_band_duration = `BAND_DURATION;
237
238       case(opmode)
239         mode_gen: begin
240           next_state = s_gencompute;
241           next_counter = 0;
242           next_tricounter = 0;
243         end
244
245         mode_band: begin
246           next_state = s_bandcompute;
247           next_counter = 0;
248           next_tricounter = 0;
249         end
250
251         default: begin
```

```
252           next_state = s_done;
253         end
254       endcase
255     end
256     /*
257 ----------------------------------------
258 ----------------------------------------
259 ----------------------------------------
260 --------------GENERIC-------------------
261 ----------------------------------------
262 ----------------------------------------
263 ----------------------------------------
264 ----------------------------------------
265     */
266 
267     s_gencompute: begin
268       // go into this state with counter = 0 and substate = 0;
269       //cleanup the re signals from previous state
270       //for generic operation, reading and writing are separate in time,
271       //thus we do not need to check if the inputs and the outputs are both free to enable the register
272 
273     //-------deal with inputs ----------/
274       a_gen_re  = 0;
275       b_gen_re  = 0;
276       next_counter  =  counter;
277       array_en_reg = 0;
278       inputs_zero = 0;
279       c_gen_we = 0;
280 
281       case(gen_substate)
282         ss_0: begin //read inputs
283           if (a_gen_empty_n && b_gen_empty_n) begin // overwrite previous statements in necessary
284             a_gen_re  = 1;
285             b_gen_re  = 1;
286             next_counter = counter + 1;
287             array_en_reg = 1;
288           end
289         end
290 
291         ss_1: begin //dead cycling
292           a_gen_re = 0;
293           b_gen_re = 0;
294           next_counter = counter + 1;
295           array_en_reg = 1; //corresponding 0 is up top
296           inputs_zero = 1;
297         end
298 
299         ss_2: begin //write outputs
300           if (c_gen_full_n) begin
301             c_gen_we = 1;
302             next_counter = counter + 1;
303             array_en_reg = 1;
304           end
305         end
306       endcase
307 
308       //------mark end of operation-------/
```

```
309       next_state = current_state;
310       if(end_of_gen_op) begin
311         next_state = s_done;
312         next_counter = 0; //reset the counter, you never know
313       end
314     end
315     /*
316 ----------------------------------------
317 ----------------------------------------
318 ----------------------------------------
319 -----------------BAND-------------------
320 ----------------------------------------
321 ----------------------------------------
322 ----------------------------------------
323 ----------------------------------------
324     */
325     s_bandcompute: begin // reachable now. The design has to be smart.
326       inputs_zero = 0;
327       array_en_reg = 0;
328       next_counter = counter;
329       next_tricounter = tricounter;
330       must_read = 0;
331       must_write = 0;
332 
333       a_band_re = 0;
334       b_band_re = 0;
335       c_band_we = 0;
336       if (counter >= band_duration) begin
337         inputs_zero = 1;
338       end
339       if(tricounter == 2'b00 && counter < band_duration) begin // if the counter is below the band
    ↪  duration then we must read
340         must_read = 1;
341       end
342 
343       if (counter >= start_band_validity && tricounter == 2'b00) begin
344         must_write = 1;
345       end // then we must write
346 
347 
348       //put a case statement here
349       case(band_substate)
350         ss_0: begin// must only read
351           if(a_band_empty_n && b_band_empty_n) begin //read and cycle
352             a_band_re = 1;
353             b_band_re = 1;
354             next_counter = counter + 1;
355             next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
356             array_en_reg = 1;
357           end //else jsut wait
358         end
359 
360         ss_1: begin// must read and write
361           if(a_band_empty_n && b_band_empty_n && c_band_full_n) begin
362             a_band_re = 1;
363             b_band_re = 1;
364             c_band_we = 1;
```

```
365             next_counter = counter + 1;
366             next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
367             array_en_reg = 1;
368           end // else just wait
369         end
370
371         ss_2: begin // must only write
372           if(c_band_full_n) begin
373             c_band_we = 1;
374             next_counter = counter + 1;
375             next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
376             array_en_reg = 1;
377           end
378         end
379
380         ss_3: begin //must do nothing but still cycle
381           next_counter = counter + 1;
382           next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
383           array_en_reg = 1;
384         end
385
386         default: begin // must do nothing, same as ss_3
387           next_counter = counter + 1;
388           next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
389           array_en_reg = 1;
390         end
391       endcase
392
393
394       next_state = s_bandcompute;
395       if (end_of_band_op) next_state = s_done;
396     end
397
398     s_done: begin // put all the output we to 0;
399       opmode_re     = 1;
400       size_re       = 1;
401       band_type_re  = 1;
402       c_gen_we = 0;
403       c_band_we = 0;
404       next_state = s_idle;
405     end
406
407     s_idle: begin
408       opmode_re     = 0;
409       size_re       = 0;
410       band_type_re  = 0;
411       next_state = s_reset;
412     end
413
414     default: begin
415       a_band_re     = 0;
416       b_band_re     = 0;
417       a_gen_re      = 0;
418       b_gen_re      = 0;
419       opmode_re     = 0;
420       size_re       = 0;
421       band_type_re  = 0;
```

```
422         c_band_we     = 0;
423         c_gen_we      = 0;
424 
425         array_en_reg  = 0;
426         inputs_zero   = 0;
427 
428         next_state = s_reset;
429         next_counter = 0;
430         next_tricounter = 0;
431 
432         must_read = 0;
433         must_write = 0;
434       end
435     endcase
436 end
437 
438 
439 endmodule
```

Listing 62: mmc2.v

```
1  //
2  // Copyright 2021 Xilinx, Inc.
3  //
4  // Licensed under the Apache License, Version 2.0 (the "License");
5  // you may not use this file except in compliance with the License.
6  // You may obtain a copy of the License at
7  //
8  //   http://www.apache.org/licenses/LICENSE-2.0
9  //
10 // Unless required by applicable law or agreed to in writing, software
11 // distributed under the License is distributed on an "AS IS" BASIS,
12 // WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 // See the License for the specific language governing permissions and
14 // limitations under the License.
15 //
16 `timescale 100ps/100ps
17 
18 `define CLOG2(x) \
19    (x <= 2) ? 1 : \
20    (x <= 4) ? 2 : \
21    (x <= 8) ? 3 : \
22    (x <= 16) ? 4 : \
23    (x <= 32) ? 5 : \
24    (x <= 64) ? 6 : \
25    (x <= 128) ? 7 : \
26    (x <= 256) ? 8 : \
27    (x <= 512) ? 9 : \
28    (x <= 1024) ? 10 : 0 //go unrealistically high to cover the basis
29 
30 `define BAND_IN_SIZE      (DATA_WIDTH*(2*MAT_SIZE-1))
31 `define GEN_IN_SIZE       (DATA_WIDTH*MAT_SIZE)
32 `define BAND_OUT_SIZE     (DATA_WIDTH*(2*(2*MAT_SIZE-1)-1))
```

```
`define GEN_OUT_SIZE      (DATA_WIDTH*MAT_SIZE)
`define ARRAY_SIZE        (2*MAT_SIZE-1)

`define START_BAND_VALIDITY ((2*MAT_SIZE-1)*4-1-3*band_type)
`define BAND_DURATION (3*size)

(* use_dsp = "simd" *)
(* dont_touch = "true" *)
module mmc #(parameter MAT_SIZE=16, DATA_WIDTH=8)
      (input                        ap_clk, ap_rst, ap_ce, ap_start, ap_continue,
      output                       ap_idle, ap_done, ap_ready,
      input [`GEN_IN_SIZE-1:0]     a_gen,
      input                        a_gen_empty_n,
      output  reg                     a_gen_re,
      input [`GEN_IN_SIZE-1:0]     b_gen,
      input                        b_gen_empty_n,
      output  reg                     b_gen_re,
      input [`BAND_IN_SIZE-1:0]    a_band,
      input                        a_band_empty_n,
      output  reg                     a_band_re,
      input [`BAND_IN_SIZE-1:0]    b_band,
      input                        b_band_empty_n,
      output   reg                    b_band_re,
      input                        opmode,
      input                        opmode_empty_n,
      output reg                      opmode_re,
      input [31:0]                 size,
      input                        size_empty_n,
      output  reg                     size_re,
      input [31:0]                 band_type,
      input                        band_type_empty_n,
      output   reg                    band_type_re,
      output [`BAND_OUT_SIZE-1:0] c_band,
      input                        c_band_full_n,
      output   reg                    c_band_we,
      output [`GEN_OUT_SIZE-1:0]   c_gen,
      input                        c_gen_full_n,
      output  reg                     c_gen_we);

/*---------declare states and substates as local parameters-----*/

localparam [4:0] s_reset        = 5'd0;
localparam [4:0] s_write        = 5'd1;
localparam [4:0] s_done         = 5'd2;
localparam [4:0] s_idle         = 5'd3;
localparam [4:0] s_readparams   = 5'd4;
localparam [4:0] s_readinputs   = 5'd5;
localparam [4:0] s_gencompute   = 5'd6;
localparam [4:0] s_bandcompute  = 5'd7;
localparam [4:0] s_selectopmode  = 5'd8;

localparam [4:0] ss_0 = 5'd0;
localparam [4:0] ss_1 = 5'd1;
localparam [4:0] ss_2 = 5'd2;
localparam [4:0] ss_3 = 5'd3;
localparam [4:0] ss_4 = 5'd4;
localparam [4:0] ss_5 = 5'd5;
```

```
90
91  localparam mode_gen = 1'b0;
92  localparam mode_band = 1'b1;
93
94  /*FSM defined by states and a counter*/
95
96    reg [4:0] current_state;
97    reg [4:0] next_state;
98
99    reg [31:0] counter;
100   reg [31:0] next_counter;
101
102   reg [1:0] tricounter;
103   reg [1:0] next_tricounter;
104
105   reg [`CLOG2(MAT_SIZE)-1:0] gen_counter;
106   reg [`CLOG2(MAT_SIZE)-1:0] next_gen_counter;
107
108
109 /* define speedup parameters */
110
111   reg [31:0] start_band_validity;
112   reg [31:0] next_start_band_validity;
113
114   reg [31:0] counter_for_end_of_gen_op;
115   reg [31:0] next_counter_for_end_of_gen_op;
116
117   reg [31:0] band_duration;
118   reg [31:0] next_band_duration;
119
120   reg must_read;
121   reg must_write;
122
123 /*--------declare necessary stuff--------*/
124   wire [4:0] gen_substate = (must_read && must_write)? ss_1 : (must_read)? ss_0 : (must_write)? ss_2:
    ↪  ss_3;
125   wire end_of_gen_op = (counter == counter_for_end_of_gen_op);
126
127   wire band_output_valid = (counter >= start_band_validity);
128   wire end_of_band_op = counter == (start_band_validity + band_duration);
129
130
131
132   wire [4:0] band_substate = (must_read && must_write)? ss_1 : (must_read)? ss_0 : (must_write)? ss_2 :
    ↪  ss_3;
133
134 /*assign bb signals*/
135 assign ap_ready = 1;
136 assign ap_idle = ap_start ? 0 : (current_state == s_idle || current_state == s_reset);
137 assign ap_done = current_state == s_done;//(current_state == s_done || current_state == s_idle);
138
139 /*signals related to array and array itself*/
140   reg array_en_reg;
141   reg inputs_zero;
142
143   wire [31:0] counter_to_array = opmode ? tricounter : gen_counter;
144   wire array_en = ap_ce ? array_en_reg : 0 ;
```

```
  wire array_rst = ap_rst ? 1 : current_state == s_reset;

  wire [`GEN_IN_SIZE-1:0] a_gen_in = inputs_zero ? 0 : a_gen;
  wire [`GEN_IN_SIZE-1:0] b_gen_in = inputs_zero ? 0 : b_gen;

  wire [`BAND_IN_SIZE-1:0] a_band_in = inputs_zero ? 0 : a_band;
  wire [`BAND_IN_SIZE-1:0] b_band_in = inputs_zero ? 0 : b_band;


unified_array #(.MAT_SIZE(MAT_SIZE), .DATA_WIDTH(DATA_WIDTH)) array0
    (
      .clk(ap_clk),
      .reset(array_rst),
      .counter(counter_to_array),//last two bits are used as tricounter when in band operation
      .array_en(array_en),
      .opmode(opmode),        //0 for generic, 1 for band
      .A_band_flattened(a_band_in),
      .B_band_flattened(b_band_in),
      .C_band_flattened(c_band),
      .A_generic_flattened(a_gen_in),
      .B_generic_flattened(b_gen_in),
      .C_generic_flattened(c_gen)
    );



/*---------------FSM STARTS HERE-------------*/
/*-----CLOCKED PART-------------------------*/
always @ (posedge ap_clk)
if (ap_rst) begin
  current_state <= s_reset;
  counter <= 0;
  tricounter <= 0;
  gen_counter <= 0;

  band_duration <= 1;
  start_band_validity <= 1;
  counter_for_end_of_gen_op <= 1;
end
else if (ap_ce) begin
  current_state <= next_state;
  counter <= next_counter;
  tricounter <= next_tricounter;
  gen_counter <= next_gen_counter;

  band_duration <= next_band_duration;
  start_band_validity <= next_start_band_validity;
  counter_for_end_of_gen_op <= next_counter_for_end_of_gen_op;
end


always @(*)
begin
  a_band_re     = 0;
  b_band_re     = 0;
  a_gen_re      = 0;
  b_gen_re      = 0;
```

```
202   opmode_re     = 0;
203   size_re       = 0;
204   band_type_re  = 0;
205   c_band_we     = 0;
206   c_gen_we      = 0;
207
208   array_en_reg  = 0;
209   inputs_zero   = 0;
210
211   next_state = s_reset;
212   next_counter = counter;
213   next_tricounter = tricounter;
214   next_gen_counter = gen_counter;
215
216   must_read = 0;
217   must_write = 0;
218
219   next_start_band_validity = start_band_validity;
220   next_band_duration = band_duration;
221   next_counter_for_end_of_gen_op = counter_for_end_of_gen_op;
222
223   case(current_state)
224     s_reset: begin
225        // reset all the output registers in the reset state
226       a_band_re     = 0;
227       b_band_re     = 0;
228       a_gen_re      = 0;
229       b_gen_re      = 0;
230       opmode_re     = 0;
231       size_re       = 0;
232       band_type_re  = 0;
233       c_band_we     = 0;
234       c_gen_we      = 0;
235
236       array_en_reg  = 0;
237       inputs_zero   = 0;
238
239       next_state = s_reset;
240       next_counter = 0;
241       next_tricounter = 0;
242       next_gen_counter = 0;
243
244       if (ap_start) begin
245         next_state = s_readparams;
246       end
247     end
248
249     s_readparams: begin
250       opmode_re     = 0;
251       size_re       = 0;
252       band_type_re  = 0;
253       next_state = s_readparams;
254       if (opmode_empty_n && band_type_empty_n && size_empty_n) begin
255         next_state = s_selectopmode;
256       end
257     end
258
```

```
259     s_selectopmode: begin
260
261       next_start_band_validity = `START_BAND_VALIDITY; // make the actual calculation here
262       next_band_duration = `BAND_DURATION;
263
264       next_counter_for_end_of_gen_op = size*band_type + (`ARRAY_SIZE);
265
266       case(opmode)
267         mode_gen: begin
268           next_state = s_gencompute;
269           next_counter = 0;
270           next_gen_counter = 0;
271         end
272
273         mode_band: begin
274           next_state = s_bandcompute;
275           next_counter = 0;
276           next_tricounter = 0;
277         end
278
279         default: begin
280           next_state = s_done;
281         end
282       endcase
283     end
284     /*
285 ----------------------------------------
286 ----------------------------------------
287 ----------------------------------------
288 --------------GENERIC-------------------
289 ----------------------------------------
290 ----------------------------------------
291 ----------------------------------------
292 ----------------------------------------
293     */
294
295     s_gencompute: begin
296       // go into this state with counter = 0 and substate = 0;
297       //cleanup the re signals from previous state
298       //for generic operation, reading and writing are separate in time,
299       //thus we do not need to check if the inputs and the outputs are both free to enable the register
300
301     //-------deal with inputs ----------/
302       a_gen_re  = 0;
303       b_gen_re  = 0;
304       next_counter  =  counter;
305       next_gen_counter = gen_counter;
306       array_en_reg = 0;
307       inputs_zero = 0;
308       c_gen_we = 0;
309
310       if(counter < counter_for_end_of_gen_op - `ARRAY_SIZE) begin
311         must_read = 1;
312       end
313       if(counter >= `ARRAY_SIZE ) begin
314         must_write = 1;
315       end
```

```
316
317       case(gen_substate)
318         ss_0: begin //must_read
319           if (a_gen_empty_n && b_gen_empty_n) begin // overwrite previous statements in necessary
320             a_gen_re  = 1;
321             b_gen_re  = 1;
322             next_counter = counter + 1;
323             next_gen_counter = (gen_counter == MAT_SIZE-1) ? 0 : gen_counter + 1;
324             array_en_reg = 1;
325           end
326         end
327
328         ss_1: begin //must read and write
329           if (a_gen_empty_n && b_gen_empty_n && c_gen_full_n) begin
330           a_gen_re = 1;
331           b_gen_re = 1;
332           c_gen_we = 1;
333           next_counter = counter + 1;
334           next_gen_counter = (gen_counter == MAT_SIZE-1) ? 0 : gen_counter + 1;
335           array_en_reg = 1; //corresponding 0 is up top
336           end
337         end
338
339         ss_2: begin //must_write
340           if (c_gen_full_n) begin
341             c_gen_we = 1;
342             next_counter = counter + 1;
343             next_gen_counter = (gen_counter == MAT_SIZE-1) ? 0 : gen_counter + 1;
344             array_en_reg = 1;
345           end
346         end
347
348         ss_3: begin // deadcycling
349             c_gen_we=0;
350             a_gen_re=0;
351             b_gen_re=0;
352             next_counter = counter + 1;
353             next_gen_counter = (gen_counter == MAT_SIZE-1) ? 0 : gen_counter + 1;
354             inputs_zero=1;
355             array_en_reg=1;
356         end
357       endcase
358
359       //------mark end of operation-------/
360       next_state = current_state;
361       if(end_of_gen_op) begin
362         next_state = s_done;
363         next_counter = 0; //reset the counter, you never know
364         next_gen_counter = 0;
365       end
366     end
367     /*
368 ----------------------------------------
369 ----------------------------------------
370 ----------------------------------------
371 -----------------BAND-------------------
372 ----------------------------------------
```

```
373 ----------------------------------------
374 ----------------------------------------
375 ----------------------------------------
376     */
377     s_bandcompute: begin // reachable now. The design has to be smart.
378       inputs_zero = 0;
379       array_en_reg = 0;
380       next_counter = counter;
381       next_tricounter = tricounter;
382       must_read = 0;
383       must_write = 0;
384 
385       a_band_re = 0;
386       b_band_re = 0;
387       c_band_we = 0;
388 
389       if (counter >= band_duration) begin
390         inputs_zero = 1;
391       end
392       if(tricounter == 2'b00 && counter < band_duration) begin // if the counter is below the band
    ↪  duration then we must read
393         must_read = 1;
394       end
395 
396       if (counter >= start_band_validity && tricounter == 2'b00) begin
397         must_write = 1;
398       end // then we must write
399 
400 
401       //put a case statement here
402       case(band_substate)
403         ss_0: begin// must only read
404           if(a_band_empty_n && b_band_empty_n) begin //read and cycle
405             a_band_re = 1;
406             b_band_re = 1;
407             next_counter = counter + 1;
408             next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
409             array_en_reg = 1;
410           end //else jsut wait
411         end
412 
413         ss_1: begin// must read and write
414           if(a_band_empty_n && b_band_empty_n && c_band_full_n) begin
415             a_band_re = 1;
416             b_band_re = 1;
417             c_band_we = 1;
418             next_counter = counter + 1;
419             next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
420             array_en_reg = 1;
421           end // else just wait
422         end
423 
424         ss_2: begin // must only write
425           if(c_band_full_n) begin
426             c_band_we = 1;
427             next_counter = counter + 1;
428             next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
```

```
429             array_en_reg = 1;
430           end
431         end
432
433         ss_3: begin //must do nothing but still cycle
434           next_counter = counter + 1;
435           next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
436           array_en_reg = 1;
437         end
438
439         default: begin // must do nothing, same as ss_3
440           next_counter = counter + 1;
441           next_tricounter = (tricounter == 2'b10 )? 0 : tricounter + 1;
442           array_en_reg = 1;
443         end
444       endcase
445
446
447       next_state = s_bandcompute;
448       if (end_of_band_op) next_state = s_done;
449     end
450
451     s_done: begin // put all the output we to 0;
452       opmode_re     = 1;
453       size_re       = 1;
454       band_type_re  = 1;
455       c_gen_we = 0;
456       c_band_we = 0;
457       next_state = s_idle;
458     end
459
460     s_idle: begin
461       opmode_re     = 0;
462       size_re       = 0;
463       band_type_re  = 0;
464       next_state = s_reset;
465     end
466
467     default: begin
468       a_band_re     = 0;
469       b_band_re     = 0;
470       a_gen_re      = 0;
471       b_gen_re      = 0;
472       opmode_re     = 0;
473       size_re       = 0;
474       band_type_re  = 0;
475       c_band_we     = 0;
476       c_gen_we      = 0;
477
478       array_en_reg  = 0;
479       inputs_zero   = 0;
480
481       next_state = s_reset;
482       next_counter = 0;
483       next_tricounter = 0;
484
485       must_read = 0;
```

```
      must_write = 0;
    end
  endcase
end


endmodule
```

Listing 63: Host_gen.cpp

```
/*******************************************************************************
Vendor: Xilinx
Associated Filename: vadd.cpp
Purpose: VITIS vector addition

*******************************************************************************
Copyright (C) 2019 XILINX, Inc.

This file contains confidential and proprietary information of Xilinx, Inc. and
is protected under U.S. and international copyright and other intellectual
property laws.

DISCLAIMER
This disclaimer is not a license and does not grant any rights to the materials
distributed herewith. Except as otherwise provided in a valid license issued to
you by Xilinx, and to the maximum extent permitted by applicable law:
(1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND WITH ALL FAULTS, AND XILINX
HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY,
INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR
FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether
in contract or tort, including negligence, or under any other theory of
liability) for any loss or damage of any kind or nature related to, arising under
or in connection with these materials, including for any direct, or any indirect,
special, incidental, or consequential loss or damage (including loss of data,
profits, goodwill, or any type of loss or damage suffered as a result of any
action brought by a third party) even if such damage or loss was reasonably
foreseeable or Xilinx had been advised of the possibility of the same.

CRITICAL APPLICATIONS
Xilinx products are not designed or intended to be fail-safe, or for use in any
application requiring fail-safe performance, such as life-support or safety
devices or systems, Class III medical devices, nuclear facilities, applications
related to the deployment of airbags, or any other applications that could lead
to death, personal injury, or severe property or environmental damage
(individually and collectively, "Critical Applications"). Customer assumes the
sole risk and liability of any use of Xilinx products in Critical Applications,
subject only to applicable laws and regulations governing limitations on product
liability.

THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS PART OF THIS FILE AT
ALL TIMES.

*******************************************************************************/
```

```
#define OCL_CHECK(error, call)                                                                   \
    call;                                                                                        \
    if (error != CL_SUCCESS) {                                                                   \
        printf("%s:%d Error calling " #call ", error code is: %d\n", __FILE__, __LINE__, error); \
        exit(EXIT_FAILURE);                                                                      \
    }

#include <stdlib.h>
#include <fstream>
#include <iostream>
#include "Host.h"
#include "../src/HLS_wrapper.h"
//#include "xcl2.hpp"

//Operation specs
static const unsigned int size = MAT_SIZE; // only full operations for now
unsigned int opcount = 10;
bool opmode = OPMODE_GEN; // choose the opmode here
size_t gen_matrix_size_in_bytes = opcount*size*size*DATA_WIDTH/8; // full size of the matrix

static const std::string error_message =
    "Error: Result mismatch:\n"
    "i = %d CPU result = %d Device result = %d\n";

//static const std::string print_results ="C element: %5d\n";

int main(int argc, char* argv[]) {

    //TARGET_DEVICE macro needs to be passed from gcc command line
    if(argc != 3) {//this line used to be !=2
                std::cout << "Usage: " << argv[0] <<" <xclbin>" << std::endl;
                return EXIT_FAILURE;
        }

    std::string xclbinFilename = argv[1];

    // Compute the size of array in bytes
    //size_t size_in_bytes = DATA_SIZE * sizeof(int);

    // Creates a vector of DATA_SIZE elements with an initial value of 10 and 32
    // using customized allocator for getting buffer alignment to 4k boundary

    std::vector<cl::Device> devices;
    cl::Device device;
    cl_int err;
    cl::Context context;
    cl::CommandQueue q;
    cl::Kernel krnl_matrix_mult;
    cl::Program program;
    std::vector<cl::Platform> platforms;
    bool found_device = false;

    //traversing all Platforms To find Xilinx Platform and targeted
    //Device in Xilinx Platform
    cl::Platform::get(&platforms);
    for(size_t i = 0; (i < platforms.size() ) & (found_device == false) ;i++){
        cl::Platform platform = platforms[i];
```

```
        std::string platformName = platform.getInfo<CL_PLATFORM_NAME>();
        if ( platformName == "Xilinx"){
            devices.clear();
            platform.getDevices(CL_DEVICE_TYPE_ACCELERATOR, &devices);
            if (devices.size()){
                    device = devices[0];
                    found_device = true;
                    break;
            }
        }
    }
    if (found_device == false){
       std::cout << "Error: Unable to find Target Device "
           << device.getInfo<CL_DEVICE_NAME>() << std::endl;
       return EXIT_FAILURE;
    }

    // Creating Context and Command Queue for selected device
    OCL_CHECK(err, context = cl::Context(device, NULL, NULL, NULL, &err));
    OCL_CHECK(err, q = cl::CommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &err));

    std::cout << "INFO: Reading " << xclbinFilename << std::endl;
    FILE* fp;
    if ((fp = fopen(xclbinFilename.c_str(), "r")) == nullptr) {
        printf("ERROR: %s xclbin not available please build\n", xclbinFilename.c_str());
        exit(EXIT_FAILURE);
    }

    // Load xclbin
    std::cout << "Loading: '" << xclbinFilename << "'\n";
    std::ifstream bin_file(xclbinFilename, std::ifstream::binary);
    bin_file.seekg (0, bin_file.end);
    unsigned nb = bin_file.tellg();
    std::cout << "number of program bytes: " << nb << std::endl;
    bin_file.seekg (0, bin_file.beg);
    char *buf = new char [nb];
    bin_file.read(buf, nb);


    std::cout << "Creating Program from binary file..." << std::endl;
    // Creating Program from Binary File
    cl::Program::Binaries bins;
    //std::cout << "CP1.1" << std::endl;
    bins.push_back({buf,nb});
    //std::cout << "CP1.2" << std::endl;
    devices.resize(1);
    std::cout << "OK" << std::endl << "Programming Device...";
    OCL_CHECK(err, program = cl::Program(context, devices, bins, NULL, &err));

    std::cout << "OK" << std::endl << "Calling Kernel...";
    // This call will get the kernel object from program. A kernel is an
    // OpenCL function that is executed on the FPGA.
    OCL_CHECK(err, krnl_matrix_mult = cl::Kernel(program,"hls_wrapper", &err));
    std::cout <<  "OK" << std::endl << "Allocating memory...";
    // These commands will allocate memory on the Device. The cl::Buffer objects can
    // be used to reference the memory locations on the device.
    //OCL_CHECK(err, cl::Buffer buffer_a(context, CL_MEM_READ_ONLY, size_in_bytes, NULL, &err));
```

```
158
159     OCL_CHECK(err, cl::Buffer buffer_A(context, CL_MEM_READ_ONLY, gen_matrix_size_in_bytes, NULL, &err));
160     OCL_CHECK(err, cl::Buffer buffer_B(context, CL_MEM_READ_ONLY, gen_matrix_size_in_bytes, NULL, &err));
161     //OCL_CHECK(err, cl::Buffer buffer_b(context, CL_MEM_READ_ONLY, size_in_bytes, NULL, &err));
162     OCL_CHECK(err, cl::Buffer buffer_C(context, CL_MEM_WRITE_ONLY, gen_matrix_size_in_bytes, NULL,
    ↪  &err));
163     //OCL_CHECK(err, cl::Buffer buffer_result(context, CL_MEM_WRITE_ONLY, size_in_bytes, NULL, &err));
164     std::cout <<  "OK" << std::endl << "Setting Kernel Arguments...";
165     //set the kernel Arguments
166     //int narg=2;
167
168
169     OCL_CHECK(err, err = krnl_matrix_mult.setArg(0,buffer_A));
170     OCL_CHECK(err, err = krnl_matrix_mult.setArg(1,buffer_B));
171     OCL_CHECK(err, err = krnl_matrix_mult.setArg(2,buffer_C));
172     OCL_CHECK(err, err = krnl_matrix_mult.setArg(3,size));
173     OCL_CHECK(err, err = krnl_matrix_mult.setArg(4,opmode));
174     OCL_CHECK(err, err = krnl_matrix_mult.setArg(5,opcount));
175     std::cout << "OK" << std::endl << ((opmode==OPMODE_GEN)?
176         "Running Generic Operation with operation size " : "Running Band Operation with operation size ")
177          << size << " and data width " << DATA_WIDTH <<std::endl;
178
179     std::cout << "Mapping OpenCL buffer to data pointers..." << std::flush;
180
181
182     data_t *ptr_A;
183     data_t *ptr_B;
184     data_t *ptr_C;
185
186
187     OCL_CHECK(err, ptr_A = (data_t*)q.enqueueMapBuffer (buffer_A , CL_TRUE , CL_MAP_WRITE , 0,
    ↪  gen_matrix_size_in_bytes, NULL, NULL, &err));
188     OCL_CHECK(err, ptr_B = (data_t*)q.enqueueMapBuffer (buffer_B , CL_TRUE , CL_MAP_WRITE , 0,
    ↪  gen_matrix_size_in_bytes, NULL, NULL, &err));
189     OCL_CHECK(err, ptr_C = (data_t*)q.enqueueMapBuffer (buffer_C , CL_TRUE , CL_MAP_READ  , 0,
    ↪  gen_matrix_size_in_bytes, NULL, NULL, &err));
190     std::cout << "OK" << std::endl << "Preparing the input data..." << std::flush;
191
192
193     //fill in the matrices with relevant numbers here.
194     for(unsigned int j=0; j < opcount; j++)
195         for(unsigned int k=0;k<size*size;k++){
196             ptr_A[j*size*size + k] = k;
197             ptr_B[j*size*size + k] = k;
198         }
199
200
201     static const std::string print_results = "%4d";
202
203     unsigned int loopcount = 1;
204
205     std::chrono::duration<double> full_time(0);
206     std::chrono::duration<double> kernel_time(0);
207
208     std::cout << "OK" << std::endl << "Starting " << loopcount << " Operations" << std::flush;
209
210
```

```
211     auto kernel_start = std::chrono::high_resolution_clock::now();
212 
213     for(unsigned int i=0;i<loopcount;i++){
214         //std::cout << "OK" << std::endl << "Migrating data to the kernel space..." << std::flush;
215 
216         // Data will be migrated to kernel space
217         OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_A},0/* 0 means from host*/));
218         OCL_CHECK(err, q.finish());
219         OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_B},0/* 0 means from host*/));
220         OCL_CHECK(err, q.finish());
221         //std::cout << "OK" << std::endl << "Launching Kernel..." << std::endl << std::flush;
222         //Launch the Kernel
223         OCL_CHECK(err, err = q.enqueueTask(krnl_matrix_mult));
224         OCL_CHECK(err, q.finish());
225         //std::cout << "EXECUTION FINISHED" << std::endl << "Migrating datafrom the kernel space..."<<
    ↪ std::flush;
226         // The result of the previous kernel execution will need to be retrieved in
227         // order to view the results. This call will transfer the data from FPGA to
228         // source_results vector
229         OCL_CHECK(err, q.enqueueMigrateMemObjects({buffer_C},CL_MIGRATE_MEM_OBJECT_HOST));
230         OCL_CHECK(err, q.finish());
231         //std::cout << "OK" << std::endl << std::flush;
232         //std::cout << std::endl;
233     }
234 
235     std::cout << "EXECUTION FINISHED" << std::endl << std::flush;
236 
237     auto kernel_end = std::chrono::high_resolution_clock::now();
238 
239     full_time = std::chrono::duration<double>(kernel_end - kernel_start);
240 
241     kernel_time = full_time / (double) loopcount;
242 
243     std::cout << "time per kernel execution:" << kernel_time.count() << "s" << std::endl << std::flush;
244 
245 
246 
247     //Verify the result
248     //int match = 0;
249     for(unsigned int k = 0; k< opcount ; k++) {
250         std::cout << std::endl;
251         std::cout << "C=" << std::endl;
252         //simple generic printer
253         for (unsigned int i = 0; i < size; i++)
254             for (unsigned int j = 0; j < size; j++){
255                 printf(print_results.c_str(), (int) ptr_C[k*size*size + i*size + j]);
256                 cout <<" \n"[j == size-1];
257             }
258 
259     }
260 
261     std::cout << "Cleaning up...";
262     //OCL_CHECK(err, err = q.enqueueUnmapMemObject(buffer_a , ptr_a));
263     OCL_CHECK(err, err = q.enqueueUnmapMemObject(buffer_A , ptr_A));
264     OCL_CHECK(err, err = q.enqueueUnmapMemObject(buffer_B , ptr_B));
265     OCL_CHECK(err, err = q.enqueueUnmapMemObject(buffer_C , ptr_C));
266     //OCL_CHECK(err, err = q.enqueueUnmapMemObject(buffer_result , ptr_result));
```

```
267     OCL_CHECK(err, err = q.finish());
268 
269 
270     std::cout << "OK" << std::endl << "KERNEL_DONE" << std::endl;
271     //std::cout << "TEST " << (match ? "FAILED" : "PASSED") << std::endl;
272     return 0;//(match ? EXIT_FAILURE :  EXIT_SUCCESS);
273 
274 }
```

Listing 64: hls_wrapper_FINAL.cpp

```
1  /*
2   * Copyright 2021 Xilinx, Inc.
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *   http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13  * See the License for the specific language governing permissions and
14  * limitations under the License.
15  */
16 
17 #include "HLS_wrapper.h"
18 
19 //------------------------------------------------------
20 #ifndef __SYNTHESIS__ // dont care for hw_emu
21 void mmc(
22     hls::stream<band_in_t >& A_band_stream, //this will be used to tranfer a stream of data to the kernel
23     hls::stream<band_in_t >& B_band_stream,
24     hls::stream<gen_in_t >&  A_gen_stream,
25     hls::stream<gen_in_t >&  B_gen_stream,
26 
27     hls::stream<bool>& opmode_stream,
28     hls::stream<int>& size_stream,
29     hls::stream<int>& band_type_stream,
30 
31     hls::stream<band_out_t >& C_band_stream,
32         hls::stream<gen_out_t >&  C_gen_stream)
33 {
34 #pragma HLS inline off
35 
36     C_gen_stream << 1;
37     C_gen_stream << 1;
38     C_gen_stream << 1;
39     C_gen_stream << 1;
40 }
41 #endif
42 
```

```
#ifdef __SYNTHESIS__
void mmc(
    hls::stream<band_in_t >& A_band_stream, //this will be used to tranfer a stream of data to the kernel
    hls::stream<band_in_t >& B_band_stream,
    hls::stream<gen_in_t >&  A_gen_stream,
    hls::stream<gen_in_t >&  B_gen_stream,

    hls::stream<bool>& opmode_stream,
    hls::stream<int>& size_stream,
    hls::stream<int>& band_type_stream,

    hls::stream<band_out_t >& C_band_stream,
        hls::stream<gen_out_t >&  C_gen_stream);
#endif
//------------------------------------------------------------
/*-----------extra function bodies--------------*/
unsigned int get_c_rect_index(int i,
                          int j,
                          int width){
//calculate the rectangular indices:
    i++;
    j++;
    unsigned int Ir = (j>i)? i : j ;
    unsigned int Jr = width - i + j;
    Ir--;
    Jr--;

//calculate the index of the 1D arry and return it
    return Ir*(2*width-1) + Jr;
}
unsigned int get_c_i_index(int c_i_rect,
                           int c_j_rect,
                           int w){
//calculate the rectangular indices:
    c_i_rect++;
    c_j_rect++;
    unsigned int c_i = (c_j_rect>=w) ? c_i_rect : w + c_i_rect - c_j_rect ;
    c_i--;
    return c_i;
}

unsigned int get_c_j_index(int c_i_rect,
                           int c_j_rect,
                           int w){
//calculate the rectangular indices:
    c_i_rect++;
    c_j_rect++;
    unsigned int c_j = (c_j_rect>=w) ? c_j_rect - w + c_i_rect : c_i_rect ;
    c_j--;
    return c_j;
}

unsigned int get_a_index(int i,
                         int j,
                         int width,
                         int p){
//calculate the rectangular indices:
```

```
100     i++;
101     j++;
102     unsigned int Ir = j;
103     unsigned int Jr = p - j + i;
104 //calculate the index of the 1D arry and return it
105     Ir--;
106     Jr--;
107     return Ir*width + Jr;
108 }
109 bool is_in( int c_i_rect,
110             int c_j_rect,
111             int size,
112             int in_width,
113             int out_width)
114 {
115     int k = size-in_width;
116     if(c_i_rect <= k) return true;//for the first portion the criterion is simple
117     else{
118         if (c_j_rect > c_i_rect-k-1 && c_j_rect < out_width - (c_i_rect-k)) return true;
119         else return false;
120     }
121 }
122 unsigned int min(unsigned int a, unsigned int b){
123     return (a<b)? a : b;
124 }
125 void hls_mmc(
126     hls::stream<band_in_t >& A_band_stream,
127     hls::stream<band_in_t >& B_band_stream,
128     hls::stream<gen_in_t >&  A_gen_stream,
129     hls::stream<gen_in_t >&  B_gen_stream,
130 
131     hls::stream<bool>& opmode_stream,
132     hls::stream<int>& size_stream,
133     hls::stream<int>& band_type_stream,
134 
135     hls::stream<band_out_t >& C_band_stream,
136         hls::stream<gen_out_t >&  C_gen_stream){
137 
138 #pragma HLS dataflow
139 
140         bool opmode;
141         int band_type;
142         int size;
143 
144         size_stream >> size;
145         opmode_stream >> opmode;
146         band_type_stream >> band_type;
147 
148         int index;
149         unsigned int index_a;
150         unsigned int index_b;
151 
152         if(opmode==OPMODE_GEN){
153 /*-------------------------------------------------GENERIC
    ↪  OPERATION-------------------------------------------------------*/
154             //let's cut up the input lines of data and fill an array with the individual data
155 
```

```
156             const unsigned int in_array_size = MAT_SIZE*MAT_SIZE;
157             const unsigned int out_array_size = MAT_SIZE*MAT_SIZE;
158 
159             data_t a_data_array[in_array_size];
160             data_t b_data_array[in_array_size];
161             data_t c_data_array[out_array_size];
162 
163             for(unsigned int opcount = 0; opcount < band_type; opcount++){
164             #pragma HLS loop_tripcount max=1000
165                 for(unsigned int i=0; i < MAT_SIZE; i++){ //loop through the lines
166                     gen_in_t a_temp;
167                     gen_in_t b_temp;
168                     A_gen_stream >> a_temp;
169                     B_gen_stream >> b_temp;
170                     for(unsigned int j=0; j<MAT_SIZE; j++){//loop through each line and cut it up and
    ↪  fill the array
171                         index_a = i*MAT_SIZE + j;
172                         index_b = j*MAT_SIZE + i; // in order to match the rtl design
173                         a_data_array[index_a] = a_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
174                         b_data_array[index_b] = b_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
175                     }
176                 }
177 
178                 for(unsigned int i=0;i<MAT_SIZE;i++){
179                     for(unsigned int j=0;j<MAT_SIZE;j++){
180                         index = i*MAT_SIZE + j;
181                         c_data_array[index]=0; // start the accumulator at 0
182                         for(unsigned int k=0;k<MAT_SIZE;k++){
183                             index_a = i*MAT_SIZE + k;
184                             index_b = k*MAT_SIZE + j;
185                             c_data_array[index] += a_data_array[index_a]*b_data_array[index_b];
186                         }
187                     }
188                 }
189                 //put the data back into streams
190                 for(unsigned int i=0; i<MAT_SIZE; i++){ //loop through the lines
191                     gen_out_t c_temp;
192                     for(unsigned int j=0; j<MAT_SIZE; j++){//loop through each line and cut it up and
    ↪  fill the array
193                         index = i*MAT_SIZE + j;
194                         c_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH) = c_data_array[index];
195                     }
196                     C_gen_stream << c_temp;
197                 }
198             }
199 
200 
201         }else{
202 /*-------------------------------------------------BAND
    ↪  OPERATION-------------------------------------------------------------*/
203 
204         const int in_width = 2*MAT_SIZE-1;
205         const int out_width = 2*in_width-1;
206 
207         const unsigned int in_array_size = 2*in_width*in_width;
208         const unsigned int out_array_size = out_width;
209 
```

```
210         data_t a_data_array[in_array_size];//define arrays with sufficient size
211         data_t b_data_array[in_array_size];
212 
213         int pattern[in_array_size];
214 
215         data_t c_data_array[out_array_size];
216 
217 
218 
219         //simple generic printer
220 
221 
222         for(unsigned int i=0; i<2*in_width && i<size; i++){ //fill the circular buffer initially
223         #pragma HLS loop_tripcount max=1000
224             band_in_t a_temp;
225             band_in_t b_temp;
226             A_band_stream >> a_temp;
227             B_band_stream >> b_temp;
228             for(unsigned int j=0; j<in_width; j++){//loop through each line and cut it up and fill the
    ↪  array
229                 index = i*in_width + j;
230                 a_data_array[index] = a_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
231                 b_data_array[index] = b_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
232             }
233         }
234 
235             // now that the data is into an array we can easily make the computations with the standard
    ↪  3-loop technique
236             int index_a;
237             int index_b;
238             int index_c;
239 
240             int c_i;
241             int c_j;
242             int c_k;
243 
244             int overlap;
245             int dif;
246             int abs_dif;
247 
248             const int q = band_type + 1;
249             const int p = in_width - q + 1;
250             const int w = in_width;
251 
252             int circ_buffer_offset = 0;
253 
254         #ifndef __SYNTHESIS__
255             std::cout << "q= "<< q << " p= " << p << " w= " << in_width << std::endl;
256         #endif
257 
258             for(unsigned int c_i_rect = 0 ; c_i_rect < size ; c_i_rect++){
259             #pragma HLS loop_tripcount max=1000
260                 if(c_i_rect > in_width && c_i_rect <= size-in_width){
261                     band_in_t a_temp;
262                     band_in_t b_temp;
263 
264                     A_band_stream >> a_temp; // these lines are deadlocking in sw_emu
```

```
                    B_band_stream >> b_temp;
                    for(unsigned int j=0; j<in_width; j++){
                        index = circ_buffer_offset*in_width + j;
                        a_data_array[index] = a_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
                        b_data_array[index] = b_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH);
                    }
                    circ_buffer_offset = (circ_buffer_offset==2*in_width-1) ? 0 : circ_buffer_offset + 1;
                }
                for(unsigned int c_j_rect = 0 ; c_j_rect < out_width ; c_j_rect++){
                    index_c = c_j_rect;
                    c_data_array[index_c] = 0; //start off every output memory at 0
                    if(is_in(c_i_rect,c_j_rect,size,in_width,out_width)){
                        c_i = get_c_i_index(c_i_rect , c_j_rect , w);
                        c_j = get_c_j_index(c_i_rect , c_j_rect , w);

                        dif = c_i-c_j;
                        abs_dif = (dif < 0)? -dif : dif;
                        overlap = w - abs_dif;

                        for(unsigned int iter=0 ; iter < overlap ; iter++){
                        DO_PRAGMA(HLS loop_tripcount max=in_width)
                            c_k = (c_i>c_j) ? c_i + iter - (w-p) : c_j + iter - (w-p);
                            if(c_k>=0 && c_k<size){
                                index_a = get_a_index(c_i, c_k , in_width,p);
                                index_b = get_a_index(c_j, c_k , in_width,p);

                                c_data_array[index_c] += a_data_array[index_a % in_array_size] *
↪  b_data_array[index_b % in_array_size];
                            }
                        }
                    }
                }
                band_out_t c_temp;
                for(unsigned int j=0; j<out_width; j++){
                    c_temp.range(DATA_WIDTH*(j+1)-1,j*DATA_WIDTH) = c_data_array[j];
                }
                C_band_stream << c_temp;
            }
        }
    }

void read_data_optimised(
    BUS_TYPE* in_A, BUS_TYPE* in_B,
    const unsigned int size,
    const bool opmode,
    const unsigned int band_type,
    hls::stream<band_in_t >& A_band_line, //this will be used to tranfer a line of data to the kernel
    hls::stream<band_in_t >& B_band_line,
    hls::stream<gen_in_t >&  A_gen_line,
    hls::stream<gen_in_t >&  B_gen_line,
    hls::stream<int >& size_line,
    hls::stream<int >& band_type_line,
    hls::stream<bool >& opmode_line)
{
#pragma HLS INLINE OFF
#pragma HLS dataflow
    /*-------------optimise for 16x16 8 bit generic operation---------*/
```

```
321     BUS_TYPE chunk_A , chunk_B;
322     //put the parameters in their own fifos
323     size_line << size;
324     band_type_line << band_type;
325     opmode_line << opmode;
326 
327     unsigned int line_accu_bit=0;
328     unsigned int chunk_accu_bit;
329     unsigned int bits_to_be_pulled;
330 
331     if (opmode==OPMODE_GEN){
332         const unsigned int in_matrix_size_in_bits = band_type*size*size*DATA_WIDTH;
333         unsigned int bits_left = in_matrix_size_in_bits;
334         const unsigned int num_transfers = (in_matrix_size_in_bits % BUSWIDTH == 0) ?
335         in_matrix_size_in_bits /BUSWIDTH : in_matrix_size_in_bits /BUSWIDTH + 1; // I believe this is
    ↪  correct
336                 //if the modulo of the division is 0 then we can spare one transfer
337 
338         gen_in_t A_line_accu;
339         gen_in_t B_line_accu;
340 
341         for(unsigned int i=0;i<num_transfers;i++){
342         #pragma HLS loop_tripcount max=8000
343         #pragma HLS pipeline II=1
344             chunk_A = in_A[i];
345             chunk_B = in_B[i];
346             chunk_accu_bit = 0;
347 
348             while(chunk_accu_bit < BUSWIDTH && bits_left != 0){ //while the chunk still has data to pull
349             #pragma HLS loop_tripcount max=2
350             #pragma HLS pipeline II=1
351                 bits_to_be_pulled = min(BUSWIDTH-chunk_accu_bit,GEN_IN_SIZE-line_accu_bit); //determine
    ↪  if we are line-limited or chunk-limited
352 
353                 A_line_accu.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit) =
    ↪  chunk_A.range(chunk_accu_bit + bits_to_be_pulled - 1,chunk_accu_bit);
354                 B_line_accu.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit) =
    ↪  chunk_B.range(chunk_accu_bit + bits_to_be_pulled - 1,chunk_accu_bit);
355                 chunk_accu_bit += bits_to_be_pulled;
356                 line_accu_bit += bits_to_be_pulled;
357                 bits_left -= bits_to_be_pulled;
358                 if(line_accu_bit == GEN_IN_SIZE){ //dispatch it
359                     A_gen_line << A_line_accu;
360                     B_gen_line << B_line_accu;
361                     line_accu_bit = 0;
362                 }
363             }
364         }
365     }
366     else{ // OPMODE_BAND
367         const unsigned int width = (2*MAT_SIZE-1) ;
368         const unsigned int length = size;
369 
370         const unsigned int in_matrix_size_in_bits = width*length*DATA_WIDTH; // num elements * bytes per
    ↪  element // here = 512 bits: lucky
371         unsigned int bits_left = in_matrix_size_in_bits;
372         const unsigned int num_transfers = (in_matrix_size_in_bits % BUSWIDTH == 0) ?
```

```
373         in_matrix_size_in_bits /BUSWIDTH : in_matrix_size_in_bits /BUSWIDTH + 1; // I believe this is
    ↪  correct
374 
375         band_in_t A_line_accu;
376         band_in_t B_line_accu;
377 
378         for(unsigned int i=0;i<num_transfers;i++){
379         #pragma HLS pipeline II=1
380         #pragma HLS loop_tripcount max=967
381             chunk_A = in_A[i];
382             chunk_B = in_B[i];
383             chunk_accu_bit = 0;
384             while(chunk_accu_bit < BUSWIDTH && bits_left != 0){ //while the chunk still has data to pull
385             #pragma HLS loop_tripcount max=9
386             #pragma HLS pipeline II=1
387                 bits_to_be_pulled = min(BUSWIDTH-chunk_accu_bit,BAND_IN_SIZE-line_accu_bit);//determine
    ↪  if we are line-limited or chunk-limited
388 
389                 A_line_accu.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit) =
    ↪  chunk_A.range(chunk_accu_bit + bits_to_be_pulled - 1,chunk_accu_bit);
390                 B_line_accu.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit) =
    ↪  chunk_B.range(chunk_accu_bit + bits_to_be_pulled - 1,chunk_accu_bit);
391                 chunk_accu_bit += bits_to_be_pulled;
392                 line_accu_bit += bits_to_be_pulled;
393                 bits_left -= bits_to_be_pulled;
394                 if(line_accu_bit == BAND_IN_SIZE){ //dispatch it
395                     A_band_line << A_line_accu;
396                     B_band_line << B_line_accu;
397                     line_accu_bit = 0;
398                 }
399             }
400         }
401     }
402 }
403 
404 void write_data_optimised(
405         BUS_TYPE* out_C,
406         const unsigned int size,
407         const bool opmode,
408         const unsigned int band_type,
409         hls::stream<band_out_t >& C_band_line,
410         hls::stream<gen_out_t >&  C_gen_line)
411 {
412 #pragma HLS INLINE OFF
413 #pragma HLS dataflow
414 
415     BUS_TYPE chunk_C;
416 
417     unsigned int line_accu_bit;
418     unsigned int chunk_accu_bit=0;
419     unsigned int bits_to_be_pulled;
420     unsigned int chunk_counter = 0;
421 
422     if (opmode==OPMODE_GEN){
423         gen_out_t Temp_C;
424 
425         unsigned int in_matrix_size_in_bits = band_type*size*size*DATA_WIDTH;
```

```
426         unsigned int bits_left = in_matrix_size_in_bits;
427         const unsigned int num_transfers = (in_matrix_size_in_bits % BUSWIDTH == 0) ?
428             in_matrix_size_in_bits /BUSWIDTH : in_matrix_size_in_bits /BUSWIDTH + 1;
429 
430         for(unsigned int i=0; i<band_type*size; i++){ // for each line, work until the line is empty
431         #pragma HLS loop_tripcount max=16000
432         #pragma HLS pipeline II=1
433             C_gen_line >> Temp_C;
434             line_accu_bit = 0;
435 
436             while(line_accu_bit < GEN_OUT_SIZE){ //while there is still data in the line to dump into
    ↪  chunk
437             #pragma HLS loop_tripcount max=1
438             #pragma HLS pipeline II=1
439                 bits_to_be_pulled = min(BUSWIDTH-chunk_accu_bit,GEN_OUT_SIZE-line_accu_bit);//determine
    ↪  how many bits we can pull
440                 #ifndef __SYNTHESIS__
441                     std::cout << "line_accu_bit: " << line_accu_bit << std::endl << std::flush;
442                     std::cout << "chunk_accu_bit: " << chunk_accu_bit << std::endl << std::flush;
443                 #endif
444                 chunk_C.range(chunk_accu_bit + bits_to_be_pulled - 1, chunk_accu_bit)=
    ↪  Temp_C.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit);
445                 chunk_accu_bit += bits_to_be_pulled;
446                 line_accu_bit += bits_to_be_pulled;
447                 bits_left -= bits_to_be_pulled;
448                 if(chunk_accu_bit == BUSWIDTH || bits_left == 0){
449                     out_C[chunk_counter++] = chunk_C;
450                     chunk_accu_bit = 0;
451                 }
452             }
453         }
454     }
455 
456 
457     else if(opmode==OPMODE_BAND){
458         band_out_t Temp_C;
459 
460         const unsigned int width = 2*(2*MAT_SIZE-1)-1;
461         const unsigned int length = size;
462 
463         unsigned int in_matrix_size_in_bits = width*length*DATA_WIDTH;
464         unsigned int bits_left = in_matrix_size_in_bits;
465         const unsigned int num_transfers = (in_matrix_size_in_bits % BUSWIDTH == 0) ?
466             in_matrix_size_in_bits /BUSWIDTH : in_matrix_size_in_bits /BUSWIDTH + 1;
467 
468         for(unsigned int i=0; i<length; i++){ // for each line, work until the line is empty
469         #pragma HLS loop_tripcount max=1000
470         #pragma HLS pipeline II=1
471             C_band_line >> Temp_C;
472             line_accu_bit = 0;
473 
474             while(line_accu_bit < BAND_OUT_SIZE){ //while there is still data in the line to dump into
    ↪  chunk
475             #pragma HLS loop_tripcount max=2
476             #pragma HLS pipeline II=1
477                 bits_to_be_pulled = min(BUSWIDTH-chunk_accu_bit,BAND_OUT_SIZE-line_accu_bit);//determine
    ↪  how many bits we can pull
```

```
                chunk_C.range(chunk_accu_bit + bits_to_be_pulled - 1, chunk_accu_bit)=
↪  Temp_C.range(line_accu_bit + bits_to_be_pulled - 1, line_accu_bit);
                chunk_accu_bit += bits_to_be_pulled;
                line_accu_bit += bits_to_be_pulled;
                bits_left -= bits_to_be_pulled;
                if(chunk_accu_bit == BUSWIDTH || bits_left == 0){//if the chunk is full or there's no
↪  more data to pull
                    out_C[chunk_counter++] = chunk_C;
                    chunk_accu_bit = 0;
                }
            }
        }
    }
}


void hls_wrapper(
    BUS_TYPE* in_A,
    BUS_TYPE* in_B,
    BUS_TYPE* out_C,
    const unsigned int size,
    const bool opmode,
    const unsigned int band_type) {

//how to set this one up ?
    #pragma HLS INTERFACE m_axi port=in_A offset=slave bundle=gmem0
    #pragma HLS INTERFACE m_axi port=in_B offset=slave bundle=gmem1
    #pragma HLS INTERFACE m_axi port=out_C offset=slave bundle=gmem2

    #pragma HLS INTERFACE s_axilite port=in_A bundle=control
    #pragma HLS INTERFACE s_axilite port=in_B bundle=control
    #pragma HLS INTERFACE s_axilite port=out_C bundle=control

    #pragma HLS INTERFACE s_axilite port=size bundle=control
    #pragma HLS INTERFACE s_axilite port=opmode bundle=control
    #pragma HLS INTERFACE s_axilite port=band_type bundle=control

    #pragma HLS INTERFACE s_axilite port=return bundle=control

    band_out_t C_band_fake;
    gen_out_t  C_gen_fake;


#pragma HLS dataflow

    hls::stream<gen_in_t > A_gen_stream("readAGen");
    hls::stream<gen_in_t > B_gen_stream("readBGen");
    hls::stream<gen_out_t > C_gen_stream("writeCgen");

    hls::stream<band_in_t > A_band_stream("readABand");
    hls::stream<band_in_t > B_band_stream("readBBand");
    hls::stream<band_out_t > C_band_stream("writeCBand");

    hls::stream<int> size_stream("sizeStream");
    hls::stream<int> band_type_stream("band_typeStream");
    hls::stream<bool> opmode_stream("opmodeStream");

```

```
533
534 #define FIFODEPTH 64
535 DO_PRAGMA(HLS STREAM variable=A_gen_stream depth=FIFODEPTH)
536 DO_PRAGMA(HLS STREAM variable=B_gen_stream depth=FIFODEPTH)
537 DO_PRAGMA(HLS STREAM variable=C_gen_stream depth=FIFODEPTH)
538
539 DO_PRAGMA(HLS STREAM variable=A_band_stream depth=FIFODEPTH)
540 DO_PRAGMA(HLS STREAM variable=B_band_stream depth=FIFODEPTH)
541 DO_PRAGMA(HLS STREAM variable=C_band_stream depth=FIFODEPTH)
542
543 //these fifos are used for simple parameter transfer
544 DO_PRAGMA(HLS STREAM variable=size_stream depth=1)
545 DO_PRAGMA(HLS STREAM variable=band_type_stream depth=1)
546 DO_PRAGMA(HLS STREAM variable=opmode_stream depth=1)
547
548
549 #if OPTIMISED == 1
550     read_data_optimised(
551 #else
552     read_data(
553 #endif
554         in_A,
555         in_B,
556         size,
557         opmode,
558         band_type,
559         A_band_stream,
560         B_band_stream,
561         A_gen_stream,
562         B_gen_stream,
563         size_stream,
564         band_type_stream,
565         opmode_stream
566         );
567
568 #if USERTL == 1
569     mmc(
570 #else
571     hls_mmc(
572 #endif
573         A_band_stream,
574         B_band_stream,
575         A_gen_stream,
576         B_gen_stream,
577
578         opmode_stream,
579         size_stream,
580         band_type_stream,
581
582         C_band_stream,
583         C_gen_stream);
584
585
586 #if OPTIMISED == 1
587     write_data_optimised(
588 #else
589     write_data(
```

```
#endif
        out_C,
        size,
        opmode,
        band_type,
        C_band_stream,
        C_gen_stream);
}
```

# List of Figures

# List of Tables

# List of Listings

# Acknowledgements

I'd like to thank my parents, Alice and Ergun, for believing in me and supporting me financially during my entire life and specifically during my 15 months abroad in Italy. I'd also like to thank my sister Dilara for reminding me to take things less seriously and taunting me with pictures of the beach while I was working.

Next, I'd like to apologize to my friends, Dennis Karp and Charlotte Stumme, for having to hear the words "*Guys, I'm making so much progress*" several times per day for the last 5 months. Of my friends, I'd like to particularly thank Mattia Carini and Marco Paolini for hosting me during the last month of my thesis, when I no longer had a home in Milan.

An honorable mention must go to my home-university coordinator, Prof. Claude Oestges, who allowed me to take part in a double degree program in PoliMi.

I would also like to thank my advisor, Prof. Christian Pilato, for always asking the right questions and single-handedly elevating the quality of my work.

The most special thanks have to go to Stephanie Soldavini, who throughout this thesis has played the role of advisor, assistant, helper, consoler, consultant, Linux-guru, deadlock resetter, documentation magician and last but certainly not least, friend. I will always admire her knowledge and hard work, and I am very lucky to have had the opportunity to work with her.