Design of Configurable and Extensible Accelerator Architectures for Machine Learning Algorithms

by

Chester Liu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)
in The University of Michigan
2021

Doctoral Committee:

Associate Professor Zhengya Zhang, Chair
Associate Professor Reetuparna Das
Professor Michael P. Flynn
Professor Wei D. Lu

Chester Liu

cwhliu@umich.edu

ORCID iD: 0000-0003-0115-9630

© Chester Liu 2021

ACKNOWLEDGEMENTS

I would've never been able to finish this thesis if it wasn't for all the guidance and support I got, especially from my advisor Professor Zhengya Zhang, who is always patient and insightful. I'd also like to thank the lab mates who had been easy to work with and given me a lot help, Phil, Chia-Hsiang, Jung Kuk, Shuanghong, Shiming, Wei, Thomas, Vishi, Alex, Sung-Gun, Teyuh, Jie-Fang, Jacob and Reid. Thanks to my parents who are always supportive, and my wife who takes good care of the family and shares the burden with me, and of course my son who brings a lot of joy to my life, without you I won't be able to make it this far.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

LIST OF FIGURES v

LIST OF TABLES viii

ABSTRACT ix

CHAPTER

I. Introduction 1

1.1 Convolutional Sparse Coding 2
1.2 2.5D Integration of Chiplets 5
1.3 Dissertation Outline 8

II. Configurable Convolutional Sparse Coding Accelerator . . . 9

2.1 Background 10
2.2 Sparse Neural Coding Algorithm and Mapping 11
2.2.1 Locally Competitive Algorithm (LCA) 13
2.2.2 LCA Learning Operation 14
2.2.3 Convolutional Formulation of LCA (CLCA) 14
2.3 Inference Hardware Architectures Comparison 15
2.3.1 Distributed Feature Feedback Architecture 16
2.3.2 Distributed Coefficient Feedback Architecture . . . 17
2.3.3 Centralized Reconstruction Feedback Architecture . 18
2.4 System Architecture 20
2.5 Configurable Convolver 22
2.5.1 Fractional Partition and Combine 23
2.5.2 Maze Convolution 26
2.5.3 Convolver Design Specification 27
2.6 Sparsity Optimization 28
2.7 Spike-Driven Reconstruction 30

2.8 Globally Asynchronous Interfaces 32
2.9 Implementation Results 33
2.10 Conclusions 36

III. AIB Chiplet Design for 2.5D Integration in MCPs 39

3.1 Background 39
3.2 AIB Interface 41
3.3 I/O Cell Design 42
3.3.1 I/O Driver Design 42
3.3.2 I/O Logic Design 44
3.4 AIB Channel 45
3.4.1 AIB Adaptor 46
3.4.2 Test Agent 48
3.4.3 Transfer Protocol 49
3.4.4 Open-RISC SoC 49
3.5 Chiplet design 50
3.5.1 Bump Map Design 50
3.5.2 AIB Channels in the Chiplet 52
3.6 Homogeneous Integration Based on Silicon Interposer 53
3.6.1 Silicon Interposer 53
3.6.2 Chiplet Testing 57
3.7 Heterogeneous Integration Based on EMIB 58
3.8 Summary 60

IV. UMAI Chiplet Interface Protocol Design 62

4.1 I/O Driver Optimization 62
4.2 AIB Channel 64
4.2.1 I/O Remapping for Chiplet Rotation 64
4.2.2 Word Marking and Multi-Channel Alignment 65
4.2.3 Data Format on the AIB Interface 66
4.2.4 UofM AIB Interface (UMAI) On-Chip Bus Interface 67
4.3 Chiplet Implementation 68
4.4 Summary 69

V. Conclusion and Outlook 71

BIBLIOGRAPHY 73

LIST OF FIGURES

Figure

1.1 Deep learning chip shipment by type. (Source: Tractica) 1

1.2 High-level illustration of the sparse coding operation. 3

1.3 Comparison of a conventional SoC chip to a multi-chiplet package (MCP). 6

2.1 Depth estimation based on feature extraction on stereo images. . . . 10

2.2 Hardware mapping of spiking convolutional sparse coding algorithm. In the first iteration, input image is selected and the convolution result is stored in the excitation map. In the subsequent iterations, image reconstructed by the previous iteration's feedback operation is selected. Neuron generates spikes when the potential exceeds a threshold. The collection of spikes forms the sparse representation of the input image. 15

2.3 DFFA architecture: neuron features are broadcast for a distributed feedback computation. 17

2.4 DCFA architecture: neuron output coefficients are broadcast for a distributed feedback computation. 18

2.5 CRFA architecture: neuron output coefficients are sent to the hub for a centralized feedback computation. 19

2.6 Block diagram showing the system containing a hub and 48 neurons, all running in different clock domains. 21

2.7 Three modes of a 2×2 configurable convolver: (a) Mode 1 adds four products and accumulates the sum to one of the four buffer entries. (b) Mode 2 operates in 2-way parallel, each adding two products and accumulating the sum to one of the two buffer entries. (c) Mode 3 operates in 4-way parallel, each accumulating one product to one buffer entry. 23

2.8 Computation of the four output values for a convolution of a 3×3 kernel with a 4×4 image. 24

2.9 Compute the convolution in 9 steps using the 2×2 convolver. The convolver is set to mode 1 for step 1 through 4, mode 2 for step 5 and 6, mode 3 for step 7, and mode 2 for step 8 and 9. 25

2.10 Maze path for a 3×3 kernel. (a) When overlaid on image, the path represents how it is traversed by the 2×2 convolver. (b) When overlaid on kernel, the sub-paths cover the sub-kernels used in the corresponding steps. 26
2.11 Construction of maze path for larger kernels. (a) To maximize data reuse for 5×5 kernel, two new sub-path types are needed. (b) Maze path constructed by adding one 2×5 segment and one 3×2 segment to the 3×3 maze path, all using the sub-paths already defined for 3×3 kernel. (c) Maze path for 7×7 constructed by adding segments to the 3×3 maze path. 27
2.12 An example of zero-patch skipping. (a) NZ map obtained from scanning the image with a NZ detector. (b) Maze path guided by a NZ map. 29
2.13 A spike-driven reconstruction example with two spikes. 31
2.14 Asynchronous interface design. (a) Token-based asynchronous FIFO. (b) Modified logic for the asynchronous FIFO to support broadcast. 32
2.15 Chip microphotograph. 34
2.16 Chip power and area breakdown. (a) power breakdown of the entire chip. (b) area breakdown of the entire chip. (c) power breakdown of a single neuron. (d) area breakdown of a single neuron. 35
2.17 Measured power and throughput of (a) feature extraction application, (b) depth extraction application. 36
3.1 Schematic design and layout view of the I/O driver. 42
3.2 I/O driver test bench setup and the simulated eye diagram on the far side (FS) of the interface. 43
3.3 Serializer and de-serializer logic design and waveform. 44
3.4 Block diagram of an AIB channel. 46
3.5 Clocking scheme when two chiplets are connected. The left chiplet is the master and the right chiplet is the slave. 47
3.6 One Wishbone write command packed into four 20b entries to be transmitted on the AIB interface. 50
3.7 Chiplet bump map views in Innovus and the bump map design tool developed in this work. 51
3.8 Chiplet die photo. 52
3.9 Silicon interposers on a 180nm wafer. 54
3.10 Interposer pattern for the mini AIB channel. 55
3.11 Interposer pattern for the regular AIB channels. 55
3.12 Silicon interposer based multi-chiplet package inside a PGA package. 56
3.13 Chiplet testing configuration for multi-chiplet package based on silicon interposer. 57
3.14 Chiplet integrated with an Intel Stratix 10 FPGA on the same substrate via EMIB. 58
3.15 Chiplet testing configuration for multi-chiplet package based on Intel EMIB. 59
4.1 Schematic design and layout view of the AIBv2 I/O driver. 63

4.2 I/O remapping example with 4 I/O cells: (a) No I/O remapping when chiplets are not rotated. (b) I/O cells are remapped when the right chiplet is rotated. 64
4.3 Data format as seen on the AIB interface. 66
4.4 UMAI bridging two chiplets with 4 AIB channels. 68
4.5 Intel 22nm chiplet bump map. Red, black, and white circles represent power, ground, and signals, respectively. Our designs are in the light blue region. 69
5.1 Chips designed in this dissertation. 72

LIST OF TABLES

Table

2.1 Comparison of Memory and Communication Requirement 20
2.2 Comparison of Sparsity Utilization Techniques 29
2.3 Comparison With Prior Work 37
3.1 Comparison Table of State-of-the-Art 2.5D Integration Technologies 60

ABSTRACT

Machine learning has gained a lot of attention over the past few years because of the wide range of applications it can be applied to. However, machine learning algorithms are typically computation-intensive and require hardware acceleration in order for them to be usable in real-time. As the technology node continues to shrink, the design effort and manufacturing cost of a chip are becoming prohibitively high, thereby limiting the scale of a single chip hardware accelerator. In this work an accelerator architecture was designed for a class of machine learning algorithm called sparse coding, and through advanced packaging technology, an extensible hardware system can be constructed using the 2.5D integration of chiplets.

The goal of sparse coding is to find a sparse representation of an input. A comprehensive comparison of different accelerator architectures for sparse coding is conducted to identify the most efficient architecture. A novel convolution computation method was proposed to support convolution for a variable kernel size using a fixed number of compute elements. By zero-patch skipping, the throughput can be increased by up to 40% at a 90% input sparsity. With a globally-asynchronous locally-synchronous clocking structure, the power consumption can be reduced by a maximum of 22%. A 2.56mm^2 configurable convolutional sparse coding accelerator chip is designed and fabricated in a 40nm CMOS technology. The chip demonstrates a competitive performance of 718GOPS running at 380MHz while consuming 257mW. The chip can be programmed for a variety of applications for learning and extracting features, and performing classifications.

A 2.5D integration technology allows one to construct a scalable and extensible hardware system using chiplets. A 2.5 mm × 2.5 mm chiplet with 3 independent Advanced Interface Bus (AIB) channels is designed and fabricated in a 16nm CMOS technology. When running at 1GHz with a 0.9V supply, the measured energy efficiency of the implemented AIB interface is 0.83pJ/b. A silicon interposer is fabricated, and two chiplets are assembled on the interposer to demonstrate homogeneous integration of chiplets. The chiplet is also verified with an Intel 14nm Stratix 10 FPGA, demonstrating heterogeneous integration of chiplets and the inter-operability of the AIB interface. A chiplet data transfer protocol, called University of Michigan AIB Interface (UMAI), is designed as an IP that provides a clean and simple interface to the user applications. A 4 mm × 4 mm chiplet with 8 AIB channels that are controlled by UMAI is designed and fabricated in a 22nm CMOS technology, and UMAI's functionality has been verified in silicon.

CHAPTER I

Introduction

Artificial intelligence (AI) has created yet another huge wave that attracted a tremendous amount of attention from not only the research community but also the industry. Unlike the other AI waves that happened in the past, which created more hypes than actual applications, in this time around, AI with advanced machine learning algorithms is making its way into our daily life, sometimes even without us knowing it. The content recommendation system when you make purchases online, the face recognition system when you post a photo on social media, the driver assistance system when you drive your car, just to name a few of the AI applications that are making our life easier. With many big companies and startups from a variety of industries going after it, AI is now one of the most important technologies that, with

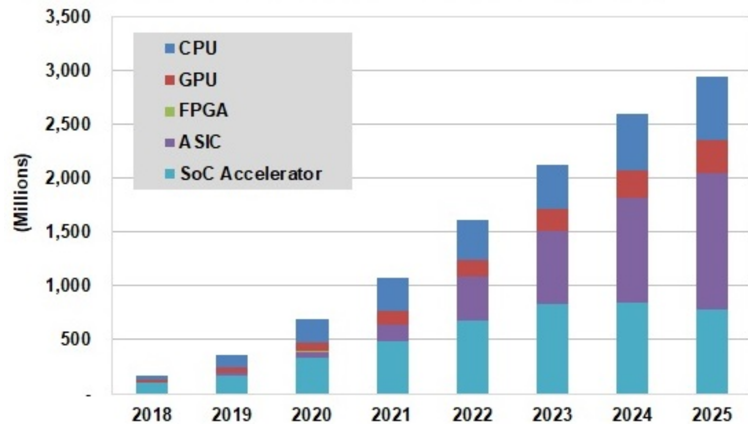


Figure 1.1: Deep learning chip shipment by type. (Source: Tractica)

hope, will make our life in the future not only easier, but better.

So what is making AI this time a reality rather than just a concept found in science fiction? There is no one-for-all reason behind this; it happens after all the key components are ready. Many factors have made the current AI wave stronger than ever, and the following are particularly important and interesting to the research community.

1. Over the years, new machine learning algorithms have been developed, achieving or even exceeding human-level performance at tasks involving perception and planning.

2. Advanced IC and packaging technology have made high performance computing hardware readily accessible, which is a key enabler to running AI applications in real time.

3. The Internet has allowed researchers all over the world to collaborate and share information, for example, large datasets like ImageNet and KITTI would have never existed without the collective effort, and these datasets play a important role in training the machine learning algorithms.

1.1 Convolutional Sparse Coding

Sparse coding [1] is a neuromorphic computing algorithm that learns the most salient features autonomously from an input dataset. These learned features, or dictionary, needs to be over-complete, i.e., the number of learned features is much larger than the size of the input, for the sparse coding algorithm to better capture patterns that exist in the input data. In sparse coding, an input is represented as a linear combination of all the features in the dictionary. However, the over-complete dictionary means that there exists many different representations with different linear coefficients for a given input. The goal of sparse coding is to find a sparse representation

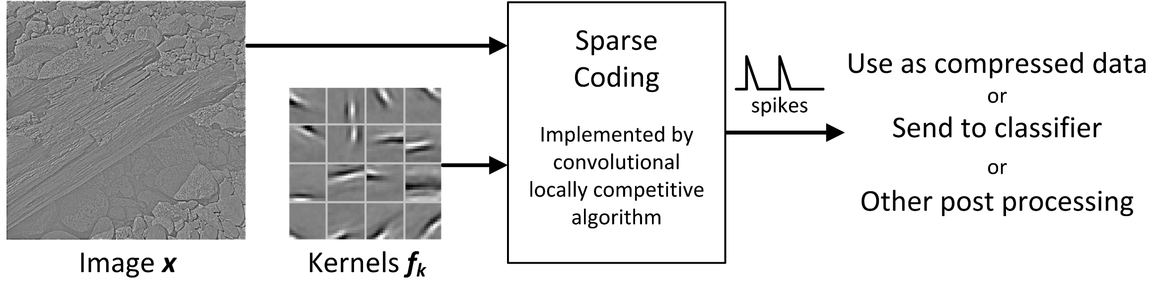


Figure 1.2: High-level illustration of the sparse coding operation.

for an input using as few features as possible.

Unlike other contemporary machine learning algorithms such as the deep convolutional neural network (DCNN) [2], which is often constructed based on empirical experience, sparse coding is based on a solid mathematical foundation. Sparse coding is an unsupervised learning algorithm, meaning that it is able to learn directly from the input dataset without the need of a list of expected results or labels. DCNN, on the other hand, is a supervised algorithm and it requires a labelled input dataset, which may not always be obtainable. As demonstrated in [1], when presented with a dataset of natural images, the features learned by the sparse coding algorithm resemble the Gabor filters (see the kernel images in Figure 1.2) that our visual cortex responds to.

Sparse coding has been applied to many applications in fields like machine learning [3, 4], pattern recognition [5], image processing [6, 7], and image compression [8]. All these applications are going to benefit from an efficient hardware accelerator for sparse coding. Mathematically, sparse coding is formulated as an optimization problem that tries to find the minimum value for (1.1), in which $\mathbf{x}$ is a given input, $\mathbf{\Phi}$ is the dictionary, $\mathbf{a}$ is the coefficient vector, T is a threshold function, and λ controls the weighting between the two term.

$$\arg\min_{\mathbf{a}} \left(\|\mathbf{x} - \mathbf{a}\mathbf{\Phi}\|^2 + \lambda T(\mathbf{a}) \right), \tag{1.1}$$

A numeric solver is usually required to solve this kind of optimization problem. For example, in [1] a conjugate gradient descendent algorithm is used to find the minimum of (1.1). However, a numeric solver is not suitable for a hardware implementation because it is typically sequential and it requires complex computation such as trigonometry and exponential.

Locally competitive algorithm (LCA) proposed in [9] solves the sparse coding optimization problem by using a neural network with recurrent connections between neurons. A neuron in the network represents one feature in the dictionary. Each neuron has an internal state called the potential, inspired by the actual neuron's membrane potential, that is charged by the input and discharged by the other neurons' activities. The charging and discharging continue until all neurons reach a stable state in which there's no significant change in neuron potentials, and this stable state is the optimization result of LCA. Prior works have already demonstrated high-performance hardware accelerators for LCA [4, 7]. However, LCA does not scale well with the input size because features are the same size as the input, meaning that the memory storage requirement for the dictionary grows rapidly as the input size increases.

Convolutional LCA (CLCA) is proposed in [10] to overcome this scalability problem. In CLCA, features are allowed to be smaller than the input, which decouples the dictionary size from the input size, allowing the algorithm to process larger input size compared to LCA. Instead of calculating the inner product of the features and the input, which is what LCA does, the features are used to scan the input and the convolution of the features (often referred to as the kernels because of the convolution operation) and the input is calculated. The feature size is application dependent, and as a result, a hardware accelerator supporting a fixed-size convolution operation [11] will be limited to specific applications.

1.2 2.5D Integration of Chiplets

Over the years we have enjoyed the outcome of Moore's Law and kept putting more and more transistors in a single chip. Nowadays, chips with multi-billion transistors inside are becoming commodities. Such a large-scale integration of integrated circuits allows us to put a complete system on a chip. Compared to a system composed of discrete chips, this kind of system-on-chip (SoC) integration offers many advantages such as lower inter-block communication overhead and smaller PCB footprint. However, the cost of building a monolithic SoC chip increases drastically as the chip size increases, especially in the more advanced technology nodes. Larger chip size also leads to lower manufacturing yield and longer design, verification, and testing time, which are all part of the overall cost. The system architecture is fixed in an SoC chip, meaning that it is impossible to optimize or fine-tune the system architecture after the chip is fabricated. Selecting one process node that works well for all the SoC blocks can be challenging, or even impossible if a specialized circuit is only available in certain nodes.

Using chiplets to overcome the aforementioned problems has been proposed and it has gained a lot momentum lately. Instead of integrating everything in a single SoC chip, the system is broken up into chiplets that are integrated at the package level. Broadly speaking chiplet integration can be categorized into one of the two kinds: (1) heterogeneous integration in which different types of chiplets (e.g. CPU, memory) are used, or (2) homogeneous integration in which all chiplets are of the same type (e.g. ASIC accelerator). Note that a chiplet is nothing new but simply a die that is being integrated inside a package. This divide-and-conquer approach allows each chiplet to be fabricated separately in a process node that is most suitable and cost-effective for the chiplet. With chiplets, the system architecture is determined at the package level, allowing the architecture to be tailored for different applications or product segments. For example, the Intel Stratix-10 FPGA [12] supports up to 6

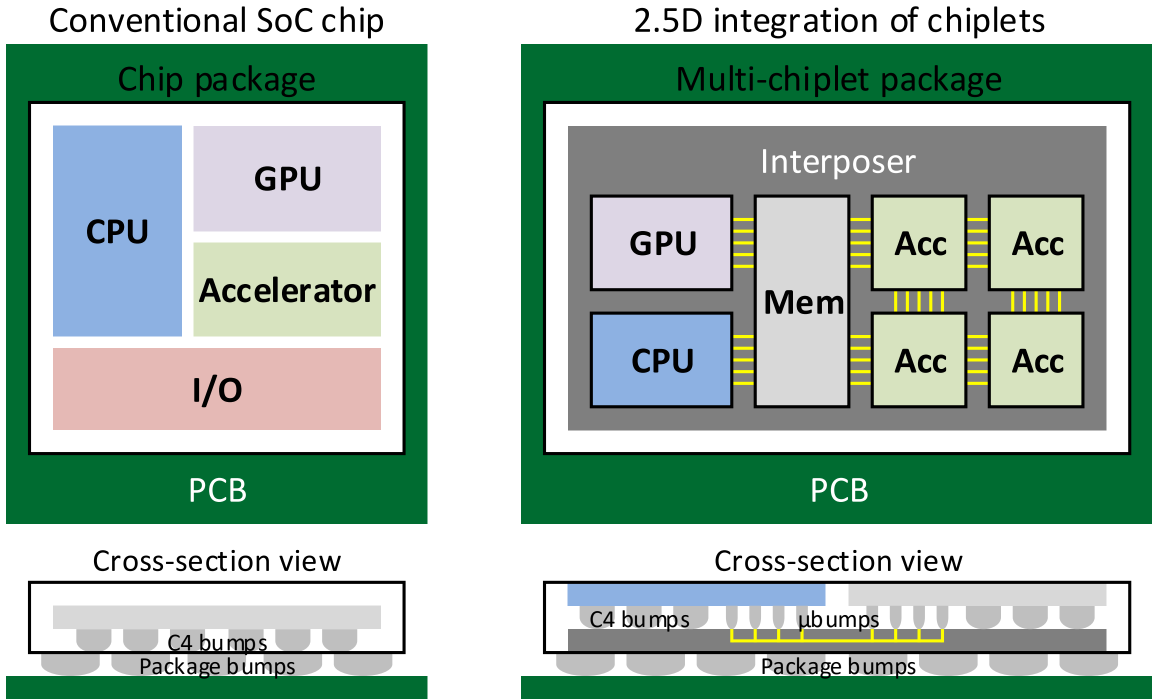


Figure 1.3: Comparison of a conventional SoC chip to a multi-chiplet package (MCP).

transceiver chiplets to be connected to the FPGA chiplet in the package, and it is up to the product definition to determine how many transceivers are needed.

Despite offering many advantages, chiplet-based systems will likely incur extra overhead in the inter-chiplet communication compared to the SoC approach in which all communication is within one die. One of the most important decisions for the inter-chiplet communication interface is serial versus parallel. Standard high-speed serial interfaces, such as PCIe and USB, are designed to transmit signals reliably off-package and across a long distance up to several meters, at the cost of design complexity and area, power and latency overhead. However, this kind of serial interface is an overkill for the inter-chiplet communication given that all chiplets are in the same package with a distance of a few millimeters between chiplets. So when it comes to the inter-chiplet communication, a parallel interface that is simple and light-weight is likely going to be a better solution. The main drawback of a parallel interface is the large number of data pins that need to be routed, and as will be discussed in the following,

the key enabler to constructing systems using chiplets with a simple parallel data interface is the advanced packaging technology.

As illustrated in Figure 1.3, typically an SoC chip includes a flip-chip die that is attached to the package substrate via the controlled collapse chip connection (C4) bumps. The C4 bumps are fanned out through the package substrate to the package bumps, which will be soldered on top of the PCB. Conventionally the minimum bump pitch, i.e., the distance between two bump center points, is around 150 µm, and the minimum routing width and spacing on the package substrate is around 10 µm and 10 µm, respectively. With advanced packaging technology, the bump pitch has been reduced to below 55 µm. Using silicon-based substrates such as a silicon interposer or Intel's embedded multi-die interconnect bridge (EMIB) [13], the routing width and spacing can be reduced to below 2 µm and 2 µm. This type of integration is called 2.5D integration because the chiplets are placed in a plane on top of the interposer. Since the throughput between chiplets is determined by the number of data pins that can fit within a unit area, a smaller bump pitch and more compact routing capability will lead to higher data throughput between chiplets.

A standard data interface needs to be defined for the chiplets to communicate with each other. This interface definition should at minimum define the number of data and clock pins, the clocking scheme between chiplets, operating frequency and voltage, and how the signals are mapped on the bump array. Advanced Interface Bus (AIB) [14] is an interface definition developed by Intel with the goal of becoming the standard chiplet data interface. Started as a proprietary data interface used by their FPGA, Intel decided to open-source AIB and make it publicly available to everyone. AIB is a digital parallel data interface that runs at 1GHz with double data rate (DDR) and an I/O voltage of 0.9V. 20 transmit (Tx) and 20 receive (Rx) data pins are grouped in an AIB channel, providing a total throughput of 80Gb/s per AIB channel. Source synchronous clocking scheme is adopted by AIB, meaning that

clocks are transmitted on the interface along with the data. An array of micro-bumps (μbumps) with a 55 µm is defined for the AIB channel. AIB only specifies the I/O operation at the PHY level, and it leaves it to the chiplet designer to define a data transfer protocol.

1.3 Dissertation Outline

In Chapter II, a configurable hardware accelerator for the convolutional sparse coding algorithm is presented. The novel convolution computation method supports convolution with different kernel sizes using a fixed size array of computing elements. By exploiting sparsity and incorporating neuron's spiking behavior [15], the configurable accelerator achieves a competitive power and performance results compared to other state-of-the-art works.

In Chapter III, the building blocks that are needed to enable chiplet operations are presented. An I/O driver and an AIB channel that is compliant with the AIB standard are implemented in this work. The chiplet is verified against another chiplet on a silicon interposer and showed correct function when running at 1GHz. Interoperability of the design between different technology nodes is demonstrated in a multi-chiplet package (MCP) containing an FPGA and the chiplet.

In Chapter IV, a protocol for chiplets to exchange data over multiple AIB channels is presented. In this work, the I/O driver is optimized to improve timing characterization, and the AIB channel is optimized to support full-speed operation on FPGA. This design is provided as a chiplet interface IP to two other ASIC designs, and the functionality has been verified in silicon.

We conclude the dissertation in Chapter V, giving an outlook on how future hardware accelerator designs may benefit from using chiplets with 2.5D integration.

CHAPTER II

Configurable Convolutional Sparse Coding Accelerator

In this chapter we present a configurable spiking convolutional sparse coding accelerator that incorporates three new features to advance the state of the art: (1) A new programmable convolution architecture, named maze convolution, that supports configurable kernel sizes and the full utilization of the hardware for all supported kernel sizes; (2) A new zero-patch skipping technique that effectively exploits the sparsity in the input to increase the performance and efficiency of sparse convolutions; (3) A scalable globally asynchronous locally synchronous (GALS) hardware architecture that does not require specialized CAD tools and can be implemented in the standard digital design flow.

A chip was designed and fabricated in 40nm. The chip demonstrates a performance of 718GOPS at 380MHz while consuming 257mW. The chip can be programmed for a variety of applications for learning and extracting features, and performing classifications. Figure 2.1 shows an interesting application using feature extraction. In this application, two chips are used to extract features from the left and right images, respectively. With the extracted features, a simple matching algorithm can be applied to estimate depth information and create the disparity map, in which objects in brighter color are closer to the camera.

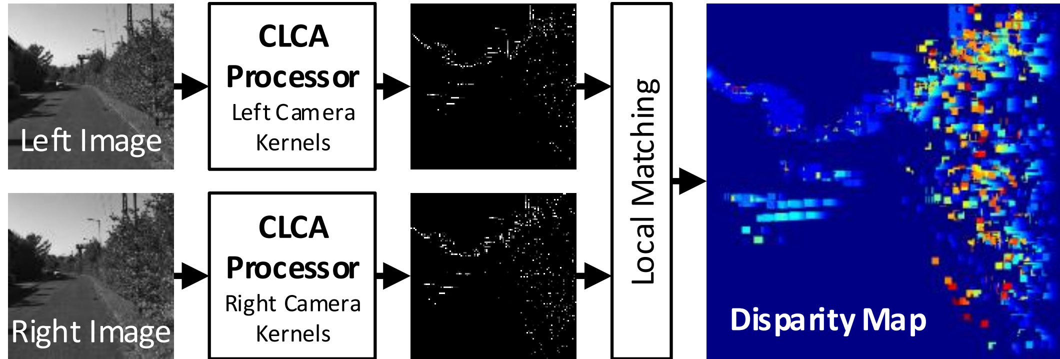


Figure 2.1: Depth estimation based on feature extraction on stereo images.

2.1 Background

Neuro-inspired sparse coding algorithms have been applied to various types of sensory inputs, including image [1, 6, 7], audio [16, 17], and video [18, 19, 20], for feature extraction in a wide range of applications such as denoising [21, 22], super-resolution [23, 24], object recognition [3, 4], and face recognition [5].

The classic sparse coding is mapped to a fully-connected recurrent neural network [9], and ASICs have been designed to achieve impressive performance and efficiency [4, 25]. However, the ASIC designs have been limited to relatively small feature sizes, e.g., 4×4, and small input image patch sizes, e.g., up to 16×16. The designs are not scalable to efficiently support applications that require larger feature sizes or larger input patch sizes. Convolutional sparse coding was therefore introduced to improve sparse coding's scalability by exploiting the shift-invariant characteristic commonly found in sensory inputs [26, 10].

Although convolutional sparse coding is scalable in theory, a number of important challenges still remain. First, sparse coding is meant to be a universal encoding algorithm that is input agnostic, but until now it is unclear whether a universal, or programmable, hardware can be made to extend the applicability of sparse coding to more than one application. Second, convolutional sparse coding is implemented in a convolutional recurrent neural network (RNN) that requires iterations of feedforward

and feedback convolution operations. Compared to popular feedforward convolutional neural networks (CNNs), convolutional RNN requires possibly an order of magnitude or more operations than a comparable CNN.

To address the programmability and complexity challenges, we note two design opportunities. First, if a configurable convolution engine can be made to support various kernel sizes, a convolutional sparse coding hardware can be made programmable to support different applications. Second, convolutional sparse coding, and sparse coding in general, provides inherent data sparsity that can be leveraged to reduce the computational complexity to improve both performance and efficiency.

Convolution engines supporting configurable kernel size have been reported recently [27, 28, 29, 30, 31]. However, in these designs, the hardware, i.e., multipliers and adders, cannot be fully utilized in supporting all kernel sizes, and these designs are unable to further improve their performance by exploiting input sparsity effectively. A sparse convolution engine was presented to increase the throughput by skipping zeros in the input [11]; however, it only supports a fixed kernel size so its application is limited. Scalable hardware architecture consisting of asynchronous modules has been designed for spiking neural networks [32, 33], but the development of such asynchronous designs requires specialized CAD tools, which are not readily accessible.

2.2 Sparse Neural Coding Algorithm and Mapping

Sparse coding is a neuro-inspired coding algorithm that attempts to find efficient, sparse representation of input stimulus. Through learning, sparse coding can be used to develop a dictionary of basis functions, or features, that is representative of the underlying structure in the data.

For sparse coding, the learning is unsupervised, so it can be deployed in the field and learn directly from unlabeled data. The resulting dictionary is overcomplete,

meaning that the size of the dictionary is larger than the input dimension. The dictionary is developed in a way so as to maximize the sparsity of the representation, which is a key feature of sparse coding. After learning converges, sparse coding can be used for inference to encode an input stimulus using the learned dictionary.

Mathematically, sparse coding can be described by $\mathbf{x} = \mathbf{a}\mathbf{\Phi}$, where $\mathbf{x}$ is a given input, $\mathbf{\Phi}$ is the dictionary, and $\mathbf{a}$ is the coefficient vector that is inferred by sparse coding. The objective of sparse coding is to find $\mathbf{a}$ that minimizes encoding error and maximizes the sparsity of $\mathbf{a}$, i.e.,

$$\arg\min_{\mathbf{a}} \left(\|\mathbf{x} - \mathbf{a}\mathbf{\Phi}\|^2 + \lambda T(\mathbf{a}) \right), \tag{2.1}$$

where T is a threshold function, and λ controls the weighting between the encoding error term and the sparsity term.

It is advantageous to adopt sparse coding in designing practical image, video, and audio processing systems, as it learns a good dictionary that is representative of the data, and it allows the compression of large, dense inputs to sparse coefficient vectors, akin to compressive sampling.

From the hardware design point of view, sparse coding produces sparse coefficient vectors, which simplify downstream processing to improve throughput and energy efficiency. Unlike conventional image and video codecs whose dictionary of basis functions conveys little information about the input, the dictionary learned by sparse coding consists of representative features. As such, the encoding contains meaningful information as to which features are existent and important in the input. After sparse coding, downstream processing can be carried out directly in the encoded, i.e., compressed, domain.

2.2.1 Locally Competitive Algorithm (LCA)

One of the earliest sparse coding algorithm Sparsenet solves the optimization of (2.1) using a conjugate gradient descent method [1]. A hardware implementation of conjugate gradient descent is however inefficient. Rozell *et al.* introduced a sparse coding algorithm named locally competitive algorithm (LCA) [9] that makes use of a RNN to solve the optimization of (2.1). The RNN can be efficiently parallelized and mapped to hardware, making it more appealing for practical applications.

In the LCA formulation, each neuron retains a feature and a potential that is charged with input stimuli through feedforward connections, and discharged with lateral inhibition through feedback connections. Inference using LCA is carried out over iterations. An iteration consists of a feedforward step for input stimuli to excite the neurons and a feedback step for the neurons to inhibit or "compete" with each other to represent the input. A typical inference converges in a few tens of iterations.

In performing the feedforward and feedback steps, a few simple rules are followed: 1) the closer a neuron's feature resembles the input, the faster the neuron's potential is charged; 2) a pair of neurons' inhibition is the strongest if the pair shares similar features; and 3) a neuron of high potential leaks faster than a neuron of low potential. Combining these rules, LCA's inference is described mathematically by the following equations

$$\begin{aligned} \Delta\mathbf{u} &= \Phi^T\mathbf{x} - (\Phi^T\Phi - I)T(\mathbf{u}) - \mathbf{u} \\ \mathbf{u}' &= \mathbf{u} + \alpha \cdot \Delta\mathbf{u} \end{aligned} \tag{2.2}$$

where $\mathbf{u}$ is a N-dimensional vector that stores neuron potentials, $\mathbf{x}$ is a M-dimensional vector that stores input stimuli, Φ is a $M \times N$ matrix that stores the dictionary, I is an identity matrix that is used to remove self inhibition, T is the threshold function, α is the step size, and $\mathbf{u}'$ is the updated neuron potential vector. In (2.2), $\Phi^T\mathbf{x}$ describes the feedforward excitation; $-(\Phi^T\Phi - I)T(\mathbf{u})$ describes the feedback inhibition; and $-\mathbf{u}$ describes the leakage term.

Thanks to the RNN formulation, LCA is suitable for hardware implementation, and it has been successfully demonstrated in hardware for image classification [4]. However, LCA relies on a fully-connected network with both feedforward and feedback connections, i.e., each neuron is connected to all the inputs and all the other neurons. The fully-connected network does not scale with input dimension.

2.2.2 LCA Learning Operation

Learning for LCA is unsupervised as it self-adapts the features Φ to better represent the input without requiring any label. In [9] Rozell *et al.* the optimization of (2.1) over Φ is mapped to the recurrent neural network and derived the following learning rule

$$\begin{aligned} \Phi' &= \Phi + \alpha \Delta\Phi \\ \Delta\Phi &= (\mathbf{x} - \Phi\mathbf{a})\mathbf{a}^T \end{aligned} \tag{2.3}$$

where $\Delta\Phi$ is the feature update, Φ' is the updated feature set, and α is a user-defined learning rate. For a given input, inference is performed before learning can be carried out as the inference result, i.e., $\mathbf{a}$, is required in calculating the feature update. In (2.3), $\Phi\mathbf{a}$ is the reconstructed input and $(\mathbf{x} - \Phi\mathbf{a})$ represents the coding error. Intuitively, in performing learning, (2.3) can be understood as minimizing the coding error introduced by each individual neuron.

2.2.3 Convolutional Formulation of LCA (CLCA)

Features in images tend to be shift-invariant, meaning that a feature may appear at different locations in an image. Scanning (i.e., convolving) the image with small-sized features known as kernels, is much more efficient than processing the image using features of the same size as the image itself.

Taking advantage of this insight, Schultz *et al.* introduced convolution to the original LCA algorithm to improve scalability by using small-sized kernels [10]. In

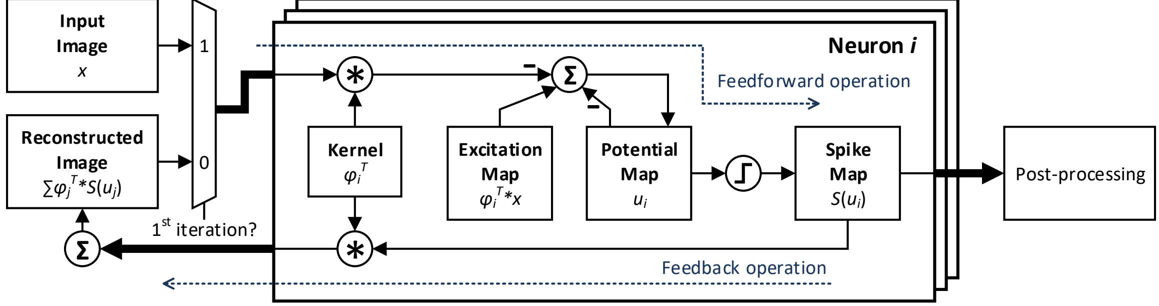


Figure 2.2: Hardware mapping of spiking convolutional sparse coding algorithm. In the first iteration, input image is selected and the convolution result is stored in the excitation map. In the subsequent iterations, image reconstructed by the previous iteration's feedback operation is selected. Neuron generates spikes when the potential exceeds a threshold. The collection of spikes forms the sparse representation of the input image.

convolutional LCA, the operational steps are the same as in LCA, but each neuron is equipped with a potential map, as opposed to a single potential, to keep track of the potential updated by the following equation

$$\Delta u_i = (\phi_i^T * x) - \phi_i^T * (\sum_j^N \phi_j * S(u_j)) - u_i. \tag{2.4}$$

Equation (2.4) largely resembles (2.2) except that matrix multiplications are all replaced by 2D convolutions, and a binary threshold function S is applied, i.e., $S(x) = 1$ if x is above a threshold, otherwise $S(x) = 0$. Equation (2.4) can be implemented in a convolutional RNN that consists of: 1) $\phi_i^T * x$ as the feedforward operation in the first iteration, 2) $\sum_j \phi_j * S(u_j)$ as the feedback operation, 3) $\phi_i^T * \sum_j \phi_j * S(u_j)$ as the feedforward operation in subsequent iterations, and 4) $-u_i$ as the leakage term in every iteration. Mapping of (2.4) to hardware is shown in Figure 2.2.

2.3 Inference Hardware Architectures Comparison

In this chapter we compare three different architectures designed to accelerate the inference operation for LCA. If we use $\mathbf{y}$ to represent $T(\mathbf{u})$, then the potential update

part of (2.2) becomes

$$\Delta \mathbf{u} = \Phi^T \mathbf{x} - (\Phi^T \Phi - I)\mathbf{y} - \mathbf{u} \tag{2.5}$$

which can be rearranged into

$$\Delta \mathbf{u} = \Phi^T (\mathbf{x} - \Phi \mathbf{y}) + \mathbf{y} - \mathbf{u} \tag{2.6}$$

Note that inhibitions between neurons are calculated in (2.5), whereas in (2.6) a reconstruction is carried out. Both (2.5) and (2.6) are in vector form as they calculate the potential update for all neurons. For an individual neuron it is calculated by

$$\Delta u_i = \boldsymbol{\phi}_i^T \mathbf{x} - \sum_{j=1, j\neq i}^{N} ((\boldsymbol{\phi}_j^T \boldsymbol{\phi}_j) y_j) - u_i \tag{2.7a}$$

$$\Delta u_i = \boldsymbol{\phi}_i^T (\mathbf{x} - \sum_{j=1}^{N} (\boldsymbol{\phi}_j y_j)) + y_i - u_i \tag{2.7b}$$

where (2.7a) and (2.7b) are the scalar form of (2.5) and (2.6), respectively.

The first two architectures are designed based on (2.7a). Neurons in these architectures have fully-connected feedback connections with other neurons, and the feedback computation is distributed to the neurons. The two architectures differ in the type of data sent from neurons for feedback computation. Neurons in the third architecture, which is based on (2.7b), are connected to a centralized hub responsible for the feedback computation and there is no connection between neurons.

2.3.1 Distributed Feature Feedback Architecture

In this DFFA architecture, the potential update is calculated using (2.7a), and each neuron retains a feature, a potential, and an output coefficient, as illustrated in Figure 2.3. In computing the feedforward excitation, the input is sent to all neurons, and is multiplied with the feature of each individual neuron in parallel. When a

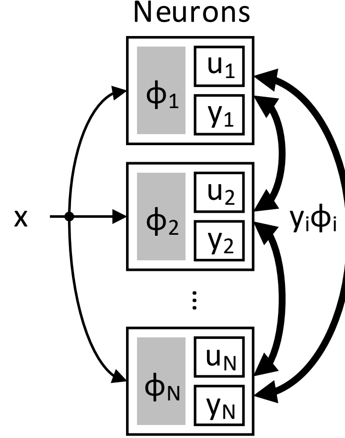


Figure 2.3: DFFA architecture: neuron features are broadcast for a distributed feedback computation.

neuron's potential exceeds the threshold, namely $y > 0$, the neuron inhibits other neurons by broadcasting its feature, which will be used by other neurons in computing the feedback inhibition.

The total memory storage requirement for features in this architecture is $O(NM)$, as each neuron needs to store its own feature. To support broadcast, a neuron requires M connections to the input and $(N-1)M$ connections to the other neurons, requiring in total $O(NM)$ feedforward connections and $O(N^2M)$ feedback connections. By distributing the feedforward and feedback computation to neurons, this architecture achieves high modularity and parallelism. However, this architecture has limited scalability since the number of feedback connections grows quadratically with the neuron number.

2.3.2 Distributed Coefficient Feedback Architecture

This DCFA architecture shares a similar structure with DFFA, as illustrated in Figure 2.4. The feedforward computation is the same as in DFFA, and the potential update is also calculated using (2.7a). But unlike DFFA, in which each neuron only retains its own feature, neurons in this architecture retain the entire feature set. When

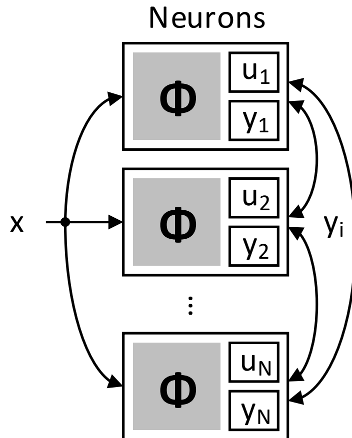


Figure 2.4: DCFA architecture: neuron output coefficients are broadcast for a distributed feedback computation.

a neuron's potential exceeds the threshold, the neuron broadcasts its output, which will be used by other neurons to lookup the feature set in computing the feedback inhibition.

This architecture achieves the same level of modularity and parallelism as in DFFA. Compared to DFFA, this architecture requires less feedback connections, i.e., $O(N^2)$, as features are not broadcast from neurons. As a trade-off, the total memory requirement for features is increased to $O(N^2M)$ in order to store the feature set in every neuron. Implementations based on this architecture typically lead to a better hardware utilization than DFFA. An ASIC design based on this architecture has demonstrated very high throughput for a network of 256 neurons [4]. This architecture, however, also has limited scalability since the memory requirement grows quadratically with the neuron number.

2.3.3 Centralized Reconstruction Feedback Architecture

Neurons in this CRFA architecture retain their own feature. A centralized hub which retains the entire feature set is added and connected to all neurons, as illustrated in Figure 2.5. The potential update is calculated using (2.7b), of which the

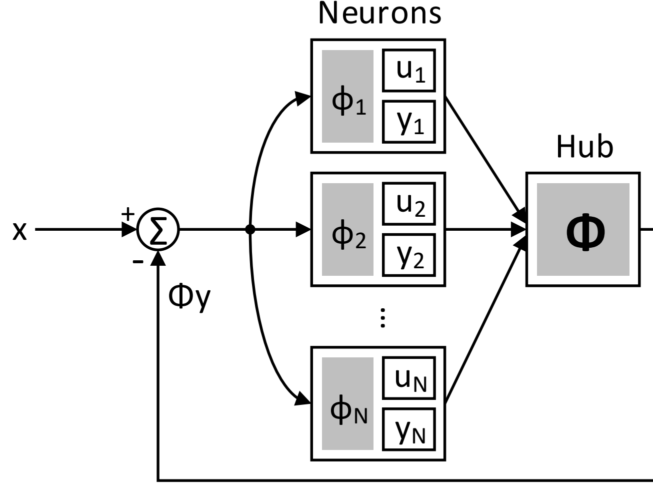


Figure 2.5: CRFA architecture: neuron output coefficients are sent to the hub for a centralized feedback computation.

feedforward and the feedback computation are carried out by the neurons and the hub, respectively. In performing the feedforward computation, the coding error, i.e., the difference between the original input and the reconstructed input, is sent to all neurons. Unlike the two distributed architectures, in which neurons are fully-connected with each other, neurons in this architecture are only connected to the hub and there is no inter-neuron connection. When a neuron's potential exceeds the threshold, the neuron sends its output to the hub. In performing the feedback computation, the hub collects output from neurons and reconstructs the input, which will be used in the next iteration.

Each neuron and the hub require $O(M)$ and $O(NM)$ memory storage for the feature and the feature set, respectively, requiring in total $O(NM)$ feature memory storage. The number of feedforward and feedback connection in this architecture are both $O(NM)$. This architecture achieves the best scalability by combining the advantage of the two distributed architectures: (1) storing individual feature in neurons for the feedforward computation, and (2) having another copy of the feature set to reduce the number of feedback connections. As a result, the accelerator presented in

Table 2.1: Comparison of Memory and Communication Requirement

Architecture	Memory	Feedforward connection	Feedback connection
DFFA	$O(NM)$	$O(NM)$	$O(N^2M)$
DCFA	$O(N^2M)$	$O(NM)$	$O(N^2)$
CRFA	$O(NM)$	$O(NM)$	$O(NM)$

this work is design based on this CRFA architecture. Compared to the distributed architectures, which can be made fully parallel and each neuron has similar workload, the hub in this architecture could become the performance bottleneck as the workload of the hub is significantly higher than the neurons. In preventing the hub from becoming the bottleneck, we designed an event-driven reconstruction to reduce the computation carried out at the hub by taking advantage of the sparsity. To further improve the throughput, a globally-asynchronous locally-synchronous structure is deployed so as to balance the throughput between neurons and the hub by adjusting their clock frequency.

2.4 System Architecture

To implement a convolutional RNN for sparse coding, a modular hardware architecture consisting of a single hub and a multitude of neurons is designed as shown in Figure 2.6, where the feedforward operations are distributed to the neurons, and the neuron output, in the form of binary spikes, is sent to the central hub for feedback operations. The sparse neuron spikes allow us to share one hub for feedback operations.

In a feedforward operation, the hub broadcasts a dense input image (in the first iteration) or a sparse reconstructed image (in subsequent iterations) to neurons. Upon receiving an image, each neuron convolves it with its kernel and accumulates the result to the potential map. Spikes will be generated for locations in the map that exceed a

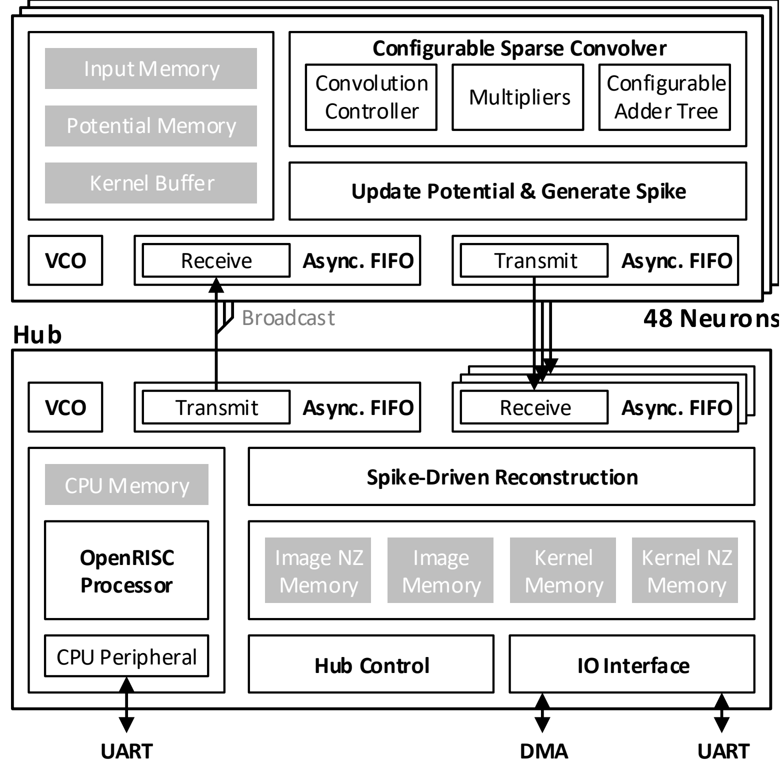


Figure 2.6: Block diagram showing the system containing a hub and 48 neurons, all running in different clock domains.

threshold. The convolution is performed by the proposed convolution engine that is optimized for both dense and sparse input with a configurable kernel size. The design of the convolution engine is described in Section 2.5.

In a feedback operation, neuron spikes are convolved with their kernels to reconstruct the input image. A direct implementation of this convolution is computationally expensive. We take advantage of the binary spikes to replace all multiplications in the convolution by additions, and further make use of the high sparsity of the spikes (typically >80% sparse) to design a sparsely activated spike-driven reconstruction that saves computation, power, and chip area. The design of the spike-driven reconstruction is described in Section 2.7.

Each neuron is equipped with one 8Kb (32×32×8b) excitation memory which buffers the excitation computed in the first iteration, one 8Kb (32×32×8b) poten-

tial memory, and a 1.8Kb (15×15×8b) latch-based kernel buffer. The hub contains 96Kb (48×16×16×8b) kernel memory, 19Kb kernel non-zero (NZ) memory, 32Kb (32×32×16b×2) image memory, and 4Kb image NZ memory. NZ memory provides fast non-zero entry lookup and is described in Section 2.6. Image memory and image NZ memory are double-buffered, enabling seamless data transfer from one feedback operation to the next iteration's feedforward operation. With these memory configurations, the system can process up to 15×15 kernel size and 32×32 image size. An input image exceeding this size needs to be divided into sub-images, with overlaps if necessary to minimize edge artifacts. Note that the kernel and image size are not limited by the architecture, but by the available on-chip memory.

The modular structure is made possible by deploying a voltage-controlled oscillator (VCO) in every neuron and the hub. The hub broadcasts commands and data to all neurons through a 128-bit (time-multiplexed 128-bit command and 16 8-bit data) asynchronous FIFO, and each neuron sends neuron spikes back to the hub through a 10-bit (5-bit x and y coordinates) asynchronous FIFO. The design of asynchronous FIFOs is described in Section 2.8.

The accelerator is initialized by populating the kernel and kernel NZ memories with data loaded through a 16-bit bi-directional DMA interface in the hub, which is also used to off-load neuron spikes for verification. A UART interface in the hub is used for controlling operations and setting configurations such as the kernel size and the number of iterations. As a demonstration of an integrated system, the accelerator is integrated with an OpenRISC processor, which can be tasked with learning and other post-processing operations.

2.5 Configurable Convolver

A 2D convolution operation can be viewed graphically by sliding the kernel on top of the input image and computing one inner product of the kernel with the covered

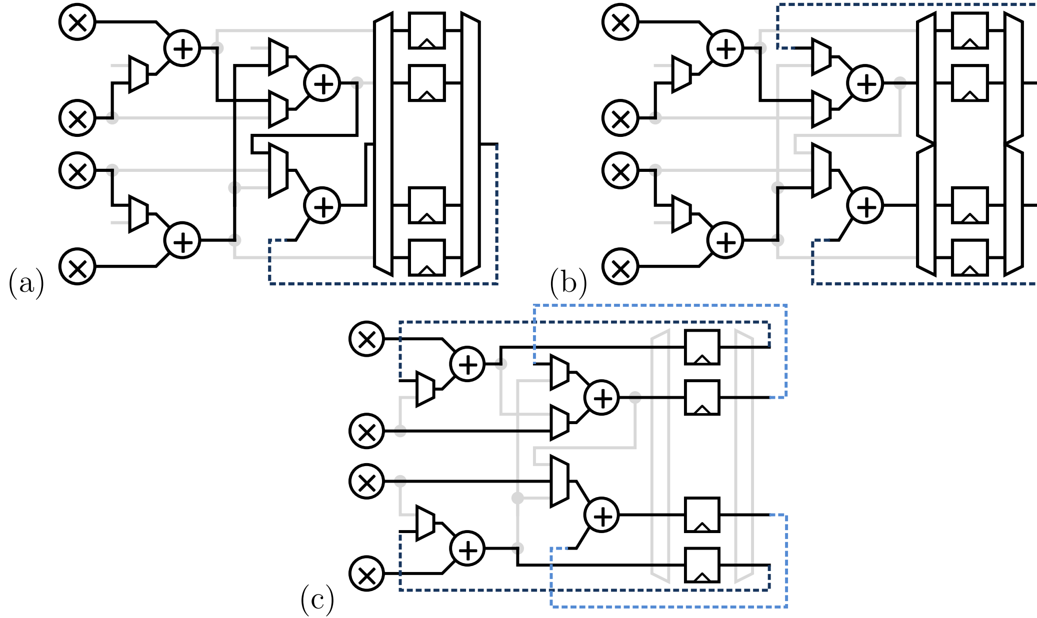


Figure 2.7: Three modes of a 2×2 configurable convolver: (a) Mode 1 adds four products and accumulates the sum to one of the four buffer entries. (b) Mode 2 operates in 2-way parallel, each adding two products and accumulating the sum to one of the two buffer entries. (c) Mode 3 operates in 4-way parallel, each accumulating one product to one buffer entry.

image portion per step of the slide. Each inner product involves element-wise product of the kernel and the covered image portion followed by summation of the elements. If the kernel size is fixed, a parallel compute array can be designed to support the inner product computation. In the following, we will refer to the inner product compute array as a convolver. If the size of the kernel varies, a convolver of a fixed size cannot be efficiently utilized. How to design a configurable convolver to support different kernel sizes while achieving the highest utilization is a challenge.

2.5.1 Fractional Partition and Combine

To address this challenge, our high-level idea is to design a fixed $C \times C$ convolver, and partition a $K \times K$ ($K > C$) kernel into $C \times C$ sub-kernels that can be directly mapped to the $C \times C$ convolver. Convolutions by sub-kernels produce partial sums, and the complete convolution results are formed by adding the partial sums. If a ker-

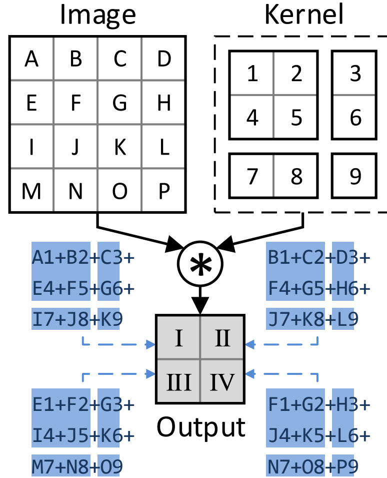


Figure 2.8: Computation of the four output values for a convolution of a 3×3 kernel with a 4×4 image.

nel cannot be partitioned into an integer number of $C \times C$ sub-kernels, a naïve design would result in the underutilization of the convolver. To overcome this inefficiency, we apply a new fractional partition and combine scheme to ensure the full utilization of the hardware.

For simplicity, we will use a $C = 2$, 2×2 convolver for illustration. A 2×2 convolver is made of four multipliers and four adders. We add configurable input connections to the adders, and a multi-port buffer memory with four entries to make a configurable convolver that supports three basic modes.

In mode 1, shown in Figure 2.7, the convolver computes four products and a 4:1 summation, i.e., sums the four products and accumulates the sum in one of the four buffer entries. In mode 2, shown in Figure 2.7, the convolver functions as two independent sub-convolvers working in parallel. Each sub-convolver computes two products, sums them and accumulates in one of the buffer entries. In mode 3, shown in Figure 2.7, the convolver functions as four independent sub-convolvers. Each sub-convolver performs one product followed by accumulation in one buffer entry.

With the configurable convolver, we illustrate our fractional partition and combine scheme in Figure 2.8 for an example of a 2×2 configurable convolver computing the

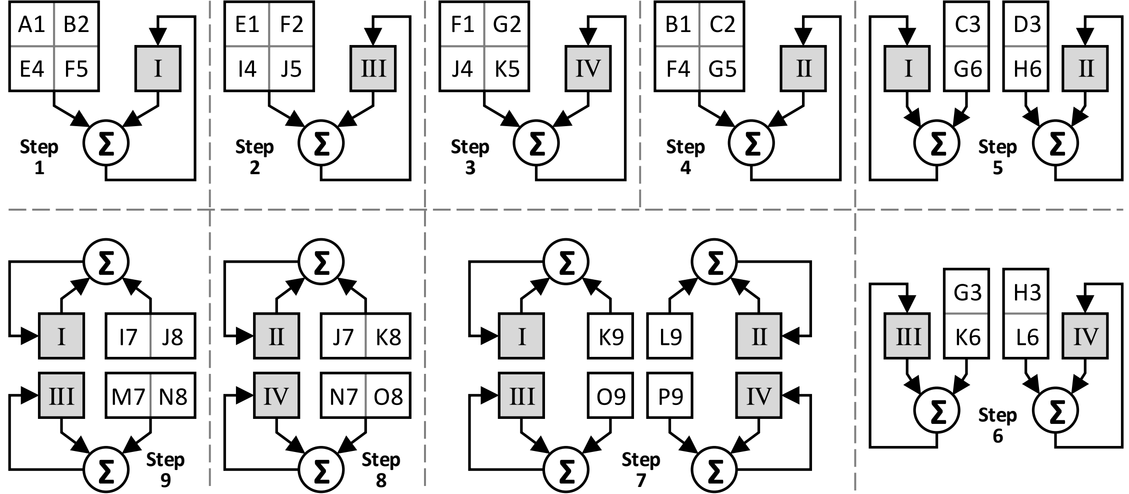


Figure 2.9: Compute the convolution in 9 steps using the 2×2 convolver. The convolver is set to mode 1 for step 1 through 4, mode 2 for step 5 and 6, mode 3 for step 7, and mode 2 for step 8 and 9.

convolution of a 3×3 kernel with a 4×4 image. The convolution will create four output values. In this case $K = 3$ is not divisible by $C = 2$, so fractional partition is done as follows: the 3×3 kernel is partitioned into four sub-kernels: a 2×2 square, a 2×1 column, a 1×2 row, and a 1×1 element. We show the four sub-kernels in Figure 2.8 and highlight how each is used to compute partial sums. Note that except for the 2×2 square, the other three partitions are fractional.

The convolution proceeds as illustrated in Figure 2.9 by convolving sub-kernels with the input image using the 2×2 configurable convolver. In step 1, mode 1 is set to convolve the 2×2 sub-kernel with the upper-left 2×2 input image patch. In step 2, the 2×2 sub-kernel slides one row down the input image to compute another 2×2 convolution. Similarly in steps 3 and 4, the 2×2 sub-kernel slides one column right and one row up, respectively, to compute the 2×2 convolutions. In step 5, mode 2 is set to convolve the 2×1 sub-kernel with two 2×1 columns in the input image. The two 2×1 fractional partitions are combined to fully utilize the convolver. A similar operation is done in step 6. In step 7, mode 3 is set to convolve the 1×1 sub-kernel with four 1×1 elements in the input image. The four 1×1 fractional partitions are

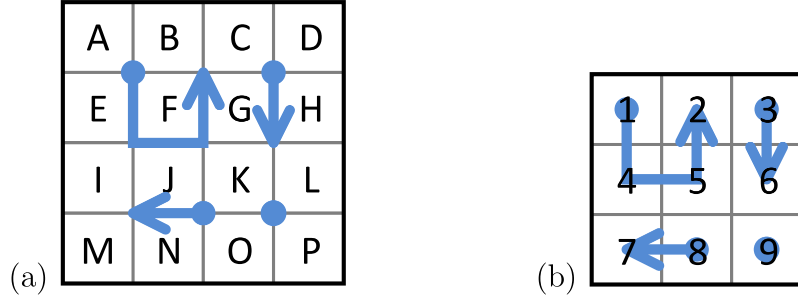


Figure 2.10: Maze path for a 3×3 kernel. (a) When overlaid on image, the path represents how it is traversed by the 2×2 convolver. (b) When overlaid on kernel, the sub-paths cover the sub-kernels used in the corresponding steps.

combined to ensure the full utilization of the hardware. In steps 8 and 9, mode 2 is set to convolve the 1×2 sub-kernel with two 1×2 rows in the input image. Again, the two 1×2 fractional partitions are combined.

By partitioning a kernel to sub-kernels and mapping fractional sub-kernel convolutions from consecutive steps on the convolver, we ensure the full utilization of the hardware and reduce the number of processing steps from 16 to 9 in this small example, achieving the highest efficiency and the minimum latency.

2.5.2 Maze Convolution

Notice from the small example that the steps are designed to maximize data reuse following two principles: 1) steps that use the same convolver mode are grouped together to maximize the reuse of the same sub-kernel; and 2) only allow a single row or column slide in the input image between steps to maximize the reuse of the input. These principles lead to a carefully designed maze walking path that traverses the input image, as illustrated in Figure 2.10. The maze walking path consists of a set of sub-paths that start with a dot and end with an arrow, and each sub-path corresponds to a convolver mode, as illustrated in Figure 2.10. We name the convolution following the maze walking path "maze convolution".

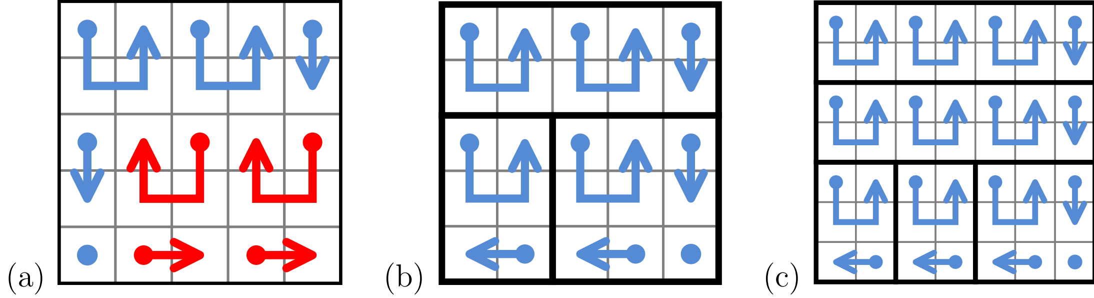


Figure 2.11: Construction of maze path for larger kernels. (a) To maximize data reuse for 5×5 kernel, two new sub-path types are needed. (b) Maze path constructed by adding one 2×5 segment and one 3×2 segment to the 3×3 maze path, all using the sub-paths already defined for 3×3 kernel. (c) Maze path for 7×7 constructed by adding segments to the 3×3 maze path.

Having a dedicated maze path for each kernel size allows us to maximize data reuse, but it also increases the storage requirement and control complexity. Figure 2.11 illustrates a maze path for a 5×5 kernel that maximizes data reuse. The maze path uses two new types of sub-paths, as highlighted in Figure 2.11, adding extra storage requirement and control complexity. Instead, we can design the maze path for a large kernel incrementally based on the sub-paths that are defined for a small kernel. As illustrated in Figure 2.11, the maze path for the 5×5 kernel can be composed of the sub-paths already defined for the maze path for the 3×3 kernel shown in Figure 2.10. High modularity is achieved by constructing maze path incrementally, as exemplified by the maze path for the 7×7 kernel in Figure 2.11; and maze paths for larger kernels, e.g., 9×9, 11×11, and so on, can be constructed based on the same principle.

2.5.3 Convolver Design Specification

To generalize, a $C \times C$ configurable convolver is made of C^2 multipliers, C^2 adders, and C^2 buffer entries. A $C \times C$ convolver supports any convolution kernel size and image size. Full utilization of the convolver can be achieved by mapping the convolu-

tion of a $K \times K$ ($K > C$) kernel with a $(C+K-1) \times (C+K-1)$ image, producing a $C \times C$ output. Such a convolution requires K^2C^2 multiplications and accumulations, and only K^2 steps to complete on the $C \times C$ convolver. A smaller image size can also be mapped, but it would result in underutilization of the hardware. A larger image can also be mapped, but for full utilization, the image needs to be of the size $((C+K-1)+NK) \times ((C+K-1)+NK)$ ($N \in \mathbb{Z}, N \geq 0$) or padded to the size.

In our design, padding is performed implicitly by the controller to maintain a compact and constant memory size for the convolver, which is especially important for a configurable convolver because the size of padding varies for different kernel sizes. After padding, a larger image is processed by the configurable convolver one patch at a time. Each patch is $(C+K-1) \times (C+K-1)$, and adjacent patches are offset by K columns or K rows. The total number of patches is $(N+1)^2$ and thus the total number of steps to complete the convolution is $(K(N+1))^2$.

2.6 Sparsity Optimization

If the input to a convolver is sparse, i.e., the input contains many zero entries, there is a potential opportunity to improve the design to increase the processing throughput and efficiency. In particular, three methods have been demonstrated in prior work. The first method, which we name zero-entry masking [27], disables the multiply-accumulate circuit when encountering a zero in the input. Zero-entry masking reduces the dynamic power consumption, but the throughput remains constant because it does not skip over a zero entry. The second method, which we name zero-line skipping [11], skips over an input line containing all zeros. Zero-line skipping increases the throughput as the input sparsity increases, and it can be combined with zero-entry masking to reduce the power consumption. However, we notice that in practice, it is more likely to have a $n \times n$ ($n \in \mathbb{Z}, n > 0$) square patch of zeros than a $1 \times n^2$ line of zeros. The third method, which we name zero-entry skipping [34, 35], aims to skip

Table 2.2: Comparison of Sparsity Utilization Techniques

Technique	Throughput	Overhead
Zero-entry masking [27]	low	low
Zero-line skipping [11]	mid	mid
Zero-entry skipping [34, 35]	highest	highest
Proposed zero-patch skipping	high	mid

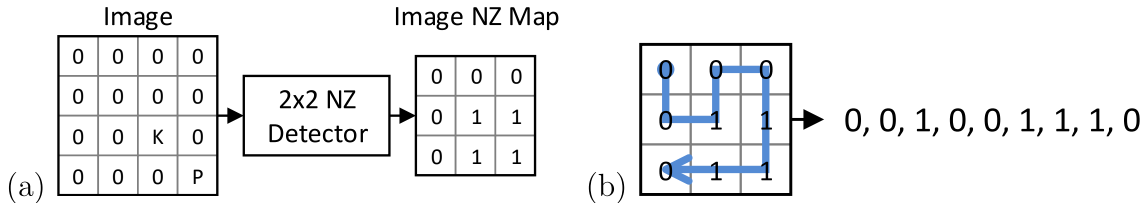


Figure 2.12: An example of zero-patch skipping. (a) NZ map obtained from scanning the image with a NZ detector. (b) Maze path guided by a NZ map.

over all zeros in the input. Zero-entry skipping, albeit theoretically optimal, requires complex control and incurs large hardware overhead.

In this work, we design a new sparsity optimization method called zero-patch skipping to skip over square patches in the input that contain all zeros. As shown in Table 2.2, compared to the three existing methods, zero-patch skipping offers several advantages: 1) higher throughput than zero-entry masking; 2) more effective than zero-line skipping due to the higher likelihood of encountering a square patch of zeros than a line of zeros, and 3) lower hardware overhead than zero-entry skipping thanks to the simpler per-patch based control than entry-by-entry based control.

To enable zero-patch skipping, we introduce a new data structure called NZ map to associate with each input. Conceptually, an NZ map is constructed by scanning an $I \times I$ input image with a $C \times C$ NZ detector. The NZ detector outputs 1 if at least one entry in an $C \times C$ patch is non-zero, otherwise it outputs 0. After scanning the $I \times I$ input, an $(I - C + 1) \times (I - C + 1)$ NZ map is constructed. An NZ map example is shown in Figure 2.12 for the given input image.

In practice, we do not use a NZ detector because it would be too costly in terms

of latency. Instead, NZ map is built incrementally using very simple operations. In the first iteration, the NZ map is initialized with all 1's because the input image is assumed to be dense. The NZ map is updated in every subsequent iteration in the reconstruction process based on the neurons, or kernels, that are activated. Since sparse coding enforces sparse neuron spike, the NZ map will be sparsely populated with 1's. The construction of NZ map will be discussed in detail in Section 2.7.

Maze convolution natively supports zero-patch skipping. Guided by the NZ bit sequence obtained from traversing the NZ map using the maze path, maze convolution skips steps where the NZ bit is 0 to realize sparsity-proportional throughput increase. We show in Figure 2.12 the NZ bit sequence for the NZ map constructed in Figure 2.12; and that five steps are skipped in performing the maze convolution, effectively doubling the throughput. Compared to [11], maze convolution with zero-patch skipping increases the throughput by up to 40% at 90% input sparsity. The proposed maze convolution with zero-patch skipping is equally applicable to CNNs [36].

2.7 Spike-Driven Reconstruction

Image reconstruction is performed at the end of each iteration to compute the input for the next iteration. Reconstruction latency therefore needs to be minimized so as not to halt the next iteration. Reconstructing an image according to (2.4) is costly in terms of both computation and latency as it involves convolution of every neuron's output with the neuron's kernel followed by summation of all convolution results. However, in computing convolution, we are able to replace the expensive multiplication with simple addition by exploiting neuron's spiking output to reduce the computation cost significantly. Furthermore, the sparse spikes allow us to hide the latency by performing image reconstruction incrementally.

Triggered by a neuron's spike, the hub performs image reconstruction by retriev-

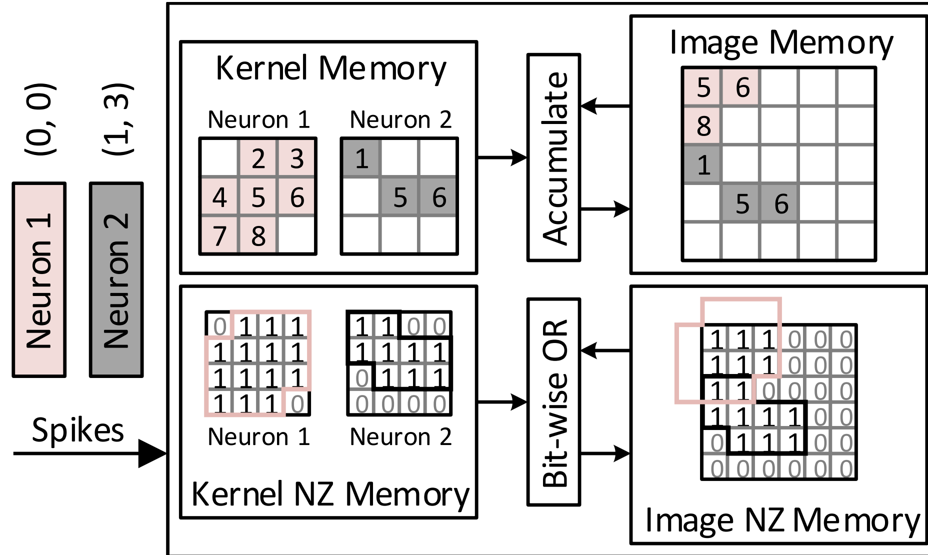


Figure 2.13: A spike-driven reconstruction example with two spikes.

ing the neuron's kernel from the kernel memory and accumulating the kernel in the image memory, with the kernel's center aligned to the spike location. In parallel, the hub incrementally constructs the NZ map of the reconstructed image by OR'ing the kernel's NZ map retrieved from the kernel NZ memory with the image NZ memory. Figure 2.13 shows an example in which neuron 1 has generated a spike at (0, 0) and neuron 2 has generated a spike at (1, 3). The neuron kernels are read out from the kernel memory and accumulated to the image memory, and their NZ maps are read out from the kernel NZ memory and OR'ed with the image NZ memory. Constructing an NZ map incrementally eliminates the need to scan the reconstructed image, saving both computation and latency. Kernel NZ maps are pre-computed and loaded to the kernel NZ memory during initialization. The spike-driven reconstruction eliminates the need to store spike map within each neuron, and a 16-entry FIFO in the hub is sufficient for buffering spikes, cutting the storage by 2.5×.

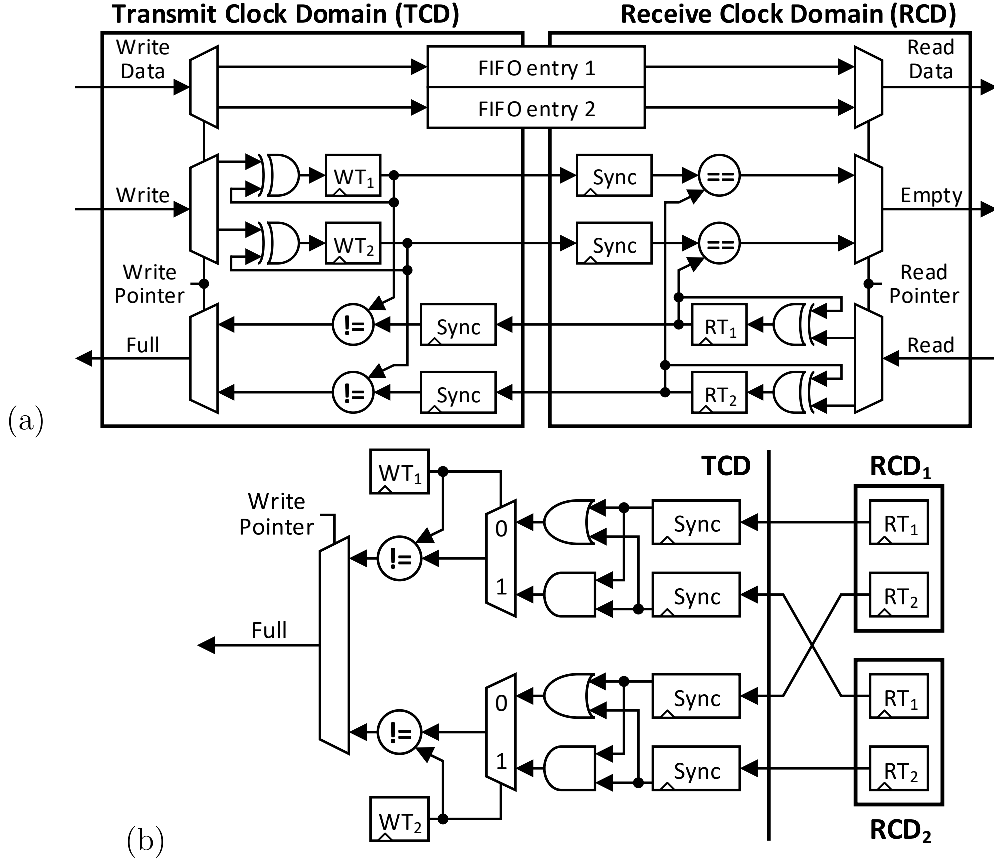


Figure 2.14: Asynchronous interface design. (a) Token-based asynchronous FIFO. (b) Modified logic for the asynchronous FIFO to support broadcast.

2.8 Globally Asynchronous Interfaces

We use a token-based asynchronous interface [37] to enable the exchange of messages between neurons and the hub that operate in different clock domains. The asynchronous interface consists of a transmit clock domain (TCD) located in the sender, a receive clock domain (RCD) located in the receiver, and an N-entry memory storage bridging the two clock domains as shown in Figure 2.14.

TCD consists of a write pointer that selects the next entry, N write token (WT) bits, and N synchronized read token (RT) bits from RCD. To send a message from TCD to RCD, the message data is written to the entry pointed by the write pointer, and the corresponding WT bit is toggled before the write pointer is incremented. If

the WT bit and the synchronized RT bit pointed by the write pointer have the same value, the entry has been read and is vacant for write; on the other hand, if the WT bit and the synchronized RT bit pointed by the write pointer in TCD have different values, the data stored in the entry has not been read, indicating that the FIFO is full. The RCD design follows the same principle.

We modify the FIFO full condition logic to allow one TCD to broadcast data to M RCDs. When the WT bit pointed by the write pointer is 0, the FIFO is full if any of the M synchronized RT bits pointed by the write pointer is 1, which can be detected by OR'ing these M synchronized RT bits; on the other hand, when the WT bit is 1, the FIFO is full if any of the M synchronized RT bits is 0, which can be detected by AND'ing these M synchronized RT bits. An example with $N = 2$ and $M = 2$ is shown in Figure 2.14. Compared to a conventional handshake-based asynchronous FIFO that requires two-way handshake, the token-based asynchronous FIFO has lower transmission latency; however, it consumes more routing resources because, instead of just the transmitted data, all entries are routed from TCD to RCD.

2.9 Implementation Results

A 4.1mm^2 test chip is implemented in a 40nm CMOS process, and the core design occupies 2.56mm^2. The chip microphotograph overlaid with the floorplan is shown in Figure 2.15. We use a mixture of 80.5% high-V_T and 19.5% standard-V_T cells to reduce the logic leakage power by 33% (8.3mW). Dynamic clock gating is applied to reduce the dynamic power by 24% (52mW). A total of 49 VCOs, 48 for neurons and one for the hub, are instantiated, with each VCO occupying only 250μm^2. 4×4 configurable convolver and 4×4 NZ detection are implemented, supporting odd kernel size from 5×5 to 15×15; the maximum supported kernel size is limited by the on-chip memory resource, not by the proposed maze convolution. The chip achieves

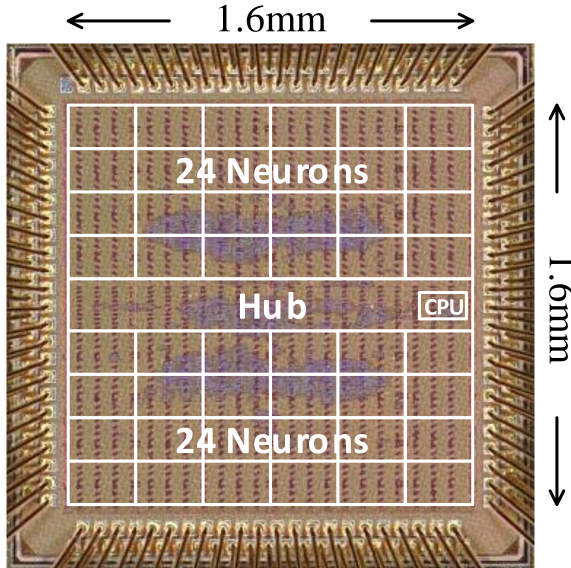


Figure 2.15: Chip microphotograph.

a maximum 718GOPS running at 380MHz with a nominal 0.9V supply at room temperature; here we define an OP as an 8-bit multiply or a 16-bit add.

Figure 2.16 shows the power and area breakdown of the test chip. As shown in Figure 2.16 the hub typically has more workload than individual neurons, and our experiments show that a balanced clock setting, in which neuron clock frequency is lowered to 70% of hub clock, can further reduce the power consumption by a maximum of 22% with less than 5% drop in the throughput. Maze convolution and zero-patch skipping require a more complex controller, and yet this overhead is within 15% of the power and area budget for a neuron We use two sample applications to demonstrate the proposed accelerator: extracting sparse representation of images and extracting depth information from stereo images.

To extract sparse representation, we ran the accelerator with 7×7 kernels for 10 iterations per input image. The inference result, i.e., the collected neuron spikes, represents the sparse representation of the image. Kernels are trained using unsupervised learning with the natural scene image dataset provided in [1]. When running at 380MHz with a 0.9V supply, the accelerator consumes 195mW while providing 24.6M

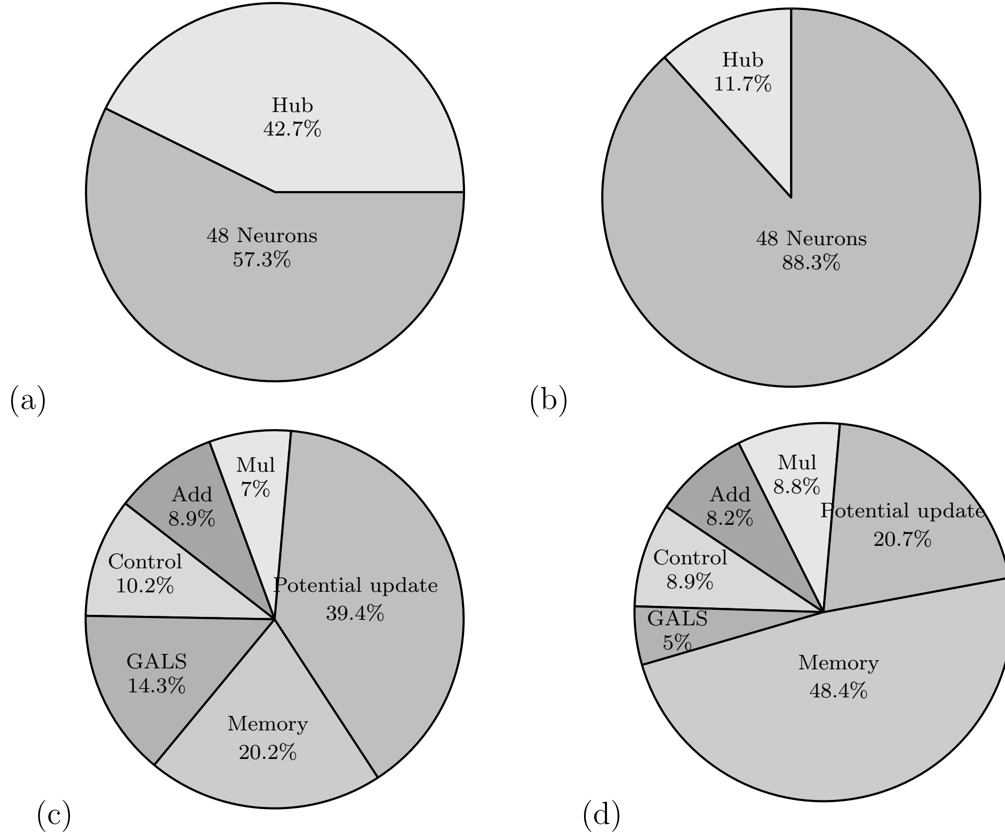


Figure 2.16: Chip power and area breakdown. (a) power breakdown of the entire chip. (b) area breakdown of the entire chip. (c) power breakdown of a single neuron. (d) area breakdown of a single neuron.

pixel/s throughput. Running at 120MHz with a 0.6V supply, the power consumption is lowered to 35mW, as shown in Figure 2.17, providing 7.6M pixel/s throughput.

To estimate depth from stereo images [38], we use one accelerator per channel (left or right) to extract features. Both accelerators use 15×15 kernels and run for 10 iterations per input image. Kernels are learned by training on the natural scene image dataset. A simple matching algorithm can be programmed on the on-chip OpenRISC processor to estimate depth by comparing the distance between the locations where the same feature appears in the two channels. When running at 380MHz with a 0.9V supply, each accelerator consumes 257mW while providing 7.68M pixel/s throughput. Running at 120MHz with a 0.6V supply, the power consumption is lowered to 45mW, as shown in Figure 2.17, providing 2.4M pixel/s throughput. Compared to

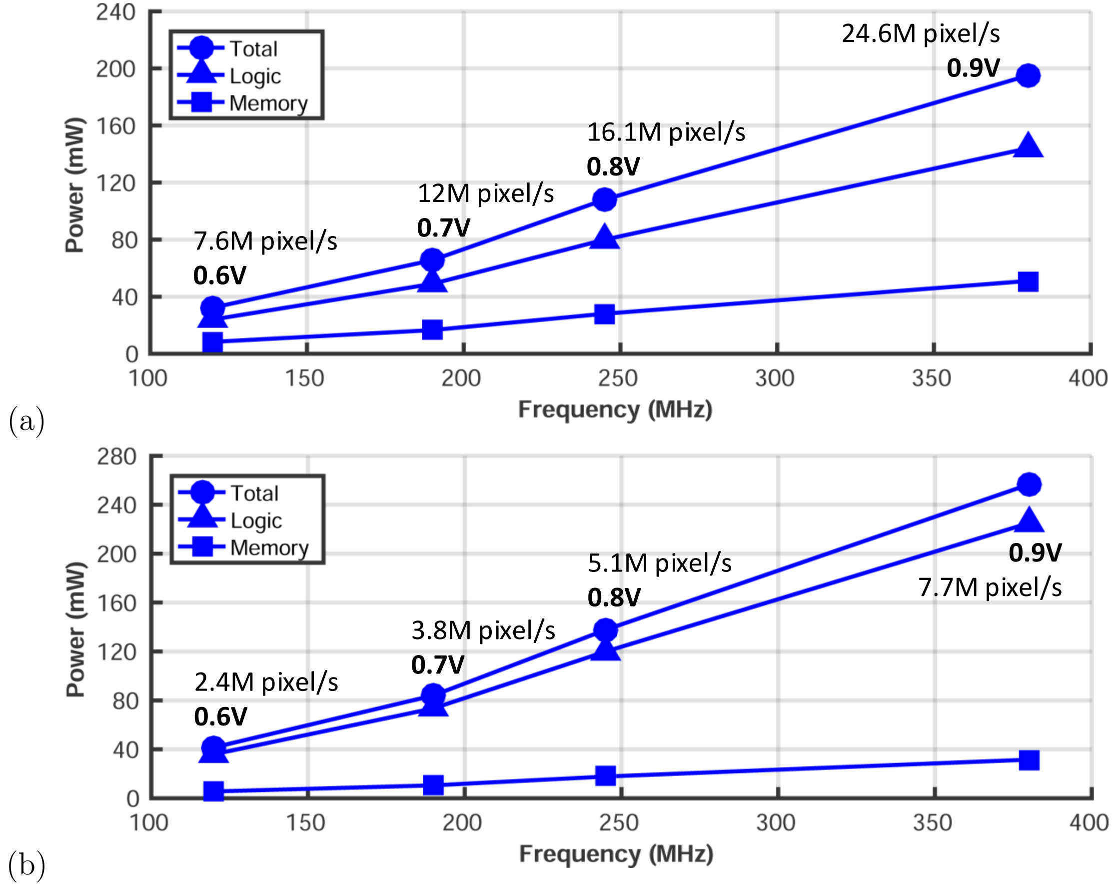


Figure 2.17: Measured power and throughput of (a) feature extraction application, (b) depth extraction application.

the optimal baseline designs without exploiting sparsity, the throughput of the two tasks is improved by 7.7× and 9.7×, respectively.

2.10 Conclusions

In this work, we present a configurable convolver, maze convolution with sparsity optimization, and modular architecture and load balancing enabled by GALS. Using these techniques, we designed a convolutional RNN that implements convolutional sparse coding for a range of applications. The techniques are equally applicable to CNNs.

Compared to other state-of-the-art inference accelerators, the proposed accelera-

Table 2.3: Comparison With Prior Work

	This Work	VLSI 2015 [4]	VLSI 2016 [11]	JSSC 2016 [27]
Architecture	Recurrent	Recurrent	Feedforward	Feedforward
Algorithm	Convolutional sparse coding	Sparse coding	Convolutional neural network	Convolutional neural network
Technology	40nm GP CMOS	65nm GP CMOS	40nm GP CMOS	65nm LP CMOS
Configurable	Yes	No	No	Yes
Exploit Sparsity	Yes	No	Yes	No
Core Area	2.56mm^2	1.82mm^2	1.4mm^2	12.25mm^2
Kernel Size	5×5 to 15×15 (odd only)	16×16	8×8	Width: 1 to 32 Height: 1 to 12
Image Size	32×32	16×16	100×100	configurable
SRAM Size	120KB	37.6KB	93KB	181.5KB
Weight Width	8-bit	4-bit	8-bit	16-bit
Voltage	0.9V	1V	0.9V	1V
Frequency	380MHz	635MHz	240MHz	200MHz
Performance	718GOPS	1138GOPS	898.2GOPS	33.6GOPS
Throughput†	24.6M pixel/s	10.16G pixel/s	96.4M pixel/s	1.8M pixel/s
Power	257mW	268.2mW	140.9mW	278mW
Power Efficiency	2.79 TOPS/W	4.24 TOPS/W	6.37 TOPS/W	0.12 TOPS/W
Area Efficiency‡	280 GOPS/mm^2	1.65 TOPS/mm^2	641.6 GOPS/mm^2	7.24 GOPS/mm^2

†Based on different benchmark due to the lack of a common benchmark. [4] reported input throughput while others reported output throughput.
‡Normalized to 40nm by multiplying the square of the ratio between node sizes.

tor implements a convolutional RNN that is configurable to support different kernel sizes, and exploits sparsity to increase throughput and efficiency. Compared to an ASIC implementation of sparse coding based on fully-connected RNN [4], this design is scalable, configurable and applicable to a wide range of applications. Compared to popular feedforward CNNs [11, 27, 28], the convolutional RNN implementing sparse

coding can be more versatile as it supports unsupervised learning. The design sacrifices power efficiency and area efficiently by a small factor compared to [11] to gain configurability. Note that the supported image and kernel size in this work are limited by the available on-chip memory, not by the architecture, which in theory supports any image and kernel size. By exploiting sparisity, the power efficiency and area efficiency of this design easily overtake those of dense CNN accelerators [27, 28].

CHAPTER III

AIB Chiplet Design for 2.5D Integration in MCPs

In this chapter, a 16nm chiplet incorporating the standard-conforming AIB interface is presented. A multi-chiplet package (MCP) with two chiplets assembled on top of a silicon interposer demonstrates a viable way to construct scalable systems by tiling chiplets of the same type. The inter-operability of the AIB interface, which is critical to enabling heterogeneous integration, is verified in a MCP that integrates the chiplet with an Intel 14nm Stratix-10 FPGA using EMIB technology. Running at 1GHz with a 0.9V supply, the measured energy efficiency of the implemented AIB interface is 0.83pJ/b.

3.1 Background

Designing and fabricating a large-scale IC requires a tremendous team effort and high cost. It also incurs a high risk of failure due to the high complexity. Process scaling escalates the design effort, cost and risk of failure, which limits the access to the most advanced technology nodes to only a handful of large companies that produce high-volume consumer electronics. Aside from the high design effort and fabrication cost, a large-scale monolithic IC design requires compromises in the process technology to accommodate different functions, e.g., digital vs. analog, logic vs. memory, resulting in sub-optimal performance in at least some of the building blocks.

Fabrication of a large-scale monolithic IC also incurs a lower yield, resulting in an increased cost and longer time-to-market.

A modular design approach breaks down a large-scale IC to individual IPs that are independently designed and reused, enabling a higher productivity. Reuse of soft IPs, in the forms of hardware description language (HDL), schematics, or layouts, is often done at the chip level. However, chip-level soft IP reuse does not solve the high cost and low yield problems associated with large-scale monolithic ICs as described above. Chip-level IP integration is also faced with challenging system verification problems that frequently arise due to nonconforming standards and mismatches of IP specifications.

Another common modular design approach is hardware IP reuse at the package or board level. Package or board-level IP reuse offers the flexibility of integrating heterogeneous components, each separately designed and fabricated in the most cost-effective and/or performance-optimal technology node, as well as a quick turnaround time and high yield thanks to a mature process. Despite these benefits, package or board-level hardware IP integration does not meet the density and performance required by the most demanding applications. More specifically, package or board-level integration limits the communication bandwidth between IPs, and ultimately limits the system performance.

A 2.5D side-by-side integration of hardware IPs on interposer offers a promising approach towards hardware IP reuse in constructing large-scale systems to deliver a performance comparable to monolithic integration, but without the high risks, cost and effort of monolithic integration. Successful commercial products using 2.5D integration have already emerged, including High Bandwidth Memory (HBM) [39, 40, 41], and high-performance FPGAs [42, 43]. A dual-SoC-chiplet was demonstrated on CoWoS using an 8Gb/s/pin low-voltage in-package-interconnect [44], and a 36-chiplet DNN accelerator was demonstrated on an organic substrate using a 25Gb/s/pin

ground-referenced signaling [45, 46]. These products have demonstrated significant improvements in functionality, performance and power that directly benefit from the 2.5D integration. However, what is missing in the existing work is the lack of a common interface standard, building blocks, as well as standard-conforming IPs, preventing the rapid buildup of large-scale systems.

3.2 AIB Interface

This work adopts the open-source Advanced Interface Bus (AIB) GEN1 [14] as the chiplet-to-chiplet interface standard. AIB transfers parallel digital data with forwarded source clock over a short distance (<3mm) between chiplets at 2Gb/s/pin. The relatively low-speed, short-reach AIB interface follows standard digital design, with a lower complexity, cost, and a better portability than high-speed serial interfaces. The simplicity of the AIB interface leads to over 10× shorter latency than a serial interface [14] and yet a competitive energy efficiency. Supported by advanced packaging technology including microbumps (μbumps) and silicon interposer, the AIB interface rivals the competing serial interfaces in terms of silicon area and bandwidth density.

AIB specifies a maximum 1GHz clock frequency on the interface. Data is transmitted in double data rate, resulting in a throughput of 2Gb/s per pin. One AIB channel includes 20 pins for transmitting and 20 pins for receiving, achieving a total throughput of 80Gb/s per AIB channel. From the application point of view, an AIB channel can be viewed as a FIFO connecting the chiplets. One chiplet pushes data at one end of the FIFO, and the other chiplet pops the data at the other end.

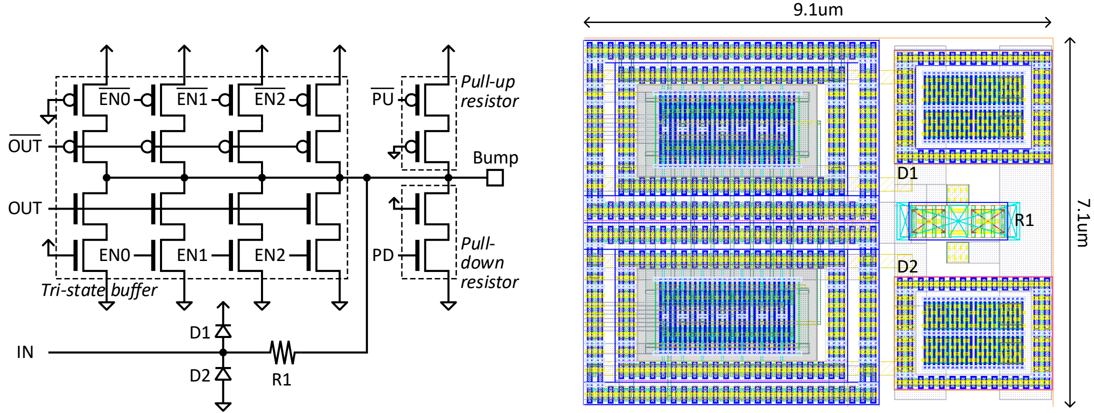


Figure 3.1: Schematic design and layout view of the I/O driver.

3.3 I/O Cell Design

An I/O cell encapsulates the low-level details of the I/O operation. On one side, it is connected to the core logic that transmits and receives data in single data rate (SDR). On the other side, it is connected via the bumps to the external wires, on which signals are transmitted and received in double data rate (DDR).

In Section 3.3.1, the design and implementation of a driver with configurable driving strength is presented. A serializer and a de-serializer are implemented to perform the data rate conversion between the core and the AIB interface, which is discussed in Section 3.3.2.

3.3.1 I/O Driver Design

Figure 3.1 shows the schematic and layout of the I/O driver. The main component on the transmitting path is the tri-state buffer that drives the signal to the bump. This tri-state buffer is designed with four driving strength modes. More transistors are enabled when switching to a stronger driving strength mode. The number and sizes of the transistors are determined by running SPICE simulations. Since the distance between chiplets is very short, for simplicity inductance was not considered in the simulation. In the simulation, the far side of the interface is a RC load with values

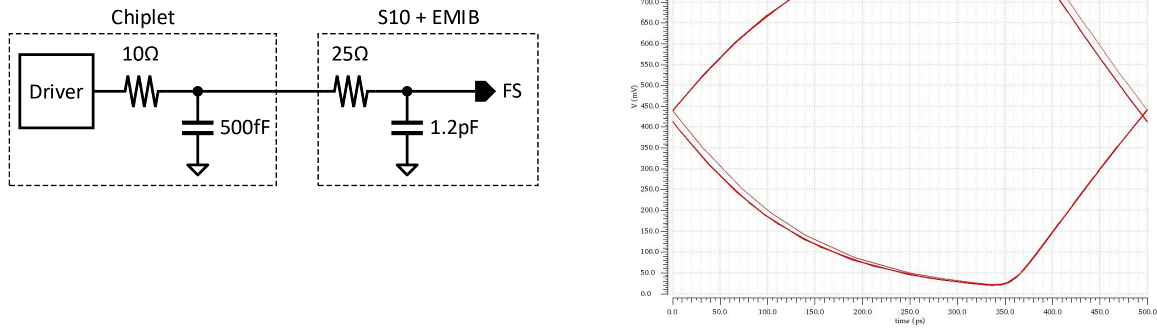


Figure 3.2: I/O driver test bench setup and the simulated eye diagram on the far side (FS) of the interface.

provided by Intel (as shown in Figure 3.2), which models the resistance and load capacitance of an I/O on the Stratix-10 FPGA. The near side is a RC load based on an estimation of our physical design result. To make sure the driver is strong enough to drive the heavy load on the FPGA, we sized the transistors such that they are capable of driving the far side signal with full-swing as shown in Figure 3.2.

A weak pull-up and pull-down resistor is connected to the transmitting path, between the tri-state buffer and the bump, allowing us to set the driver output to a known state when no one is driving the link. By default these pull resistors are disabled to avoid contention.

On the receiving path, a buffer is used to receive signals from the bump. For a standard I/O design, ESD protection circuit must be added to protect transistor's gate terminal that is connected to the bump from an ESD event that can easily punch-through the gate oxide. Unlike standard I/Os, which are designed for off-package connection, AIB I/Os are always in-package so the ESD protection requirement is not as strict as the standard I/O. In this design, we followed guideline by Intel and added a small-valued ESD cell and a current-limiting resistor. For AIB I/O, latch-up is less likely to happen compared to the standard I/O because AIB I/O operates at

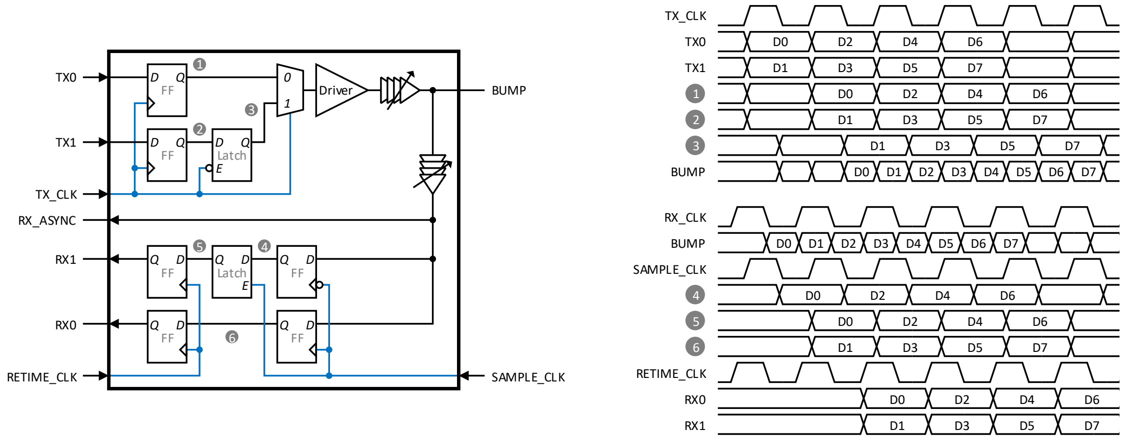


Figure 3.3: Serializer and de-serializer logic design and waveform.

0.9V, which is considerably lower than the standard 1.8V-3.3V I/O voltage. However, to be on the safe side, we followed TSMC's guideline and placed double guard-rings around the PMOS transistors and the NMOS transistors to prevent latch-up from happening.

The size of a driver is 9.1 µm × 7.1 µm, and it is clear in Figure 3.1 that most of the driver area is occupied by the ESD circuit and the guard-rings, meaning that the driver can be made much smaller if ESD and latch-up event can be guaranteed not to happen.

3.3.2 I/O Logic Design

On the transmitting path, a 2-to-1 serializer converts the 2b SDR data received from the AIB adaptor (Section 3.4.1) to a 1b DDR data and then sends it to the driver. The serializer is implemented with two flip-flops, one latch, and one mux, and the serialization waveform is shown in Figure 3.3. AIB requires clocks to be sent along the data, which is implemented by feeding the clock to the I/O cell and tie TX0 and TX1 inputs to 0 and 1, respectively. As shown in Figure 3.3, the transmitting latency in the I/O cell is 1.5 clock cycles.

On the receiving path, a de-serializer converts the 1b DDR data received from the

driver to a 2b SDR data and returns it to the AIB adaptor. Two clocks are used in receiving the data. The sample clock is used to capture the data received from the driver, and the retime clock is used to transfer the captured data to the AIB adaptor. As shown in Figure 3.3, there's a phase difference between the two clocks to ensure correct timing, which will be discussed in Section 3.4.1. The receiving latency in the I/O cell is 1.5 clock cycles. Combined with the transmitting latency, the total latency of the interface is 3 clock cycles plus the signal propagation delay between chiplets. With a 1GHz interface clock, the total latency is approximately 4ns.

A configurable delay chain, with a maximum delay of approximately 500ps, is inserted to both the transmitting path and the receiving path for post-silicon I/O timing fine-tuning. As will be discussed in Section 3.4.1, this delay chain is also used to create the phase difference between the sample clock and the retime clock.

3.4 AIB Channel

Figure 3.4 illustrates the block diagram of an AIB channel implemented in this work. A channel consists of three functional blocks: AIB I/O block, AIB adaptor, and the application hardware block. I/O cells for data and clock pins are instantiated in the I/O block. The application block includes an OpenRISC-based SoC and a built-in self test agent. A conversion layer, called AIB adaptor, is inserted between the I/O block and the application block to handle clock-domain crossing and data width conversion. An on-chip ring oscillator is used to generate clocks for the channel. A channel is configured to be either a master or a slave. When connecting two chiplets by the AIB interface, one is configured as the master and the other needs to be the slave. Each channel has an UART interface that can be used to read and write configuration registers for debugging purposes.

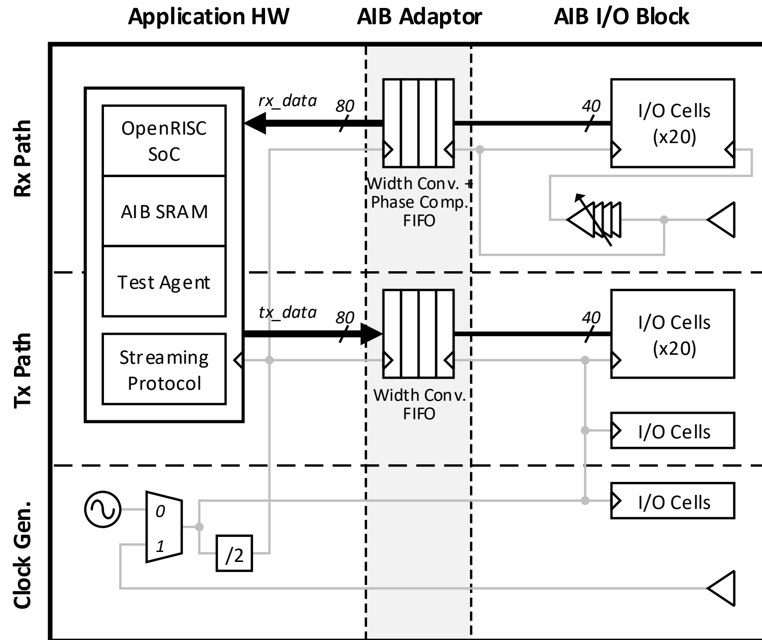


Figure 3.4: Block diagram of an AIB channel.

3.4.1 AIB Adaptor

AIB adaptor plays a critical role in the channel as it handles the details of clock-domain crossing and data width conversion, and presents a clean and easy-to-use synchronous data interface to the application block. AIB is a source-synchronous clocking interface, meaning that the clock used to generate the transmit data is also transmitted on the interface for the receiver to sample the data. The transmit clock exists on both the master and the slave chiplet. In addition to the transmit clock, the master chiplet also sends out its system clock to the interface. This clock is called the forwarded clock and only the master chiplet will output this clock.

Figure 3.5 shows the clocking scheme when two chiplets are connected by the AIB interface. In the master chiplet on the left, the ring oscillator generates a 1GHz transmit clock that is used to transmit data on the AIB interface, and a divided 500MHz core clock that is used by the SoC. In this chiplet the 1GHz clock is also used as the system clock that is being forwarded from the master to the slave. The core clock and the transmit clock are synchronous to each other since they are generated

Figure 3.5: Clocking scheme when two chiplets are connected. The left chiplet is the master and the right chiplet is the slave.

from the same clock source, therefore, a single register stage would be sufficient for the data to cross clock domains if the core clock and the transmit clock are the same frequency. In this chiplet the core clock is fixed to $\frac{1}{2}$ of the transmit clock. In order to maintain the same throughput, bus width in the core clock domain is $2\times$ the transmit clock domain, as a result, a width conversion FIFO is needed to convert the data width between the two clock domains.

In the slave chiplet on the right, the ring oscillator is bypassed and the forwarded clock sent from the master is used as the clock source to generate the core clock and the transmit clock on the slave chiplet. Doing so ensures the master and slave chiplets are synchronized with each other. Since the core clock and the transmit clock on the slave chiplet are also generated from the same clock source, the same clocking logic can be used when the chiplet is used as a slave.

The receiving logic is also the same for chiplet in either master mode or slave mode. To sample the received data in double data rate at the middle of the data eye, the sample clock needs to be have a $\frac{1}{4}$ cycle of delay. The delay chain in the I/O cell that received the clock is used to generate this delay. On the receiving path, the transmit clock from the other chiplet is delayed by the delay chain, and this delayed clock becomes the sample clock that is used to sample the data at the first stage of

the I/O cell. The second stage of the I/O cell is clocked by the retime clock, which is the un-delayed version of the received transmit clock. A width conversion FIFO is also needed to convert the data width from the AIB clock domain to the core clock domain. In addition, this FIFO also allows reading with extra cycles of delay to compensate for the potential phase difference between the retime clock and the core clock.

3.4.2 Test Agent

A controller called test agent is designed to perform two critical tasks: debugging and stress testing for the AIB channel. The test agent can be programmed with a loop count and a pattern with 8 data entries. Once started, the test agent on a chiplet that is configured in master mode will send data to the AIB channel repeatedly with the programmed data pattern. The test agent expects to receive the same data from the AIB channel, so the slave chiplet on the other side of the AIB channel needs to return the data it receives from the AIB channel. When the slave chiplet is the same chiplet, the test agent on the slave chiplet will directly loop-back the data within the AIB channel.

Two types of failure may happen: the received data does not match the transmitted data, and timeout. If the received data is not the same as the transmitted data, the test agent will record the received data and stop. The recorded data can be read out via the UART interface for debugging purpose. If the test agent does not receive any data within 256 cycles, it will set the timeout flag and stop.

During the initial bring-up stage, the test agent can be used to test if the AIB channel works at a lower clock frequency. Running the AIB channel at a lower clock frequency allows us to focus on the fundamental aspects such as connectivity and logic operation. After the AIB channel has been verified at a lower clock frequency, the test agent can be used again to test the AIB channel running full speed at 1GHz.

The test agent will send data every clock cycle, thereby stressing the AIB channel by pushing the bandwidth to the maximum.

3.4.3 Transfer Protocol

The AIB specification only specifies how the data is transferred on the AIB interface, and it is up to the user to decide how this data interface is used. In this work a low-overhead protocol is designed to allow chiplets to stream data through the AIB channel. The MSB in the 20 data pins is reserved as the valid flag. This bit is set to 1 if the transmitter is sending a data, otherwise this bit is set to 0. The receiver can easily tell if a valid data is being transferred by checking the valid bit.

3.4.4 Open-RISC SoC

Each AIB channel in this work includes an OpenRISC-based SoC as shown in Figure 3.4. The OpenRISC processor is connected to an on-chip bus, allowing it to access instruction memory, data memory, peripherals, and the AIB channel. On-chip peripherals include an UART for communication, and a timer to support real-time operating system porting. The instruction and data memories share a single 64KB SRAM. Another memory, called AIB memory, is directly connected to the AIB channel, meaning that it is only accessible from another chiplet via the AIB channel. The SoC also includes a debug UART interface, which translates UART data to APB commands, allowing us to read and write configuration registers in the SoC part of the chiplet.

The OpenRISC processor uses a bus interface protocol called Wishbone. To bridge the processor and the AIB channel, a converter is designed to take the command and data received from the processor and the AIB channel, and pass it to the AIB channel and the processor, respectively. For example, when the processor sends a write request to the AIB channel, the converter will pack the write command and data into a packet

1	0	WDATA[31:19]
1	WDATA[18:0]	
1	0	BYTE_EN	ADDR[31:18]
1	ADDR[17:0]	WE

Figure 3.6: One Wishbone write command packed into four 20b entries to be transmitted on the AIB interface.

of four 20b entries and pass the packet to the AIB channel as shown in Figure 3.6.

3.5 Chiplet design

A 2.5 mm × 2.5 mm chiplet is designed and fabricated in a TSMC 16nm FinFET technology. The chiplet includes three regular AIB channels that can be used in parallel, and one mini AIB channel with only 4 Tx and 4 Rx data pins for debugging and characterization purposes. Note that the silicon area is shared with other unrelated designs and the presented AIB design occupies less than half of the total die area.

3.5.1 Bump Map Design

The bump map of a chiplet specifies the types and coordinates of the bumps on the chiplet. The bump map design is part of the packaging design process, as the chiplet is attached to the package substrate via the bumps. Two types of package substrates are designed for this chiplet. The first substrate is a custom-designed silicon interposer (Section 3.6.1), with which we have the freedom in designing the bump map. The second substrate is Intel's EMIB (Section 3.7), which already specifies a bump map designed for the Stratix-10 FPGA.

A bump map is often exchanged in the form of a comma-separated value (CSV) file, which simply lists all the information in a text file. To help visualization, a tool is developed to facilitate displaying or modifying a bump map. The coordinates specified in a bump map is the design coordinates, which is not equivalent to the physical

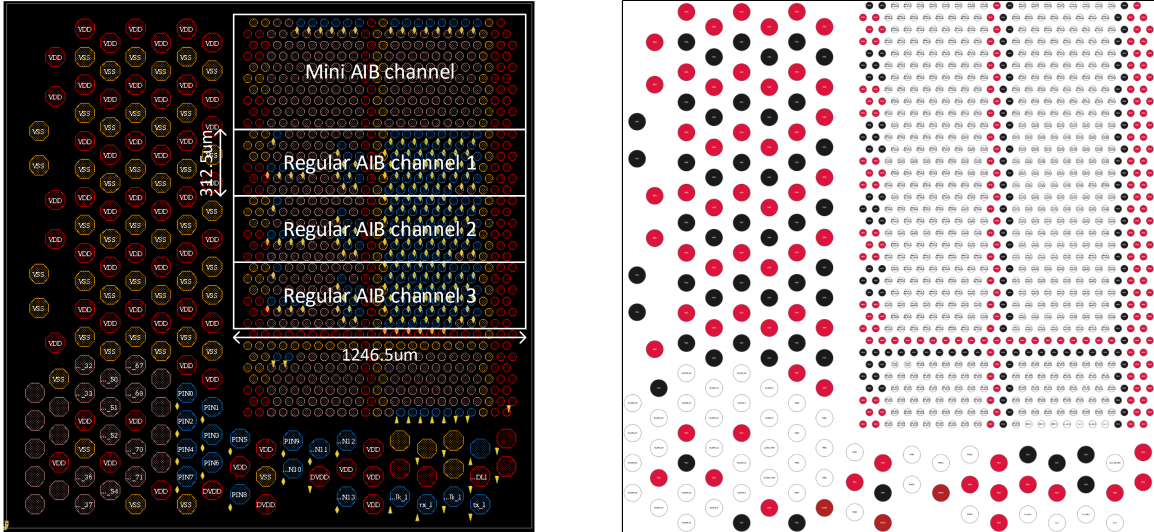


Figure 3.7: Chiplet bump map views in Innovus and the bump map design tool developed in this work.

coordinates if the fabrication process has a shrinkage. For example, TSMC 16nm technology has a 98% shrinkage, meaning that the physical coordinates are obtained by multiplying the design coordinates by 0.98. The tool is able to convert between design and physical coordinates with any shrinkage automatically. In addition, the tool can convert the bump map to various formats for different tools, such as the TCL format for Innovus and the SKILL format for Virtuoso.

Figure 3.7 shows the bump map of a the chiplet viewed in Innovus and the bump map design tool. The larger circles are the C4 bumps with bump size of 75 µm and bump pitch of 130 µm. The smaller circles are the μbumps with bump size of 31 µm and bump pitch of 55 µm. In the Innovus view, the red and yellow circles represent power and ground bumps, respectively. Blue circles are the I/O bumps, whereas the pink circles are unused bumps. Some bumps in a channel are unused, and they are not required for the standard AIB interface.

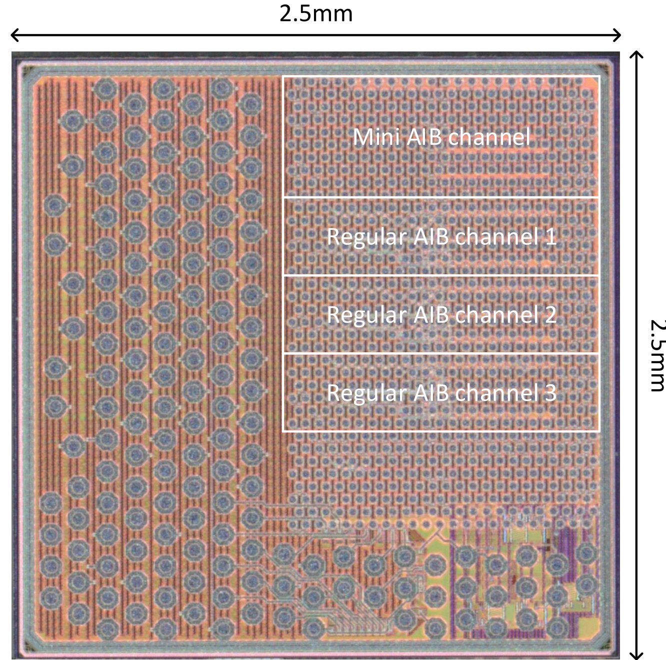


Figure 3.8: Chiplet die photo.

3.5.2 AIB Channels in the Chiplet

Two variants of the AIB channel are implemented in this chiplet. The regular AIB channel has 20 Tx and 20 Rx data pins as specified in the AIB spec. A mini AIB channel with only 4 Tx and 4 Rx pins is implemented and placed on the top right corner of the chiplet such that all 8 data pins are on the shoreline of the chiplet. Compared to the regular AIB channel, putting data pins on the edge makes it easier to make connections to these pins on the substrate. The two variants are synthesized and place-and-routed separately, and a library exchange format (LEF) file for each variant is generated and used in the top level integration.

Within each AIB channel, in order to minimize the skew between data pins, the I/O cells are placed next to the bumps such that the routing distance from the I/O cells to the bumps are kept short and of the same length for all I/Os. This is possible because of the following two reasons. First, the I/O cell area is small enough to fit in the space between μbumps. Second, the entire area enclosing the bumps for one

channel, which is 312.5 µm × 1246.5 µm, is allocated for the AIB channel. In cases when these conditions can not be met, I/O cells will need to be grouped and placed carefully to balance the skew between data pins.

The three regular AIB channels and the mini AIB channel operate independently in parallel. Each AIB channel requires 11 pins for control and debug. These pins need to be connected to the C4 bumps, however, the total number of C4 bumps available is only 14. In addition, the silicon area is shared with other designs, which also need access to the C4 bumps. An I/O mux for these low-speed I/Os is therefore created, allowing one of the design partitions to be active at any moment. Figure 3.8 shows the die photo of the chiplet.

3.6 Homogeneous Integration Based on Silicon Interposer

In this chapter, a multi-chiplet package consists of two chiplets assembled on top of a silicon interposer is constructed as a proof-of-concept for systems that can be built by tiling chiplets of the same type. This kind of integration is particularly suitable for accelerating computation with regular structures, such as the convolutional neural network commonly used in AI applications, and the FFT computation that is widely used in signal processing applications.

3.6.1 Silicon Interposer

The silicon interposer is fabricated in a TSMC 180nm technology (Figure 3.9). No active or passive device is used in the interposer, so the front-end-of-line (FEOL) process for manufacturing devices is not required and only the back-end-of-line (BEOL) process is used to fabricate the three metal layers: metal 1 (M1), metal 2 (M2), and top redistribution layer (RDL). Power and ground connections are routed mainly on the RDL layer since it has the lowest resistance. Signals are routed on M1 and M2 with a minimum width of 2.5 µm and a pitch of 4.7 µm.

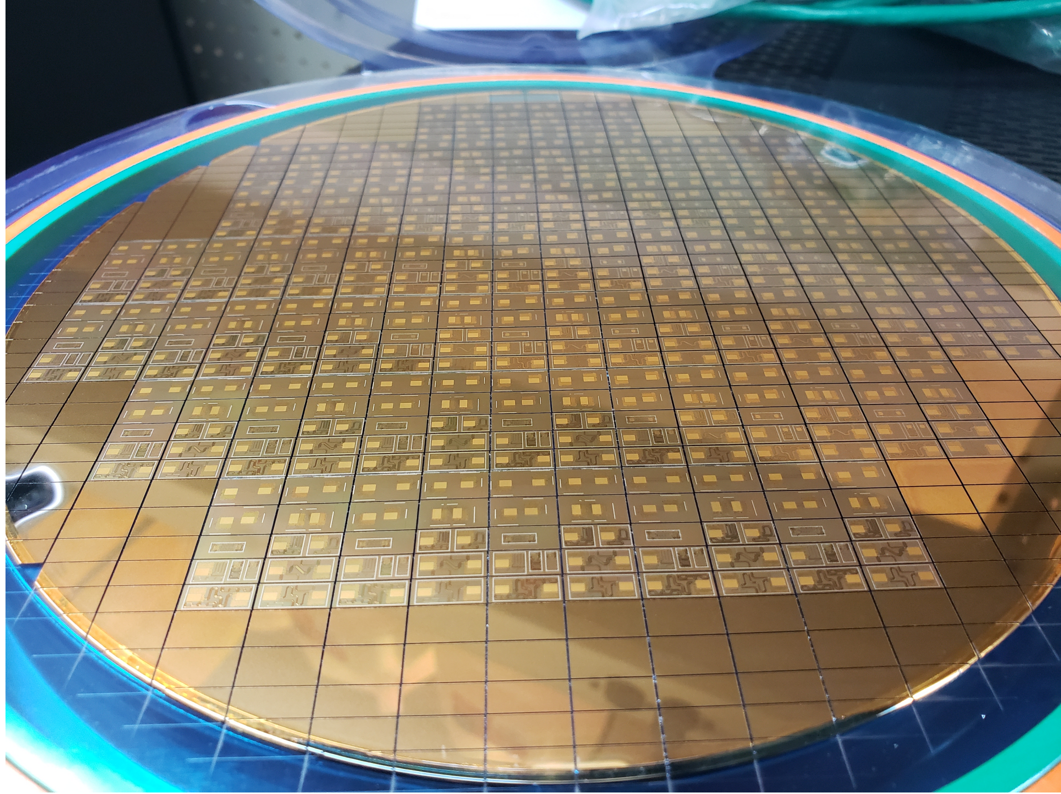


Figure 3.9: Silicon interposers on a 180nm wafer.

Two interposer patterns are designed in this work. The first pattern is designed to test the mini AIB channel that has only 4 Tx and 4 Rx data pins. As shown in Figure 3.10, in this pattern one chiplet footprint is rotated by 90 degrees to align the mini AIB channels on the edge of the chiplets. In order to equalize the routing delay, the signals are routed with a detour such that the routing length is kept the same 2 mm for all the connections, as shown in Figure 3.10. The size of the interposer is 7.6 mm × 5.1 mm, measured from the top/right bond pads to the left/bottom bond pads.

The second pattern is designed to test the regular AIB channels that have 20 Tx and 20 Rx data pins. Chiplet footprints in this pattern are placed side-by-side without rotation as shown in Figure 3.11. As described in the AIB specification, this placement will naturally lead to equal signal routing length. Routing channels in pattern 2 are more congested than in pattern 1 because of the larger number of

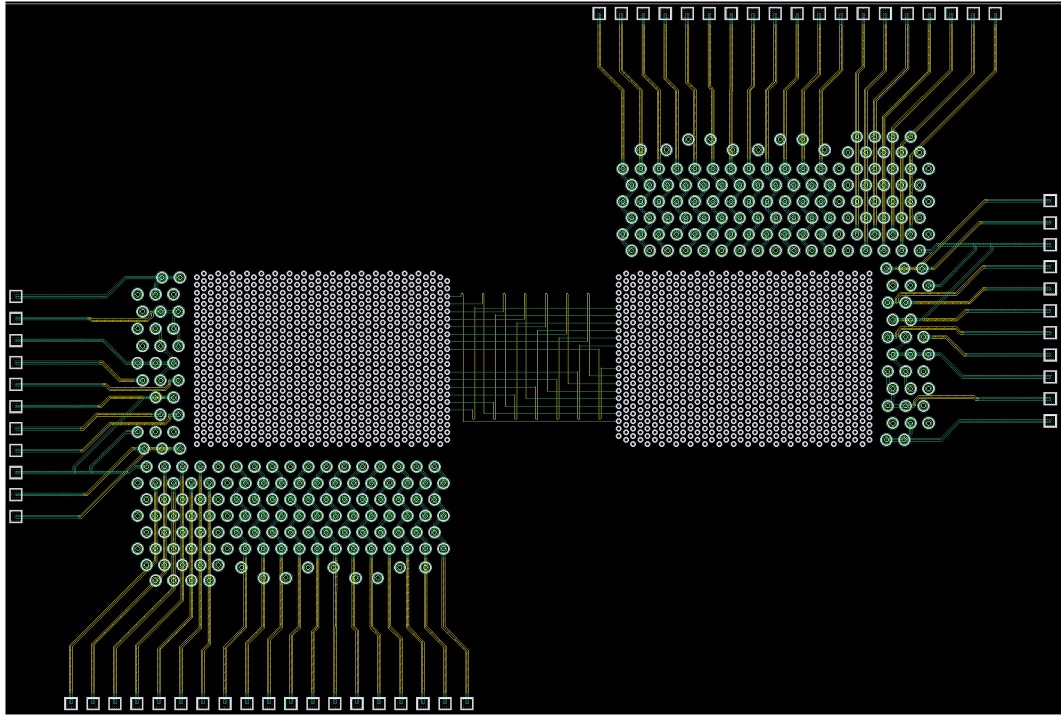


Figure 3.10: Interposer pattern for the mini AIB channel.

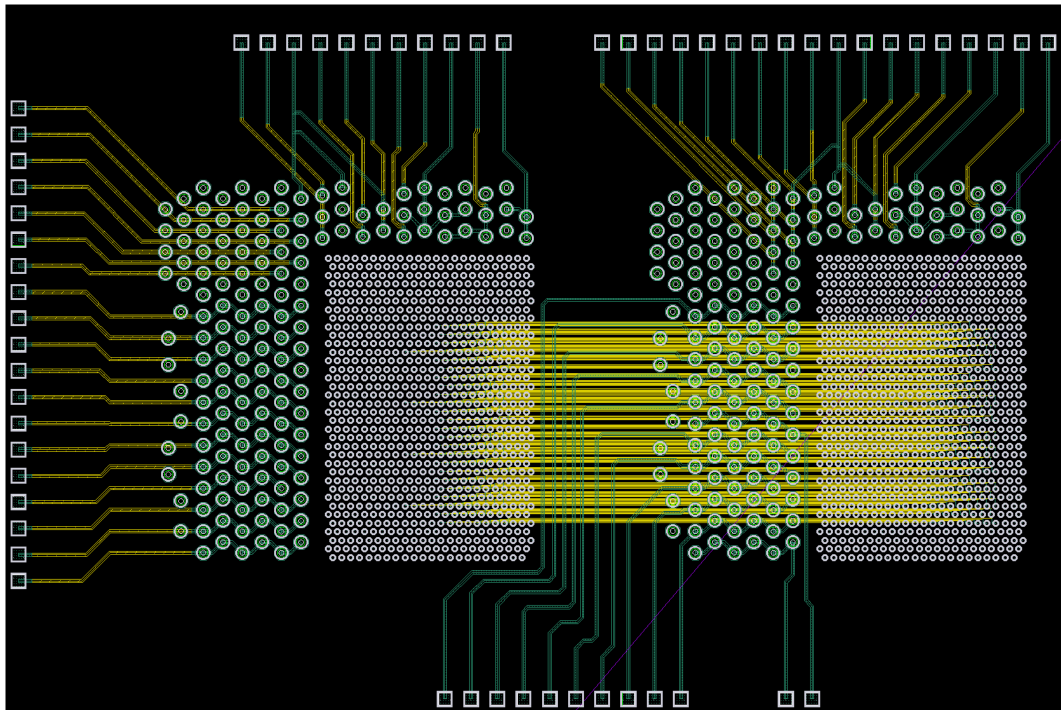
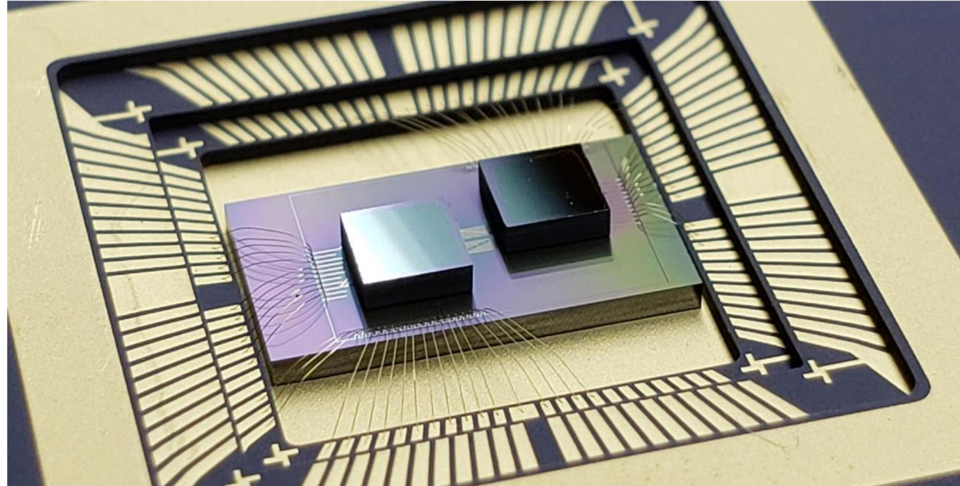


Figure 3.11: Interposer pattern for the regular AIB channels.



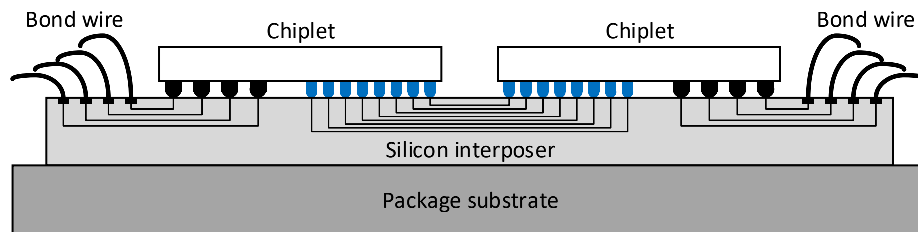


Figure 3.12: Silicon interposer based multi-chiplet package inside a PGA package.

signals that need to be routed, wires with width of 2.5 µm and pitch of 4.7 µm are used to route all the signals. The size of the interposer is 6.4 mm × 4.1 mm.

For both patterns, power, ground, and the low-speed I/Os are fanned-out to the periphery of the interposer. These are routed with a wide metal to the bond pads, which will be connected to the package via bond wires. Thanks to the regular structure of the chiplet footprint, a script is developed to route the signals automatically.

Two chiplets are assembled on top of the silicon interposer, and the interposer is packaged in a pin-grid array (PGA) package for testing (Figure 3.12). The actual interposer dimension is 9.8 mm × 5 mm, which is larger than the drawn pattern dimension because the wafer includes multiple designs and the wafer needs to be diced along the largest design. The selected PGA package has a 10 mm × 10 mm cavity for the interposer to fit inside. The bond pads on the interposer are connected to the package pins via bond wires.

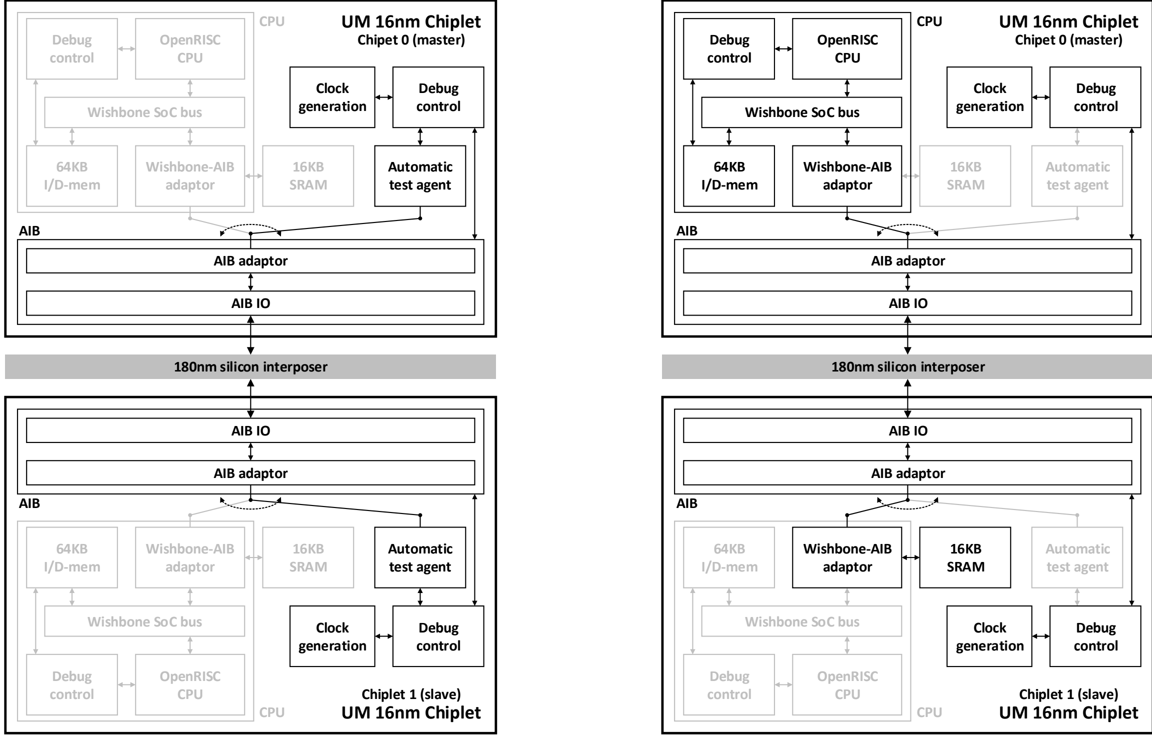


Figure 3.13: Chiplet testing configuration for multi-chiplet package based on silicon interposer.

3.6.2 Chiplet Testing

To verify the performance of the AIB channel implemented in this work, we ran a stress test with the test agent using the multi-chiplet package built with interposer pattern 1 (Figure 3.10). In this test, the test agent on the master chiplet is enabled, whereas the slave chiplet will return the data received from the AIB channel directly back to the channel, as shown in Figure 3.13. The default setting for the delay chains in the I/O cells works well up to around 500MHz. A script is developed to automatically sweep different settings for clock frequency higher than 500MHz. When running the AIB interface at 1GHz and the test agent at 500MHz, with 0.9V core voltage and 0.9V AIB I/O voltage at the room temperature, the measured energy efficiency of the AIB interface is 0.83pJ/b.

As a proof-of-concept for tiling chiplets, the OpenRISC processor on the chiplet

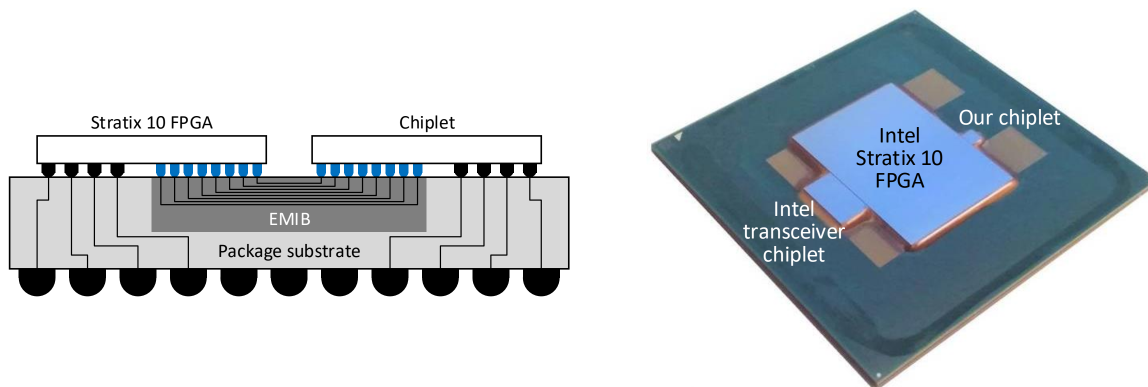


Figure 3.14: Chiplet integrated with an Intel Stratix 10 FPGA on the same substrate via EMIB.

is programmed to access the AIB channel using the multi-chiplet package built with interposer pattern 2 (Figure 3.11). In this test, the OpenRISC processor on the master continuously sends read and write commands to the AIB SRAM on the slave chiplet via the AIB channel. When running the AIB interface at 500MHz and the core at 250MHz, the OpenRISC processor always reads back the same data that it writes to the other chiplet via the AIB channel.

3.7 Heterogeneous Integration Based on EMIB

To verify the inter-operability of the AIB interface, the chiplet is packaged with an Intel Stratix 10 FPGA. The chiplet is connected to the FPGA on the same substrate via embedded multi-die interconnect bridge (EMIB), as illustrated in Figure 3.14. EMIB is an advanced packaging technology developed by Intel [13]. By embedding a small silicon bridge in the organic substrate, EMIB offers a cost-effective approach compared to other silicon interposer based integration technology such as TSMC's CoWoS, at the cost of increased complexity in the organic substrate manufacturing procedure. EMIB supports up to four routing metal layers, with a minimum 2 µm width and 2 µm pitch. As shown in Figure 3.14, a Stratix 10 FPGA has six EMIBs in total, each providing connections for 24 AIB channels, and our chiplet is assembled

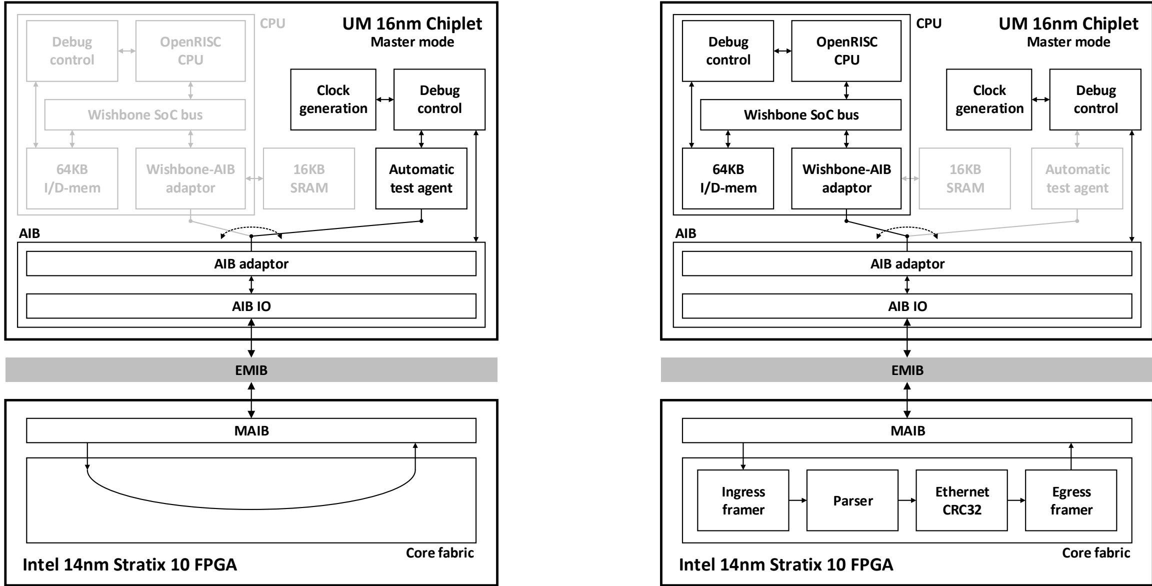


Figure 3.15: Chiplet testing configuration for multi-chiplet package based on Intel EMIB.

on the right middle EMIB.

When connected to the FPGA, our chiplet is the master and the FPGA is the slave. The test agent is first used to verify the function of the AIB channel that connects the chiplet and the FPGA. The FPGA is configured to return the data it receives from the AIB channel directly back to the channel. When running the AIB interface at 500MHz and the core at 250MHz, the test is always passed with the default delay chain setting. The maximum clock frequency for the FPGA internal logic is around 500MHz, as a result, using the clock forwarded from the chiplet as the FPGA system clock will limit the AIB clock to around 500MHz. To run the AIB clock on the FPGA at 1GHz, either an external clock source is used to generate clocks on the chiplet and the FPGA, or a half frequency system clock from the chiplet is forwarded to the FPGA via a non-standard clock pin in the AIB channel. However, both options are not supported by the chiplet, therefore, the maximum AIB interface clock frequency for this testing is set to 500MHz.

After the AIB interface is verified, a CRC acceleration design is constructed to

Table 3.1: Comparison Table of State-of-the-Art 2.5D Integration Technologies

	LIPINCON [44]	GRS [45]	This Work
Technology	7nm FinFET	16nm FinFET	16nm FinFET
I/O Voltage [V]	0.3	0.3	0.9
Bump Pitch [μm]	40	140	55
Reach [mm]	0.5	80	3
I/O Area [μm^2/b]	500	10175	203.2
I/O latency [ns]	-	<20	4
Data Rate [Gb/s/pin]	8	25	2
Energy Efficiency [pJ/b]	0.56	1.17	0.83
Shoreline BW Density [Gb/s/mm]	1600	354	256
Area BW Density [Gb/s/mm^2]	1600	516	614.4
Chiplet Substrate	CoWos 15-layer	Organic 12-layer	Silicon interposer 3-layer / EMIB 4-layer

verify the interface from the application point of view. In this test, the FPGA is configured with a CRC accelerator. The OpenRISC processor writes a block of 256 32b data to the FPGA via the AIB channel, and the CRC accelerator on the FPGA will calculate the CRC for the data. After sending the data, the OpenRISC processor calculates the CRC for the data block, and reads back the result from the FPGA for comparison. The experiment result shows 6.8× performance improvement when the CRC computation is off-loaded to the FPGA. The current Wishbone-to-AIB adaptor does not support burst transfer, and for simplicity it did not pack commands across the 20b boundary (see Figure 3.6). With an optimized Wishbone adaptor, the performance gain is expected to be up to 40×.

3.8 Summary

A standard-conforming AIB interface is designed and implemented in a 2.5 mm × 2.5 mm chiplet fabricated in a 16nm FinFET technology. The only technology-dependent part of the design is the I/O driver, meaning that the presented design can be readily ported to other technologies to enable rapid chiplet development and

deployment. A multi-chiplet package based on an interposer fabricated in a 180nm technology demonstrates an extensible way to construct a larger system by tiling multiple chiplets using the 2.5D integration technology. The 16nm chiplet is also verified with an Intel 14nm Stratix-10 FPGA, demonstrating AIB interface's interoperability between different devices, which is key to enabling heterogeneous integration of chiplets fabricated in different technology nodes.

Compared to other 2.5D integration technologies shown in Table. 3.1, this work achieves a competitive energy efficiency even though the operating voltage is at the nominal voltage, and it is higher than the other low-swing approaches. Our I/O area and latency are smaller than the other designs thanks to the simple parallel I/O design. Note that both LIPINCON and GRS are proprietary designs, whereas AIB is an open-source standard that is publicly available, making it a good candidate to be adopted as a chiplet interface standard.

CHAPTER IV

UMAI Chiplet Interface Protocol Design

In this chapter, an improved version of the AIB design is presented. When disambiguation is needed, the previous AIB design will be referred to as AIBv1 and the AIB design presented in this chapter will be referred to as AIBv2. As a proof-of-concept, AIBv1 is meant to be a simple design that can be implemented within a short time frame. AIBv2, on the other hand, is the result of an effort to push the design out as a chiplet interface IP that can be adopted by other chiplet designs. To achieve that goal, AIBv2 needs to present a clean, simple, and yet an efficient data interface to the users.

In Section 4.1, optimizations are applied to improve timing characterization of the I/O driver. A complete protocol stack designed based on the AIB interface is presented in Section 4.2, and we show the implementation result and conclude this chapter in Section 4.3 and Section 4.4, respectively.

4.1 I/O Driver Optimization

If the main driver, i.e., the tri-state buffer, is directly exposed to the other logic, which is the case for the AIBv1 driver, it is up to the place-and-route tool to insert pre-driver logic that is strong enough to drive the main driver. However, depending on the location and other factors, the tool may decide to use different logic for different

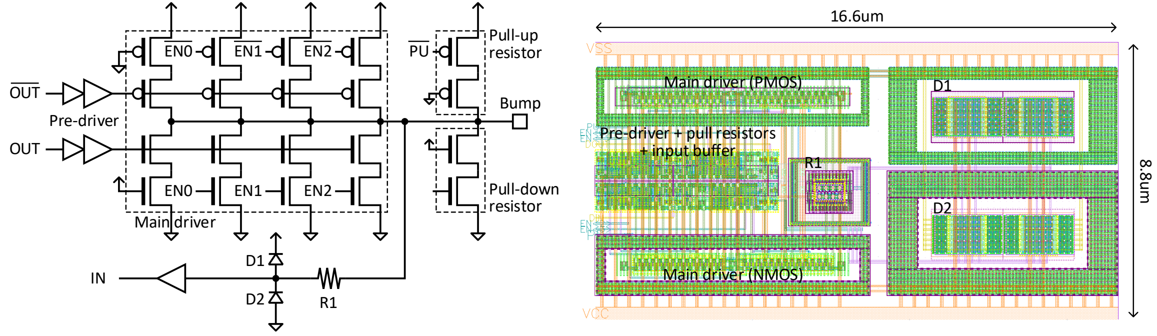


Figure 4.1: Schematic design and layout view of the AIBv2 I/O driver.

I/O pins, resulting in a delay skew between pins. To overcome this potential problem, as shown in Figure 4.1, a pre-driver stage is included before the main driver to improve the driver timing characterization. The transistors in the first stage of the pre-driver is $\frac{1}{4}$ of the transistors they drive, so that the pre-driver can be easily driven by any standard cell logic gate while still being capable of driving the main driver. The tri-state buffer and pull-up/down resistors are the same circuit design as the AIBv1 driver.

ESD circuit is also included in this driver to protect the gate of the transistors in the input buffer from ESD events. Single guard-rings are used to protect latch-up from happening. We reuse AIBv1 driver simulation testbench for this driver design, and the transistors are sized such that the eye diagram for this driver is roughly the same as the AIBv1 driver. As shown in Figure 4.1, the size of a driver is 16.6 µm × 8.8 µm. Again, just like AIBv1 driver, more than 50% of the area is occupied by the ESD circuit. The area of this driver is larger than the AIBv1 driver, however, this driver includes an internal pre-driver which is not included in the AIBv1 driver.

Note that the logical function of the I/O cell is the same as the AIBv1 I/O cell, so the serializer and de-serializer structure shown in Figure 3.3 remain the same. As a result, the latency of the AIBv2 I/O cell is the same as the AIBv1 I/O cell, which is 3 clock cycles plus signal propagation delay, or approximately 4ns when the AIB interface operates at 1GHz.

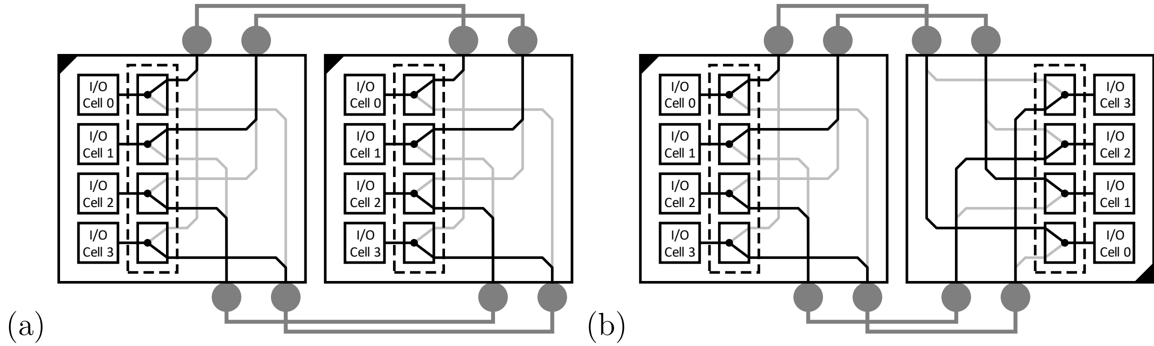


Figure 4.2: I/O remapping example with 4 I/O cells: (a) No I/O remapping when chiplets are not rotated. (b) I/O cells are remapped when the right chiplet is rotated.

4.2 AIB Channel

4.2.1 I/O Remapping for Chiplet Rotation

One difficulty that the AIBv1 chiplet encountered is the silicon interposer routing problem. As shown in Figure 3.11, the right chiplet is placed in the same orientation as the left chiplet in order to guarantee the same routing length on the substrate for all I/O pins. The same routing length requirement is defined in the AIB standard in order to to minimize the skew between pins. Because of this constraint, the two chiplets are placed in a back-to-face orientation, which requires extra routing length as the signals need to be routed through the C4 bumps of the right chiplet. As shown in Figure 3.10, rotating the right chiplet by 180° allows the chiplets to be face-to-face so that signals can be routed without having to go through one of the chiplets. With rotation, however, a signal routing needs to cross-over other signals to reach the matching pin on the other chiplet. The complexity of routing with cross-over for the mini AIB channel is manageable since it only has only 4 Tx and 4 Rx pins. But rotating the regular AIB channel requires crossing 20 Tx, 20 Rx, and the clock pins, which not only will congest the routing channel, but will also lead to signal integrity issues.

An extra stage designed to solve this problem by remapping I/Os is added between

the AIB adaptor and the AIB I/O cells. This remapping logic is essentially a collection of multiplexers that determines which I/O cell is connected to which bump. As opposed to the single connection between the I/O cell and bump in the AIBv1, the I/O cells are now connected to two bumps, one is used when the chiplet is not rotated, and the other one is used when the chiplet is rotated. When the chiplets are placed back-to-face as shown in Figure 4.2 (a), no remapping is needed and the I/O cells are connected to the default bumps. When the chiplets are placed face-to-face as shown in Figure 4.2 (b), the slave chiplet on the right is rotated and the I/Os on the slave chiplet are remapped. With the help of this remapping logic, routing for the AIB signals on the substrate is (1) simpler because there is no cross-over, and (2) shorter because there is no need to go through the back side of the chiplet. The remapping logic incurs negligible area overhead, however, special care must be taken to minimize the skew introduced to the on-chip routing due to the added fanout.

4.2.2 Word Marking and Multi-Channel Alignment

When testing the AIBv1 chiplet with the Stratix 10 FPGA, the maximum clock frequency on the AIB interface is about 500MHz because the FPGA's internal fabric can not run at a higher clock frequency. In order for the FPGA to run with a 1GHz AIB interface clock, the AIB adaptor on the FPGA needs to be configured in a mode called 2× mode. To support this mode, the chiplet needs to transmit a half frequency 500MHz clock in addition to the standard 1GHz clock. These two clocks are fed to the width conversion FIFO in the AIB adaptor on the FPGA. Similar to the width conversion FIFO in our AIB adaptor, the data width in the 500MHz clock domain is 2× the width in the 1GHz clock domain. And in this case, the FPGA requires word marking to be embedded in order for it to identify the lower half word from the upper half word. The lower half will have marking bit equals to 0, and the upper half will have marking bit equals to 1. The embedding of this word marking bit will be

TX1	WM	AF	PAYLOAD[35:18]
TX0	DV	0	PAYLOAD[17:0]

Figure 4.3: Data format as seen on the AIB interface.

discussed in Section 4.2.3.

The AIB channels in the AIBv1 chiplet are designed to be used independently. When a single AIB channel is not able to provide sufficient bandwidth for applications that require high inter-chiplet data transfer, multiple channels can be combined and used as a single channel with a bandwidth proportional to the number of channels in use. Data received from different AIB channels may not arrive at the same time because of the skew between channels, therefore, an alignment logic is needed to ensure proper data recombination. In AIBv2, channel alignment is fulfilled by allocating a FIFO in each channel to buffer the received data, and these FIFOs are read out at once when all FIFOs have valid data, which can be checked by ORing the non-empty status of all FIFOs.

4.2.3 Data Format on the AIB Interface

Figure 4.3 depicts how the 20 data pins of the AIB interface is used. Note that the AIB interface runs at 1GHz DDR, and TX0 and TX1 are transmitted at the positive and negative edge of the clock, respectively. DV stands for data valid, which is 1 when both TX0 and TX1 contain valid information sent out from the transmitter. WM is the word marking bit described in Section 4.2.2, and since TX0 and TX1 are concatenated by the AIB adaptor to form one 40b entry at 1GHz SDR, the word width marked by the bit is 40b. AF is the almost full status flag, which is set to 1 by the receiver to inform the transmitter that the receiving FIFO in the receiver's AIB adaptor is almost full so the transmitter needs to stop transmitting until this flag is clear.

4.2.4 UofM AIB Interface (UMAI) On-Chip Bus Interface

The streaming protocol in the AIBv1 chiplet is designed to be simple and light-weight. As a result, the utilization rate of the available bandwidth provided by the AIB channel is low because burst transfer is not supported, meaning that every data transfer needs to be coupled with the command (see Figure 3.6). Another problem with the streaming protocol is that it is built for a single AIB channel, so it's unable to handle the higher total bandwidth provided by combining multiple AIB channels. The goal of the new on-chip bus interface design, called UofM AIB interface (UMAI), is (1) to provide an efficient and clean interface for the user, (2) to support multiple AIB channels, and (3) allow channel direction to be configurable at run-time.

Inspired by the AMBA AXI protocol, UMAI's inteface to the application logic includes the following 3 bus groups: (1) command group, (2) write data group, and (3) read data group. Each group has a set of handshake signals, i.e., valid and ready, for flow control. The valid signal indicates a valid command or data has been placed on the bus by the transmitter, and the ready signal is for the receiver to indicate whether or not it can accept the command or data. The command group includes a 6b length signal, meaning that the protocol supports up to burst of 64 data for every command, reducing the command to data overhead from 1:1 (as in the streaming protocol in AIBv1) to 1:64. Both read and write data width is 512b regardless of how many AIB channels are actually used to transfer data. The packing and unpacking of data is handled inside the UMAI design, thereby freeing the application logic from having to deal with a variable data bus width.

Figure 4.4 shows the architecture of the UMAI design in a chiplet with one UMAI master and one UMAI slave built on top of 4 AIB channels. A UMAI slave (1) receives commands and write data from, and (2) returns read data to an user UMAI master. A UMAI master (1) outputs commands and write data to, and (2) receives read data from an user UMAI slave. UMAI master and slave are connected to the AIB channels

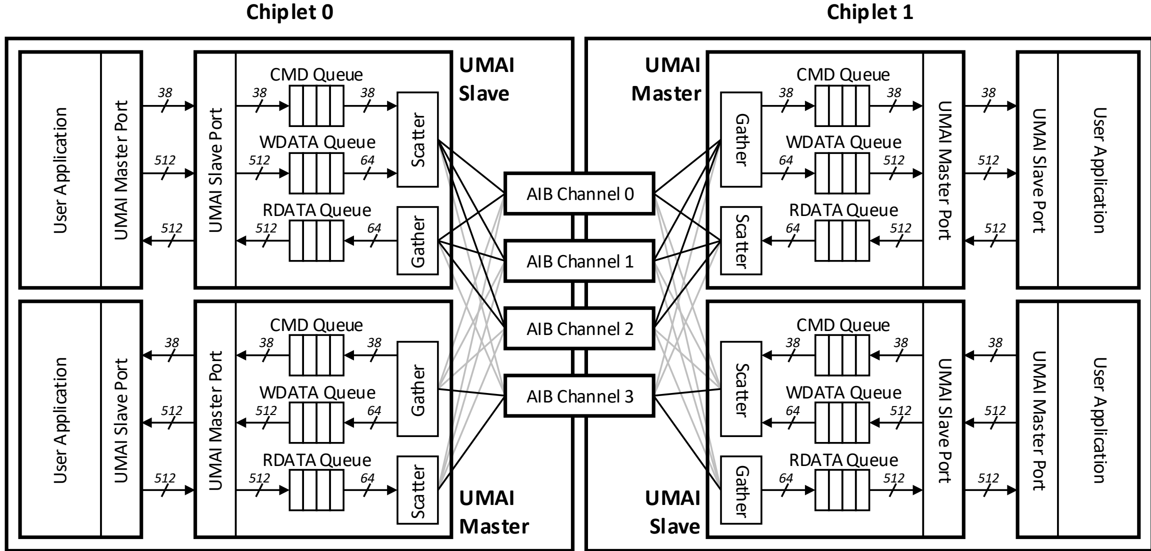


Figure 4.4: UMAI bridging two chiplets with 4 AIB channels.

via a configurable crossbar switch that allows splitting the total bandwidth between the two applications at run-time. In Figure 4.4, 3 AIB channels are assigned to the top user application and 1 AIB channel is assigned to the bottom user application, meaning that the top user application is given 3× the bandwidth of the bottom user application.

The UMAI slave includes a down-sizing write data queue that splits the 512b write data into 8 64b entries. A scatter logic reads the command and write data queue and distribute these to the AIB channels that are assigned to this UMAI slave. A gather logic reads the AIB channels and pushes the read data into the up-sizing read data queue that concatenates 8 64b entries to form 1 512b read data.

4.3 Chiplet Implementation

A 4 mm × 4 mm chiplet is designed and fabricated in an Intel 22nm FinFET technology. The bump map is shown in Figure 4.5. The light blue region is allocated for our design, and it includes 8 AIB channels on the left and 28 low-speed I/Os on the right. The AIB design presented in this chapter is used as an IP by two other

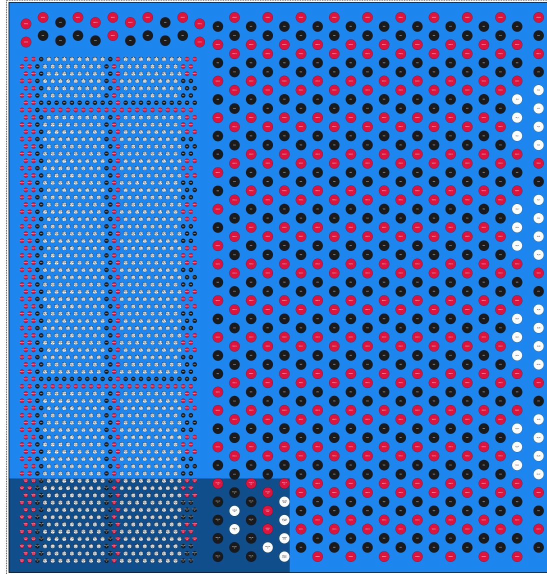


Figure 4.5: Intel 22nm chiplet bump map. Red, black, and white circles represent power, ground, and signals, respectively. Our designs are in the light blue region.

designs, and the functionality of the AIB design, including AIB I/O cell, AIB channel and the UMAI bus interface have been verified in silicon. The theoretical maximum bandwidth for 8 AIB channels is 640Gb/s. However, since for the most part only 64b out of the 80b is used to transfer data, the actual maximum bandwidth as seen from the application logic is approximately 512Gb/s, or equivalently 64GB/s.

4.4 Summary

In this chapter, an improved I/O driver in an Intel 22nm technology is presented, with pre-driver circuit included in the driver for better timing characterization. The simulated energy efficiency of this I/O design is 0.74pJ/b. To simplify routing on the interposer, I/Os are remapped on-chip to support face-to-face chiplet connection. Word marking bit and divided clock output are added to the AIB channel to support the 2$\times$ mode on the FPGA.

A chiplet data transfer protocol called UMAI is designed as an IP that provides

an AXI-like interface to the user application logic. Multiple AIB channels can be combined to provide a higher total bandwidth. A 4 mm $\times$ 4 mm chiplet with 8 AIB channels is fabricated in an Intel 22nm technology. The direction of each AIB channel is run-time configurable, giving user the flexibility in splitting bandwidth between applications. The functionality of UMAI and AIBv2 have been verified in silicon by the on-chip user application logic.

CHAPTER V

Conclusion and Outlook

This dissertation presents the accelerator architecture design for the sparse coding algorithm, and the building blocks for 2.5D integration that can be used to construct scalable systems with chiplets.

The sparse coding accelerator is designed with a novel maze convolution computation that can be configured to support a wide range of different kernel sizes using the same computing element. The natively supported zero-patch skipping is up to 40% more efficient in reducing unnecessary computation compared to the previously proposed zero-line skipping. Area and power savings are achieved by adopting event-driven reconstruction and globally-asynchronous locally-synchronous structure. The techniques developed for this accelerator can be applied to other algorithms such as the widely used convolutional neural networks.

Two chiplets incorporating the AIB interface have been designed and tested. The AIBv1 chiplet includes 3 standard-conforming AIB channels that can be used to transfer data at up to 80Gb/s per channel. The interface design can be easily ported to another CMOS technology because it is synthesizable for most of the parts except for the driver, making it a strong candidate to be adopted as the standard chiplet interface. Experiment results from a silicon interposer based multi-chiplet package showed the energy efficiency of the implemented I/O is 0.83pJ/b. This module also

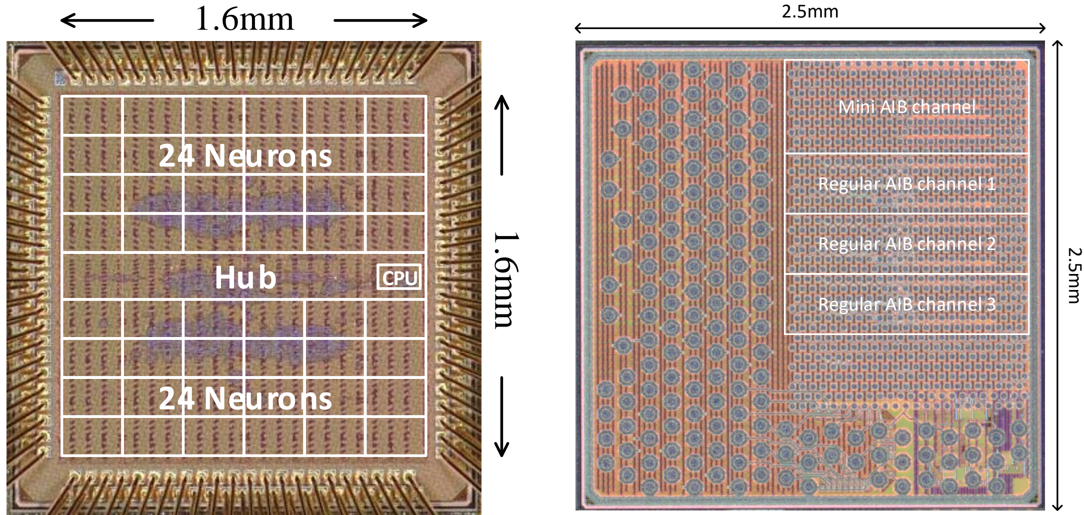


Figure 5.1: Chips designed in this dissertation.

demonstrates the possibility of constructing a larger system by tiling chiplets. When packaged with an Intel 14nm Stratix 10 FPGA, the 16nm chiplet is verified to be inter-operable with another chiplet fabricated in a different technology, via the AIB channels. The AIBv2 chiplet includes 8 standard AIB channels that can be used over UMAI to provide a total of 640Gb/s bandwidth. UMAI is designed with a clean and simple interface for the user application to transfer data between chiplets.

Currently UMAI only uses 80% of the total bandwidth, a possible direction for improvement is to pack data more tightly to increase the bandwidth utilization. Another direction is to improve the energy efficiency of the I/O interface by low-swing signaling. A lower I/O voltage requires more complex I/O circuit design, but the impact on the energy efficiency can be significant, as evidenced by LIPINCON and GRS. Low-swing signaling has already been proposed by the next generation AIB standard.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Bruno A. Olshausen and David J. Field. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 381(6583):607–609, June 1996.

[2] Karen Simonyan and Andrew Zisserman. Very deep concolutional networks for large-scale image recognition. *arXiv: 1409. 1556*, 2014.

[3] Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. Fast inference in sparse coding algorithms with applications to object recognition. *arXiv:1010.3467*, October 2010.

[4] Jung Kuk Kim, Phil Knag, Thomas Chen, and Zhengya Zhang. A 640M pixel/s 3.65mW sparse event-driven neuromorphic object recognition processor with on-chip learning. In *IEEE Symposium VLSI Circuits*, pages 50–51, June 2015.

[5] Meng Yang, Lei Zhang, Jian Yang, and David Zhang. Robust sparse coding for face recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 625–632, August 2011.

[6] Honglak Lee, Alexis Battle, Rajat Raina, and Andrew Y. Ng. Efficient sparse coding algorithms. In *International Conference on Neural Information Processing Systems (NIPS)*, pages 801–808, December 2006.

[7] Jung Kuk Kim, Phil Knag, Thomas Chen, and Zhengya Zhang. Efficient hardware architecture for sparse coding. *IEEE Transactions on Signal Processing*, 62(16):4173–4186, August 2014.

[8] Yijing Watkins, Oleksandr Iaroshenko, Mohammad Sayeh, and Garrett Kenyon. Image compression: Sparse coding vs. bottleneck autoencoders. *arXiv preprint arXiv:1710.09926v2*, January 2018.

[9] Christopher J. Rozell, Don H. Johnson, Richard G. Baraniuk, and Bruno A. Olshausen. Sparse coding via thresholding and local competition in neural circuits. *Neural Computation*, 20(10):2526–2563, October 2008.

[10] Peter F Schultz, Dylan M Paiton, Wei Lu, and Garrett T Kenyon. Replicating kernels with a short stride allows sparse reconstructions with fewer independent kernels. *arXiv preprint arXiv:1406.4205*, June 2014.

[11] Phil Knag, Chester Liu, and Zhengya Zhang. A 1.40mm^2 141mW 898GOPS sparse neuromorphic processor in 40nm CMOS. In *IEEE Symposium VLSI Circuits*, pages 180–181, June 2016.

[12] Intel. Intel stratix 10 fpgas overview.

[13] Ravi Mahajan, Robert Sankman, Neha Patel, Dae-Woo Kim, Kemal Aygun, Zhiguo Qian, Yidnekachew Mekonnen, Islam Salama, Sujit Sharan, Deepti Iyengar, and Debendra Mallik. Embedded multi-die interconnect bridge (emib) – a high density, high bandwidth packaging interconnect. In *IEEE Electronic Components and Technology Conference (ECTC)*, June 2016.

[14] David Kehlet. Accelerating innovation through a standard chiplet interface: The advanced interface bus (aib).

[15] Yijing Watkins, Austin Thresher, Peter F. Schultz, Andreas Wild, Andrew Sornborger, and Garrett T. Kenyon. Unsupervised dictionary learning via a spiking locally competitive algorithm. In *International Conference on Neuromorphic Systems*, pages 1–5, July 2019.

[16] Roger Grosse, Rajat Raina, Helen Kwong, and Andrew Y. Ng. Shift-invariant sparse coding for audio classification. In *Twenty-Third Conference on Uncertainty in Artificial Intelligence*, pages 149–158, July 2007.

[17] Mark D. Plumbley, Thomas Blumensath, Laurent Daudet, Rémi Gribonval, and Mike E. Davies. Sparse representations in audio and music: From coding to source separation. *Proceedings of the IEEE*, 98(6):995–1005, June 2010.

[18] Bruno A. Olshausen. Sparse coding of time-varying natural images. In *International conference on independent component analysis and blind source separation*, pages 603–608, 2000.

[19] Ching-En Lee, Thomas Chen, and Zhengya Zhang. A 127mW 1.63TOPS sparse spatio-temporal cognitive SoC for action classification and motion tracking in videos. In *IEEE Symposium VLSI Circuits*, pages 226–227, August 2017.

[20] Sheng Y. Lundquist, Dylan M. Paiton, Peter F Schultz, and Garrett T Kenyon. Sparse encoding of binocular images for depth inference. In *IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI)*, pages 121–124, April 2016.

[21] Michael Elad and Michal Aharon. Image denoising via sparse and redundant representations over learned dictionaries. *IEEE Transactions on Image Processing*, 15(12):3736–3745, November 2006.

[22] Matan Protter and Michael Elad. Image sequence denoising via sparse and redundant representations. *IEEE Transactions on Image Processing*, 18(1):27–35, January 2009.

[23] Jianchao Yang, John Wright, Thomas S. Huang, and Yi Ma. Image super-resolution via sparse representation. *IEEE Transactions on Image Processing*, 19(11):2861–2873, November 2010.

[24] Shuyuan Yang, Min Wang, Yiguang Chen, and Yaxin Sun. Single-image super-resolution reconstruction via learned geometric dictionaries and clustered sparse coding. *IEEE Transactions on Image Processing*, 21(9):4016–4028, September 2012.

[25] Jung Kuk Kim, Phil Knag, Thomas Chen, and Zhengya Zhang. A 6.67mW sparse coding ASIC enabling on-chip learning and inference. In *IEEE Symposium VLSI Circuits*, pages 61–62, June 2014.

[26] Felix Heide, Wolfgang Heidrich, and Gordon Wetzstein. Fast and flexible convolutional sparse coding. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5135–5143, June 2015.

[27] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid-State Circuits*, 52(1):127–138, Nov 2016.

[28] Bert Moons, Roel Uytterhoeven, Wim Dehaene, and Marian Verhelst. Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm FDSOI. In *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, pages 246–247, March 2017.

[29] Arash Ardakani, Carlo Condo, Mehdi Ahmadi, and Warren J. Gross. An architecture to accelerate convolution in deep neural networks. *IEEE Transactions on Circuits and Systems I*, PP(19):1–14, October 2017.

[30] Yue-Jin Lin and Tian Sheuan Chang. Data and hardware efficient design for convolutional neural network. *IEEE Transactions on Circuits and Systems I*, PP(99):1–14, November 2017.

[31] Fengbin Tu, Shouyi Yin, Peng Ouyang, Shibin Tang, Leibo Liu, and Shaojun Wei. Deep convolutional neural network architecture with reconfigurable computation patterns. *IEEE Transactions on Very Large Scale Integration Systems*, 25(8):2220–2233, Aug 2017.

[32] Paul Merolla, John Arthur, Filipp Akopyan, Nabil Imam, Rajit Manohar, and Dharmendra S. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm. In *IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4, October 2011.

[33] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, Brian Taba, Michael Beakes, Bernard Brezzo, Jente B. Kuang, Rajit Manohar, William P. Risk, Bryan Jackson, and Dharmendra S. Modha.

TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, August 2015.

[34] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *International Symposium on Computer Architecture (ISCA)*, pages 1–13, August 2016.

[35] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*, pages 27–40, June 2017.

[36] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, May 2015.

[37] Ivan Miro Panades and Alain Greiner. Bi-synchronous FIFO for synchronous circuit communication well suited for network-on-chip in GALS architectures. In *Int. Symp. Networks-on-Chip (NOCS)*, pages 83–94, May 2007.

[38] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the KITTI vision benchmark suite. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3354–3361, July 2012.

[39] Bryan Black. Die stacking is happening. In *Intl. Symp. Microarchitecture, Davis, CA*, 2013.

[40] Mike O'Connor. Highlights of the high-bandwidth memory (hbm) standard. In *Memory Forum Workshop*, 2014.

[41] Dong Uk Lee et al. A 1.2 v 8 gb 8-channel 128 gb/s high-bandwidth memory (hbm) stacked dram with effective i/o test circuits. *IEEE J. Solid-State Circuits*, 50(1):191–203, 2015.

[42] Kirk Saban. Xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency. *Xilinx White Paper*, 1:1–10, 2011.

[43] Christophe Erdmann et al. A heterogeneous 3d-ic consisting of two 28 nm fpga die and 32 reconfigurable high-performance data converters. *IEEE J. Solid-State Circuits*, 50(1):258–269, 2015.

[44] Mu-Shan Lin, Tze-Chiang Huang, Chien-Chun Tsai, King-Ho Tam, Cheng-Hsiang Hsieh, Tom Chen, Wen-Hung Huang, Yu-Chi Chen Jack Hu, Sandeep Kumar Goel, Chin-Ming Fu, Stefan Rusu, Chao-Chieh Li, Sheng-Yao Yang, Mei Wong, Shu-Chun Yang, and Frank Lee. A 7nm 4ghz arm-core-based cowos chiplet design for high performance computing. In *IEEE Symposium VLSI Circuits*, June 2019.

[45] Brian Zimmer, Rangharajan Venkatesan, Yakun Sophia Shao, Jason Clemons, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel S. Emer, C. Thomas Gray, Stephen W. Keckler, and Brucek Khailany. A 0.11 pj/op, 0.32-128 tops, scalable, multi-chip-module-based deep neural network accelerator with ground reference signaling in 16nm. In *IEEE Symposium VLSI Circuits*, June 2019.

[46] John W. Poulton, John M. Wilson, Walker J. Turner, Brian Zimmer, Xi Chen, Sudhir S. Kudva, Sanquan Song, Stephen G. Tell, Nikola Nedovic, Wenxu Zhao, Sunil R. Sudhakaran, C. Thomas Gray, and William J. Dally. A 1.17-pj/b, 25-gb/s/pin ground-referenced single-ended serial link for off- and on-package communication using a process- and temperature-adaptive voltage regulator. *IEEE J. Solid-State Circuits*, 54(1):43–54, Nov 2018.

[47] Chester Liu, Sung-Gun Cho, and Zhengya Zhang. A 2.56mm^2 718GOPS configurable spiking convolutional sparse coding processor in 40nm CMOS. In *IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pages 233–236, November 2017.

[48] Intel. Heterogeneous 3d system-in-package integration.

[49] Intel. Embedded multi-die interconnect bridge.

[50] Hyo Gyuem Rhew, Michael P Flynn, and JunYoung Park. A 22gb/s, 10mm on-chip serial link over lossy transmission line with resistive termination. In *Proc. ESSCIRC*, pages 233–236, 2012.

[51] Jaehun Jeong, Nicholas Collins, and Michael P Flynn. A 260 mhz if sampling bit-stream processing digital beamformer with an integrated array of continuous-time band-pass σ modulators. *IEEE J. Solid-State Circuits*, 51(5), 2016.