METHODOLOGIES, ARCHITECTURES, AND PROTOTYPES FOR SCALING ON- AND OFF-CHIP INTERCONNECTS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by
Yanghui Ou
November 2024

© 2024 Yanghui Ou
ALL RIGHTS RESERVED

METHODOLOGIES, ARCHITECTURES, AND PROTOTYPES FOR SCALING ON- AND OFF-CHIP INTERCONNECTS

Yanghui Ou, Ph.D.

Cornell University 2024

The slowdown of Moore's Law and the end of Dennard scaling have driven modern computing systems to embrace parallelism, both within single chips and across multiple compute devices, in order to meet the growing computational demands. Efficient data movement, both on-chip and off-chip, has thus become increasingly critical. However, scaling on- and off-chip interconnects each presents unique challenges in both methodology and architecture. For on-chip interconnects, challenges include: (1) the methodology challenge of developing a robust framework to model, test, and evaluate on-chip network (OCN) designs across a vast design space, and (2) the architecture challenge of bridging the gap between theoretical advances and practical implementation of scalable, low-diameter OCN topologies. For off-chip interconnects, challenges include: (1) the methodology challenge of modeling large-scale distributed systems accurately, and (2) the architecture challenge of breaking the capacity, latency, and bandwidth trade-offs inherent in current off-chip interconnect technologies. This thesis addresses these challenges by developing new methodologies, proposing architectural solutions, and validating their feasibility through practical silicon prototypes.

The first part of this thesis focuses on OCNs for manycore architectures. I first present PyOCN, a unified Python-based framework for modeling, testing, and evaluating on-chip networks, which vertically integrates multiple research methodologies and enables productive design space exploration of OCNs. Next, I propose practical low-diameter OCN topologies that can be effectively implemented with a tiled physical design methodology, bridging the gap between principle and practice. Finally, the CIFER chip tape-out demonstrates the feasibility and effectiveness of PyOCN as well as the tiled physical design approach.

The second part of the thesis addresses challenges in scaling off-chip interconnects, particularly for machine learning workloads. I first present LLMCompass-E2E, a comprehensive framework for modeling large-scale distributed LLM training performance. I then explore the po-

tential of emerging co-packaged silicon photonic interconnects by proposing an optically connected multi-stack HBM module which can effectively break the trade-off between memory bandwidth and capacity. Lastly, the PIPES chip tape-out demonstrates a practical implementation of such co-packaged silicon photonic interconnects, highlighting their potential for scalable, high-performance interconnect solutions in large-scale distributed systems.

BIOGRAPHICAL SKETCH

Yanghui Ou was born on April 23, 1995 to Hongmei Yang and Shaojia Ou. At a young age, Yanghui's greatest passion was actually basketball. He dreamed of becoming the next big basketball star, starting his journey with the school team in elementary school. Little did he know, his true calling would not involve slam dunks but rather solving bugs. During his time at Beijing No. 171 High School, Yanghui first learned about Java and was fascinated by the idea of solving problems with code. He managed to juggle playing in the school basketball team and being a dedicated member of the school's robotics club. Obviously not talented enough for the NBA, Yanghui decided to pursue a career in technology.

Yanghui's high school years were a rollercoaster of ups and downs. Before the college entrance exam, he only listed one university on his application, refusing to consider any alternatives. Unfortunately, his score fell short. He accepted the failure and returned to high school to attend the senior year again. Doubt crept in throughout the year as he sometimes performed worse than the previous year in practice exams, but he gritted his teeth and pushed through. On exam day, Yanghui was in the zone, channeling everything he had learned. This time, he scored high enough to choose any university. He decided to take an adventure far away from home, going all the way south and attending the Hong Kong University of Science and Technology.

Yanghui cherished the beautiful scenery and vibrant campus life at HKUST. Other than winning the championship of the undergraduate basketball league for mainland students with his awesome teammates and coaches, he also found his passion for computer architecture. In fall 2016, he participated in an exchange program at Cornell University, where he took the world's best computer architecture course taught by Prof. Christopher Batten. It was at that time that he decided to pursue a PhD career in computer architecture.

In 2018, Yanghui fulfilled his dream of joining the PhD program at Cornell University. He joined Prof. Christopher Batten's group without hesitation. In the next six years, he expanded his knowledge on various aspects of computer architecture and contributed to many projects, ranging from methodologies to silicon prototypes. Beyond technical expertise, he also learned how to conduct good research and witnessed exemplary leadership from his advisor.

Yanghui is deeply grateful for his PhD experience at Cornell University, a period of invaluable professional and personal growth, guided by an exceptional advisor and accompanied by many good friends.

This document is dedicated to my parents, my grandparents, and Yiran Hao.

ACKNOWLEDGEMENTS

My PhD journey has been a mix of joyful moments and challenges. I am grateful to many individuals who have supported, encouraged, and guided me along the way.

First of all, I cannot express enough gratitude to my advisor, Prof. Christopher Batten. His computer architecture course was the spark that inspired me to pursue a PhD in this field. To this day, I can still vividly recall the six key concepts for advanced computer architecture that we repeated before each class. Beyond the countless brainstorming sessions that fueled my research, Chris has also helped me grow in many other ways throughout my PhD. Chris made me a better communicator through his honest and constructive feedback. He constantly encouraged me to step up, talk to people and grow my professional network. Chris taught me the value of open and honest communication, never pretending to understand and nodding my head but instead asking questions, no matter how simple or stupid they might seem. He instilled in me an appreciation for details, such as writing well-formatted code, thoroughly unit testing my own design, and maintaining consistency and clarity in my writing. From him, I learned how to make good slides, how to deliver compelling presentations, how to create clear and effective figures, how to write well-crafted papers, and how to be a good teacher. His mentorship has left an indelible mark on both my professional and personal growth.

I am also immensely thankful to all my PhD friends in the Batten Research Group (BRG) for their support and collaboration. I have learned a great deal from each of you, both professionally and personally. I would like to thank Christopher Torng, who not only taught me a lot about ASIC flow and VLSI but also shared many useful terminal and Vim tricks, arguably the most important survival skills for any PhD student. I joined the group just as Berkin Ilbeyi was about to graduate. but I still learned the concept of JIT and Python acceleration techniques from his impressive work and his paper presentation in our social reading group. Moyang Wang, your expertise in work-stealing runtime and software-centric cache coherence impresses me as much as your skiing and snowboarding skills. I will always treasure the memory of us snowboarding through the woods at Greek Peak and ended up rolling on the ground together. Shunning Jiang, you were a fantastic mentor, and our collaboration on the PyMTL3 project was one of the highlights of my PhD. I will never forget the night when the whole group were crammed in Chris' hotel room, working together before the day of our PyMTL3 tutorial at FCRC'19. Khalid Al-Hawaj, thank you for teaching me gem5 and for your jokes that always lifted my spirits. Tuan Ta, thank you too for showing me

how to hack gem5 and for introducing me to badminton. Lin Cheng, your lessons on memory consistency and cache coherence were invaluable, and so were the board game nights at your place. And thank you for dragging me to Greek Peak to learn snowboarding together. I still recall the day when we finally figured out how to properly make an S-turn and conquered the green track without crashing. Peitian Pan, your work ethic is truly inspiring and I really enjoyed working with you. I am still proud of our epic final project for the HLS course and our snoopy-based cache coherent memory system for the ASIC course. Nick Cebry, you are an amazing friend. Our snowboarding trips to Greek Peak were some of the best times of my PhD. I will never forget the semester when we were the only two BRG members left in Ithaca. I wish you the best for the rest of your PhD journey. Derin Ozturk, though our overlap was brief, it was refreshing to have your robotics expertise add a new dimension to our group. Elton Shih and Niklas Schmelzle, it was a pleasure to finally see some new faces in the group. The future of BRG is in your hands and I am excited to see where you take it. Max Doblas, thank you for visiting our group from Barcelona Supercomputing Center. Your work on genomics was amazing. I was also fortunate to work with three amazing postdoc researchers in BRG. Cheng Tan, thank you for leading the PyOCN paper and for creating a compiler for the UE-CGRA that spared me from manually mapping the kernels on the whiteboard. Thank you for taking me to the gym and teaching me how to build muscle. Shady Agwa, you were always willing to help and thank you for teaching me about SRAM compilers and floorplanning. Austin Rovinski, you were such a physical design wizard and the PIPES project would have been impossible without your help. I will never forget our sleepless nights before the tape-out deadline. I also learned so much about VLSI and open-source EDA from you. I wish you a successful career as a professor and a thriving REALISE lab, or whatever even more creative name you may come up with next.

The Computer Systems Laboratory (CSL) has been a supportive and caring community and I am grateful for my CLS mentors, friends, and colleagues. I would also like to thank Prof. Zhiru Zhang and Prof. Adrian Sampson, for agreeing to be on my committee and providing invaluable insights and feedback on my research. I would also like to thank my fellow PhD friends at CSL. A special mention goes to Zhuangzhuang Zhou, who was also my neighbor in Warrenwood Apartments for almost five years. Thank you for cooking me delicious food and for sharing your stories with me. Weizhe Hua, you will always be my tennis coach and thank you for teaching me how to play tennis. Yi Jiang, thank you for making so many delicious cakes and cookies for the gatherings

at Lin's place. Philip Bedoukian, your cheerful energy never failed to brighten my day. Yanqi Zhang, Yichi Zhang, and Chenhui Deng, thank you for bringing me, a complete badminton newbie, to your pro badminton party and teaching me how to play. I am also grateful for the friendship and inspiring conversations of many other CSL researchers: Ritchie Zhao, Mulong Luo, Kailin Yang, Mingyu Liang, Hanchen Jin, Zichao Yue, Jie Liu, Yixiao Du, and Niansong Zhang.

I would like to thank Prof. David Wentzlaff and the exceptional researchers in his research group for their collaboration on the CIFER project, the optically connected HBM project, and the LLMCompass-E2E project. I am especially grateful to Jonathan Balkind for teaching me about OpenPiton and how to hack the framework. I appreciate Ang Li and Fei Gao for their insightful feedback and advice on the OCN design for CIFER. To the rest of the CIFER team, thank you for your hard work and dedication in making the tape-out a success. A special callout goes to Hengrui Zhang for our productive collaboration on the optically connected HBM project, and the LLMCompass-E2E project. We had so many inspiring discussions and I learned a lot from him about GPU architecture and LLM workloads. Similar thanks go to August Ning and Rohan Prabhakar for our engaging discussions about LLMCompass and LLM training. I learned a lot from your perspectives and expertise.

I would like to express my gratitude to my PIPES teammates. First, I would like to thank my collaborators at Intel. Kaveh Hosseini and Tim Tri Hoang led the PIPES project and I am grateful for their feedback and guidance. Sung-Gun Cho offered tremendous support on the ASIC flow, and meeting the deadline would not have been possible without his help. Ching-Chi Chang helped a lot on pre-silicon verification and AIB debugging. I am also grateful to Prof. Alyosha Molnar and his team, especially Christine Ou, Devesh Khilwani, and Sunwoo Lee, for their collaboration on the TRX integration. My sincere thanks also go to Prof. Keren Bergman and her team for their work on the PIC and the final end-to-end demo. I am particularly thankful to Songli Wang and Yuyang Wang for their generosity in hosting me at Columbia University and for teaching me so much about silicon photonics.

I would like to thank all the other mentors who have inspired and guided me throughout my journey. I thank Prof. Jiang Xu and Prof. Wei Zhang for their inspiring courses at HKUST. Prof. Wei Zhang also supervised my undergraduate final-year project and taught me a lot about FPGA and hardware accelerators. I thank Prof. Hoi-Sing Kwok for supervising some of my undergraduate research projects. Though I did not end up pursuing a career in display technologies,

I explored many interesting topics and broadened my perspective under his guidance. I thank Prof. Charles Sodini for hosting me as a visiting student at MIT for a summer research program. I thank Mr. Jianpeng Qin, my high school English teacher, whose encouragement and guidance helped me persevere through my lowest moments.

I would like to thank all my undergraduate friends and high school friends who, despite being scattered all around the world, have remained a constant source of support and joy. A special thanks to my friend at Hong Kong, Yunhao Feng. Thank you for playing hundreds of hours of Valheim, Satisfactory, and particularly Factorio with me, which turned me into a master of train networks and interstellar logistics, adding a new dimension to my engineering skill set. I would also like to thank Xiangju Chen, Xuanyin Xian, and Zepu Sun for their enduring friendship. Guangyu He, thank you for coming to New York all the way from Germany to visit me and celebrate my birthday. I also appreciate the friendship of Linwei Zhang, Shiyu Li, Jiaziyu Lang, and Zhe Feng. Your thoughtful birthday wishes every year remind me how lucky I am to have such amazing friends.

Finally, I would like to thank my family for raising me up and supporting me unconditionally throughout my life. I look forward to going back home and spending more time with them. I also want to thank Yiran Hao for her support and companionship. Your presence has made my PhD journey brighter and more worthwhile. Special thanks to the true co-author of all my work, my cat, MianMian, for her loyal company during countless late nights. Her emotional support has been a constant source of comfort and strength.

In terms of funding, this thesis was supported in part by DARPA POSH Award #FA8650-18-2-7852, DARPA PIPES Award HR00111920014, DARPA CHIPS Award HR00111830002, DARPA SDH Award #FA8650-18-2-7863, NSF EVE Award #CCF-2008471, NSF PPoSS Award #CCF-2118709, equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . ix
List of Figures . . . xii
List of Tables . . . xiv
List of Abbreviations . . . xv

1 Introduction **1**
- 1.1 On-Chip Interconnects . . . 2
- 1.2 Off-Chip Interconnects . . . 5
- 1.3 Thesis Overview . . . 9
- 1.4 Collaboration, Previous Publications, and Funding . . . 11

I Scaling On-Chip Interconnects **16**

2 Methodology: A Unified Framework for On-Chip Networks **17**
- 2.1 Introduction . . . 17
- 2.2 Related Work . . . 18
  - 2.2.1 Modeling OCNs . . . 18
  - 2.2.2 Testing OCNs . . . 20
  - 2.2.3 Evaluating OCNs . . . 21
- 2.3 PyOCN Framework . . . 23
- 2.4 PyOCN for Modeling OCNs . . . 23
- 2.5 PyOCN for Testing OCNs . . . 27
- 2.6 PyOCN for Evaluating OCNs . . . 29
- 2.7 Case Study . . . 31
- 2.8 Conclusion . . . 35

3 Architecture: Low-Diameter On-Chip Networks for Manycore Processors **36**
- 3.1 Introduction . . . 36
- 3.2 Manycore OCN Topologies . . . 38
- 3.3 Manycore OCN Analytical Modeling . . . 42
- 3.4 Manycore OCN Physical Design . . . 50
- 3.5 Conclusions . . . 52

4 Prototype: CIFER Chip Tape-Out **53**
- 4.1 CIFER Architecture . . . 53
- 4.2 CIFER On-Chip Networks . . . 56
  - 4.2.1 Supporting Multi-Flit Packets in PyOCN . . . 56
  - 4.2.2 Logical vs. Physical Hierarchy Trade-Offs . . . 58
  - 4.2.3 Data Transfer Interface Trade-Offs: Val/Rdy vs. En/Rdy vs. Credit-Based . 60
  - 4.2.4 Timing Optimization: Speculative Switch Allocation . . . 63

4.2.5 TinyCluster Resource-Sharing Crossbar and Memory Crossbar Design . . . 65
4.3 Conclusion . . . 67

II Scaling Off-Chip Interconnects **69**

5 Methodology: A Performance Modeling Framework for End-to-End LLM Training **70**
5.1 Introduction . . . 70
5.2 Background . . . 72
5.2.1 LLM Training . . . 73
5.2.2 The LLMCompass Framework . . . 76
5.3 LLMCompass-E2E . . . 80
5.3.1 Frontend . . . 81
5.3.2 System-Level Performance Model . . . 83
5.4 Evaluation . . . 84
5.5 Conclusion . . . 87

6 Architecture: Optically Connected Multi-Stack HBM Module **89**
6.1 Introduction . . . 89
6.2 Background . . . 91
6.3 System Architecture . . . 93
6.4 Evaluation . . . 96
6.5 Related Work . . . 100
6.6 Conclusions . . . 100

7 Prototype: PIPES Chip Tape-Out **101**
7.1 Introduction . . . 101
7.2 PIPES System Architecture . . . 103
7.2.1 EIC Architecture . . . 105
7.2.2 EIC Scaling Challenges . . . 107
7.2.3 PIC Architecture . . . 108
7.2.4 PIC Scaling Challenges . . . 108
7.3 EIC Design, Implementation, and Verification . . . 110
7.3.1 Crossbar Design . . . 110
7.3.2 EIC Implementation . . . 114
7.3.3 Pre-Silicon Verification . . . 116
7.3.4 Post-Silicon Verification . . . 118
7.4 Link-Level Evaluation . . . 119
7.4.1 Simulation-Based Evaluation . . . 119
7.4.2 Experimental Evaluation . . . 119
7.5 System-Level Evaluation . . . 121
7.6 Conclusion . . . 123

8 Conclusion **125**
8.1 Thesis Summary and Contributions 125
8.2 Future Work 126
8.2.1 Testing Methodology for On-Chip Networks 126
8.2.2 LLMCompass-E2E 127
8.2.3 Co-Packaged Optics for Memory and System Interconnects 128

Bibliography **131**

LIST OF FIGURES

1.1 Trend of Processor Core Count . . . 2
1.2 Examples of Manycore Processors . . . 3
1.3 Trend of Per-Device Memory Bandwidth and Capacity . . . 6
1.4 Trend of Off-Chip Interconnect Bandwidth . . . 7
1.5 Tension in Off-Chip Interconnect Technologies . . . 8

2.1 Overview of PyOCN Framework . . . 22
2.2 PyOCN Generic Router Architecture . . . 24
2.3 FL Implementation of Ring Network . . . 24
2.4 CL Implementation of SwitchUnit . . . 25
2.5 RTL Implementation of SwitchUnit . . . 26
2.6 Physical Elaboration . . . 27
2.7 Unit Test for a Router in 4×4 Mesh . . . 28
2.8 Property-Based Random Testing for Mesh Network Generator . . . 29
2.9 RTL Simulation Results . . . 30
2.10 Router Characterization . . . 30
2.11 4-ary 3-fly Butterfly Network . . . 32
2.12 4-ary 3-fly Butterfly Network Floorplan . . . 33
2.13 Parameterization System Example . . . 33
2.14 Post Place-and-Route Layout of 4-ary 3-fly Butterfly . . . 34

3.1 Twelve Topologies Implemented Using a Tiled Physical Design Methodology . . . 39
3.2 OCN Component-Level Results . . . 40
3.3 Latency and Area Trade-Offs . . . 43
3.4 Latency and Bandwidth Trade-Offs . . . 44
3.5 Bandwidth, Latency, and Area Trade-Offs for Post-Place-and-Route Results . . . 47
3.6 Example Macro-Level Post-Place-and-Route Layouts . . . 48
3.7 Example Chip-Level Post-Place-and-Route Layouts . . . 49

4.1 CIFER Package and Die Photo . . . 54
4.2 CIFER SoC Architecture . . . 55
4.3 CIFER NoC Header Format . . . 57
4.4 CIFER Router Architecture . . . 57
4.5 CIFER SoC Logical Hierarchy . . . 59
4.6 CIFER SoC Physical Hierarchy . . . 60
4.7 Different Handshake Interfaces . . . 61
4.8 Speculative Switch Allocation . . . 64
4.9 TinyCore Cluster Architecture . . . 65
4.10 Resource-Sharing Crossbar Design . . . 66
4.11 Memory Crossbar Design . . . 66

5.1 Architecture of Decoder-Only Transformer Layer . . . 73
5.2 LLMCompass Overview . . . 76
5.3 LLMCompass Hardware Description Template . . . 77

5.4 LLM Model Description in LLMCompass 78
5.5 Matrix Multiplication Performance Modeling in LLMCompass 79
5.6 LLMCompass-E2E Overview 80
5.7 LLMCompass-E2E Code Example 81
5.8 LLMCompass-E2E Operator Example 82
5.9 LLM Model Description in LLMCompassE2E 85
5.10 Computational Graph of A Multi-Head Attention Block 86

6.1 Memory Requirement for Training LLMs and HBM Capacity Per Device Over Time 90
6.2 Silicon Photonics Integration Technologies 92
6.3 Example System Architecture 94
6.4 Optical Channel Datapath 95
6.5 EIC and PIC Architecture 95
6.6 Evaluation Results for Training 98
6.7 Per Layer Speedup for Inference 99

7.1 Approaches to Tightly Integrated Optical Interconnect 102
7.2 PIPES Package Photo 104
7.3 EIC and PIC Die Photos 106
7.4 EIC Architecture Block Diagram 107
7.5 Photonic Link Architecture 109
7.6 Crossbar Block Diagram 110
7.7 SPI Configuration Stack 111
7.8 SPI Minion Diagram 112
7.9 AIB Interface Unit Datapath Diagram 113
7.10 TRX Interface Unit Datapath Diagram 114
7.11 PIPES ASIC Flow 115
7.12 EIC Floorplan and Layout 116
7.13 EIC Bumps 117
7.14 Experimental Setup 120
7.15 Eye Diagrams for Opto-Electrical EIC/PIC Channel 120
7.16 Optimal Mappings for LLM Training 122

LIST OF TABLES

2.1 Comparison with Prior Art 19
2.2 PyOCN Multi-Level Simulation 26

3.1 Analytical Modeling Results 40
3.2 Post-Place-and-Route Macro Results 47

4.1 Comparison of Data Transfer Interfaces 63

5.1 Shape of intermediate Tensors in Transformer Layer 72
5.2 Comparison Against Real Hardware Results 87

LIST OF ABBREVIATIONS

OCN	on-chip network
LLM	large language model
CIFER	coherent interconnect and FPGA enabling reuse
PIPES	photonic in package for extreme scalability
ASIC	application-specific integrated circuit
FPGA	field-programmable gate array
GPU	graphics processing unit
TPU	tensor processing unit
MCM	multi-chip module
RTL	register-transfer level
DDR	double data rate
DIMM	dual inline memory module
GDDR	graphics double data rate
HBM	high bandwidth memory
TSV	through-silicon via
PCIe	peripheral component interconnect express
SoC	system-on-chip
FL	functional level
CL	cycle level
PL	physical level
eFPGA	embedded FPGA
LLC	last-level cache
TRI	transaction response interface
MDU	multiply-division unit
FPU	floating-point unit
LUT	look-up table
BSV	Bluespec SystemVerilog
IR	intermediate representation
FLOP	floating-point operation
MFU	model FLOPs utilization
EIC	electrical interface chiplet
PIC	photonic interface chiplet
CW	continuous-wave
TX	transmitter
RX	receiver
TRX	transceiver
UCIe	universal chiplet interconnect express
PHY	physical layer
AIB	advanced interface bus
PRBS	pseudo-random binary sequence
AFE	analog front end
SPI	serial peripheral interface

CHAPTER 1
INTRODUCTION

The slowdown of Moore's Law and the end of Dennard scaling have fundamentally reshaped the landscape of computing. For decades, these two principles drove consistent improvement in single-chip performance by enabling more transistors to fit within the same area and improving the power efficiency per transistor. However, as physical and technical limits have been reached, the exponential gains in performance from these traditional scaling methods have diminished. This shift has compelled the industry to explore new strategies for enhancing performance, placing greater emphasis on *parallelism*. Modern computing systems, ranging from supercomputers to edge devices, now require more innovative design solutions to meet the demands of increasingly complex and data-intensive workloads.

One of the key strategies to address the limitations of traditional scaling is the adoption of *manycore architectures*, which integrate a large number of simple, lightweight processing cores on a single chip. Examples include thread-parallel manycore processors such as Epiphany-V [Olo16] and Celerity [RZAH+19b], as well as data-parallel manycore processors such as graphics processing units (GPU) [nvi20, nvi23] and tensor processing units (TPUs) [JKL+23]. Manycore systems can scale to hundreds or even thousands of cores, enabling massive parallelism and significantly higher throughput for data-intensive workloads. However, the efficiency of manycore architectures hinges on the ability to move data efficiently across the chip, making the design of scalable, high-performance on-chip networks (OCNs) essential.

While on-chip interconnects are critical for efficient data movement within the compute chip, off-chip interconnects play an equally important role in connecting these processors to external components, such as memory, storage, and other compute chips. Off-chip interconnects, including *memory interconnects* that connect processing units to external memory and *system interconnects* that connect multiple compute chips or systems, enable scalable and distributed compute systems by facilitating high-bandwidth, low-latency communication between multiple chips. These interconnects can range from short-reach connections spanning a few millimeters, such as the die-to-die interconnects in modern multi-chip modules (MCM), to long-reach connections covering up to a few kilometers, such as optical links in data center networks. The effectiveness of these distributed systems depends heavily on the design of scalable, high-performance off-chip interconnects that can handle the increasing demands of data movement across components.

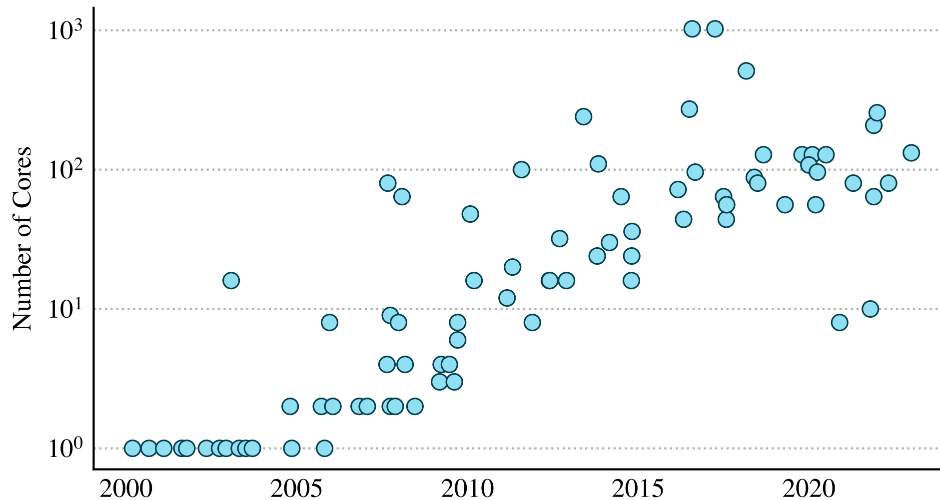
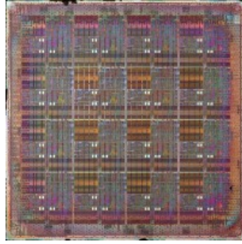


Figure 1.1: Trend of Processor Core Count – The figure illustrates the core count of selected processors from 2000 to 2020. The data is partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten, and K. Rupp [Rup24].

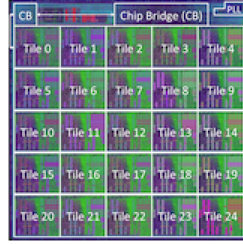
In this chapter, I provide an overview of the challenges and opportunities in scaling on-chip and off-chip interconnects, highlighting the importance of efficient data movement in modern computing systems. Section 1.1 first discusses the trend of on-chip interconnects and then introduces the methodology and architecture challenges in designing scalable, high-performance on-chip networks. Section 1.2 first discusses the trend of off-chip interconnects, including *memory interconnects* and *system interconnects*, and then presents the methodology and architecture challenges in designing scalable, high-performance off-chip interconnects. Section 1.3 provides an overview of the thesis. Section 1.4 discusses the collaboration, previous publications, and funding sources related to the work presented in this thesis.

1.1 On-Chip Interconnects

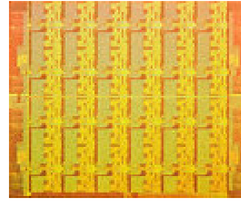
As the computing industry seeks to overcome the limitations of traditional scaling, manycore architectures have emerged as a prominent solution. Over the past decades, there has been a steady trend towards increasing the core counts in processors, driven by the need for greater parallelism to handle the growing computational demands of modern workloads. Compared to general-purpose multi-cores, the manycore approach can improve throughput and energy efficiency per unit area, particularly for highly parallel workloads. As is illustrated in Figure 1.1, the core count in pro-



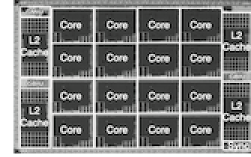
(a) 16-core MIT RAW Processor [TKM+03]



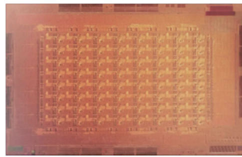
(b) 25-core Piton [MFN+17]



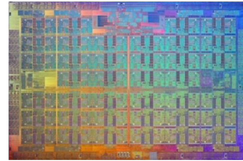
(c) 48-core Intel Single-Chip Cloud Computer [HDH+10]



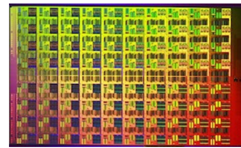
(d) 64-core ICT Godson-T [TFZ+08]



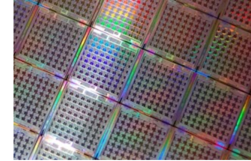
(e) 64-core Meta MTIA [FCL+23]



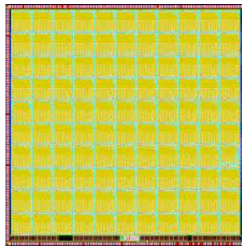
(f) 72-core Intel Knights Landing [SGC+16]



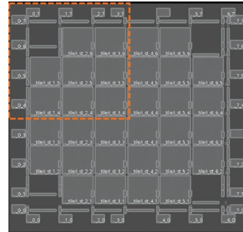
(g) 80-core Intel Teraflops Research Chip [HVS+07]



(h) 100-core TILE-GX100 [Ram11]

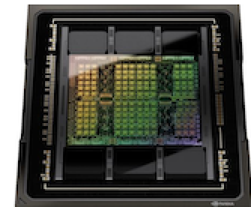


(i) 110-core EM^2 [LSC+13]

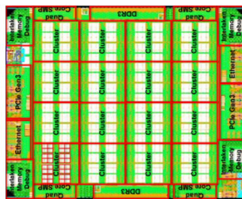


(j) 128-core Ampere Altra Max [Whe20]

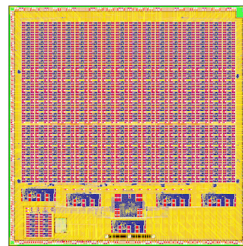
(k) 128-core Sunway SW26010 [LFF+18]



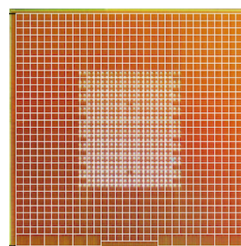
(l) 132-core NVIDIA H100 [nvi23]



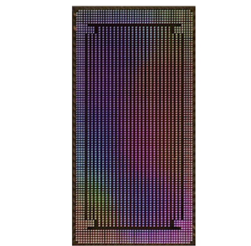
(m) 256-core Kalray MPPA-256 [kal24a]



(n) 511-core Celerity Research Chip [DXT+18]



(o) 1024-core KiloCore [BSP+17]



(p) 1024-core Adapteva Epiphany-V [Olo16]

Figure 1.2: Examples of Manycore Processors – Chip plots or die photos of selected manycore processors.

cessors has scaled dramatically in the past two decades, reaching hundreds and even a thousand in recent years.

Figure 1.2 shows various examples of manycore processors that have been designed and manufactured over the years. Early *thread-parallel* manycore research prototypes integrated up to 110 cores within a single die. The MIT RAW processor [TKM+03] integrated 16 simple cores connected by a 4×4 mesh OCN. The Intel Teraflops research chip [HVS+07] contained 80 tiles arranged as a 10×8 mesh OCN. The Intel Single-Chip Cloud Computer (SCC) [HDH+10] was a manycore processor with 48 cores connected by a 4×6 mesh OCN. The 110-core Execution Migration Machine (EM^2) [LSC+13] demonstrated a directory-less shared-memory manycore with a 10×11 mesh OCN. Over time, the industry has adopted the manycore approach as well. Examples include the 64-core Tile64 [BEA+08], the 72-core Knights Landing [SGC+16], the 100-core Tile GX100 [Ram11], the 128-core Ampere Altra Max [Whe20], and the 128-core Sunway SW26010 [LFF+18]. Recent research prototypes have scaled core counts to over a thousand cores, such as the 1000-core KiloCore [BSP+17], the 1024-core Epiphany-V [Olo16], and the 4096-core Manticore [ZSB21].

Data-parallel manycore processors are also widely adopted by the industry, with GPUs being the most prominent example. GPUs usually have around a hundred cores (known as stream multiprocessors in NVIDIA GPUs and compute units in AMD GPUs), and they are capable of supporting thousands of concurrent hardware threads. In addition to GPUs, there are also custom accelerators designed for specific workloads, such as the Google TPU [JKL+23], Meta MTIA [FCL+23], Kalray MPPA-256 [kal24a], and Tenstorrent Grayskull [kal24b]. These custom accelerators typically incorporate multiple processing elements (PEs) arranged into a 2D systolic array.

The effectiveness of manycore systems relies heavily on efficient on-chip data movement, making the design of scalable, high-performance OCNs critical. However, achieving scalability and high-performance in OCN design presents substantial challenges, both in methodology and architecture.

Methodology Challenge – Developing on-chip interconnects involves overcoming several methodological challenges, particularly in the design, testing, and evaluation of networks. The design space for OCNs is vast, encompassing various factors such as topology, routing algorithms, flow control mechanisms, and physical design considerations. Exploring this design space requires a unified approach that can model, simulate, and evaluate OCNs across different levels of abstrac-

tion, from functional models to cycle-accurate simulations and hardware implementations. However, existing tools often struggle to balance the need for rapid design-space exploration with the accuracy required for hardware-level evaluation. For example, many widely used on-chip network simulators use cycle-level modeling for early design-space exploration and verifying cycle-level behavior [APM+12, AKPJ09, CHB+10, JBM+13, TB12, LSC+10]. However, these simulators do not support register-transfer-level (RTL) modeling and cannot easily generate synthesizable Verilog, which is essential for accurate evaluation of area, energy, and timing. OCN generators use RTL modeling to accurately characterize area, energy, and timing, but they lack the high-level design abstractions that enable fast design-space exploration [CP04, PH13, KK17]. This gap creates challenges in assessing trade-offs between performance, area, and energy consumption early in the design process, making it difficult to iterate and refine OCN designs efficiently. To address these challenges, this thesis presents a unified framework that seamlessly integrates modeling, testing, and evaluation to support the development of scalable and robust OCNs.

Architecture Challenge – A notable gap exists a noticeable gap between the theoretical advancements in OCN design and their practical implementation in manycore processors. While research literature has proposed numerous innovative solutions to improve OCN performance, such as novel flow-control schemes [KPKJ07, MWM04, PD01], custom circuits [KS08, CPK+13], and novel network topologies [BD06, BD06, KBD07, GHKM09, GHKM11], most manycore processors still adopt a simple 2D-mesh OCN topology [BEA+08, WGH+07, MFN+17, LSC+13, BSP+17, Whe20, Hal20, RZAH+19b], even though it is well known that the high diameter of 2D-mesh topologies can significantly increase packet latency and thus reduce system-level performance [DT04]. The key reason is that implementing manycore processors relies on a tiled physical design methodology, yet these novel solutions are often incompatible with this approach. To bridge this gap, this thesis explores practical architectures that can reduce the network diameter while remaining compatible a tiled physical design methodology.

1.2 Off-Chip Interconnects

As the number of cores within a single chip continues to scale, the demand for higher memory bandwidth has risen sharply to ensure that compute cores receive sufficient data. Additionally, modern data-intensive workloads, such as large language model (LLM) training, require a substan-

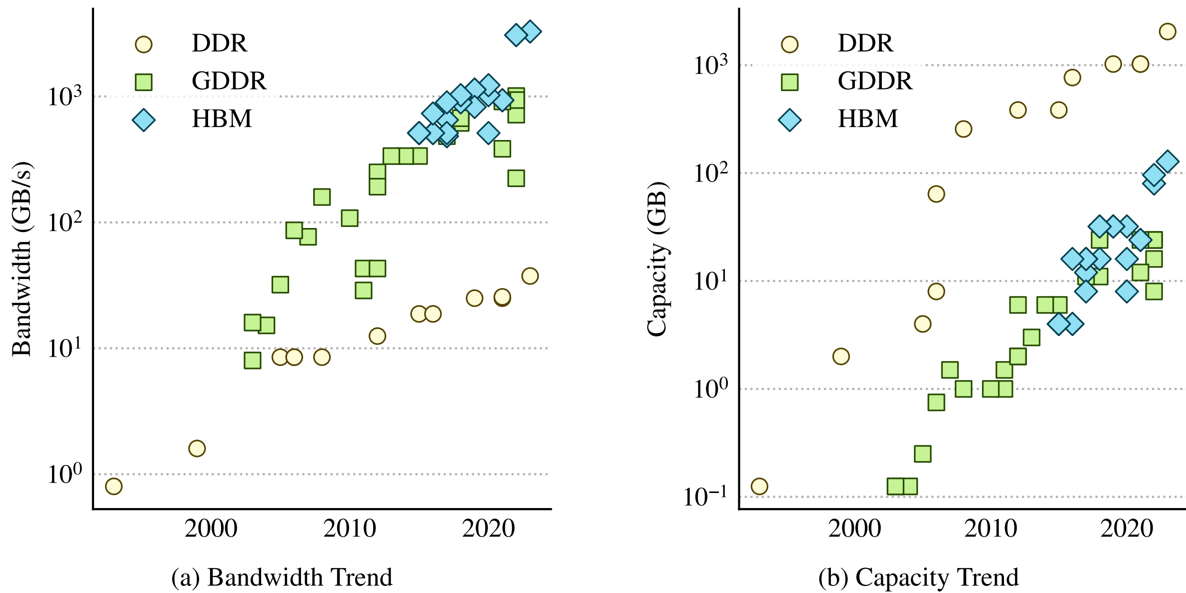


Figure 1.3: Trend of Per-Device Memory Bandwidth and Capacity – This plot shows how the per-device memory bandwidth and capacity of different memory technologies scale over time. Data is collected from [tec24b, tec24a]

tial amount of memory capacity. This has driven the need for off-chip *memory interconnects* that can provide high bandwidth and are easy to expand memory capacity. Meanwhile, while tremendous efforts have been made to scale up single-chip performance, the physical limitations of chip design, such as reticle size, mean that further gains are increasingly difficult. Therefore, scalable, high-performance *system interconnects* that can enable efficient scale-out of distributed compute systems has become increasingly critical.

For off-chip *memory interconnects*, a key tension exists between achieving high bandwidth and maintaining scalable memory capacity. As illustrated in Figure 1.3(a) and Figure 1.3(b), both the bandwidth and capacity of off-chip memory have improved significantly over the years across various technologies, each enabled by different types of interconnects and offering unique trade-offs. Double data rate (DDR) memory is typically integrated on separate circuit boards known as dual inline memory modules (DIMMs), which are installed on the motherboard of a compute node. This setup allows for flexible memory expansion, but the off-chip interconnect between the compute chip and the DIMMs limits bandwidth compared to more integrated memory technologies. Graphics double data rate (GDDR) memory, on the other hand, is integrated on the same board as the compute chip and is positioned close to its chip package. This shorter off-chip connection allows for higher bandwidths compared to DDR, making GDDR well-suited for bandwidth-hungry workloads such as graphics processing. However, expanding the memory capacity of GDDR is

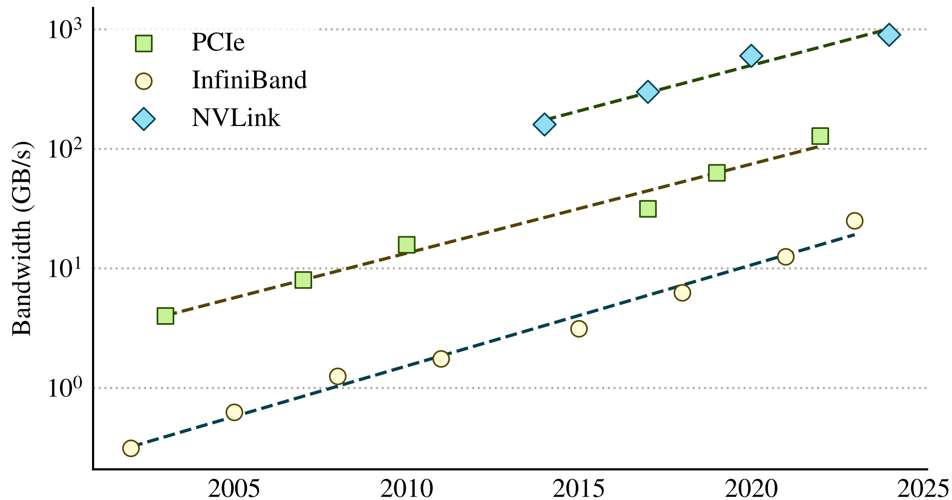
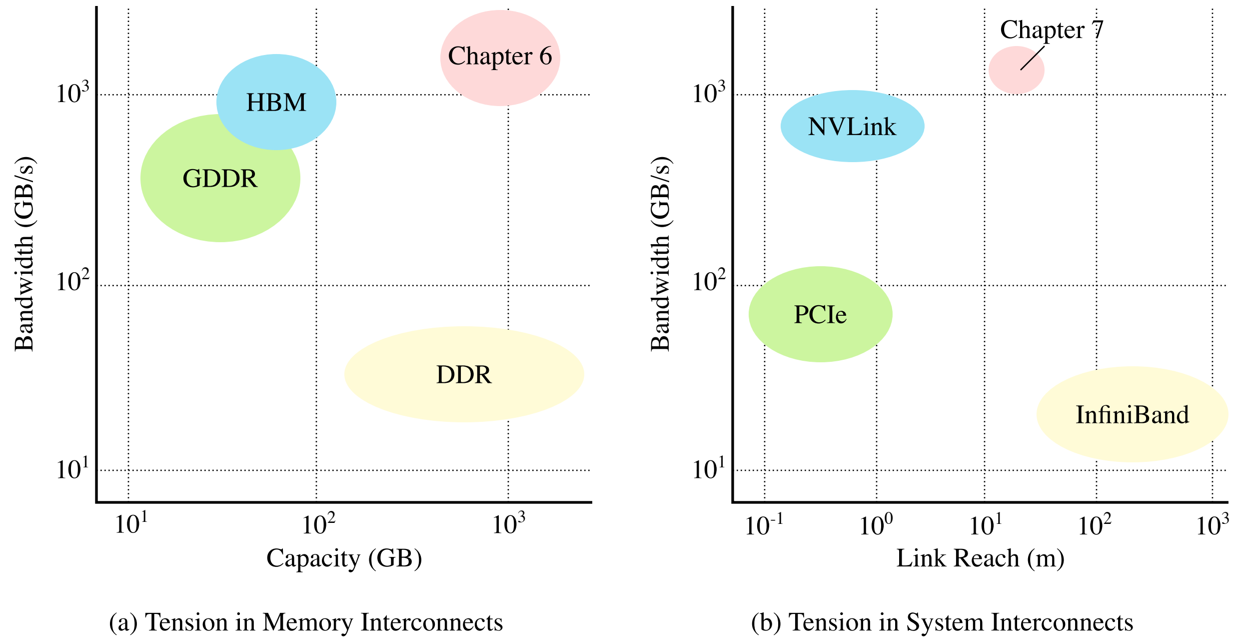


Figure 1.4: Trend of Off-Chip Interconnect Bandwidth – This plot shows how the bandwidth of selected off-chip interconnect technologies scales over time. Data is adapted from various online sources [Sha22, inf24, nvl24]

less flexible than DDR. High bandwidth memory (HBM) is a more recent memory technology and takes integration a step further by being tightly integrated within the same package as the compute chip. HBM is connected to the compute chip via short-reach high-density chiplet I/Os which enables significantly higher bandwidths compared to DDR and GDDR, making it indispensable for modern data-intensive applications. Although HBM achieves higher memory density compared to DDR and GDDR by leveraging advanced technologies such as 3D stacking and through-silicon vias (TSVs), the number of HBM stacks that can be integrated within a single package is fundamentally limited by the perimeter of the compute chip. This limitation makes it challenging to scale the memory capacity of HBM-based memory systems.

For off-chip *system interconnects*, a key tension exists between high bandwidth and link reach, which is closely related to the scalability of the interconnect. As illustrated in Figure 1.4, system interconnect technologies such as Peripheral Component Interconnect Express (PCIe), InfiniBand, and NVLink address specific requirements based on different use cases, and they have seen significant increase in bandwidth over the years. PCIe is commonly used for connecting different components, such as CPU, GPU, network cards, and storage devices. While PCIe has seen steady bandwidth improvements over the years, from a few GB/s in the late 1990s to 128 GB/s in the latest PCIe 6.0 standard, it is mostly limited to short-reach connections within a server node. InfiniBand is widely used for interconnecting different compute nodes over a long distance. While its bandwidth of InfiniBand has scaled similarly over the years, reaching up to 25 GB/s per port in the lasted



(a) Tension in Memory Interconnects

(b) Tension in System Interconnects

Figure 1.5: Tension in Off-Chip Interconnect Technologies – Conceptual illustration of the tension in off-chip interconnect technologies.

XDR version, the bandwidth offered by InfiniBand is still lower than short-reach interconnects like PCIe. Since the 2010s, the rise of deep learning and AI workloads demanding high throughput has led to the development of specialized high-bandwidth interconnects like NVLink, offering up to 900 GB/s per GPU. However, the limited link reach of NVLink restricts its scalability, and it is primarily used for tightly-coupled GPUs within a single node (e.g., DGX H100).

Figure 1.5 conceptually illustrates the tensions in various memory interconnect and system interconnect technologies. These tensions give rise to challenges in scaling off-chip interconnects, both in methodology and architecture.

Methodology Challenge – Developing an accurate model for large-scale distributed systems is challenging primarily due to the complex interactions between interconnected components. Unlike on-chip networks, where interactions are confined within a single chip, off-chip interconnects must manage data transfer across multiple devices, including CPUs, GPUs, memory modules, and storage units, often spanning multiple compute nodes. Modeling such a diverse range of interactions is difficult, as it involves considering multiple layers of communication across various interconnects in the system. Additionally, large-scale distributed systems with multiple compute nodes and sophisticated interconnects are not readily accessible to most researchers, making it challenging

to calibrate models against real hardware performance. This thesis focuses on the methodology challenge of modeling distributed LLM training.

Architecture Challenge – The design of off-chip interconnects involves navigating trade-offs between many aspects, such as latency, bandwidth, energy efficiency, and scalability. Modern workloads, however, demand high performance across almost all these metrics. Optimizing one of these aspects usually often comes at the expense of others. For instance, HBM-based memory systems, enabled by tightly integrated memory interconnect, offer exceptional bandwidth but suffer from limited memory capacity. High-bandwidth system interconnects such as NVLink come at the cost of limited scalability. Emerging technologies, such as tightly integrated silicon photonics, present opportunities to break some of these trade-offs. This thesis investigates the use of co-packaged optics to break these traditional trade-offs, advancing interconnect designs to meet the demands of future large-scale, high-performance workloads.

1.3 Thesis Overview

This thesis explores new methodologies, architectures, and silicon prototypes to address the challenges of scaling both on-chip and off-chip interconnects in modern computing systems. The work is divided into two main parts: the first focuses on OCNs for manycore architectures, while the second addresses off-chip interconnects, particularly for machine learning workloads. By presenting comprehensive frameworks and practical design solutions, this thesis aims to improve the scalability, performance, and efficiency of interconnects, and demonstrate their feasibility through silicon prototypes.

Chapter 2 presents PyOCN, a unified Python-based framework for modeling, testing, and evaluating OCNs. PyOCN enables rapid design-space exploration of OCNs by providing a library of highly parametrized router and network components, which can be easily configured and composed to form complex network topologies with various routing algorithms and flow control mechanisms.

Chapter 3 proposes a tiled physical design methodology for implementing low-diameter OCNs, which closes the gap between theoretical principle and practical implementation. I, concurrently with the work by Jung et. al [JDZ$^+$20], propose the ruche channel to fully exploit the VLSI wiring capability in modern technology nodes. It demonstrates that low-diameter OCNs for manycore processors can be realized by adapting mesh/torus topologies with concentration and ruche channels.

Through analytical modeling and realistic layout-level evaluations, I demonstrate that 2D-mesh topologies with modest concentration factors and modest length ruche channels can significantly reduce network diameter at similar area and bisection bandwidth.

Chapter 4 presents the CIFER chip tape-out, in which PyOCN was used for developing OCNs for different use cases. This chapter discusses the CIFER architecture, the OCN implementations, as well as the findings and reflections I made during the tape-out process.

Chapter 5 introduces LLMCompass-E2E, a performance evaluation framework for large-scale distributed training workloads. LLMCompass-E2E is built on top of the existing LLMCompass framework, which provides kernel-level performance model for inference. By incorporating a kernel-level compute graph intermediate representation (IR), kernel-level auto-gradient, and a pipeline scheduler simulator, LLMCompass-E2E effectively supports modeling the end-to-end distributed training performance on a given compute system.

Chapter 6 proposes optically connected multi-stack HBM modules that leverage co-packaged silicon photonics interconnect. The proposed design extends the HBM memory system off the compute interposer, circumventing the chip packaging constraint and allowing more HBM stacks to be connected to the compute chip while also improving off-chip bandwidth. Evaluations using LLMCompass-E2E show significant improvements in training and inference efficiency for large-scale large language models (LLMs).

Chapter 7 presents the PIPES silicon photonic tape-out, which validates the proposed off-chip interconnect in Chapter 6. This chapter details the PIPES system architecture as well as the design, implementation, and verification of the electrical interface chiplet (EIC).

Chapter 8 concludes the thesis by summarizing the contributions of this thesis and discussing future research directions. The primary contributions of this thesis are as follows:

- I develop PyOCN, a unified Python-based framework for modeling, testing, and evaluating on-chip networks that enables rapid design-space exploration.
- I propose and evaluate practical OCN topologies that reduce the network diameter while remaining practical to implement using a tiled physical design methodology.
- I demonstrate the benefits of PyOCN through the CIFER chip tape-out, highlighting practical design trade-offs and optimizations.

- I develop LLMCompass-E2E, a performance evaluation framework for large-scale distributed training workloads.
- I propose and evaluate the use of co-packaged silicon photonic interconnect to scale the memory capacity and bandwidth of HBM-based memory systems, which are essential for LLM workloads.
- I demonstrate a practical implementation of the proposed co-packaged silicon photonic interconnect through the PIPES tape-out.

1.4 Collaboration, Previous Publications, and Funding

This thesis would not have been possible without the support and mentorship of my advisor, Christopher Batten, as well as the collaboration of my colleagues at Cornell University and external partners. Throughout my Ph.D. journey, Christopher Batten was a constant source of inspiration and guidance, consistently shaping the direction and depth of my research. His feedback, along with our countless discussions, played a crucial role in refining my ideas and driving my work forward.

I was one of the core contributors to the PyMTL3 project, where I collaborated closely with Shunning Jiang and Peitian Pan to design and implement the PyMTL3 framework from scratch. Shunning Jiang led the project, spearheading the design and implementation of key components, including the core DSL, the pass mechanism, the scheduling and simulation passes, and the initial PyMTL3 standard library. Peitian Pan was responsible for designing and implementing the translation mechanism, enabling extensible conversion from the PyMTL3 DSL to various backends. I focused on developing the method-based interfaces, mixed-level modeling, the parameter mechanism within the DSL, and contributed to the creation of several IPs using PyMTL3. Shunning Jiang led the paper we published at IEEE Micro in 2020 [JPOB20].

I co-led the PyH2 (Python Hypothesis for Hardware) project with Shunning Jiang. Zac Hatfield-Dodds, the creator of the Hypothesis framework, provided valuable feedback and guidance on the integration of Hypothesis with PyMTL3. I carried out the initial pathfinding of PyH2. I worked on leveraging Hypothesis to effectively test hardware design generators, with the help of Cheng Tan. Peitian Pan and Kaishuo Cheng led the work on generating random instruction sequences to

automatically test processors. Xiaoyu Yan and Eric Tang led a case study on testing a PyMTL3 cache generator with PyH2. Yixiao Zhang contributed to exploring Hypothesis stateful testing for hardware data structures. Peitian Pan developed a random bug injector for evaluating PyH2. Shunning Jiang led the paper we published at IEEE Design & Test in 2020 [JOP+20].

I led the PyOCN project. I implemented the initial version of the PyOCN framework from scratch. Cheng Tan joined the project later and contributed to the development of several router components and network topologies. Shunning Jiang provided support and guidance on the PyMTL3 integration. Peitian Pan provided support on PyMTL3 translation which was essential for generating synthesizable Verilog. Christopher Torng and Shady Agwa helped with setting up the ASIC toolflow for area, energy, and timing analysis. Cheng Tan led the paper we published at ICCD 2019 [TOJ+19].

I led the tiled OCN project. I pushed various router and channel designs through the ASIC flow and built analytical models showing trade-offs in latency, bandwidth, and area. I designed and implemented the RTL model for the routers, hard macros, and networks. I conducted post-place-and-route evaluations for a variety of hard macros and network topologies. Shady Agwa provided guidance on hierarchical physical design and supported the chip-level results. I led the publication of our work at NOCS 2020 [OAB20].

I was a key contributor to the CIFER chip tape-out presented in Chapter 4. The CIFER project was a collaborative effort between Professor David Wentzlaff's research group at Princeton University and Professor Christopher Batten's research group at Cornell University. The Princeton team, including Ting-Jung Chang, Ang Li, Fei Gao, Georgios Tziantzioulis, Jinzheng Tu, Kaifeng Xu, Paul Jackson, August Ning, Grigory Chirkov, Marcelo Orenes-Vera, and Jonathan Balkind, led the overall project and was in charge of developing the Ariane tile, developing the embedded FPGA, top-level integration, physical design, and post-silicon testing. The Cornell team was in charge of developing the TinyCore cluster and the OCN. Tuan Ta led the TinyCore cluster development. He developed the TinyCore RTL model. Xiaoyu Yan, and Eric Tang developed the software-managed coherent cache in the TinyCore cluster, with help and guidance from Moyang Wang. Moyang Wang led the development of a task-parallel runtime system for the TinyCore cluster, with support from Tuan Ta. Shady Agwa and I assisted in setting up gate-level testing and helped with preliminary timing and area analysis by pushing the TinyCore cluster through an ASIC flow. I led the development of the tile-level OCN as well as the OCNs within the TinyCore

cluster. With tremendous help from Jonathan Balkind, I successfully integrated PyOCN into the OpenPiton framework. Fei Gao and Professor David Wentzlaff provided valuable feedback and guidance on the OCN design and timing optimization. Ting-Jung Chang and Ang Li led the papers we published at CICC and SSCL in 2023 [CLG+23, LCG+23].

I was a key contributor to the LLMCompass-E2E framework, which was an extension of the LLMCompass framework. I worked closely with the first author of LLMCompass, Hengrui Zhang from Professor David Wentzlaff's group to extend the original LLMCompass framework to support end-to-end distributed training performance modeling. Hengrui Zhang provided guidance on the LLMCompass framework and we had many productive discussions on distributed LLM training. I refactored the original LLMCompass software model and added a kernel-level compute graph IR. I implemented kernel-level auto-gradient in LLMCompass. I re-engineered the frontend of the framework to be more automated. I implemented an event-driven simulator for generating pipeline schedules which is critical for modeling the pipeline parallelism. I implemented network models for simulating the communication overhead for data parallelism and pipeline parallelism. I added support for modeling different activation recomputation strategies.

I led the optically connected HBM project. I had a lot of useful brainstorming sessions with Austin Rovinski at the beginning of the project. Yuyang Wang and Songli Wang taught me a lot about silicon photonics. Hengrui Zhang provided support on LLMCompass and we had many useful discussions on the system design. I implemented the optically connected HBM model in LLMCompass-E2E and conducted various experiments to explore its benefit in both LLM training and inference. I led our paper submission to IEEE CAL. Professor David Wentzlaff and Austin Rovinski provided valuable feedback on the paper.

I was a key contributor to the PIPES silicon photonic tape-out, presented in Chapter 7. PIPES was led by Intel and was a collaborative effort between Professor Keren Bergman's research group at Columbia University, Professor Alyosha Molnar's research group at Cornell University, and Professor Christopher Batten's research group at Cornell University. Kaveh Hosseini and Tim Tri Hoang from Intel led the overall project and provided valuable feedback and support throughout the project. The Columbia team led the development of the PIC and the Cornell teams led the development of the EIC. For Professor Keren Bergman's group, Songli Wang and Asher Novick led the development of the modulators in the PIC. Robert Parsons led the development of the ring filters in the PIC and wafer-level testing. Songli Wang and Yuyang Wang led the development of the

interleavers and de-interleavers in the PIC. For Professor Alyosha Molnar's group, Hamilton Lee and Luke James designed the offset DAC. Daria Sansoterra worked on current mirrors. Christine Ou, Devesh Khilwani, and Sunwoo Lee led the development of the TRX unit in the EIC. I co-led the EIC top-level integration and physical design with Austin Rovinski. I implemented the RTL model of the EIC, including an SPI configuration unit, a configuration network, an AIB controller, AIB interface units, and TRX interface units. Austin Rovinski implemented the RTL model of the mesochronous buffer. I led the EIC top-level pre-silicon verification. Ching-Chi Chang provided guidance on setting up the test bench and helped significantly with debugging the AIB unit. I worked closely with Christine Ou and Devesh Khilwani to design a digital testing interface for the TRX unit. I implemented most of the testing infrastructure and the test cases. Nicholas Cebry also contributed many test cases to pre-silicon verification. I co-led the top-level AISC flow and physical design of the EIC with Austin Rovinski. I worked closely with Christine Ou and Devesh Khilwani to create a mixed-signal characterization flow. I implemented the hard macros for AIB interface units and TRX interface units. Khalid Al-Hawaj helped with initial ASIC flow pathfinding. The physical design of the EIC could not have been done without Austin Rovinski's contribution to power planning, I/O placement, redistribution-layer (RDL) routing, and design rule violations fixing. Sung-Gun Cho from Intel also helped significantly with the ASIC flow. Sung-Gun Cho, Sunwoo Lee, and Austin Rovinski made tremendous contributions to fixing the design rule violations. I led the post-silicon testing of the crossbar unit in the EIC. I developed all the post-silicon testing infrastructure and test cases. I verified the SPI configuration unit on an FPGA before the chip was taped out. I conducted unit testing of the crossbar unit. I worked closely with Christine Ou and Devesh Khilwani to conduct integration testing of the crossbar unit and TRX unit. I worked closely with Songli Wang and Yuyang Wang to conduct integration testing of the EIC and PIC.

This thesis was supported in part by DARPA POSH Award #FA8650-18-2-7852, DARPA PIPES Award HR00111920014, DARPA CHIPS Award HR00111830002, DARPA SDH Award #FA8650-18-2-7863, NSF EVE Award #CCF-2008471, NSF PPoSS Award #CCF-2118709, equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or rec-

ommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

PART I
SCALING ON-CHIP INTERCONNECTS

The first part of this thesis focuses on addressing the methodology and architecture challenges in scaling on-chip interconnects. As the demand for processing power continues to grow, manycore architectures have emerged as a solution to enhance parallelism and throughput within a single chip. However, the efficiency of these architectures is heavily dependent on the performance of on-chip interconnects, which play a critical role in facilitating data movement between cores. Scaling on-chip interconnects to accommodate increasing core counts while maintaining low latency, high bandwidth, and energy efficiency poses significant challenges. Part I of the thesis addresses these challenges by presenting new methodologies and architectures that enhance the scalability and performance of OCNs. Chapter 2 presents PyOCN, a unified framework for modeling, testing, and evaluating on-chip interconnects. Chapter 3 proposes practical low-diameter network topologies along with a tiled physical design methodology to effectively implement them and explores their trade-offs using PyOCN. Finally, Chapter 4 details the CIFER chip tape-out, a heterogeneous manycore processor implemented using a tiled physical design methodology and validates the feasibility of the PyOCN framework.

CHAPTER 2
METHODOLOGY: A UNIFIED FRAMEWORK FOR ON-CHIP NETWORKS

There is a growing interest in the open-source hardware movement to amortize non-recurring engineering costs by using plug-and-play system-on-chip (SoC) designs, where the communication among different components is provided by an on-chip interconnection network. Unfortunately, building an on-chip network (OCN) that is suitable for a specific SoC design requires the exploration of a large number of design options and involves diverse research methodologies to evaluate performance, area, energy, and timing. In this chapter, I present PyOCN, a unified framework that vertically integrates multiple research methodologies to enable productively exploring the OCN design space. PyOCN is the first comprehensive framework for modeling (e.g., functional-level, cycle-level, and register-transfer-level), testing (e.g., unit testing, integration testing, and property-based random testing), and evaluating (e.g., simulating, generating, and characterizing) on-chip interconnection networks. We use a case study based on a 64-terminal butterfly network to illustrate the key features of PyOCN and to demonstrate the framework's potential in productively modeling, testing, and evaluating OCNs. PyOCN is further used in developing the OCNs in Chapter 3 and Chapter 4.

2.1 Introduction

On-chip networks (OCNs) play a significant role in chip design across many different domains. Embedded SoCs can include tens of homogeneous or heterogeneous cores to meet performance and power requirements [Gre11, TKMP18], high-end cloud servers can include tens to hundreds of cores to enable high-performance computing [Bol12, WKP11], and accelerators can include hundreds of processing elements for domain-specific computing [KTK+18, CKES16, CDS+14, KSK18]. At the same time, the costs of chip design and verification are rising. In response, there is growing interest in open-source hardware design based on plug-and-play SoC frameworks, where the communication between components is provided by an on-chip interconnection network.

Unfortunately, building an OCN that is suitable for a specific SoC design requires exploring a large design space (e.g., network size, channel bandwidth, topologies, routing algorithms, flow control schemes, arbitration techniques, physical floorplanning, and wire routing) using a combina-

tion of high- and low-level modeling to accurately estimate performance, area, energy, and timing. For example, OCN cycle-level simulators are widely used today and provide rich configuration options for early-stage design-space exploration [APM+12, AKPJ09, CHB+10, JBM+13, TB12]. However, the convenience in using CL models must be balanced against decreased accuracy and no path to real hardware implementations. There are a number of OCN register-transfer-level (RTL) generators that produce synthesizable Verilog to drive an evaluation of area, energy, and timing [cor19, fle19, CP04, PH13, KK17, FFDMS14]. These low-level generators can be difficult to use and lack support for fast simulation. Some OCN design frameworks combine various research methodologies together to facilitate design space exploration [BJM+05, PCSV08]. However, area, energy, and timing characterization in these frameworks is often based on high-level first-order modeling. There is a growing need for a vertically integrated OCN framework that can effectively characterize performance, area, energy, and timing across a large design space.

This paper presents PyOCN, a unified framework for modeling, testing, and evaluating on-chip interconnection networks. The concrete contributions of this work are the following: (1) PyOCN enables multi-level modeling to facilitate rapid design-space exploration and OCN implementation; (2) PyOCN provides sophisticated test harnesses for testing OCN designs modeled at different abstraction levels; (3) PyOCN can simulate OCNs at various abstraction levels, generate synthesizable Verilog, and drive a commercial standard-cell-based toolflow for characterizing OCN area, energy, and timing.

2.2 Related Work

Table 2.1 summarizes the state-of-the-art OCN research methodologies and compares them to PyOCN.

2.2.1 Modeling OCNs

Existing state-of-the-art OCN simulators struggle to balance rapid design-space exploration (requiring high-level design abstractions) and accurate estimation of area, energy, and timing (requiring low-level detailed modeling).

	Framework	Modeling							Testing			Evaluating			Open-source
		Lang.	Topology	Routing	FL	CL	RTL	PL	Unit	Int.	PBT.	Sim.	RTL Gen.	ASIC Char.	
Simulation	**BookSim2 [JBM^{+}13]**	C++	Xbar, Ring, (C)Mesh, Butterfly, Torus, Tree	DOR, Customized	○	●	○	○	○	○	○	●	○	○	●
	Garnet [AKPJ09]	C++, Python	Xbar, Mesh, Customized	DOR, Customized	○	●	○	○	○	●	○	●	○	○	●
	Noxim [CMM^{+}05]	SystemC	Mesh, Butterfly, Wireless	DOR, Odd-Even, Dyad routing	○	●	○	○	○	●	○	●	○	●	●
Generation	**FlexNoC [fle19]**	?	Application-specific	n/a	○	○	●	○	?	?	?	◒	●	○	○
	NoCGEN [CP04]	HDL	Mesh, Customized topology	DOR routing	○	○	●	○	?	?	?	◒	●	○	○
	Connect [PH13]	BSV	Customized topology	Customized routing	○	○	●	○	?	?	?	◒	●	○	◒
	OpenPiton [BMF^{+}16]	Verilog	Xbar, Mesh	DOR routing	○	○	◒	○	○	●	○	◒	●	●	●
	Netmaker [net19]	System-Verilog	Mesh	DOR routing	○	○	◒	○	○	●	○	◒	●	○	●
	OpenSMART [KK17]	BSV Chisel	Mesh, Customized topology	DOR, Source routing	○	○	●	○	●	●	○	◒	●	○	●
	OpenSoC Fabric [FFDMS14]	Chisel	Mesh, Flattened butterfly	DOR routing, Concentration	○	●	●	○	●	●	○	◒	●	○	●
Charact.	**DSENT [SCK^{+}12]**	C++	n/a	n/a	○	○	○	○	○	○	○	○	○	◒	●
	Orion2.0 [KLPS09]	C++	n/a	n/a	○	○	○	○	○	○	○	○	○	◒	●
	COSI [PCSV08]	C++	Application-specific	n/a	○	●	●	●	○	○	○	●	●	◒	●
	NetChip [BJM^{+}05]	SystemC	Application-specific	n/a	○	●	●	○	?	?	?	●	●	◒	○
	PyOCN	PyMTL	Xbar, Ring, (C)Mesh, Butterfly, Torus, Customized topology	DOR, Source, Customized routing	●	●	●	●	●	●	●	●	●	●	●

Table 2.1: Comparison with Prior Art – Different state-of-the-art research methodologies for designing OCNs, which are categorized into three groups (i.e., Simulation, Generation, and Characterization). ○, ◒, and ● indicate the corresponding feature is not supported, partially supported, and fully supported, respectively. For example, OpenSoC Fabric can generate synthesizable Verilog (●) but relies on VCS for simulation (◒). In contrast, the simulation in PyOCN allows the test bench to be written in Python and eliminates any semantic gap. Lang. = language; FL = functional level; CL = cycle level; RTL = register-transfer level; PL = physical level; Unit = unit testing; Int. = integration testing; PBT. = property-based random testing; Sim. = simulation; RTL Gen. = RTL generation; ASIC Char. = ASIC characterization.

Cycle-Level Modeling – Many widely used on-chip network simulators use CL modeling for early design-space exploration while verifying functional- and cycle-level behavior [APM+12, AKPJ09, CHB+10, JBM+13, TB12, LSC+10]. Unfortunately, these simulators do not support RTL modeling and cannot easily generate synthesizable Verilog, which is essential for accurate evaluation of area, energy, and timing. As an exception, Noxim [CMM+05] is a cycle-level OCN simulator developed in SystemC with some capacity for power estimation. All basic elements of the OCN in Noxim are also modeled in VHDL and are synthesized with a 65 nm CMOS standard cell library at 1GHz to provide statistical power analysis.

Register-Transfer-Level Modeling – On the other hand, OCN generators use RTL modeling to accurately characterize area, energy, and timing, but they lack the high-level design abstractions that enable fast design-space exploration [CP04, PH13, KK17]. For example, OpenSMART [KK17] is an OCN RTL generator for a wide range of different network configurations. Unfortunately, simulating generated RTL can easily limit rapid design-space exploration over large parameter space.

Physical-Level Modeling – Finally, OCN frameworks rarely take physical-level (PL) modeling considerations into account (e.g., macro- and micro-floorplanning), which is critical for effectively building complex OCNs. One exception is SUNMAP [MM04], which enables PL modeling in OCN generation and uses a floorplanning algorithm [AM03] to minimize the estimated area and wire lengths for specific applications.

2.2.2 Testing OCNs

Debugging OCNs can be time-consuming and tedious, as common problems (e.g., deadlock, fairness) can be hard to trigger and the resulting trace can often contain hundreds of packets. Most OCN simulation, generation and characterization frameworks lack robust testing infrastructure to validate model correctness. Most of these frameworks only contain simple tests for a single router or a specific network instance. Many frameworks lack an automatic and systematic way to verify outputs.

2.2.3 Evaluating OCNs

Simulating OCNs – Simulation is provided by most OCN simulators using CL modeling [APM+12, AKPJ09, CHB+10, JBM+13, TB12, LSC+10]. Some multicore simulators [BBB+11, RLC+12] also integrate dedicated on-chip network simulators. Conventional OCN generators that generate synthesizable RTL code do not have the ability to simulate the generated model. They often require other Verilog or SystemC simulators to drive the simulation.

Generating OCNs – ARM's CoreLink Interconnect [cor19] and Arteris FlexNoC [fle19] are two commercial OCN generators that target mobile applications. NoCGEN [CP04] can generate VHDL code for both 2D and 3D mesh topologies based on a user-defined OCN specification in XML file format but with very limited configuration options. Connect [PH13] focuses on generating OCNs optimized for FPGA implementations. All the above OCN generators are closed-source, leading to limited visibility into the implementation details and the inability to extend the framework. Open-source OCN generators are emerging as part of the open-source hardware movement. OpenPiton [BMF+16] is an open-source many-core research framework that contains three 2D mesh OCNs to ensure deadlock-free operation and provide communication between the tiles for cache coherence, I/O and memory traffic, and inter-core interrupts. Netmaker [net19] is written in SystemVerilog and passes parameters by including a single parameter file in all modules. It provides testbenches and simulation for the entire OCN. OpenSMART [KK17] is an open-source OCN generator implemented in BSV and Chisel. It can generate SMART NoCs [KCKP14] to enable single-cycle multi-hop traversals in arbitrary topologies. However, no FL and CL simulation is provided in these prior works. OpenSoC Fabric [FFDMS14] provides an open-source OCN generator implemented in Chisel. It can generate both software (C++) and hardware (Verilog) models but without a native simulator. Users must work in multiple languages when writing testbenches in C++ or Verilog.

Parameterization is clearly a first-order concern in OCN simulators and generators. Most existing tools implement parameterization using a declarative OCN specification (e.g., configuration parameters in XML file format). Developers need to carefully modify the configuration file to make sure the new parameters can be properly passed down the hierarchy. This significantly increases the development effort especially for designing complex OCNs where different modules share the same parameter name (namespace collision) or the same modules produce differently parameterized outputs.

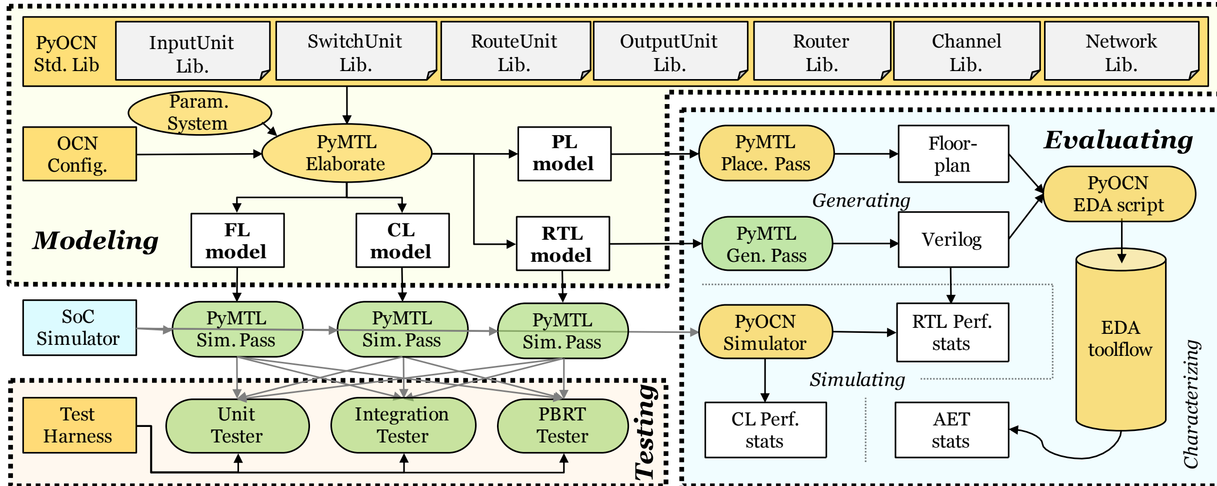


Figure 2.1: Overview of PyOCN Framework – PyOCN provides a library of OCN components to compose networks. PyOCN also provides unit test for each building block and integration test for the target OCNs. Property-based random test (PBRT) is used for stress testing network models. An OCN simulator is implemented for simulating different OCNs and backend scripts are provided to drive the EDA toolflows for area, energy, and timing characterization. PyOCN also features a parameterization system facilitating easy OCN configuration. Green components are included as part of the PyMTL framework. Orange components are added as part of the PyOCN framework.

Characterizing OCNs – Power- and area-models (e.g., Orion [WPM02], Orion2.0 [KLPS09], DSENT [SCK+12]) are widely used to characterize complete OCNs or OCN components (e.g., routers, channels) early in the design cycle. These frameworks can also be integrated into OCN generator frameworks for high-level optimization. For instance, COSI [PCSV08] is a synthesis framework for OCNs that embeds the power and area models derived from Orion to facilitate the OCN optimization. However, COSI targets synthesis without support for higher-level modeling. Similarly, SUNMAP [MM04] leverages XPipes [BB04] and Orion's power model to automatically generate SystemC descriptions of power-optimized network components. NetChip [BJM+05] integrates SUNMAP with the XPipesCompiler [JMBM04] to generate synthesizable Verilog for OCN designs. HotSniper [PH18] allows interval thermal simulation of many-cores. However, generation of synthesizable Verilog is not supported. High-level area, energy, and timing models enable early characterization, however, the lack of a detailed implementation leads to significant inaccuracy [KLN12, KLN15] and iterative development (Orion series [WPM02, KLPS09, KLN15]).

2.3 PyOCN Framework

PyOCN is a unified framework for modeling, testing, and evaluating on-chip interconnection networks. Figure 2.1 shows an overview of the PyOCN framework and illustrates its tight integration with PyMTL [LZB14, JIB18].

PyOCN extends the PyMTL framework with additional features suitable for *OCN design-space exploration*. Highly parameterized and modularized OCN components, modeled in FL, CL, RTL, and PL, serve as a standard library for building OCNs (see Section 2.4). In addition, PyOCN provides a comprehensive testing methodology based on unit testing, integration testing, and property-based random testing to test the FL, CL and RTL models (see Section 2.5). To evaluate different OCN designs, PyOCN can generate synthesizable Verilog with the geometry information for floorplanning based on the RTL and PL models via the generation pass and placement pass (see Section 2.6). A parameterization system is implemented to allow developers to flexibly parameterize any module instance. For characterizing OCN components and networks, PyOCN provides a set of electronic-design automation (EDA) scripts to drive a commercial standard-cell-based toolflows.

PyMTL is a unified hardware modeling framework. It leverages Python for behavioral specification, structural elaboration, and verification, enabling a rapid code-test-debug cycle for hardware modeling. PyMTL allows a designer to write the design under test (DUT) and test bench completely in Python for simulation and only transit to the traditional HDL workflow to push the DUT through an FPGA/ASIC toolflow. The simulation engine written in Python drastically reduces the iterative development cycle and eliminates any semantic gap.

2.4 PyOCN for Modeling OCNs

PyOCN provides a library of modular basic building blocks to compose OCNs. As shown in Figure 2.2, a router is composed of input units, route units, switch units, and output units. All these basic components have standardized latency insensitive interfaces so that each component can easily be replaced by user-customized components to create new networks. For example, if we want to implement a ring network with on/off flow control, instead of reimplementing the whole router, we only need to implement an input unit and an output unit that supports on/off flow control and swap them into the standard ring network which uses credit-based flow control. The modular

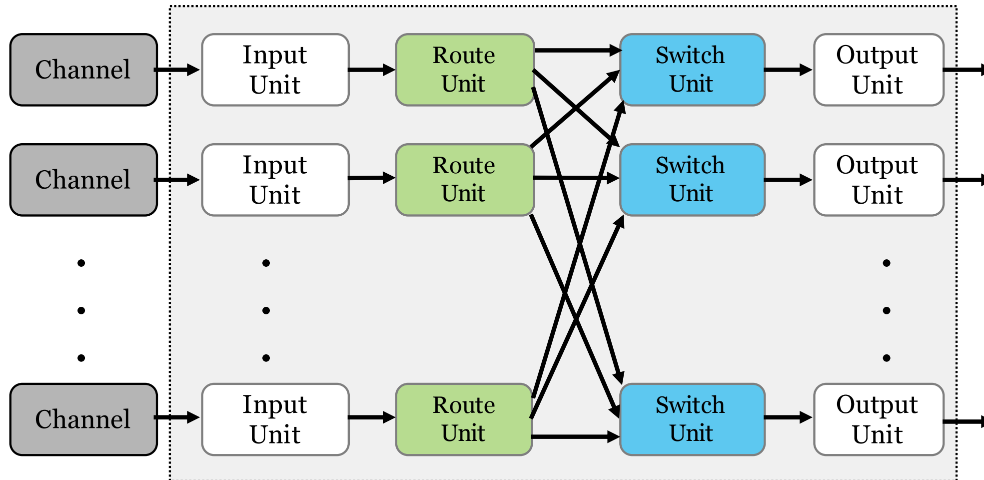


Figure 2.2: PyOCN Generic Router Architecture

```
1 def ringnet_fl( src_pkts ):
2   nterminals = len( src_pkts )
3   dst_pkts = [ [] for _ in range( nterminals ) ]
4
5   for packets in src_pkts:
6     for pkt in packets:
7       dst_pkts[ pkt.dst ].append( pkt )
8   return dst_pkts
```

Figure 2.3: FL Implementation of Ring Network – Simply redistributes an array of packet lists based on the destination field of each packet.

design approach also makes it easy to unit test the router, since we can test each basic component in isolation before we integrate them into a router.

By leveraging PyMTL, PyOCN is capable of modeling and generating OCNs at different levels of abstraction, including FL, CL, and RTL, in a unified environment, which enables a user to rapidly take an OCN design from concept to implementation. This section describes PyOCN's modeling approach spanning FL, CL, and RTL modeling.

Functional-Level Modeling – An FL network is essentially a magic crossbar. Figure 2.3 illustrates the FL implementation of a mesh network. PyOCN provides FL network models to enable early-stage validation and fast emulation of the model. We can write tests, check them first against the FL network, and then reuse these tests to verify CL and RTL networks at later design stages. Developing test cases with validation against FL network improves the credibility of these test cases. In other words, if the CL or RTL networks fail a test, it is more likely due to an error in the CL or RTL implementation, rather than an incorrect test case. Furthermore, our FL networks

```
class SwitchUnitCL( Component ):
  def construct( s, pkt_t, num_inports ):

    # Local parameters
    s.num_inports = num_inports

    # Interface
    s.get = [ \
        CallerIfc(pkt_t) for _ in range(num_inports) ]
    s.give = \
        CalleeIfc(pkt_t, method=s.give_, rdy=s.give_rdy)

    # Components
    s.priority = list( range(num_inports) )

    for i in range( num_inports ):
      s.add_constraints( M( s.get[i] ) == M( s.give ) )

  def give_rdy( s ):
    for i in range( s.num_inports ):
      if s.get[i].rdy():
        return True
    return False

  def give_( s ):
    for i in s.priority:
      if s.get[i].rdy():
        s.priority.append( s.priority.pop(i) )
        return s.get[i]()
```

Figure 2.4: CL Implementation of SwitchUnit – It is parametrized by the packet type and the number of inputs. It uses a list of integers `s.priority` to model a round-robin arbiter.

can also be composed with lower-level (i.e., CL and RTL) cores, memories, and accelerators to help develop end-to-end software that runs correctly on an SoC model.

Cycle-Level Modeling – PyOCN provides CL networks to facilitate rapid design-space exploration of cycle-level performance across a wide range of microarchitectural parameters, such as topology, routing algorithm, channel latency, type/size of input queues, and type of arbiters. The CL networks are built with the CL version of basic components. Figure 2.4 illustrates the implementation of a CL switch unit. Instead of using an arbiter, it simply instantiates a list of integers to model a round-robin arbiter. Our CL model is almost cycle-accurate (see Table 2.2 for an example), since most of the CL building blocks are cycle-accurate.

Register-Transfer-Level Modeling – PyOCN also provides RTL implementations of multiple networks for cycle-accurate performance evaluation and ASIC/FPGA synthesis. Similar to CL networks, the RTL networks are composed using the RTL version of the basic building blocks. Figure 2.5 shows the implementation of an RTL switch unit in the PyMTL domain-specific language

```
 1 class SwitchUnitRTL( Component ):
 2   def construct( s, pkt_t, num_inports ):
 3     # Local Parameters
 4     sel_width = clog2( num_inports )
 5     sel_t     = mk_bits( sel_width   )
 6     grant_t   = mk_bits( num_inports )
 7
 8     # Interface
 9     s.get = [GetIfc(pkt_t) for _ in range(num_inports)]
10     s.send = SendIfc(pkt_t)
11
12     # Components
13     s.arbiter = RoundRobinArbiterEn( num_inports )
14     s.mux = Mux( pkt_t, num_inports )(
15       out = s.send.msg,
16     )
17     s.encoder = Encoder( num_inports, sel_width )(
18       in_ = s.arbiter.grants,
19       out = s.mux.sel,
20     )
21
22     # Connections
23     for i in range( num_inports ):
24       s.connect( s.get[i].rdy, s.arbiter.reqs[i] )
25       s.connect( s.get[i].msg, s.mux.in_[i]      )
26
27     @s.update
28     def up_arb_send_en():
29       s.arbiter.en = \
30         ( s.arbiter.grants > grant_t(0) ) & s.send.rdy
31       s.send.en = \
32         ( s.arbiter.grants > grant_t(0) ) & s.send.rdy
33
34     @s.update
35     def up_get_en():
36       for i in range( num_inports ):
37         s.get[i].en = s.get[i].rdy & s.send.rdy & \
38                       ( s.mux.sel == sel_t(i) )
```

Figure 2.5: RTL Implementation of SwitchUnit – The switch unit implementation reuses the RTL arbiter, encoder, and mux from PyMTL's standard library.

Injection Rate	0.01	0.1	0.2	0.3	0.4
Performance	17.9	15.5	14.2	13.3	13.0
Accuracy	86%	87%	87%	97%	74%

Table 2.2: PyOCN Multi-Level Simulation – Normalized simulation performance (simulated cycles per second) and accuracy of average latency measurement modeled in CL with respect to RTL. The accuracy of the CL model is slightly degraded under very high load. The ideal throughput for the mesh network is 0.5

.

(DSL). PyMTL provides primitives similar to other hardware description languages: port-based interfaces for module encapsulation, structural connectivity for module composition, and combinational and synchronous concurrent blocks for logic description.

```
1  class RingNetworkRTL(Component):
2    def construct(s, pkt_t, pos_t, nrouters, chnl_lat=0):
3      ...
4    def elaborate_physical(s):
5      N = s.nrouters
6      chnl_len = s.channels[0].dim.w
7      for i, r in enumerate(s.routers):
8        if i < (N / 2):
9          r.dim.x = i * (r.dim.w + chnl_len)
10         r.dim.y = 0
11       else:
12         r.dim.x = (N - i - 1) * (r.dim.w + chnl_len)
13         r.dim.y = r.dim.h + chnl_len
14     s.dim.w = N/2 * r.dim.w + (N/2 - 1) * chnl_len
15     s.dim.h = 2 * r.dim.h + chnl_len
```

Figure 2.6: Physical Elaboration – Floorplanning code for parameterizable ring network. Geometry information is propagated hierarchically from each router and channel instance in the network component.

Physical-Level Modeling – Physical-level modeling (e.g., macro-/micro- floorplanning, cell tiling) is critical for effectively building complex OCNs. Without this kind of modeling, the structure in datapaths is destroyed by the automated place-and-route tools producing sub-optimal quality-of-results. We added a PyMTL placement pass to facilitate the physical-level modeling of the target network. The placement pass collects the geometry information of each network component and generates the floorplan script as shown in Figure 2.6.

2.5 PyOCN for Testing OCNs

PyOCN provides extensive test suites to unit test the basic network components as well as complete network instances. The highly modular design of PyOCN enables rigorous unit testing for each basic building block.

In addition, our test suites can be easily reused across all modeling levels including FL, CL, and RTL because the generated networks at different levels all have standardized interfaces. Since PyMTL is embedded in Python, PyOCN is able to leverage powerful python packages to facilitate test-driven design of our OCN models. In our framework, we extensively use *pytest* [pyt14] to generate and drive test cases and *hypothesis* [hyp19] to perform property-based random testing. This section describes PyOCN's testing strategy spanning unit, integration, and property-based random testing.

```
1   @pytest.mark.parametrize(
2     'pos_x, pos_y',
3     product( [ 0, 1, 2, 3 ], [ 0, 1, 2, 3 ] )
4   )
5   def test_simple_4x4( pos_x, pos_y ):
6     ncols = 4; nrows = 4
7     pkt_t = mk_mesh_pkt( ncols, nrows, nvcs=2 )
8
9     src_pkts = [
10      #     src_x  y  dst_x  y  opaque  vc  payload
11      pkt_t(    0, 0,     1, 1,      0,  0, 0xfaceb00c ),
12      pkt_t(    0, 2,     3, 3,      0,  0, 0xdeadface ),
13    ]
14
15    th = TestHarness( pkt_t, src_pkts )
16    # Use the elegant parameter system
17    th.set_param( "top.construct",
18      ncols=ncols, nrows=nrows,
19      pos_x=pos_x, pos_y=pos_y,
20    )
21    run_sim( th )
```

Figure 2.7: Unit Test for a Router in 4×4 Mesh – This simple test case injects two packets into the router. The test harness instantiates the router, injects the packets, and checks if the packets are ejected from the correct output ports.

Unit Testing – PyOCN provides unit tests not only for all network components such as routers and channels, but for basic components like input units and switch units. Figure 2.7 shows a simple example of one *unit test* for a router in a 4×4 mesh network. It simply injects two packets into the router and checks if they are ejected from the correct output ports. The pytest `@parametrize` decorator generates a number of test configurations from a single test definition. In this case, it generates 16 test cases that test routers with all possible positions in a 4×4 network. This test can be used for testing both CL and RTL routers. It can be reused for testing torus routers as well. The only change we need to make is to change the type of the design-under-test (DUT) in the test harness.

Integration Testing – PyOCN provides similar tests that integrate basic components into a router, compose routers and channels into a network, and then test the network as a whole. Many test cases are reusable across different topologies as they share the same FL model and have the same interfaces. RTL networks can reuse test cases for CL networks as PyMTL supports multi-level composition of CL and RTL interfaces.

Property-based Random Testing – Property-based random testing was first popularized by the Haskell library QuickCheck [CH11]. It works by using a type-based random data generator for all inputs and checking if the DUT violates the given specification. Figure 2.8 illustrates a sim-

```python
@hypothesis.given(
  ncols = st.integers(2, 8),
  nrows = st.integers(2, 8),
  pkts  = st.data(),
)
def test_hypothesis( ncols, nrows, pkts ):
  Pkt = mk_mesh_pkt( ncols, nrows, nvcs=2 )

  pkts_lst = pkts.draw(
    st.lists( mesh_pkt_strat( ncols, nrows ) ),
    label= "pkts"
  )

  src_pkts = mk_src_pkts( ncols, nrows, pkts_lst )
  dst_pkts = meshnet_fl( ncols, nrows, src_pkts )
  th = TestHarness( Pkt, ncols, nrows,
                    src_pkts, dst_pkts )
  run_sim( th )
```

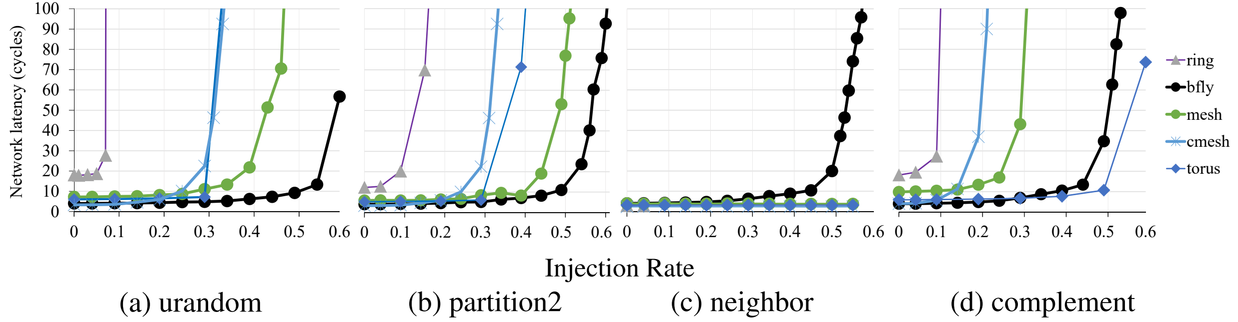
Figure 2.8: Property-Based Random Testing for Mesh Network Generator – This test shows a simple example of how PyOCN leverages hypothesis to test network generators. The test function randomly configures a mesh network and draws a list of packets as input. It verifies the DUT's output against the FL model which serves as an oracle.

ple example of how PyOCN leverages *hypothesis*, an open-source property-based random testing framework for Python. This tests more than a single network instance. Rather, it randomly configures mesh networks with different sizes on the fly and verifies the generated networks against the golden reference, which in this case is the FL model. *Hypothesis* produces readable and minimal counter-examples when encountering a failure. If it finds an example failing the specification, it takes that example and keeps simplifying it until it finds a minimal example that still triggers the problem.

2.6 PyOCN for Evaluating OCNs

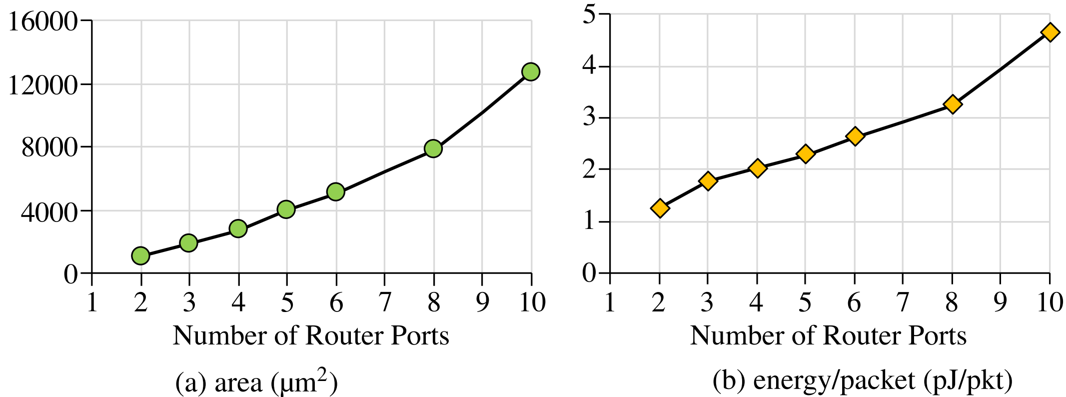
In addition to modeling and testing OCNs, PyOCN also supports evaluating OCNs using a combination of simulation (for cycle-level or cycle-accurate performance analysis), generation (for producing synthesizable RTL), and characterization (for area, energy, and timing analysis).

Simulating OCNs – The simulation in PyOCN is powered by PyMTL's simulation pass, which allows the test bench to be written in Python and eliminates any semantic gap. The simulation pass statically schedules and then calls the *@s.update* blocks every cycle. The PyOCN simulator can issue packets into different OCNs with different traffic patterns (e.g., uniform random (urandom), neighbor, partition by two (partition2), and complement) at different injection rates.



(a) urandom (b) partition2 (c) neighbor (d) complement

Figure 2.9: RTL Simulation Results – Average latency at different injection rates across different network topologies with 64 terminals. Mesh and torus both have eight rows and eight columns. Butterfly is 4-ary 3-fly. All topologies parameterize the channel latency as a single cycle and the router pipeline as a single cycle. Ring and torus leverage virtual channels to avoid deadlock.



(a) area (μm^2) (b) energy/packet (pJ/pkt)

Figure 2.10: Router Characterization – Characterization of area and energy for routers with different number of input and output ports targetting a 500 MHz clock frequency. The channel bandwidth is 32 b/cycle. During the RTL simulation, we generate hundreds of packets that traverse from each inport to each outport without contention. The dumped net activity file is passed into the EDA toolflow to drive the energy analysis.

Generating OCNs – PyOCN leverages the translation pass in PyMTL to generate synthesizable Verilog from RTL OCN models. PyOCN's *parameterization system* facilitates the configuration process of OCN components. In PyOCN, model implementations are defined as Python classes. The constructor registers each module in a dictionary based on its name and hierarchical position. The parameterization system can modify any parameter in any module registered in the dictionary by tagging a specific hierarchical component name with parameters. So instead of tediously carrying all the parameters down through the whole hierarchy during construction, developers are able to parameterize only a set of components or any single component *in the middle of hierarchy* after construction but before elaboration. During elaboration time, models are elaborated based on the updated parameters in each module.

Characterizing OCNs – PyOCN generates both synthesizable Verilog and a corresponding floorplan script that can be used to drive a commercial standard-cell-based toolflow for area, energy, and timing characterization. The PyOCN framework includes scripts for various commercial tools including Synopsys Design Compiler, Cadence Innovus, and Synopsys PrimeTime PX in order to synthesize, place, route, and estimate energy for the given design. PyOCN leverages open-source physical IP libraries including the 45 nm NanGate standard-cell library and the FreePDK45 physical design kit.

PyOCN's integration with a standard-cell-based toolflow enables highly accurate measurement of area, energy, and timing for the placed-and-routed gate-level netlist. Specifically, area and timing are both estimated post-place-and-route after meeting timing with Cadence Innovus's internal signoff-quality static timing analysis engines. Energy is estimated using Synopsys PrimeTime PX with the RTL-level switching activity information (provided by PyOCN) and the post-place-and-route gate-level netlist. The tool statistically propagates annotated switching activity to intermediate nodes before using gate/interconnect and parasitic RC information to estimate energy.

2.7 Case Study

With the help of PyOCN's standard library, an OCN can be easily configured and modeled at various abstraction levels. Currently, PyOCN supports crossbar, ring, mesh, concentrated mesh, torus, and butterfly topology models with extensive testing infrastructure. In this case study, we explore an OCN targeting a 64-terminal system.

PyOCN provides an OCN *simulator* for different topologies modeled at various levels. A developer can initially simulate the target design in CL to quickly estimate performance. Table 2.2 shows that the simulation speed for a 64-terminal mesh in CL is over 10× faster than an equivalent RTL model. The simulated performance of different topologies modeled in RTL is shown in Figure 2.9. In this case study, we choose to optimize for a unified random (i.e., *Urandom*) traffic pattern which might be representative of general memory traffic over an OCN interconnecting private L1 caches and a tiled, shared L2 cache. Given this context, we chose a butterfly OCN for further analysis.

PyOCN supports OCN *characterization* by providing scripts that semi-automatically takes the *generated* Verilog and net activity file to drive a standard-cell-based electronic-design-automation

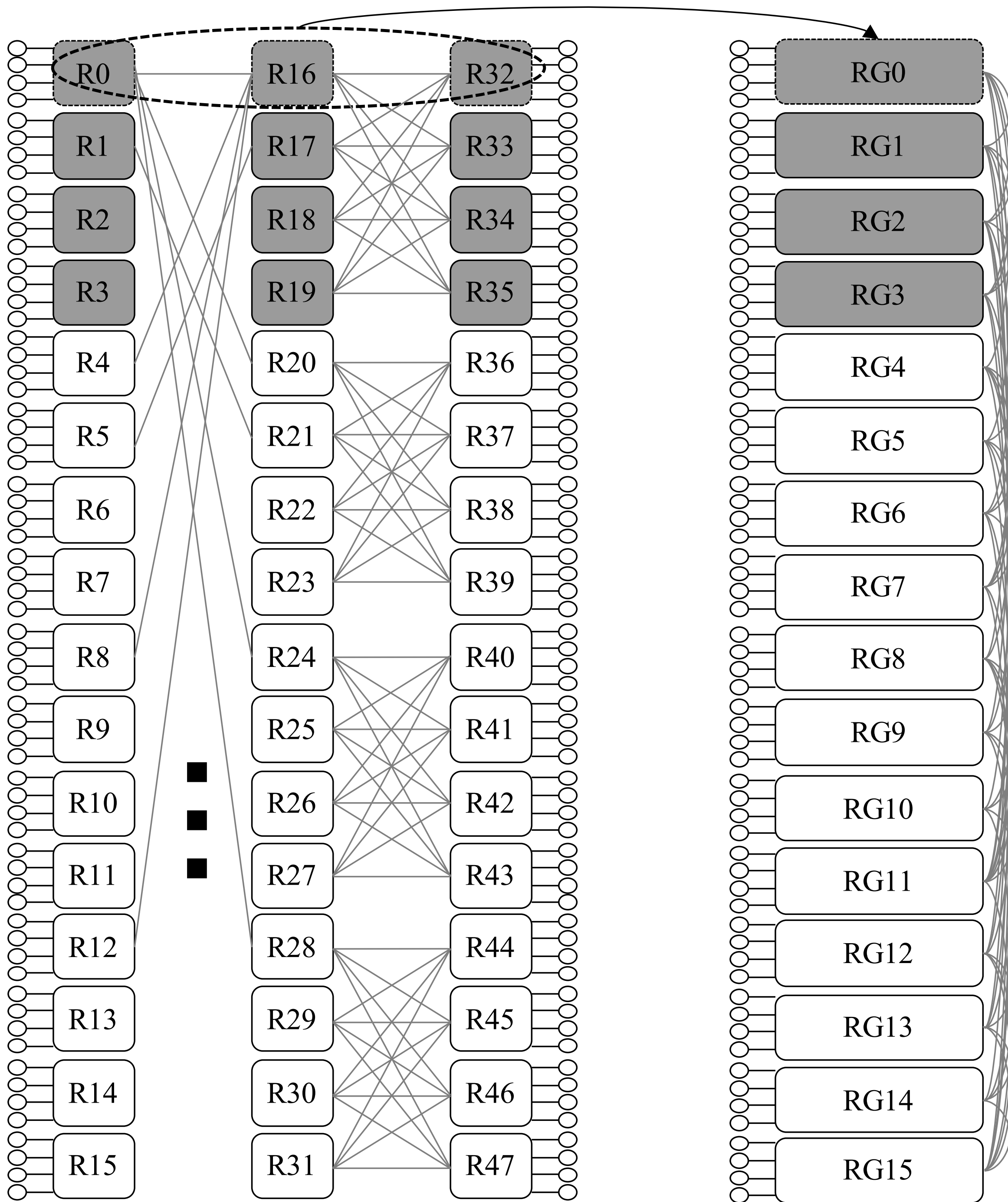


Figure 2.11: 4-ary 3-fly Butterfly Network – The routers in the same rows can be recognized as a router group. For simplicity, we use single line with two arrows to indicate bidirectional data delivery.

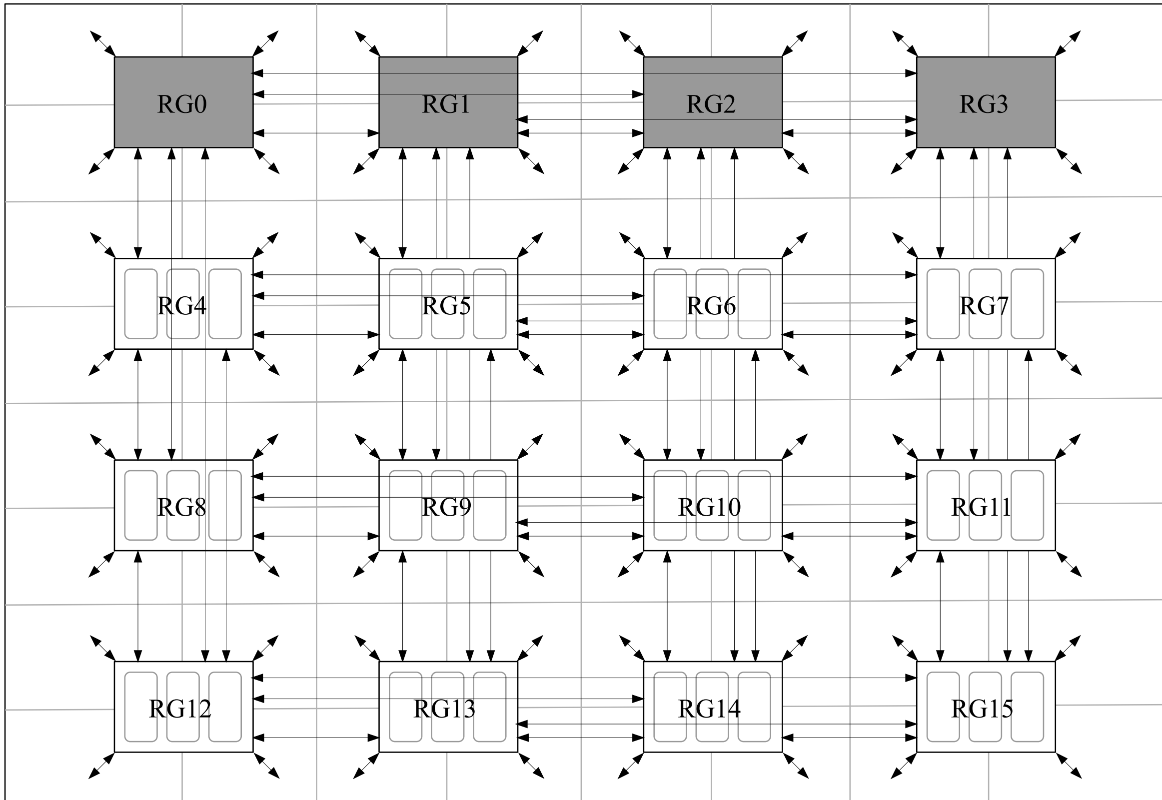


Figure 2.12: 4-ary 3-fly Butterfly Network Floorplan – Floorplan proposed in [KBD07].

```
1  net = BFlyNetworkRTL( pkt_t, k_ary=4, n_fly=3 )
2  critical_paths= [
3    "channels[82]",
4    "channels[114]",
5    ...
6  ]
7  for c in critical_paths:
8    net.set_param( f'top.{c}.construct', hops=2 )
9  net.elaborate()
```

Figure 2.13: Parameterization System Example – We collect all the critical paths violating the timing constraint reported by the EDA toolflow, add them into the `critical_paths`, and use `set_param` to change the number of channel queues on these channels.

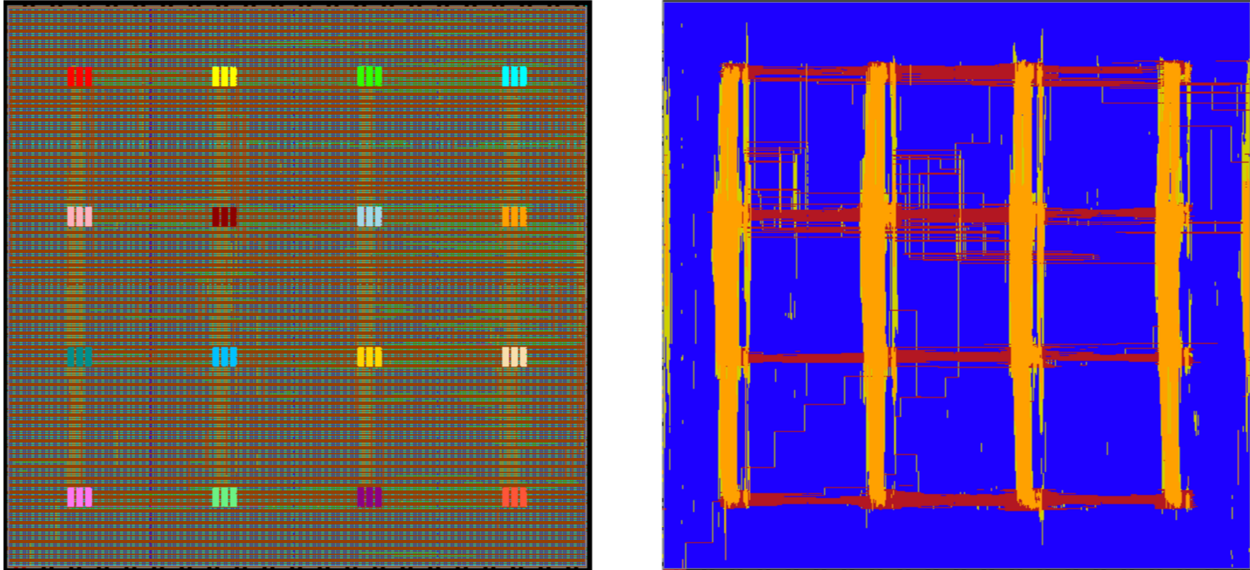


Figure 2.14: Post Place-and-Route Layout of 4-ary 3-fly Butterfly – The routers are highlighted. The floorplan is generated based on PyOCN PL modeling. PyOCN also provides script to semi-automatically drive the EDA toolflow to generate the final layout. The area is 4.8 mm x 4.6 mm and the operating frequency is 500 MHz @ 45 nm.

(EDA) toolflow for area, energy, and timing analysis. In this case study, we choose to use the FreePDK45 with the Nangate standard cell library. Figure 2.10 shows the area-energy analysis for a router with an increasing number of ports. Generally, the higher-radix routers require more area and energy per packet. We eventually decided to implement a 4-ary 3-fly rather than a 2-ary 6-fly as the zero load latency of 4-ary 3-fly is half the 2-ary 6-fly.

To place a 4-ary 3-fly butterfly, we group routers in the same row together as a router group (see Figure 2.11) and place them on the chip as shown in Figure 2.12, which is similar to the placement of the flattened butterfly topology proposed in [KBD07]. PyOCN's *PL modeling* can provide explicit geometry information for the placement of each router. We reserve 1 mm between router groups to provide enough space for the terminals. We initially target 500 MHz and set the channel latency to be one meaning that there is no channel queues between routers.

However, using the EDA toolflow revealed that a 2 ns timing constraint is not possible due to long channels between some router groups. The corresponding critical path starts from the input unit of Router28, goes through the channel, and ends at the input unit of Router46, with a negative slack of 0.13 ns. Although the channel between Router28 and Router47 seems longer according to the logical layout, the EDA toolflow's routing algorithm ultimately meant the critical path was

limited by the channel from Router28 to Router46. One straightforward way to break such critical paths is to add channel queues to channels that violate the timing constraint.

The *parameterization system* of PyOCN enables us to easily configure any network components without touching the source code of the target OCN design. As shown in Figure 2.13, only a few lines before the elaboration of the target OCN is needed to add channel queues to specific channels. Figure 2.14 shows the final layout of our target butterfly OCN, where the target frequency is achieved.

2.8 Conclusion

This paper has introduced PyOCN, a unified framework for simulating, testing, and characterizing on-chip interconnection networks. PyOCN is the first open-source framework for modeling (e.g., functional-level, cycle-level, and register-transfer-level), testing (e.g., unit testing, integration testing, and property-based random testing), and evaluating (e.g., simulating, generating, and characterizing) on-chip interconnection networks. PyOCN is an open-source framework and is available online at `https://github.com/cornell-brg/pymtl3-net`.

CHAPTER 3
ARCHITECTURE: LOW-DIAMETER ON-CHIP NETWORKS FOR MANYCORE PROCESSORS

This chapter address the architecture challenges of scaling on-chip interconnects for manycore processors. Manycore processors are now integrating up to 1000 simple cores into a single die, yet these processors still rely on high-diameter mesh on-chip networks (OCNs) without complex flow-control nor custom circuits due to three reasons: (1) manycores require simple, low-area routers; (2) manycores usually use standard-cell-based design; and (3) manycores use a tiled physical design methodology. In this chapter, we explore mesh and torus topologies with internal concentration and/or ruche channels that require low area overhead and can be implemented using a traditional standard-cell-based tiled physical design methodology. We use a combination of analytical and RTL modeling along with layout-level results for both hard macros and a 3×3 mm 256-terminal OCN in a 14-nm technology for twelve topologies. Critically, the networks we study use a tiled physical design methodology meaning they: (1) tile a homogeneous hard macro across the chip; (2) implement chip top-level routing between hard macros via short wires to neighboring macros; and (3) use timing closure for the hard macro to quickly close timing at the chip top-level. Our results suggest that a concentration factor of four and a ruche factor of two in a 2D-mesh topology can reduce latency by over $2\times$ at similar area and bisection bandwidth for both small and large messages compared to a 2D-mesh baseline.

3.1 Introduction

Today's network, embedded, and server processors already integrate tens of processor cores on a single chip, and there is growing interest in using a *manycore approach* to integrate an even larger number of relatively simple cores within a single die. Early manycore research prototypes included 16–110 cores [TLM+04,MFN+17,HDH+10,HVS+07,LSC+13], complemented by manycore processors in industry with 64–128 cores [BEA+08,WGH+07,SGC+16,Hal20,Whe20]. Recent research prototypes have scaled core counts by an order-of-magnitude including the 496-core Celerity [RZAH+19b], 1000-core KiloCore [BSP+17], and 1024-core Epiphany-V [Olo16]. The manycore approach has demonstrated significant improvements in energy efficiency and throughput per unit area for highly parallel workloads.

Almost all manycore processors use a simple 2D-mesh on-chip-network (OCN) topology [BEA+08, WGH+07,MFN+17,LSC+13,BSP+17,Whe20,Hal20,RZAH+19b] (possibly with limited external concentration [SGC+16, HVS+07]), scaling from a 4×4 mesh in the RAW processor [TLM+04] up to a 32×32 mesh in the Epiphany-V processor [Olo16]. It is well known that the high diameter of 2D-mesh topologies can significantly increase packet latency and thus reduce system-level performance [DT04]. Indeed, there is a rich body of literature proposing numerous techniques to reduce packet latency in on-chip networks. Novel OCN flow-control schemes [KPKJ07, MWM04, PD01] and/or OCN custom circuits [KS08, CPK+13] can be used to reduce router and channel latencies. Alternatively, novel OCN topologies can reduce the network diameter including concentrated mesh [BD06], fat-tree [BD06], flattened butterfly [KBD07], multi-drop express channels [GHKM09, GHKM11], Clos [KYAC11], Slim NoC [BHY+18], and asymmetric high-radix topologies [ADL+13]. However, this raises the question: **Why do manycore processor silicon implementations continue to use simple high-diameter on-chip networks given the potential benefit reported in the literature for adopting novel on-chip network flow-control schemes, custom circuits, and/or topologies?**

Based on our experiences contributing to the Celerity manycore processor [DXT+18,RZAH+19a, RZAH+19b] and building an open-source OCN generator [TOJ+19], we argue there are three primary reasons for this gap between principle and practice.

Manycores Require Simple, Low-Area Routers – Manycore processors by definition use simple cores leaving modest area for the OCN routers (e.g., 10% of chip area in [RZAH+19b, Olo16]). Therefore, manycore processors usually use single-stage routers [Olo16,BSP+17,HVS+07, RZAH+19b], and protocol deadlock is often through multiple physical networks [TLM+04,WGH+07, MFN+17, LSC+13] as opposed to using virtual channels. These simple single-stage OCN routers mitigate the need for complex flow-control schemes.

Manycores Use Standard-Cell-Based Design – Manycore processor design teams (and indeed chip design in general) have been steadily moving towards highly automated standard-cell-based design methodologies [MFN+17, LSC+13, Olo16, RZAH+19b]. Unfortunately, this complicates using more advanced circuit techniques in the literature to reduce router and/or channel latency.

Manycores Use a Tiled Physical Design Methodology – Physical design is a critical challenge in implementing manycore processors. A tiled physical design methodology is the key to

overcoming this challenge and has been used in multiple manycore implementations [MFN+17, LSC+13, Olo16, RZAH+19b]. A *tiled physical design methodology* adheres to the following constraints: (1) the design is based on tiling a homogeneous hard macro across the chip; (2) all chip top-level routing between hard macros must use short wires to neighboring macros; and (3) timing closure for the hard macro must imply timing closure at the chip top-level. Unfortunately, a tiled physical design methodology precludes using many low-diameter, high-radix topologies proposed in the literature which require long global channels routed at the chip top-level and/or heterogeneous hard macros.

In this chapter, we seek to close this gap between principle and practice by exploring techniques for implementing low-diameter on-chip networks for manycore processors based on low-area routers, standard-cell-based design, and a tiled physical design methodology. Section 3.2 describes how mesh and torus topologies with concentration and/or ruche channels can use a tiled physical design methodology. Ruche channels[1] are a novel technique concurrently proposed in this work and by Jung et. al in [JDZ+20] which provide dedicated channels for packets to skip past routers for efficient long distance communication. Ruche channels are better suited to on-chip networks using a traditional standard-cell-based physical design methodology compared to prior work on physical and virtual express channels [GHKM09, Dal91, KPKJ07]. Section 3.3 compares 12 topologies using an analytical model based on router and channel RTL implementations and a standard-cell-based flow. Section 3.4 uses PyOCN (an open-source OCN generator [TOJ+19]) to generate both hard-macro and full-chip layout for each topology suitable for use in a 3×3 mm 256-core manycore processor implemented in a 14-nm technology. Our results suggest that by leveraging a concentration factor of four and a ruche factor of one in a 2D-mesh topology, our approach can reduce latency by over $2\times$ at similar area and bisection bandwidth for both small and large messages compared to a 2D-mesh baseline.

3.2 Manycore OCN Topologies

Our target system is a manycore with 256 cores arranged in a 16×16 grid (see Figure 3.1(a)). Figure 3.1(b–m) illustrates the 12 topologies explored in this work. Figure 3.1(b) illustrates our

[1]Ruching involves gathering fabric in a repeating pattern to make a pleat or ruffle. The logical topology diagram for mesh networks with ruche channels (see Figure 3.1(e)) resembles a ruched garment.

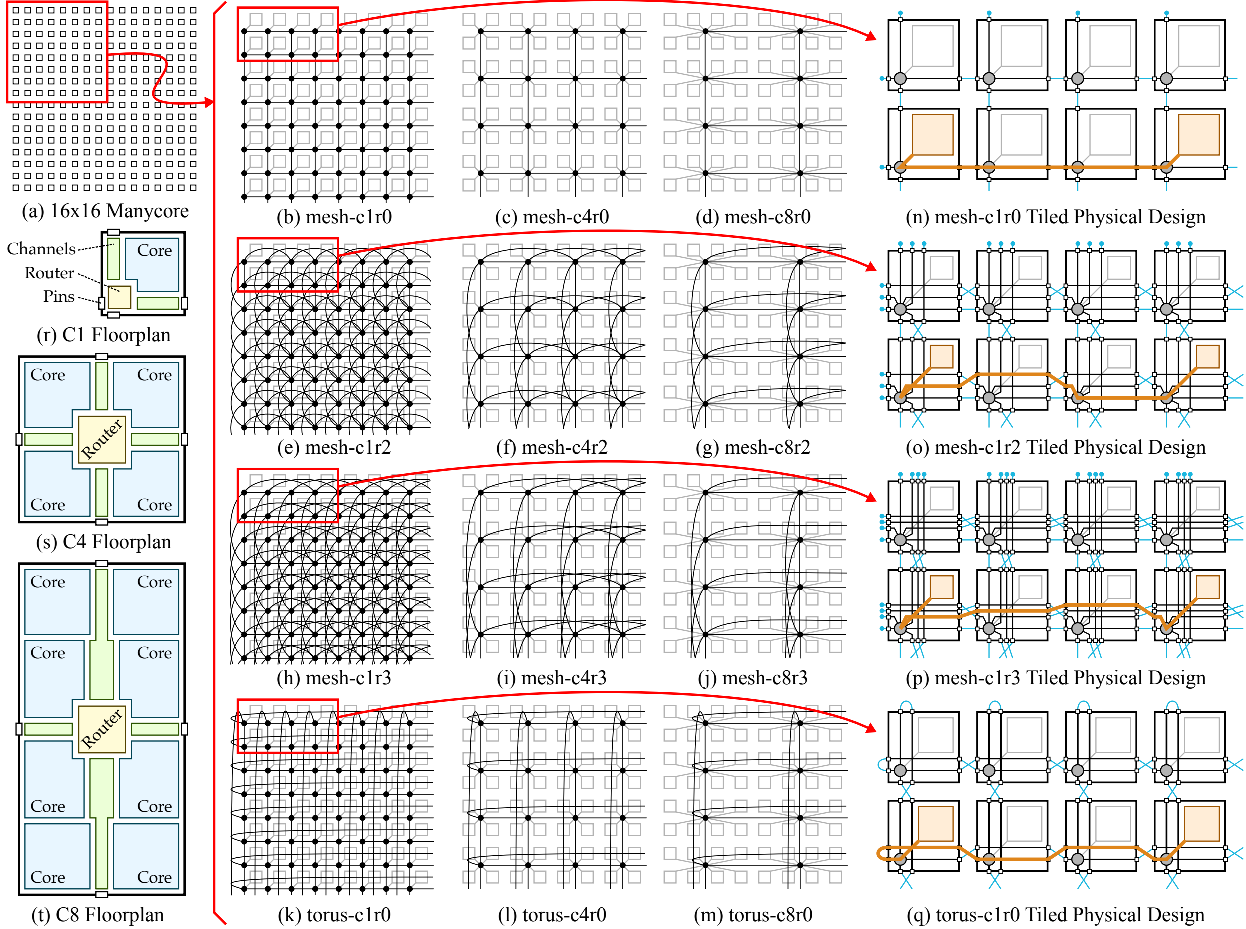


Figure 3.1: Twelve Topologies Implemented Using a Tiled Physical Design Methodology – (a) 16×16 manycore; (b–d) mesh with increasing concentration; (e–g) mesh w/ ruche factor of 2, increasing concentration; (h–j) mesh w/ ruche factor of 3, increasing concentration; (k–m) torus w/ increasing concentration; (n) = *mesh-c1r0* pin placement enables short chip top-level routing, unused channels terminated at top level; (o) = *mesh-c1r2* tile w/ feed-through channel, short cross-over chip top-level routing; (p) = *mesh-c1r3* tile w/ two feed-through channels, short cross-over chip top-level routing; (q) = *torus-c1r0* tile w/ folded torus, one feed-through channel, short cross-over chip top-level routing; (r–t) = macro floorplans for increasing concentration.

baseline 2D-mesh topology as implemented in most state-of-the-art manycore processors [BEA+08, WGH+07, MFN+17, LSC+13, BSP+17, Whe20, Hal20, RZAH+19b]. We use elastic-buffer flow-control [MBD09] and dimension-ordered routing on all mesh topologies.

We explore internal concentration where multiple cores share a single router [KPK+09]. Figure 3.1(c–d) illustrates a concentration factor of 4–8. Unlike external concentration, internal concentration reduces latency while maintaining per-terminal throughput and homogeneous channel bandwidths. Concentration reduces the number of routers, increases router radix, decreases the bisection channel count, and reduces network diameter.

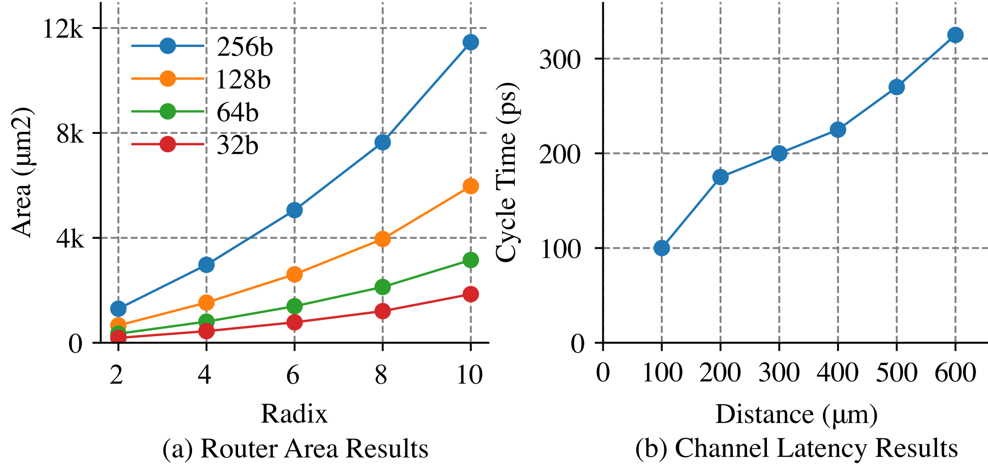


Figure 3.2: OCN Component-Level Results – (a) router area with different radices and port bitwidths under a 950 ps clock constraint; (b) minimum cycle time that can be achieved for queues manually placed at various distances with an auto-routed channel between them.

						B_B (Kb/cycle)				Area (%)			
Topology	N_R	r	N_{BC}	H_D	H_{avg}	32	64	128	256	32	64	128	256
mesh-c1r0	256	5	32	60	21.3	1	2	4	8	4.4	8.9	15.2	27.6
mesh-c4r0	64	8	16	28	10.5	0.5	1	2	4	2.0	3.9	7.3	13.6
mesh-c8r0	32	12	8	20	7.8	0.3	0.5	1	2	2.0	3.6	6.8	12.8
mesh-c1r2	256	9	96	32	11.6	3	6	12	24	11.6	20.6	36.5	61.2
mesh-c4r2	64	12	48	16	6.3	1.5	3	6	12	5.4	9.8	18.3	32.9
mesh-c8r2	32	16	24	12	4.9	0.8	1.5	3	6	4.6	8.1	15.5	28.1
mesh-c1r3	256	9	128	24	9.7	4	8	16	32	13.8	24.7	43.4	71.5
mesh-c4r3	64	12	64	12	6.0	2	4	8	16	6.7	12.2	22.6	40.5
mesh-c8r3	32	16	32	12	5.7	1	2	4	8	5.2	9.6	18.0	32.8
torus-c1r0	256	5	64	32	16.0	2	4	8	16	6.9	12.9	23.6	42.2
torus-c4r0	256	5	32	16	8.0	1	2	4	8	3.3	6.2	11.7	21.8
torus-c8r0	256	5	16	12	6.0	0.5	1	2	4	2.9	5.4	10.3	19.2

Table 3.1: Analytical Modeling Results – N_R = number of routers; r = router radix (i.e., number of ports per router); N_{BC} = number of bisection channels; H_D = diameter of the network; H_{avg} = average hop latency over all source/destination pairs; B_B = bisection bandwidth; Area = OCN area as a percentage of the full chip.

Ruche channels are a novel technique which add dedicated channels to skip past some number of routers. A *ruche factor* of two means each ruche channel skips to a router two hops away, while a ruche factor of three means each ruche channel skips to a router three hops away. A ruche factor of zero means there are no ruche channels, and a ruche factor of one means the ruche channel directly connects nearest neighbors. Figure 3.1(e,h) illustrates a ruche factor of two and three. Ruche channels maintain the number of routers, increase router radix, increase the bisection

channel count, and reduce network diameter. Ruche channels are related to but distinct from express channels [GHKM09, Dal91]. Ruche channels do not use separate interchanges and ensure all routers are homogeneous (i.e., all routers are a source and destination for exactly one ruche channel, ruche channels overlap). We use oblivious minimal routing on the ruche channels.

Figure 3.1(f–g,i–j) illustrates topologies that combine concentration and ruche channels. Finally, we explore 2D-torus topologies with similar concentration factors (see Figure 3.1(k–m)). We use minimal routing, credit-based flow-control, two virtual channels, and a dateline to avoid deadlock in the torus topologies. These 12 topologies provide a broad range of design points with different: topology styles (mesh/torus), numbers of routers, router radix, bisection channel count, channel lengths, and diameter. Figure 3.1 shows the naming convention we will use in the rest of the paper. For example, *mesh-c4r3* refers to a topology with a concentration factor of four and ruche factor of three. We will also use suffix such as *mesh-c4r3-b128* to refer to a topology with a channel bandwidth of 128 b/cycle.

Figure 3.1(n–q) illustrates how to map these 12 topologies to a tiled physical design methodology. Mapping *mesh-c1r0* simply requires careful placement of the pins for north, west, south, and east channels at the macro level to ensure short chip top-level routes (see Figure 3.1(n)). If $l \times l$ are the dimensions of each macro, then the channels in *mesh-c1r0* are approximately l long. Since all macros must be homogeneous, macros on the edge and corners require a few gates at the chip top-level to ensure input channels are never enabled and output channels are never ready. Mapping *mesh-c1r2* requires an additional set of north, west, south, and east channels, along with a set of feed-through channels (see Figure 3.1(o)). Again, careful placement of pins ensure short routes with a possible cross-over at the chip top-level. Ruche channels are approximately $2l$ long. Mapping *mesh-c1r3* requires an additional set of feed-through channels (see Figure 3.1(p)). Ruche channels are now approximately $3l$ long. Finally, mapping *torus-c1r0* requires just one set of north, west, south, and east channels along with one set of feed-through channels. Most channels are approximately $2l$ long, although the channels at the edges may be slightly shorter or longer due to the chip top-level wrap-around routing. While adding ruche channels to torus topologies is possible, it can be challenging to map these ruche channels into a tiled design methodology and/or to route on these topologies. Figures 3.1(r–t) illustrates floorplans which enable using the tiled physical designs in Figures 3.1(n–q) at higher concentration factors.

Our approach meets the three constraints of a tiled physical design methodology. First, all topologies can be implemented using a homogeneous hard macro which can then be tiled across the chip. Second, all chip top-level routing between hard macros is either short straight routing, short cross-over routing, or short wrap-around routing. Third, assuming careful consideration of timing constraints on register-to-output, input-to-register, and input-to-output paths, timing closure for all hard macros can imply timing closure at the chip top-level.

3.3 Manycore OCN Analytical Modeling

In this section, we explore trade-offs across different topologies using analytical modeling before presenting more realistic layout-level results in Section 3.4. To choose an appropriate core area, we implemented a RISC-V RV32IMAF in-order single-issue processor with 4KB instruction and data caches in RTL using PyMTL3 [JPOB20] and then used a commercial standard-cell-based toolflow in a 14-nm technology. The resulting area is 37,029 μm^2 which roughly corresponds to a 3×3 mm chip area for 256 cores. This per-core area is roughly $1.5\times$ larger than the per-core area in Celerity [RZAH+19b], but this is expected since Celerity does not support floating point and uses scratchpad memories instead of caches. Our per-core area is roughly $3.3\times$ smaller than the per-core area in Epiphany-V [Olo16], but again this is expected since Epiphany-V implements a 64-bit instruction set, supports dual-issue, and includes 64 KB of SRAM per core. Ultimately, we chose a tile size of $185 \times 185\,\mu m$ which is a reasonable target in between prior manycore implementations. We target a 1 GHz clock frequency which is comparable to the Celerity clock frequency when running at nominal voltage [RZAH+19b, DXT+18].

We construct an analytical model for area, zero-load latency, and bisection bandwidth based on the OCN component-level results shown in Figure 3.2. We model the channel latency as a function of distance between routers. We measured the minimum delay using static-timing analysis that can be achieved for two queues that are manually placed at various distances with an auto-routed channel between them. We use this data to estimate the number of channel queues that need to be inserted in each channel to meet the target 1 GHz clock frequency. We model the area of the OCNs as a function of router radix and channel bandwidth. We pushed a number of OCN routers from Py-OCN [TOJ+19] with different radices and port bitwidths through the ASIC toolflow using a 950 ps timing constraint. We use the post-place-and-route area information as an estimate of the buffering

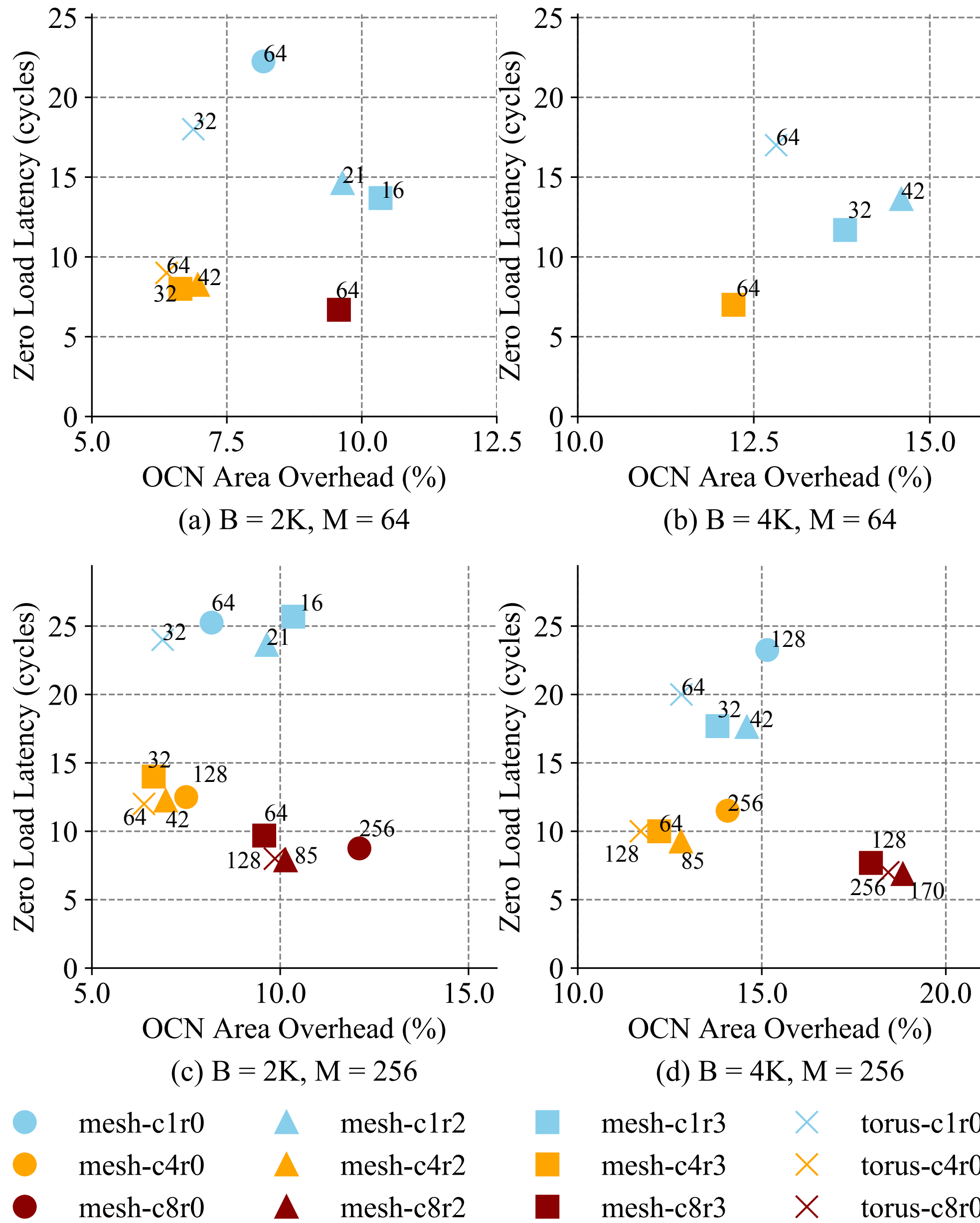


Figure 3.3: Latency and Area Trade-Offs – The zero-load latency and area overhead are compared for both 64-bit message and 256-bit message, with the bisection bandwidth normalized to 2 Kb/cycle and 4 Kb/cycle. Each topology is labeled with the channel bandwidth that corresponds to the normalized bisection bandwidth. Topologies that cannot reach the given bisection bandwidth even with channel bandwidth equal to the message size are not shown in the plot. B = normalized bisection bandwidth in bits per cycle; M = message size in bits.

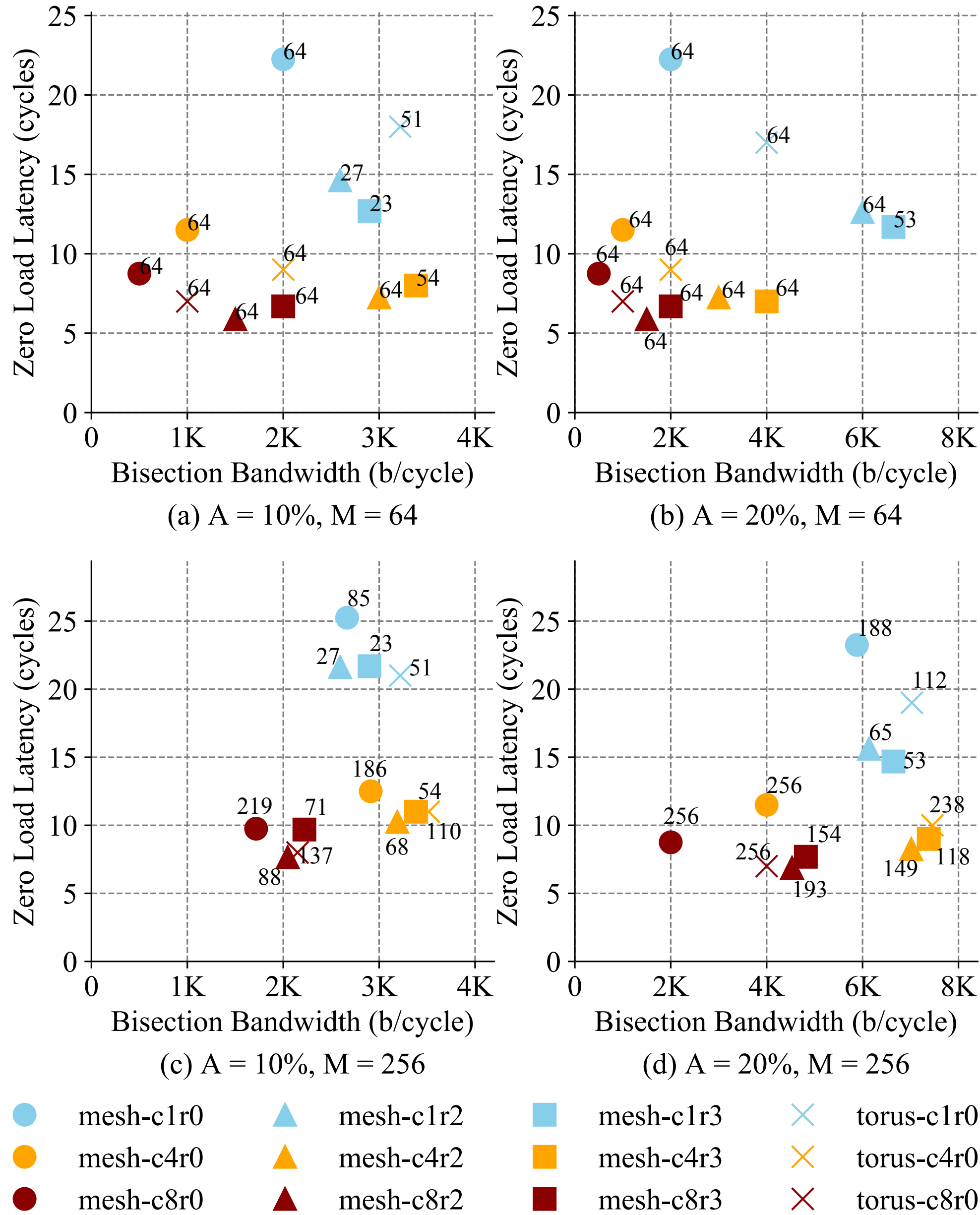


Figure 3.4: Latency and Bandwidth Trade-Offs – Zero-load latency and bisection bandwidth are compared for both 64-bit messages and 256-bit messages, with maximum area overhead for the OCN constrained at 10% and 20% of the total chip area. Each topology is labeled with the corresponding channel bandwidth that either reaches the maximum overhead or reaches the message size. A = maximum OCN area overhead; M = message size in bits.

and switching logic in the router. With the pitch and minimum width information of the metal layers that are used for local routing, we can calculate the linear wire density and estimate the area that needs to be reserved for channel pins on each edge of the hard macro. Based on the floorplans shown in Figure 3.1(r–t), we can calculate the area that is used by the OCN (both for the router and channels). We also interpolate and extrapolate component-level results to estimate the area of a router with any given radix and bitwidth. We calculate the zero-load latency under uniform random traffic and bisection bandwidth as described in [DT04]. In the analytical model, we assume one-cycle router latency and at least one-cycle channel latency. Topologies with short channels and low radix can potentially achieve a zero-cycle channel latency (i.e., router buffering/arbitration and channel traversal can be completed in a single cycle). However, this will also push the ASIC tool to use larger and faster cells which can increase area. We will explore the impact of zero-cycle channel latency in Section 3.4. The analytical modeling results are shown in Table 3.1. Figure 3.3 shows the zero-load latency vs. area overhead for a fixed bisection bandwidth. Figure 3.4 shows the zero-load latency vs. bisection bandwidth for a fixed area overhead. Results are shown for both small messages (64b) suitable for scratchpad-based manycores with word accesses and large messages (256b) suitable for cache-based manycores with cacheline accesses.

Impact of Concentration – Concentration reduces the number of routers, increases router radix, decrease the number of bisection channels, and increases the channel length. The router area model in Figure 3.2(a) suggests higher-radix single-cycle routers are still very feasible for concentration factors of 4–8. Similarly, the channel latency model in Figure 3.2(b) also suggests that longer single-cycle channels are still very feasible for these concentration factors. Thus concentration reduces network diameter by reducing the number of hops while maintaining router and channel latency (see Table 3.1). If we focus only on the mesh topology without ruche channels, then across all scenarios in Figures 3.3 and 3.4, increasing concentration reduces the zero-load latency. Concentration also reduces the number of bisection channels. This can reduce the bisection bandwidth for small messages since increasing the channel bandwidth beyond the message size has no benefit (see Figure 3.4(a–b)). However, for large messages, concentration can compensate by increasing the channel bandwidth (see Figure 3.4(c–d)). In terms of reducing latency, a concentration factor of four is more area efficient than a concentration factor of eight. The benefit from *c4* to *c8* is less significant compared to from *c1* to *c4*, which indicates that the area benefit from the reduction in the number of routers is outweighed by the increase in router area due to increased radix.

Impact of Ruche Channels – Ruche channels are long physical channels that aggressively bypass routers. Ruche channels maintain the number of routers, increase router radix, and increase the number of bisection channels. As with concentration, the router area and channel latency models suggest higher-radix single-cycle routers and longer single-cycle channels are still very feasible for ruche factors of 2–3 (with the possible exception of *mesh-c8r3* which requires ruche channels that are over 2 mm long). Even so, ruche channels do increase router area and each hard macro needs to reserve additional area to accommodate the feed-through channels. When area is constrained, adding ruche channels may require narrower channels which increase serialization latency. Compared to concentration, ruche channels are less area efficient in terms of reducing latency, but given a sufficient area budget, ruche channels can be effective. For example, in Figure 3.4(b), the channel bandwidth of all topologies are limited by the small message size and the area budget is relatively large, and thus ruche channels improve both latency and bisection bandwidth.

Combining Concentration and Ruche Channels – Concentration significantly reduces latency and area but decreases the number of bisection channels, while ruche channels increase the number of bisection channels but add area overhead. Thus combining concentration and ruche channels can provide additional benefits. With concentration, a ruche factor of two is a better trade-off; increasing the ruche factor to three adds more area overhead with marginal benefit or even a negative impact on latency. We consider *mesh-c4r2* as a promising design point. It is always on or close to the Pareto-optimal frontier across all scenarios in Figures 3.3 and 3.4, except for Figure 3.3(b) where it cannot achieve a bisection bandwidth of 4096 b/cycle because the channel bandwidth is limited by the small message size.

Torus Topologies – Practical on-chip torus topologies always use a folded torus to ensure all channels are similar in length. Compared to mesh topologies, torus topologies do not increase the router radix, but can still take advantage of longer single-cycle channels. In Figure 3.3, *torus-c1r0* consumes less area for the same bisection bandwidth compared to *mesh-c1r2* and *mesh-c1r3*. This trend is less obvious for topologies with concentration because the radix of the router is already relatively high for these topologies. While torus topologies are certainly competitive, concentrated mesh topologies with ruche channels provide higher bandwidth on short messages. Perhaps just as importantly, mesh topologies are simpler than the corresponding torus topologies which require multiple virtual channels to avoid deadlock and use more complicated routing logic.

Topology	H_D	H_{avg}	Positive Slack (ps) 32	64	128	256	Area Overhead (%) 32	64	128	256
mesh-c1r0	60	21.3	175	66	19	33	5.9	9.6	16.3	35.3
mesh-c1r0q0	30	10.6	0.5	46	–	–	7.9	13.8	–	–
mesh-c4r0	28	10.5	94	73	51	24	2.7	4.6	10.2	18.2
mesh-c4r2	16	6.3	39	49	18	–	6.4	10.7	22.5	–
torus-c1r0	32	16.0	174	19	139	–	8.6	17.7	37.6	–

Table 3.2: Post-Place-and-Route Macro Results H_D = diameter of the network; H_{avg} = average hop latency; Positive Slack = worst-case positive slack for all constrained paths given 1 ns chip-level target cycle time; Area = OCN area overhead as a percentage of the full chip. Designs that do not meet timing or have prohibitively high area overhead are not shown.

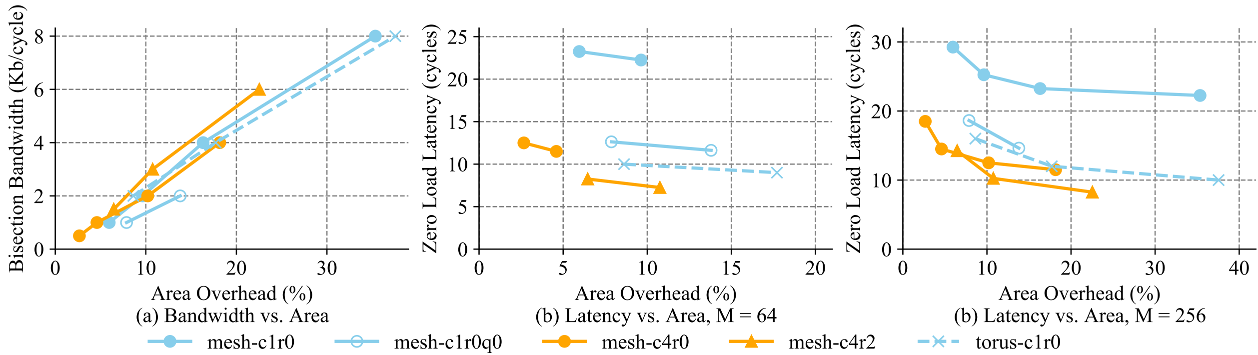


Figure 3.5: Bandwidth, Latency, and Area Trade-Offs for Post-Place-and-Route Results – M = message size in bits; (a) = bandwidth and area trade-offs; (b) = latency and area trade-offs for small messages (64b); (c) = latency and area trade-offs for large messages (256b).

Summary – Concentration is very effective in reducing area overhead and zero-load latency but may reduce the bisection bandwidth at high concentration factors and thus limit overall throughput. Ruche channels, on the other hand, reduce the average hop count and increase the number of bisection channels but may require narrower channels due to the area overhead that comes from more feed-through channels and higher radix routers. Combining concentration and ruche channels provides an elegant hybrid solution. We find that *mesh-c4r2* is a promising topology. According to our analysis, *mesh-c4r2* dominates the baseline *mesh-c1r0* in zero-load latency, area, and bandwidth under different area constraints or bisection bandwidth constraints for both small and large message sizes.

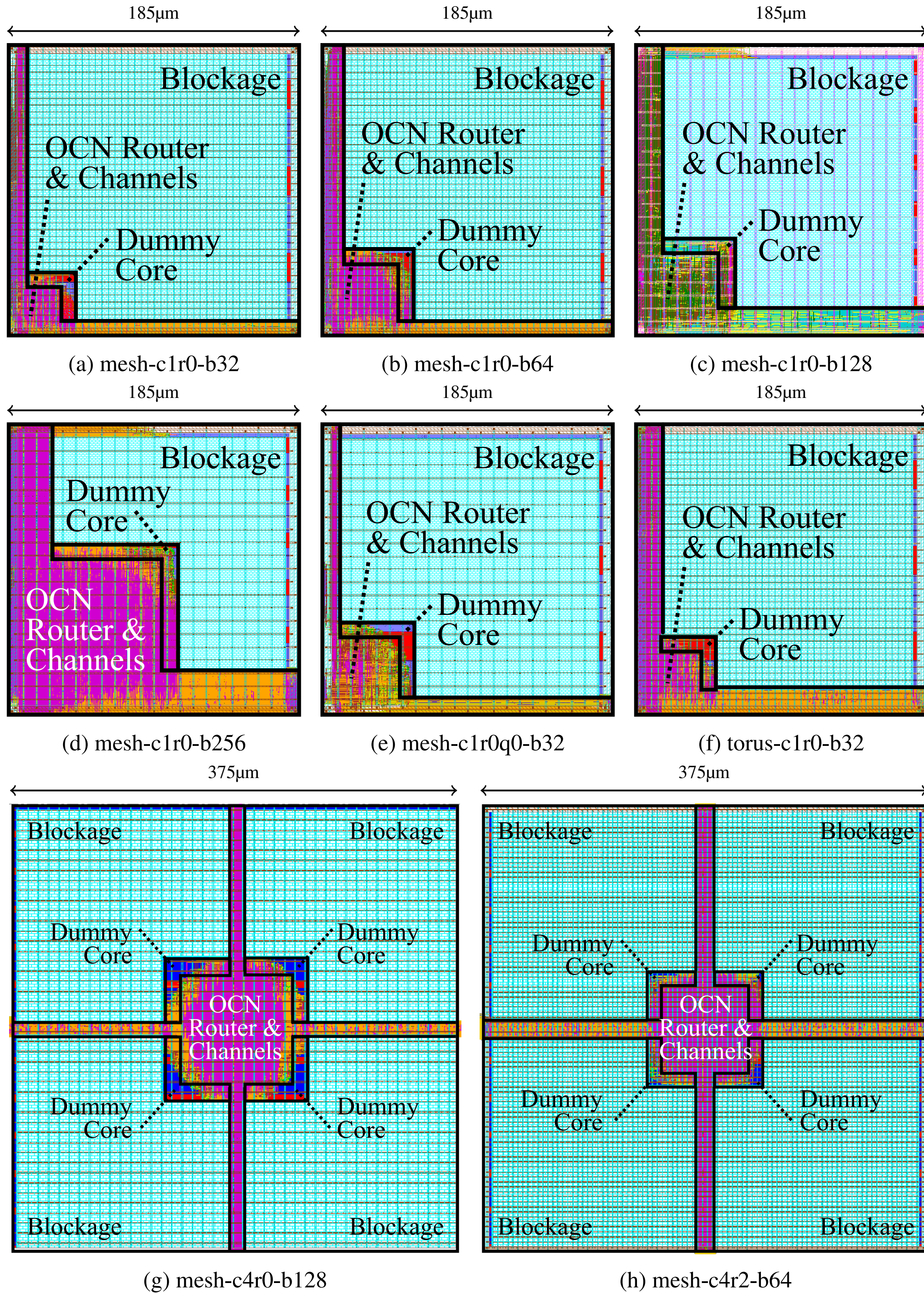
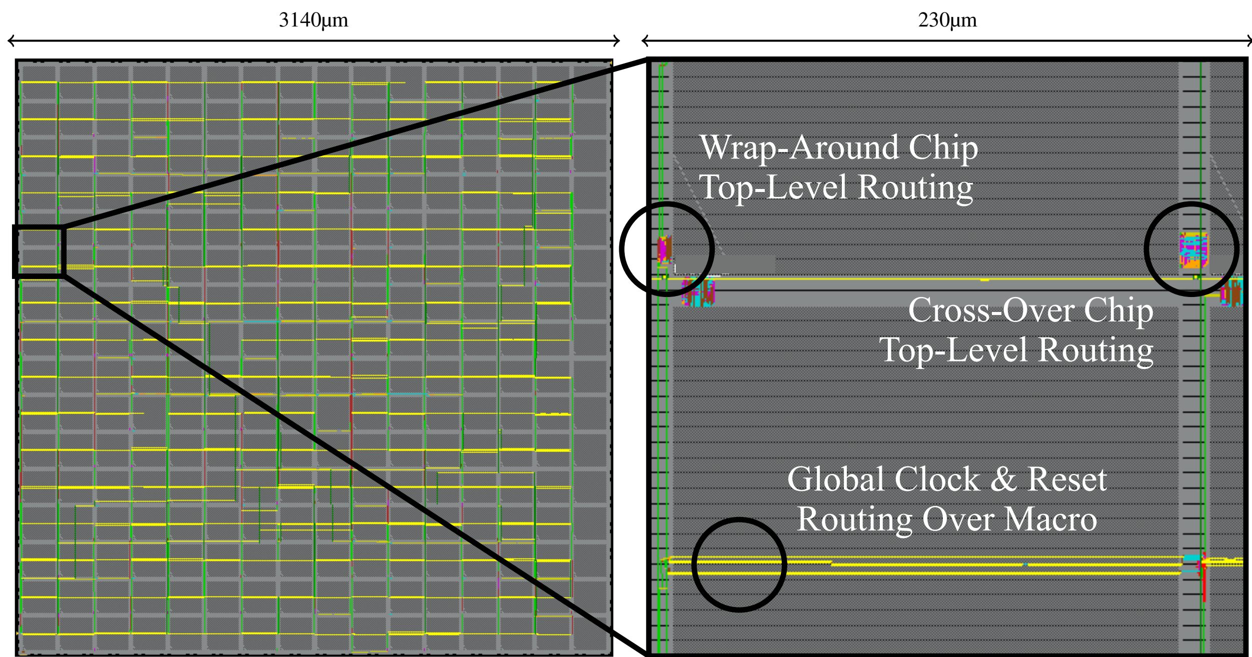
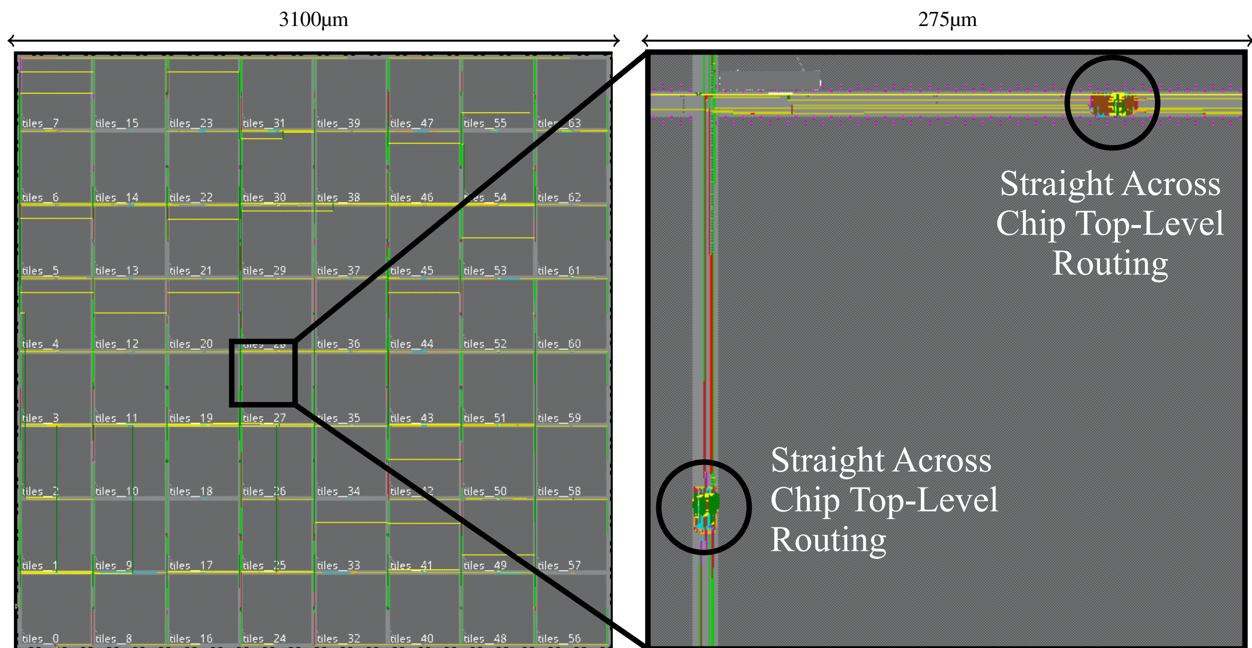


Figure 3.6: Example Macro-Level Post-Place-and-Route Layouts – All layouts are to scale and include 1–4 blockages, 1–4 dummy cores, and fences to constrain placement of the router and channels. (a–d) layouts for *mesh-c1r0* at four different channel bandwidths; (e) layout for *mesh-c1r0q0-b32* (i.e., no channel queues), OCN requires more area than *mesh-c1r0-b32*; (f) layout for *torus-c1r0-b32*, OCN requires comparable area to *mesh-c1r0-b32*; (g–h) layout for *mesh-c4r0-b128* and *mesh-c4r2-b64* both of which require comparable area (smaller OCN router "square" in *mesh-c4r2-b64* is outweighed by longer and wider channel "rectangles").



(a) torus-c1r0-b32 Full Chip

(b) torus-c1r0-b32 Close Up



(c) mesh-c4r2-b64 Full Chip

(d) mesh-c4r2-b64 Close Up

Figure 3.7: Example Chip-Level Post-Place-and-Route Layouts – (a) full-chip layout with 256 instances of the *torus-c1r0-b32* hard macro which is shown in Figure 3.6(f); (b) close-up of required chip top-level routing including cross-over routing to neighboring hard macro, wrap-around routing, and global clock and reset routing over the hard macro; (c) full-chip layout for 64 instances of the *mesh-c4r2-b64* hard macro which is shown in Figure 3.1(h); (d) = close-up of the required chip top-level routing including straight-across routing at the middle of each macro side.

3.4 Manycore OCN Physical Design

Based on the results from analytical modeling, we selected a set of promising topologies with different channel bandwidths for macro-level physical analysis. We used PyOCN to generate the hard macro as well as the full-chip layout. PyOCN is a unified Python-based framework for modeling, testing, and evaluating on-chip networks [TOJ+19]. We pushed each design through the ASIC toolflow multiple times and recorded the minimum area that meets all timing constraints. We also experimented with zero-cycle channel latencies for each topology (i.e., removing the channel queue so router buffering/arbitration and channel traversal take one cycle). We found that *mesh-c1r0* is the only topology that can achieve zero-cycle channel latency without introducing substantial area overhead. We will use *mesh-c1r0q0* to indicate a *mesh-c1r0* topology with zero-cycle channel latency. We carefully floorplan the macro and place the pins to enable short chip top-level routing (see Figure 3.1(r–t) and Figure 3.6). We use "dummy cores" to connect to the injection and ejection ports of the router to queues to prevent the ASIC toolflow from optimizing away any logic and to accurately model terminal channel latencies. We create a hard fence for each dummy core so that the router cannot place any cells into the area that is reserved for the actual processing cores. We also place routing blockages on top of the fences to prevent the router from using any routing resources that are reserved for use by the processing cores. Our 14-nm technology has a total of 13 metal layers. We use three metal layers for horizontal routing (M2, M4, M6) and three for vertical routing (M3, M5, M7). We reserve M8 and M9 for the local power grid, M10 and M11 for global routing (e.g., clock, reset, chip-level I/O), and M12 and M13 for the global power grid. The results of our macro-level analysis can be found in Table 3.2.

To ensure that timing closure for the hard macro can imply timing closure at the chip top-level, we carefully constrain the maximum delay of each register-to-output, input-to-output, and input-to-register path such that the sum of the path delays which form a register-to-register path at the chip top-level is less than a clock cycle (T_c). For example, for *mesh-c4r2* we constrain the maximum delay of register-to-output paths that end at the east ruche output port to be $0.4T_c$, west to east feed-through paths to be $0.3T_c$, and input-to-register paths that start at the west ruche input port to be $0.25T_c$. Our timing constraints are a sufficient but not necessary condition for meeting timing at the chip top-level. Ideally, we only need to constrain the sum of the delays for these paths

rather than constrain each of the three paths separately. Unfortunately, such complex constraints are not currently supported by the ASIC toolflow.

Figure 3.5(a) illustrates trade-offs between bisection bandwidth and area for several topologies. As predicted in our analytical analysis, all topologies provide similar bisection bandwidth for a given area (ignoring message size limitations). By adding ruche channels to *mesh-c4r0*, *mesh-c4r2* achieves comparable bisection bandwidth at similar area overhead compared to *mesh-c1r0*. This supports our hypothesis that ruche channels can complement the reduced bisection channel count brought by concentration. Figure 3.5(b–c) illustrates trade-offs between zero-load latency and area for both small (64b) and large (256b) messages. For both cases, *mesh-c4r2* achieves the lowest latency at similar area. Compared to *mesh-c4r0*, adding ruche channels further reduces the zero-load latency. Although ruche channels lead to narrower channels at the same area, the benefit of reduced average hop count still outweighs the increase in serialization latency. For example, *mesh-c4r2-b64* has similar area as *mesh-c4r0-b128*; it increases serialization latency by two cycles but reduces average hop latency by four cycles (see Table 3.2).

One key observation is that packets can travel long distances in a single cycle. Thus topologies with long channels are critical to reducing the diameter of the network. In the baseline *mesh-c1r0*, a single-cycle channel is of length 185 µm. In *mesh-c4r0*, a single-cycle channel is of length 370 µm, and in *mesh-c1r2*, a single-cycle ruche channel is also of length 370 µm. Combining concentration and ruche channels results in even longer single-cycle channels. In *mesh-c4r2*, a single-cycle ruche channel is of length 740 µm which starts to approach the single-cycle limit.

We also observed that *mesh-c1r0q0* significantly reduces the diameter of the network compared to *mesh-c1r0*. However, it brings area overhead as it pushes the ASIC toolflow to use larger and faster standard cells. It is hard to meet timing with zero-cycle channels when these channels are wide. In our experiments, *mesh-c1r0q0* fails to meet timing at channel bandwidths larger than 64 bits, which limits the maximum bisection bandwidth it can achieve. Overall, even though *mesh-c1r0q0* reduces the average hop latency by 2×, a combination of concentration and ruche channels still achieves lower latency and higher bisection bandwidth at similar area compared to *mesh-c1r0q0*.

In this work, we assume all hard macros are implemented internally as a single flat module. This allows the cores in a hard macro to have different shapes and/or orientations. This may make a macro with concentration harder to implement as it is now four times or even eight times larger

compared to a *mesh-c1r0* macro. We leave exploring multi-level hierarchical design methodologies which might compose core macros within a larger concentrated macro for future work.

To verify that our hard macro can indeed meet the 1 ns chip top-level timing constraint, we pushed full-chip layouts through the ASIC toolflow using each of the hard macros. The *mesh-c1r0-b64* chip has a positive slack of 47.7 ps, *mesh-c4r0-b128* chip has a positive slack of 240.1 ps, *mesh-c4r2-b128* chip has a positive slack of 292.6 ps, *torus-c1r0-b32* chip has a positive slack of 455.7 ps, and *torus-c1r0-b64* has a positive slack of 283.4 ps. The positive slack is significantly better than the worst case positive slack shown in Table 3.2 because our macro-level timing constraints are rather conservative. Figure 3.7 shows the full-chip layout for *torus-c1r0* and *mesh-c4r2*. By integrating feed-through channels into the macro, we enable short chip top-level routing for topologies that would otherwise require long and complicated chip top-level routing.

3.5 Conclusions

Practical manycore processor implementations usually avoid novel on-chip network flow-control schemes, custom circuits, and/or topologies due to various physical design issues. This chapter makes the case that it is possible to implement low-diameter on-chip networks in manycore processors by creatively adapting mesh/torus topologies with concentration and ruche channels for a tiled physical design methodology. Through a combination of analytical modeling and rigorous layout-level evaluation in a traditional standard-cell-based flow, this chapter demonstrated that 2D-mesh topologies with modest concentration factors (concentration factor of four) and modest length ruche channels (ruche factor of two) can reduce latency by over 2× at similar area and bisection bandwidth for both small and large messages compared to a 2D-mesh baseline.

CHAPTER 4
PROTOTYPE: CIFER CHIP TAPE-OUT

This chapter presents the CIFER[1] chip tape-out, a silicon prototype implemented using a tiled physical design methodology. PyOCN is validated through the CIFER tape-out, where it was used to design multiple on-chip networks (OCNs) within the chip. Although the OCN of CIFER is too small to make use of concentration or ruche channels, it reinforces the critical need for a tiled physical design methodology, which is the key motivation for Chapter 3. CIFER is the world's first open-source, many-core, CPU-FPGA system-on-chip (SoC) that offers full cache coherence across across its heterogeneous components. By Combining Linux-capable processors, tiny-core clusters, and an embedded FPGA (eFPGA), CIFER is designed to efficiently exploit both parallelism and specialization. A collaborative team of postdocs, graduate students, and undergraduate students from Princeton University and Cornell University completed the design and tape-out of CIFER within a rapid seven-month period during the COVID-19 pandemic. The design and tape-out process was made efficient by incorporating five open-source projects, including OpenPiton [BMF+16], BYOC [BLS+20], Ariane [ZB19], PyOCN [TOJ+19], and PRGA [LW21]. Fabricated on GlobalFoundries' 12 nm FinFET process, the CIFER chip measures 16 mm^2 (4 mm$\times$4 mm) and is packaged in a 208-pin ceramic quad flat pack. Figure 4.1 shows the CIFER package and die photo.

This chapter begins by presenting the overall architecture of the CIFER SoC which includes the Ariane tile, the TinyCore Cluster tile, and the eFPGA. Section 4.2 delves into the design of the CIFER OCN, and discusses trade-offs between logical and physical hierarchy as well as different data transfer interfaces. Additionally, a timing optimization technique is presented.

4.1 CIFER Architecture

CIFER integrates a scalable architecture composed of multiple heterogeneous processing units, interconnected through a distributed, fully cache-coherent system. The architecture features a 2$\times$4 mesh of tiles, each containing a shard of the coherence system and hosting one of the following components: an Linux-capable Ariane core, a TinyCore cluster, or an embedded FPGA (eFPGA) controller. Each tile is equipped with an 8 KB private L2 cache, and a 64 KB last-level cache (LLC)

[1]CIFER stands for Coherent Interconnect and FPGA Enabling Reuse

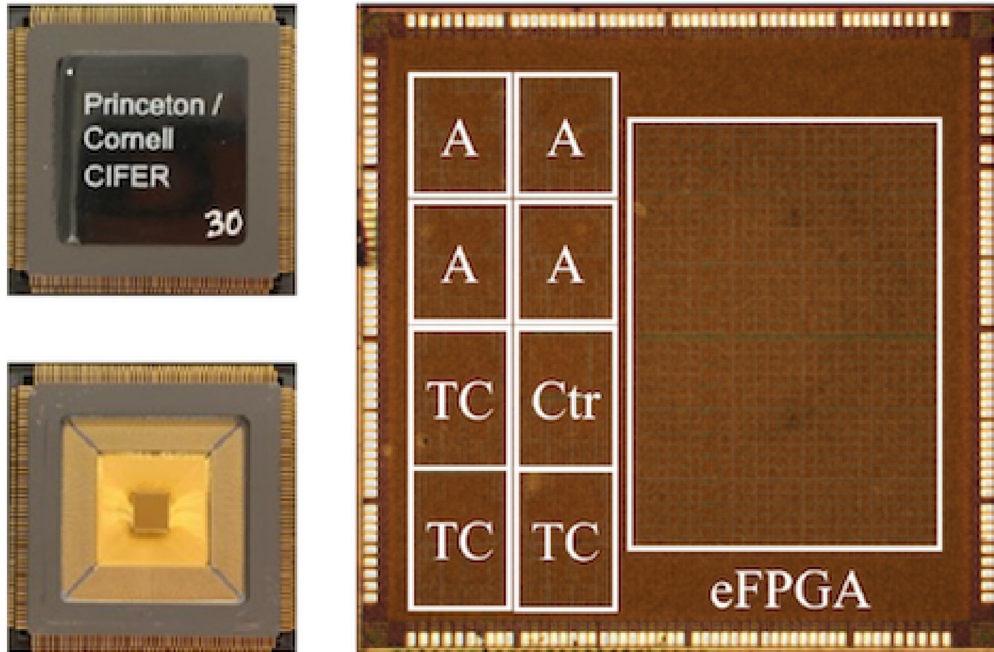


Figure 4.1: CIFER Package and Die Photo – A = Ariane Tile; TC = TinyCore Cluster Tile; eFPGA = Embedded FPGA. Figure is adapted from [LCG+23]

slice, which is part of the shared 512KB LLC distributed across the chip. Cache coherence between the L2 caches and the LLC is ensured by a hardware directory-based MESI protocol. Figure 4.2 illustrates the CIFER architecture. In this section, architecture details of the Ariane tile, tiny-core cluster tile, and eFPGA are discussed.

Ariane Tile – The CIFER chip integrates four Ariane tiles, each equipped with an Ariane core. The Ariane core is an open-source, Linux-capable, 64-bit RISC-V processor implementing the RV64GC instruction set with support for double-precision floating point arithmetic [ZB19]. It operates with a six-stage in-order pipeline and includes a 16 KB L1 instruction cache alongside an 8 KB L1 data cache. Cache coherence between the Ariane core's private L1 caches and the shared LLC is managed through adaptation to the BYOC's transaction response interface (TRI) [BLS+20], ensuring seamless data synchronization across the chip.

TinyCore Cluster Tile – The CIFER chip integrates three TinyCore cluster tiles to exploit thread-level parallelism (see Figure 4.2). Each TinyCore cluster tile includes six lightweight in-order cores arranged into three pairs, resulting in a total of 18 tiny cores. Each tiny core is a 32-bit RISC-V processor implementing the RV32IMAF instruction set. Each tiny operates with a six-stage scalar pipeline and is equipped with a 4 KB L1 data cache. Each pair of tiny cores shares a 4 KB L1 instruction cache, a 32-bit integer multiply-division unit (MDU), and a 32-bit

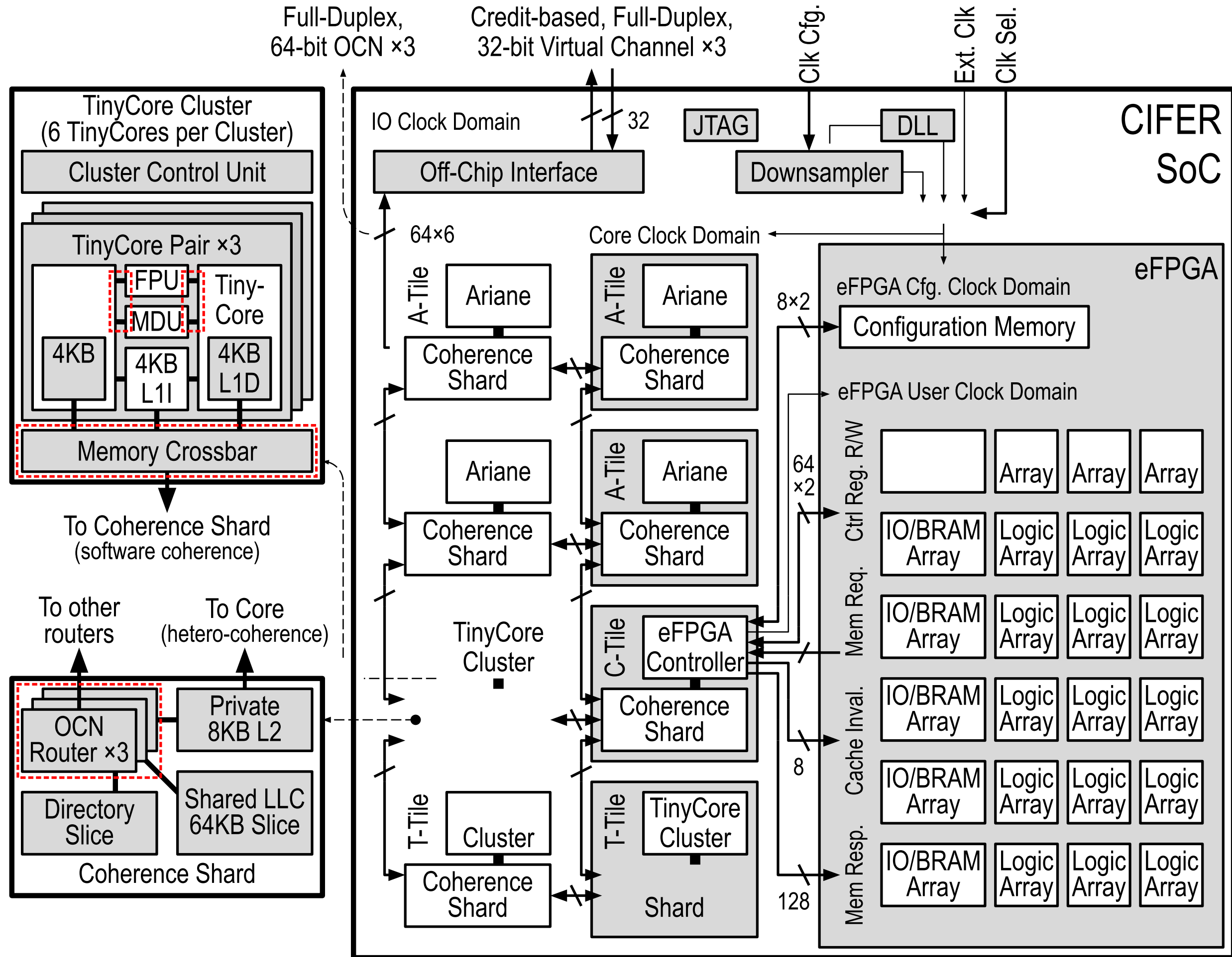


Figure 4.2: CIFER SoC Architecture – Adapted from [CLG+23]. Components designed with PyOCN are highlighted in red, including the OCN routers in each tile as well as the memory crossbar and resource-sharing networks in the TinyCore cluster tile.

single-precision floating-point unit (FPU). Cache coherence between the L1 data caches and the L2 cache is handled in software by a task-parallel work-stealing runtime, with explicit cache flush and invalidation operations.

eFPGA – The eFPGA in CIFER consists of 6720 6-input look-up tables (LUTs) and 18 dual-port block RAMs (BRAMs), each with a capacity of 24 Kb. This flexible reconfigurable fabric allows for the creation of custom accelerators through an open-source RTL-to-bitstream flow, which includes Yosys for synthesis [Wol20], VPR for place-and-route [MPZ+20], and PRGA for bitstream assembly [LW21]. The eFPGA interfaces with the rest of the system via two key connections in the eFPGA controller tile: a control register interface that enables CPU access through memory-mapped I/O, and a configurable memory interface supporting non-coherent, IO-coherent,

or fully coherent memory access. Additionally, the eFPGA supports atomic memory operations, facilitating low-overhead synchronization with other compute units. The eFPGA emphasizes programmability and cache coherence, making it adaptable to a wide range of computational tasks.

4.2 CIFER On-Chip Networks

The CIFER SoC incorporates a 2×4 mesh network that interconnects the heterogeneous processing units. The OCN in CIFER provides communication between the tiles for a variety of operations, including cache coherence, I/O, memory accesses, and inter-core interrupts. CIFER uses three physical OCNs to prevent protocol-level deadlocks. The CIFER OCNs uses dimension-order wormhole routing to ensure point-to-point ordering between any source- destination pair. The CIFER OCNs are designed and implemented using the PyOCN framework. This section details the design choices and implementation details of the CIFER OCN.

4.2.1 Supporting Multi-Flit Packets in PyOCN

The CIFER OCNs use multi-flit packets to enable transferring large data payloads between the processing units through the 64-bit channels. Since the data payload can be as large as 512 bits, adopting single-flit packet, though simpler, will require a channel width of more than 512 bits. Given that the there are three physical OCNs and that the channels are unidirectional, single-flit packets would require almost 3K wires in each direction of a tile, which will take significant amount of wiring resources within the tile and drastically complicates the routing between tiles.

The CIFER OCN packet format is inherited from OpenPiton. A CIFER OCN packet contains a header flit followed by up to ten payload flits. Figure 4.3 shows the CIFER OCN packet header format implemented in PyOCN. The `chipid` field is used for identifying whether the packet should be sent off-chip. For the router at position $(0,0)$ (upper left corner), if the `chipid` field is not equal to the chip ID of the router, the packet will be sent to the off-chip port (northern port). The `xpos` and `ypos` fields are used for identifying the precise location of the destination router. The `fbits` field informs the destination router which port the packet should be sent to. The `plen` field indicates the length of the payload (excluding the header flit) in flits. The rest of the fields are not used by the OCNs. The `mtype` field indicates the message type ID. The `mshr` field includes the tag or ID

```python
@bitstruct
class CiferNoCHeader:
  chipid : Bits14
  xpos   : Bits8
  ypos   : Bits8
  fbits  : Bits4
  plen   : Bits8
  mtype  : Bits8
  mshr   : Bits8
  opt1   : Bits6
```

Figure 4.3: CIFER NoC Header Format – CIFER NoC header implementation in PyOCN.

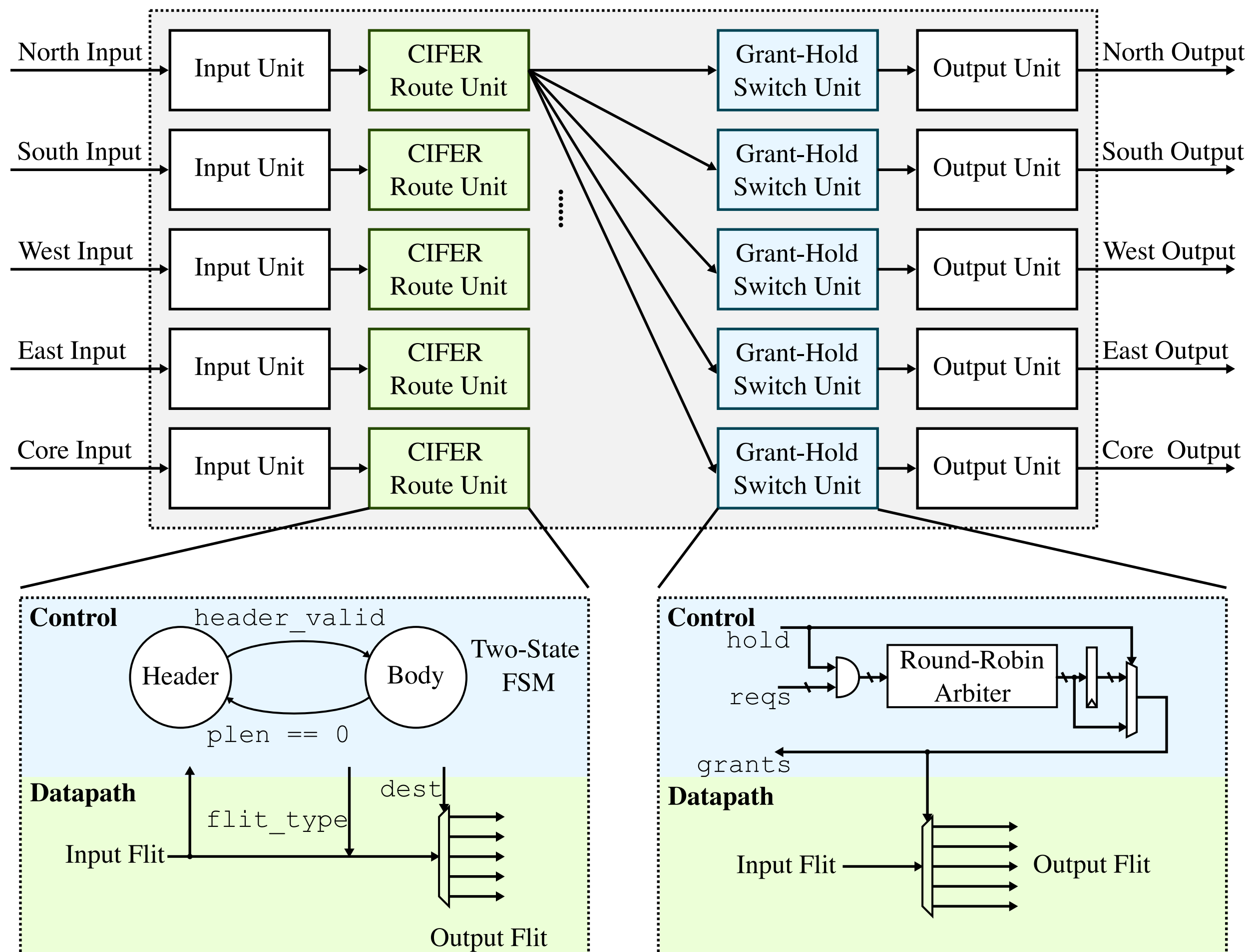


Figure 4.4: CIFER Router Architecture – CIFER router implemented with PyOCN. A CIFER router is composed of default input units, custom CIFER route units, grant-hold switch units, and default output units.

of the miss-status-handling register (MSHR). `opt1` field corresponds to the *Options 1* field in the OpenPiton specification [ope16], which is a reserved field that can be used for adding additional information.

PyOCN offers a library of router components that can be used to build custom OCN topologies. However, the default PyOCN router components (route units and switch units) only support single-flit packets. To support multi-flit packets, I implemented a route unit with an embedded finite state machine (FSM) that manages the packet state across multiple flits. I also implemented a switch unit with grant-hold arbitration to hold the arbitration results as the payload flits is going through the router. The general router architecture remains the same as described in 2.2.

4.2.2 Logical vs. Physical Hierarchy Trade-Offs

PyOCN generates an OCN as a standalone module which contains all the routers and links, with only the ports to the terminal nodes exposed as the OCN's input and output ports. This structure naturally leads to a clean logical hierarchy, as is shown in Figure 4.5. In this hierarchy, the OCN module is instantiated three times to form the three physical OCNs, with each tile only exposing one port to the OCN. This approach offers two primary benefits.

Better Testability. The logical hierarchy completely isolates the functionality of the OCN from the rest of the system, making it easier to verify and debug the OCN as a standalone module. This decoupling also simplifies the testing of individual tiles, as their interface remains straightforward. The OCN can be replaced with a simple testbench that generates and receives packets, facilitating direct testing of different behaviors of the tile.

Better Modularity. The logical hierarchy also allows the OCN module to be easily swapped for other OCN implementations featuring different topologies, routing algorithms, or flow control mechanisms, without affecting the rest of the system. The tiles do not need re-testing as long as the interface between the tiles and the OCN remains consistent. This flexibility is particularly useful for exploring various OCN designs and optimizing the OCN for different workloads.

However, CIFER, similar to many other multi/many-core SoCs, adopts a tiled physical design methodology, where each tile is made into a hard macro in the ASIC flow and replicated multiple times to form the entire chip. This leads to the physical hierarchy shown in Figure 4.6, where the routers are integrated within the tiles, and the tiles are directly connected to adjacent tiles to form a mesh network. In this physical hierarchy, the interface of the tile becomes more complex, as each tile now has four ports to connect to the neighboring tiles, as opposed to just one port to connect to the OCN. This increased complexity makes unit testing the tile more complicated. Moreover,

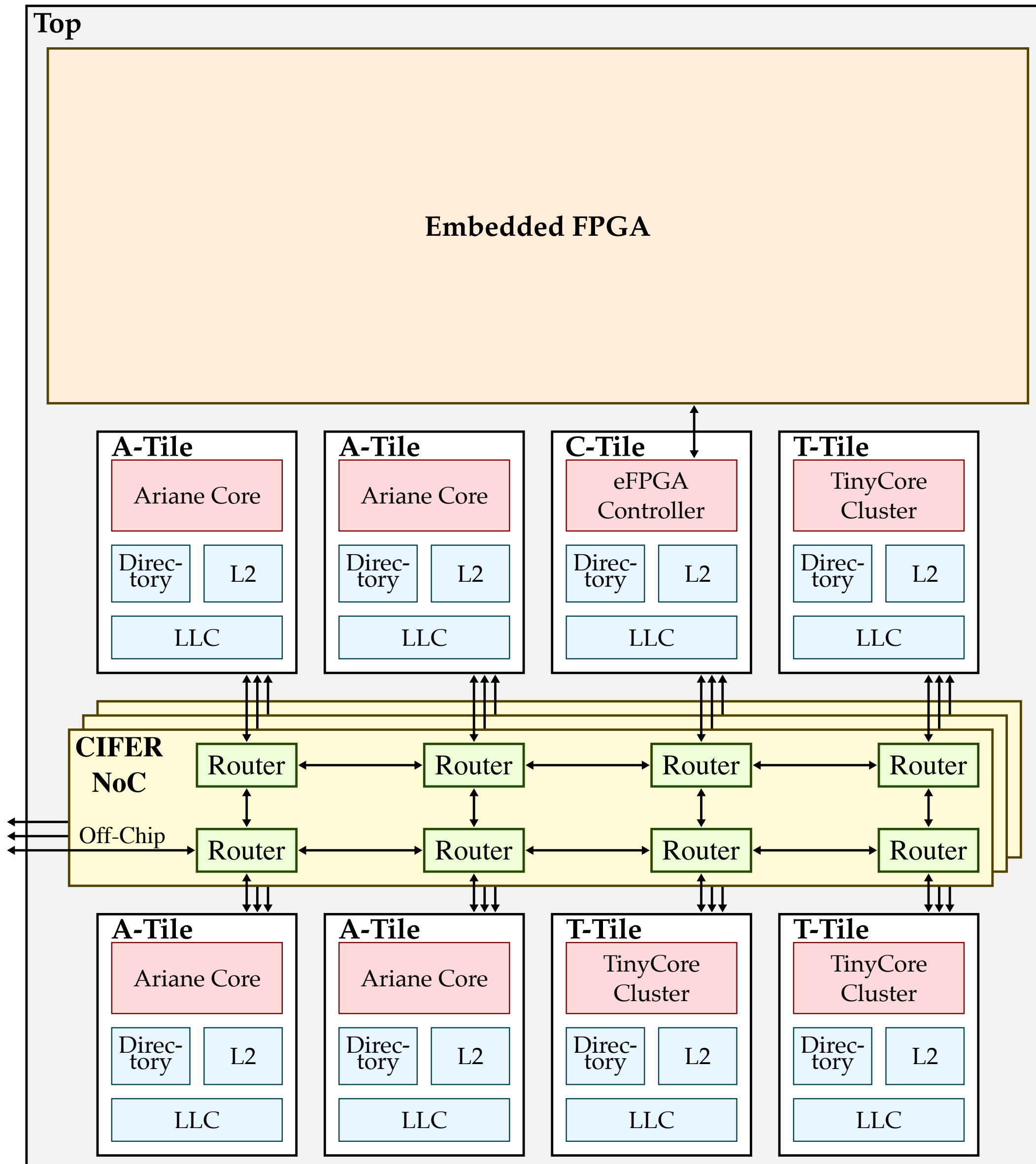


Figure 4.5: CIFER SoC Logical Hierarchy

replacing the OCN with another implementation under this physical hierarchy necessitates re-testing of the tiles, as the routers are embedded within them.

Despite these drawbacks, the physical hierarchy significantly simplifies the top-level physical design. Global routing becomes less complex, as it only needs to connect adjacent tiles with relatively short wires. In contrast, using standalone OCN modules would require more intricate global routing. Furthermore, as demonstrated by Petrisko et al. [PZD+20], integrating the router within the tile leads to better area and timing results in the ASIC flow.

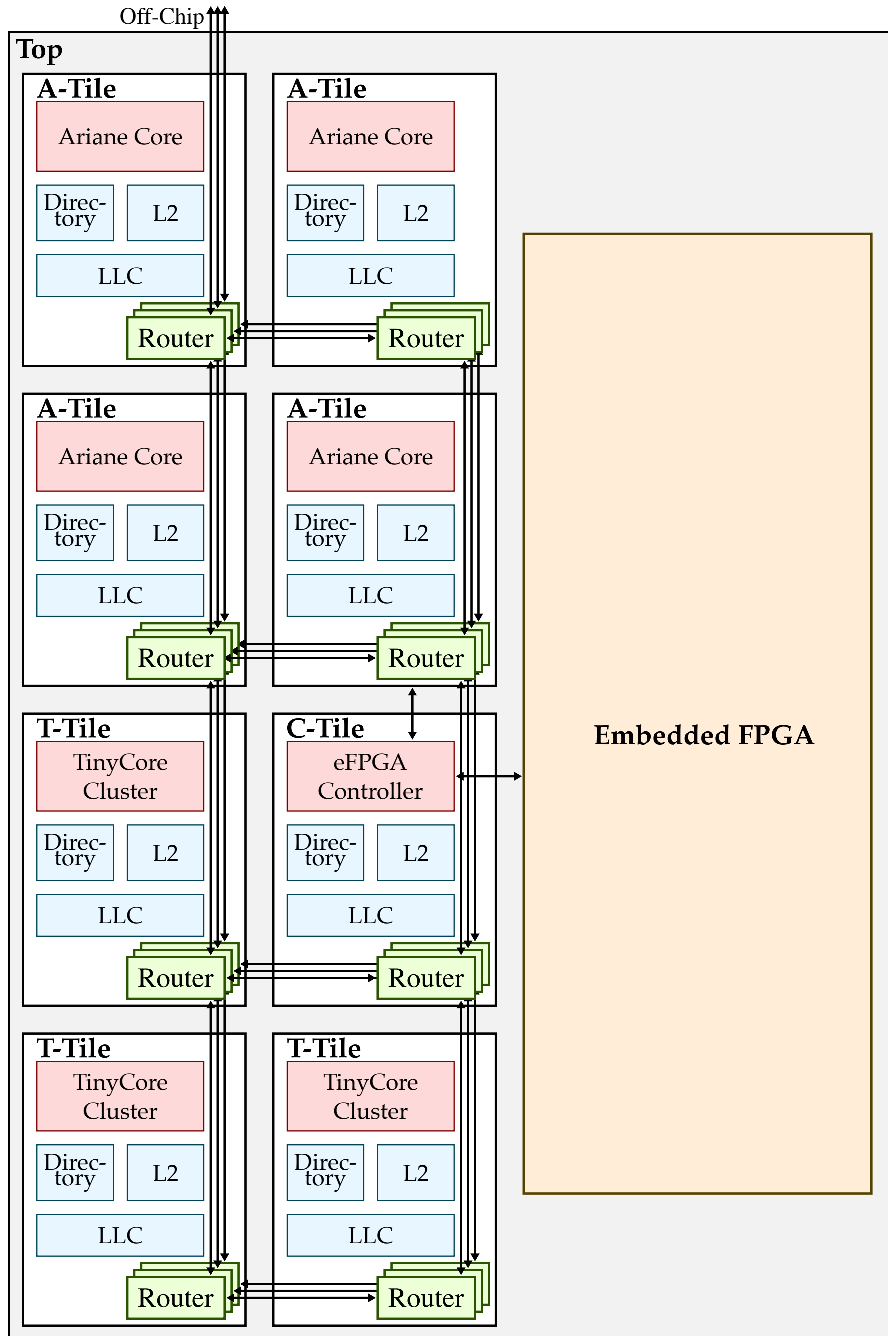


Figure 4.6: CIFER SoC Physical Hierarchy

4.2.3 Data Transfer Interface Trade-Offs: Val/Rdy vs. En/Rdy vs. Credit-Based

Data transfer interface refers to the microarchitecture-level protocol that determines how the sender and receiver of a communication channel coordinate the data transfer. Different data transfer protocols have different trade-offs in terms of timing, area, risk of combinational loop, and

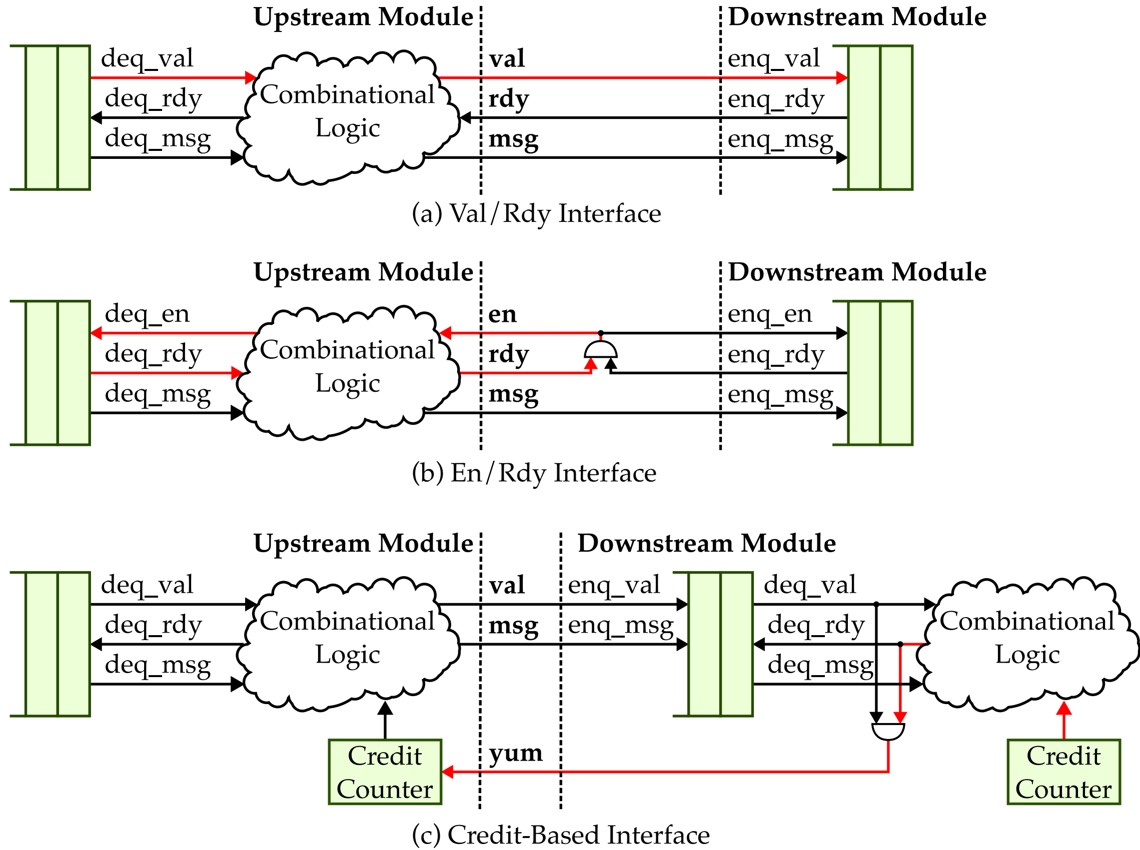


Figure 4.7: Different Handshake Interfaces – Potential critical paths are highlighted in red. Assuming input-registered design.

reusability. Three data transfer protocols are considered when implementing the CIFER OCNs: Val/Rdy, En/Rdy, and credit-based. Figure 4.7 shows how data is transferred from the sender's input queue to the receiver's input queue using these three data transfer interfaces.

Figure 4.7(a) shows the widely used Val/Rdy interface. The sender asserts the `val` (valid) signal to indicate that the current message is valid and can be taken. The receiver asserts the `rdy` (ready) signal to indicate that it is ready to receive the data. The message is transferred when both the `val` and `rdy` signals are asserted.

The `val` and `rdy` can be efficient in terms of timing, particularly when the `val` and `rdy` signals are independent of one another. However, the Val/Rdy protocol does allow dependencies between the `val` and `rdy` signals, which can sometimes lead to combinational loops. For example, if the sender's `rdy` signal is generated by its `val` signal, and the receiver's `val` signal is generated by its `rdy` signal, a combinational loop is formed. Therefore, reusing a design with Val/Rdy inference may not always be safe. The designer must understand the implementation details of the sender

and receiver or follow a strict guideline (such as only allowing one signal to depend on the other) to avoid introducing combinational loops.

Figure 4.7(b) shows the En/Rdy interface, which was initially used in the PyOCN implementation. The En/Rdy protocol is inspired by the guarded method-based interface in Bluespec SystemVerilog (BSV) language [Nik04]. With this interface, the `en` (enable) signal is an input and the `rdy` (ready) signal is an output for both the sender and receiver. The sender asserts the `rdy` signal when the message is ready to be transmitted, and the receiver asserts the `rdy` signal when it is ready to receive the message. The `en` signals on both side can only be asserted when both `rdy` signals are asserted. The `rdy` signals on both sides need to be AND-ed to generate the `en` signals. This compulsory dependency eliminates the risk of combinational loop. It can potentially achieve comparable timing performance as the Val/Rdy protocol if the AND gate used to generate the `en` signals is placed optimally (i.e. right in the middle of the critical path). However, this is often not feasible in practice due to (1) the optimal position for the AND gate may be inside a hard macro, which the ASIC tool cannot modify; and (2) the interface signals are often generated by some combinational logic and the AND gate must be placed after the long combinational logic, making the critical path impossible to balance. Therefore, the En/Rdy interface is more suitable for short-reach data transfer within a module where logic gates can be placed in close proximity. For long-distance data transfers, the En/Rdy interface may introduce additional delay due to sub-optimal placement of the AND gate, making it less efficient for inter-module communication in large design.

Figure 4.7(c) shows the credit-based interface, where a `val` (valid) signal indicates whether the current message is valid, and a `yum` (yummy) signal is used for sending back a credit from the receiver. The sender maintains a credit counter to keep track of how many messages it is allowed to send. The credit counter is initialized to be the number of entries in the receiver's input queue. It is decremented when a message is sent and incremented when a credit is received (i.e., when the `yum` signal is asserted). No handshake is required during data transfer, since by design the sender ensures that there is always sufficient space in the receiver's input queue as long as the sender has credit. The receiver asserts the `yum` signal when a message is consumed (i.e. dequeued from its input queue). The credit-based interface offers good timing performance with little risk of combinational loop, since the `val` and `yum` signals are fully decoupled. In a single-hop router design, the critical path is often the `yum` signal (see Figure 4.7(c)), which is derived from the

Interface	Timing	Area Overhead	Risk of Combi-national Loop	Reusability
Val/Rdy	Good	N/A	Yes	May introduce combinational loop
En/Rdy	Not suitable for long distance	Extra AND gates	No	Safe to reuse
Credit-Based	Best if `yum` signal is registered	Credit counter and potentially larger queue size	No	Safe to reuse

Table 4.1: Comparison of Data Transfer Interfaces

`deq_val` and `deq_rdy` signals, the latter of which is generated by some arbitration logic depending on the `deq_val` signal and the receiver's credit counter. To optimize timing, the `yum` signal can be registered on the receiver's side, since the `val` and `yum` signals do not need to be asserted in the same cycle. However, this optimization increases the round-trip latency of the credit, meaning that the input queue size must be enlarged to maintain full throughput. While this optimization improves the timing performance, it introduces additional area overhead.

Table 4.1 summarizes the trade-offs of the Val/Rdy, En/Rdy, and credit-based interfaces. In the CIFER chip tape-out, we eventually decided to use the Val/Rdy interface for the OCNs since it introduces no area overhead and achieves reasonably good timing.

4.2.4 Timing Optimization: Speculative Switch Allocation

The CIFER OCNs employ a single-hop router design, where each router consists of five stages: input, routing computation, switch allocation, switch traversal, and output (see Figure 4.8(a)). In the input stage, incoming packets are stored in an input queue. In the routing computation stage, the output port for the packet is determined based on its destination. Next, in the switch allocation stage, the router arbitrates which input port gains access to which output port, resolving any contention between multiple packets contending for the same output. In the switch traversal stage, the packet is forwarded through the switch fabric. Finally, in the output stage, the packet is sent to the next router. The routing computation, switch allocation, and switch traversal stages involve a lot of combinational logic and often end up being the critical path of a chip.

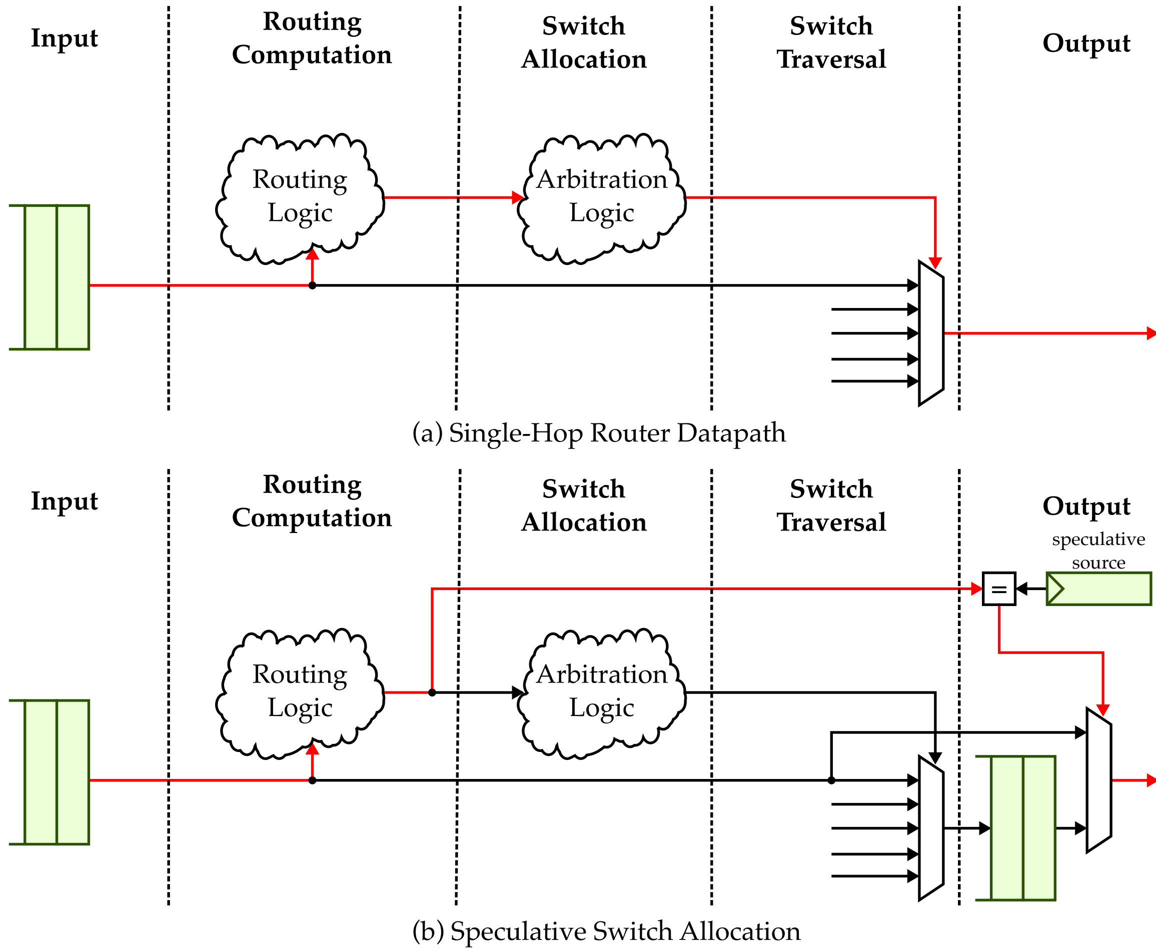


Figure 4.8: Speculative Switch Allocation – Critical paths are highlighted in red.

The CIFER OCNs employ standard dimension-order routing, where packets first traverse the X dimension and then the Y dimension. Such a routing algorithm ensures that packets turn at most once, either at the boundary between dimensions or not at all if they are traveling straight. When implementing the CIFER OCNs, I exploit such behavior and explore *speculative switch allocation* to optimize the critical path. The high-level idea is that the packet takes one cycle for the straight route and two cycles for the turning route.

Figure 4.8(b) illustrates how speculative switch allocation works. In this approach, the router stores a speculative source in a register for each output port. For example, the north output can have a speculative source of the south input, which corresponds to the straight route. After the routing computation stage, if the speculation is not correct, the packet will just go through the normal switch allocation stage. A pipeline buffer is inserted after the switch traversal stage to break the critical path. If the speculation is correct, the packet will bypass the switch allocation and directly

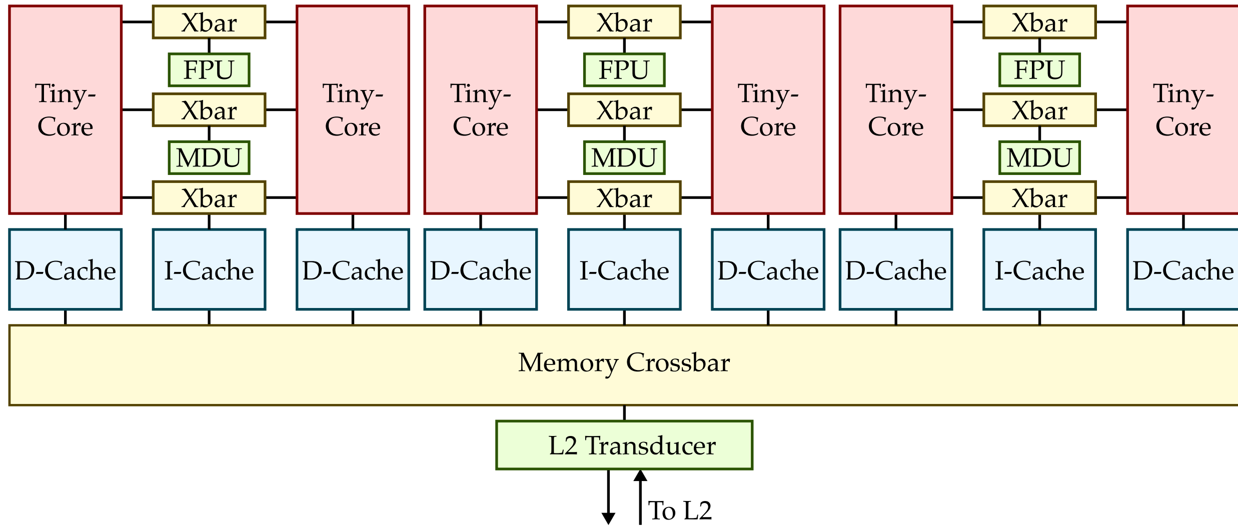


Figure 4.9: TinyCore Cluster Architecture – Xbar = Resource-Sharing Crossbar.

be forwarded to the output port. If multiple packets are contending for the same output port, the speculative route will not be used to enforce fairness. By doing this, the switch allocation stage is effectively taken out from the critical path. The switch traversal stage in the critical path also reduces from going through a 5-to-1 MUX to a 2-to-1 MUX. The zero-load latency of the OCN only increases by one cycle, and the throughput of the OCN remains the same.

I evaluate the speculative switch allocation optimization using the CIFER ASIC flow. The results show that the speculative switch allocation can shorten the critical path of the router by around 20% (from 650ps to 520ps). Eventually, the speculative switch allocation is not adopted in the actual tape-out as there are other paths longer than the router critical path. However, the speculative switch allocation optimization can be useful for other designs or future tape-outs where the router is the critical path.

4.2.5 TinyCluster Resource-Sharing Crossbar and Memory Crossbar Design

The TinyCore cluster in the CIFER SoC consists of six TinyCores, organized into three pairs. Each pair of TinyCores shares key resources: an FPU, an MDU, and an L1 instruction cache. To facilitate efficient sharing of these resources, the cluster utilizes a resource-sharing crossbar designed and implemented using PyOCN. Additionally, the cluster employs a memory crossbar, also designed with PyOCN, to manage connections between the L1 caches and the L2 cache.

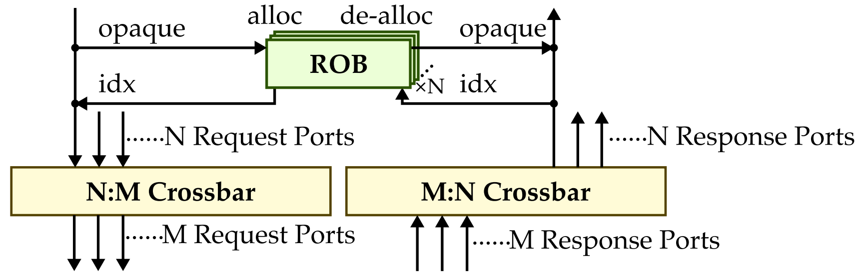


Figure 4.10: Resource-Sharing Crossbar Design – ROB = Reorder Buffer.

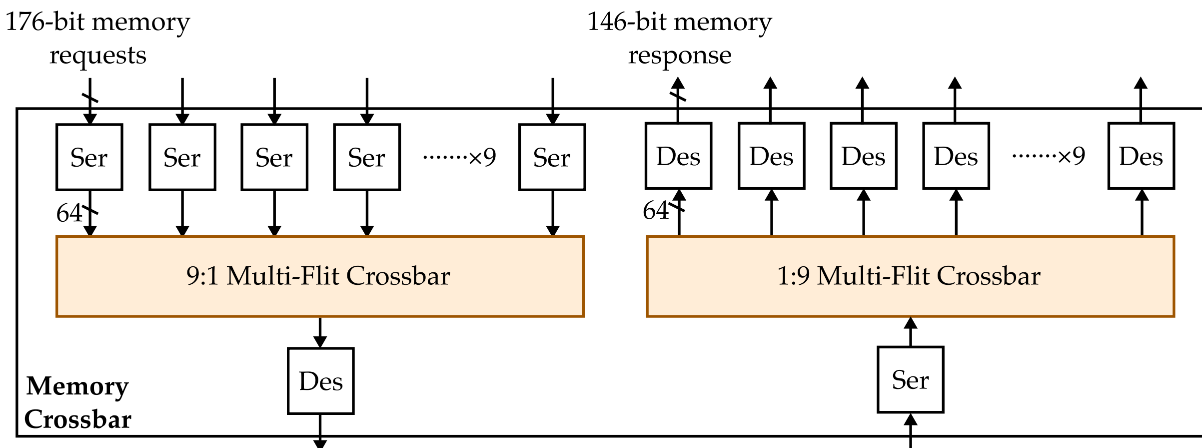


Figure 4.11: Memory Crossbar Design

The resource-sharing crossbar, illustrated in Figure 4.10, is a generic request/response network that allows M responders (such as FPUs or MDUs) to be shared among N requesters (TinyCores). The design includes two distinct networks: a request network (an N-to-M crossbar) and a response network (an M-to-N crossbar). Each requester sends requests to shared resources via the request network, while the response network ensures that responses are routed back to the correct requesters.

A key feature of this crossbar is that the requesters and responders are unaware of the network and do not need to be modified to enable the resource sharing. Requesters simply send their requests, and responders generate corresponding responses, without handling any network-specific details. The crossbar manages all necessary routing by leveraging the `opaque` field in the request messages. Each request port is equipped with a Reorder Buffer (ROB) to manage these interactions. Upon receiving a request, the ROB allocates an entry, storing the original opaque field, and replaces it with an index corresponding to the ROB entry and the source ID of the requester. This opaque field is preserved throughout processing by the responder. When a response is generated,

the crossbar uses the source ID in the opaque field to determine the destination for the response. Before finally sending back the response, the crossbar restores the original opaque field by deallocating the ROB entry using the ROB index in the opaque field, ensuring that the requester receives the response with the same opaque field it initially sent. This mechanism allows the system to handle out-of-order responses effectively, ensuring accurate delivery without requiring the requesters or functional units to handle additional network-specific details.

The TinyCore cluster also includes a memory crossbar that connects nine caches (six data caches from the TinyCores and three shared instruction caches) to the L2. The memory crossbar design, shown in Figure 4.11, consists of a request network (a nine-to-one multi-flit crossbar) and a response network (a one-to-nine multi-flit crossbar). Unlike the resource-sharing crossbar, the memory crossbar does not require a Reorder Buffer (ROB) because responses from the L2 transducer are guaranteed to be in order. Given that the memory requests and responses are relatively wide messages (176-bit and 146-bit respectively), the memory crossbar uses a multi-flit design to optimize area and timing. The requests are serialized into 64-bit flits before being transmitted across the crossbar. The serialized flits are then deserialized at the output, reconstructing the full memory request for the L2 transducer. Similarly, responses from the L2 transducer are serialized into 64-bit flits for transmission across the response network and deserialized back into complete memory responses before being sent back to caches. This multi-flit approach enables the memory crossbar to efficiently handle wide data transfers while improving timing and reducing wiring and area usage.

The integration of resource-sharing and memory crossbars within the TinyCore cluster provides a scalable and efficient on-chip communication infrastructure. Both networks are productively developed with the PyOCN framework, which streamlines the design and implementation process.

4.3 Conclusion

This chapter presents the architecture of the CIFER SoC, and explores the design of the CIFER NoCs both between the tiles and within the TinyCore cluster. It examines the trade-offs between logical and physical hierarchies and various data transfer interfaces, including Val/Rdy, En/Rdy, and credit-based protocols. Additionally, a speculative switch allocation technique is introduced as a timing optimization strategy. These design choices contribute to the overall efficiency and

scalability of the CIFER NoCs, providing a robust network solution for the CIFER SoC. The insights gained from this work can inform future SoC designs, particularly in balancing modularity, testability, and performance in large-scale NoCs.

PART II
SCALING OFF-CHIP INTERCONNECTS

The second part of this thesis shifts focus to off-chip interconnects, which are equally important for enabling scalable, high-performance computing in distributed systems. Chapter 5 presents LLMCompass-E2E, a performance modeling framework designed to capture the complexities of distributed LLM training. Following this, this part explores the potential of co-packaged optics in both *memory interconnect* and *system interconnect*. Chapter 6 proposes an optically connected multi-stack HBM module that leverages co-packaged optics for memory interconnect, breaking the traditional bandwidth and capacity trade-off. Chapter 7 presents the PIPES chip tape-out, demonstrating a practical implementation of a system interconnect based on co-packaged optics. Both the optically connected multi-stack HBM module and the PIPES chip tape-out use LLMCompass-E2E to evaluate the system-level performance for LLM training. These chapters address the key tensions in off-chip interconnects, offering robust methodologies and architectural innovations for the next generation of distributed computing systems.

CHAPTER 5
METHODOLOGY: A PERFORMANCE MODELING FRAMEWORK FOR END-TO-END LLM TRAINING

This chapter presents LLMCompass-E2E, a comprehensive framework developed to tackle the complexities of simulating large-scale LLM training and inference. Building on top of the original LLMCompass foundation, LLMCompass-E2E integrates a front-end capable of generating computational graphs for both the forward and backward passes, as well as a system-level performance model that supports system-level configurations involving multiple compute nodes and various parallelism techniques such as data, tensor, and pipeline parallelism. These additions allow LLMCompass-E2E to accurately represent the intricate interplay between compute units, memory systems, and interconnects in distributed environments, offering an in-depth analysis of hardware performance across diverse architectural and system configurations. This chapter aligns with the broader goals of the thesis by providing researchers with an advanced tool to assess the efficiency and scalability of hardware architectures designed for LLMs. LLMCompass-E2E is further used in subsequent chapters to evaluate the performance of co-packaged silicon photonic interconnects for both memory and system interconnects for LLM workloads.

5.1 Introduction

In recent years, the scale of large language models has grown significantly, driven by advances in hardware capabilities and the demand for more accurate and sophisticated models. This trend, exemplified by models such Open AI ChatGPT [BMR+20] and Google Gemini [GLB+24], aligns with the scaling law for language models [KMH+20], which suggests that increasing model size, dataset size, and compute resources yields predictable improvements in model performance across a variety of tasks. Recent models have surged into trillions of parameters [FZS22], pushing the boundaries of LLM capabilities.

However, the unprecedented scale of these models poses challenges to their training, which is extremely resource-intensive and requires large-scale computing systems and highly specialized hardware. For example, a recent report from Meta indicates that it takes 24K NVIDIA H100 GPUs to train the Llama 3 model with 405 billion parameters [met24]. As the scale of LLMs and the dataset sizes continue to grow, it will only require even more computing resources to

train these models. Access to such extensive computing resources is often limited to industry or specialized research institutions. For the majority of researchers, limited access to large-scale computing systems hinders the ability to evaluate and understand the impact of different system- or architecture-level design choices, thereby impeding the exploration of innovative approaches for LLM workloads. As LLM workloads continue to grow in scale and complexity, there is a pressing need for a new class of modeling tools to support effective design decisions in the early stages of architecture exploration for LLM workloads.

This growing gap highlights the pressing need for new modeling tools that enable researchers to evaluate and optimize hardware architectures for LLMs without requiring access to large-scale infrastructure. To be effective for LLM workloads, such tools must meet the following key criteria. **(1) Balance between simulation speed and accuracy.** Given the massive scale of LLM workloads, these tools need to deliver results quickly without sacrificing precision to capture critical performance insights. **(2) Capability of system-level evaluation.** Such tools need to have the ability to model not only single devices but also interconnected systems involving thousands of compute devices, as both LLM training and inference often rely on large-scale distributed systems.

Existing tools fall short of these requirements. Fast analytical methods like Roofline model [WWP09] are useful for general insights but lack accuracy and do not capture the impact of architectural details such as memory hierarchy and interconnects. Cycle-level simulators, such as MGPUSim [SBM+19], provide precision but are too slow and impractical for modeling end-to-end LLM workloads on complex, distributed systems. Existing system-level simulators for distributed learning, such as ASTRA-sim [RSSK20], has a greater focus on simulating the data movement over system interconnects but does not offer a compute simulator to explore the impact of different architectural design choices. To effectively evaluate and explore innovative hardware designs for large-scale LLM workloads, a unified hardware system modeling tool that can accurately model both compute and communication is needed.

This chapter presents LLMCompass-E2E, an end-to-end hardware system modeling framework for LLM training and inference. LLMCompass-E2E is an extension to LLMCompass[1], a hardware modeling framework for LLM inference [ZNPW24]. LLMCompass provides fast and accurate performance estimation for LLM inference and is architecturally descriptive, which meets the criteria (1) listed above. Section 5.2.1 introduces background on LLM training and inference. Section 5.2.2

[1] LLMCompass stands for Large Language Model Computation Performance and Area Synthesis

Tensor	Shape
X_{mha}, Y_{mha}, X_{ffn},Y_{ffn}, H_1	(b,s,d)
Q, K, V	(b,s,d_h)
A_{prob}	(b,s,s)
H_0	$(b,s,4d)$

Table 5.1: Shape of intermediate Tensors in Transformer Layer – b = batch size, s = sequence length, d = model embedding size, d_h = head size.

introduces the original LLMCompass framework. Section 5.3 describes how LLMCompass-E2E extends LLMCompass to model LLM training.

5.2 Background

Large language models are based on the Transformer architecture [VSP+17] and contain a large number of parameters pre-trained on large datasets. This chapter focuses on decoder-only Transformer models, which are widely adopted in LLMs such as GPT [BMR+20], Llama [TLI+23], PaLM [CND+24]. The fundamental building block of these LLMs is the Transformer layer. Figure 5.1 shows the architecture of a decoder-only Transformer layer. which consists of a multi-head attention block, a feed-forward neural network, two layer normalization operators, and two residual connections. The multi-head attention block, the most compute-intensive component of the Transformer layer, consists of h attention heads and each attention head includes multiple batched matrix multiplication operations. The outputs of these attention heads are concatenated to produce the final output of the multi-head attention block. In practice, to optimize hardware utilization, the operations in attention heads are usually processed in a batched manner as opposed to computing each attention head individually.

This section provides background on LLM training (Section 5.2.1). It also introduces LLMCompass (Section 5.2.2), a high-level hardware evaluation tool designed specifically for LLM inference. LLMCompass establishes a solid foundation for the LLMCompass-E2E framework, which is presented later in this chapter.

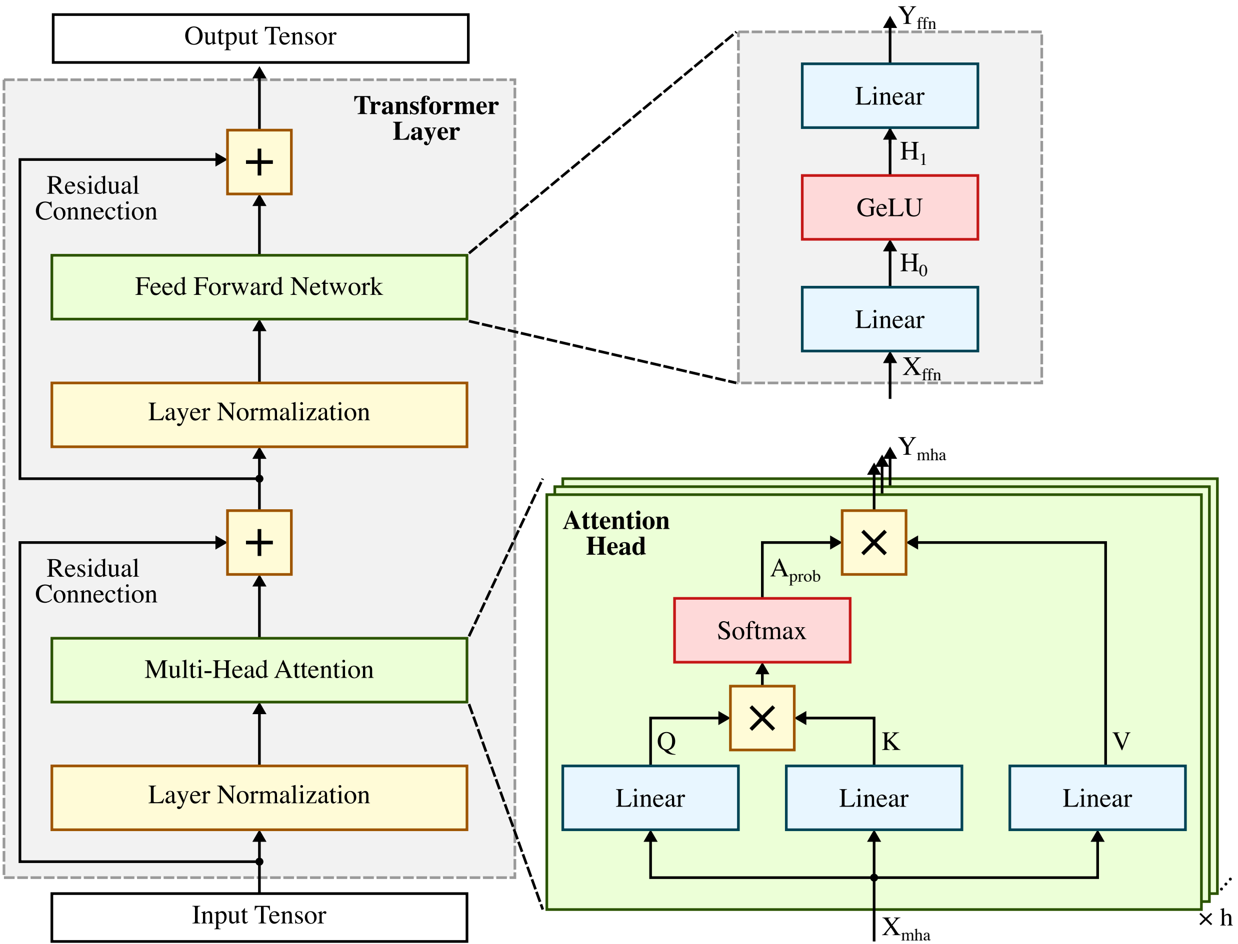


Figure 5.1: Architecture of Decoder-Only Transformer Layer – Linear layer is a fully connected layer with a weight matrix. Q, K, V, A, H, X, and Y denote intermediate tensors. $h =$ number of attention heads. Table 5.1 summarizes the shape of intermediate tensors.

5.2.1 LLM Training

Training LLMs shares the same core principles as training other deep learning models. The training process involves three main steps: *forward pass*, *backward pass*, and *parameter update*. During the forward pass, the model processes input data through its layers, generating predictions based on its current parameters. The output is then compared to the target output. A loss function is used to measure the model's prediction error. Next, in the backward pass, the gradients of the loss function with respect to the model parameters are computed using the back-propagation, which is based on the chain rule of calculus. The optimizer then updates the model parameters using the gradients to minimize the loss. Optimizers such as Adam [KB15] also maintain additional optimizer state (e.g. momentum vectors) to adaptively adjust the learning rate for each parameter.

The optimizer state is also updated after each iteration to refine the training process. This cycle of forward pass, backward pass, and parameter update is referred to as an *iteration*. The training dataset is divided into multiple *batches*, each containing multiple input and target output pairs. Each iteration processes one batch, and the process is repeated over many iterations, gradually improving the model's accuracy.

Due to the enormous scale of LLMs, the training process is highly challenging. The size of LLMs far exceeds the memory capacity of even the largest GPUs (e.g. NVIDIA H100 with 80 GB of memory). Even if we manage to fit the model in a single GPU by frequently swapping data between the GPU memory and the host memory [RRA+21], the vast amount of compute operations required can result in unrealistically long training times. This calls for parallelism. There are data parallelism and model parallelism. Model parallelism includes tensor parallelism and pipeline parallelism.

Training LLMs presents substantial challenges due to their enormous scale. The size of LLMs far exceeds the memory capacity of even the largest GPUs (e.g. NVIDIA H100 with 80 GB of memory). While it is possible to fit an LLM on a single GPU by frequently offloading data between GPU memory and host memory [RRA+21], the sheer volume of computations required would result in prohibitively long training times. To address this, various forms of parallelism are employed, including data parallelism, which distributes different training data across multiple compute nodes, and model parallelism, which partitions the model across multiple compute nodes. Mainstream model parallelism techniques include tensor parallelism and pipeline parallelism.

Data Parallelism – In data parallelism, each compute node stores a copy of the model and processes a different subset of the training data (a batch). The gradients are computed independently on each node and then aggregated across all nodes through an all-reduce operation to update the model parameters. Data parallelism is an effective Data parallelism is an effective technique for scaling out deep learning training to multiple compute nodes. However, it suffers from two key limitations. First, as the level of data parallelism increases, the per-node batch size becomes smaller which leads to reduced hardware utilization and increased communication overhead. Additionally, the maximum number of compute nodes that can be used for training is limited by the batch size, as the smallest batch size is one. This limits the scalability of data parallelism.

Tensor Parallelism – Tensor parallelism, or intra-layer model parallelism, divides matrix multiplications within each Transformer layer across multiple compute devices [SPP+19]. Tensor

parallelism requires frequent all-reduce operations (two all-reduce operations per layer) and relies on high-bandwidth interconnects to achieve high performance. Recent work demonstrates that tensor parallelism can effectively support models up to 20 billion parameters within a multi-GPU sever like the NVIDIA DGX A100, which includes eight GPUs interconnected via high-bandwidth NVLink [SCP+18, SPP+19]. However, for models exceeding this scale, tensor parallelism faces challenges. To accommodate larger models, layers must be split across multiple multi-GPU servers, introducing two significant issues. First, the all-reduce communication required for tensor parallelism now involves inter-server links, the bandwidth of which are much smaller than NVLink thus creating communication bottlenecks. Second, a high degree of model parallelism produces smaller matrix multiplications, which can reduce hardware utilization. These limitations highlight the scalability challenges of tensor parallelism as models grow beyond tens of billions of parameters.

Pipeline Parallelism – Pipeline parallelism distributes different layers of a model across multiple compute devices to form a pipeline [HCB+19, NHP+19, NPS+21]. A batch is divided into smaller microbatches and fed into the pipeline. The peer-to-peer communication between different pipeline stages introduces communication overhead. Furthermore, at the end of an iteration, a pipeline flush is required to synchronize the optimizer state and model parameters, which leads to bubble inefficiency, where stages of the pipeline are underutilized. Consequently, a larger batch size is required to reduce the ratio of pipeline bubbles and achieve high hardware utilization, which may limit the use of data parallelism.

In conclusion, LLM training is a highly complex multi-dimensional optimization problem that requires carefully balancing the trade-offs between data parallelism, tensor parallelism, and pipeline parallelism. Effective LLM training also demands careful coordination between diverse hardware components, including compute devices, memory systems, and interconnects, each of which impacts performance, scalability, and efficiency. Given the intricate interactions and dependencies in LLM training, a holistic modeling tool is needed to capture these complexities. Such a tool can enable researchers to explore and evaluate architecture- and system-level design choices when designing hardware for LLM workloads, facilitating the development of innovative hardware tailored for the demands of LLM workloads.

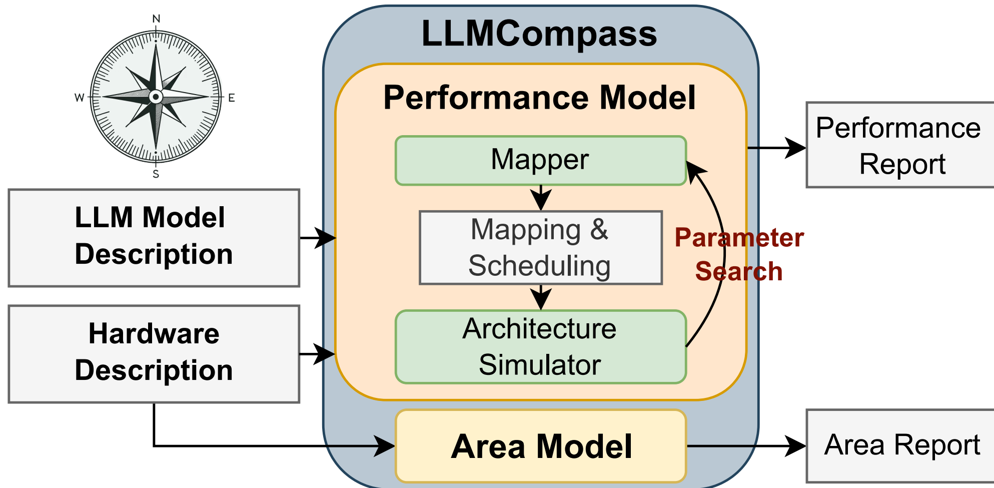


Figure 5.2: LLMCompass Overview – An example hardware system in LLMCompass. Figure adapted from [ZNPW24].

5.2.2 The LLMCompass Framework

The LLMCompass framework is a Python-based high-level hardware evaluation tool designed specifically to model and optimize hardware for *LLM inference*. Traditional approaches to hardware evaluation, such as Roofline models, cycle-level simulators, and FPGA emulation, are either not accurate, too slow, or requires too much engineering effort, making them impractical for rapid iteration and design exploration in the context of LLM inference. LLMCompass addresses these limitations by providing a fast, accurate, and flexible evaluation platform that enables hardware designers to explore a variety of architectural configurations before committing to RTL-level design and implementation.

Figure 5.2 shows an overview of the LLMCompass framework, which requires two inputs: an **LLM model description** and a **hardware description**. The **Performance Model** of LLMCompass takes both inputs and generates a performance report. The **Area and Cost Model** of LLMCompass generates area report based on the hardware description.

LLM Model Description – The LLM model description is a high-level representation of key operators and data flow in the LLM model. It is composed of key computational operators such as matrix multiplication, softmax, layer normalization, and activation functions like GeLU. Figure 5.4 shows a software model of a multi-head attention operator in a transformer layer. The parameters and the operators of the multi-head attention operator are defined in the `MultiHeadAttention`

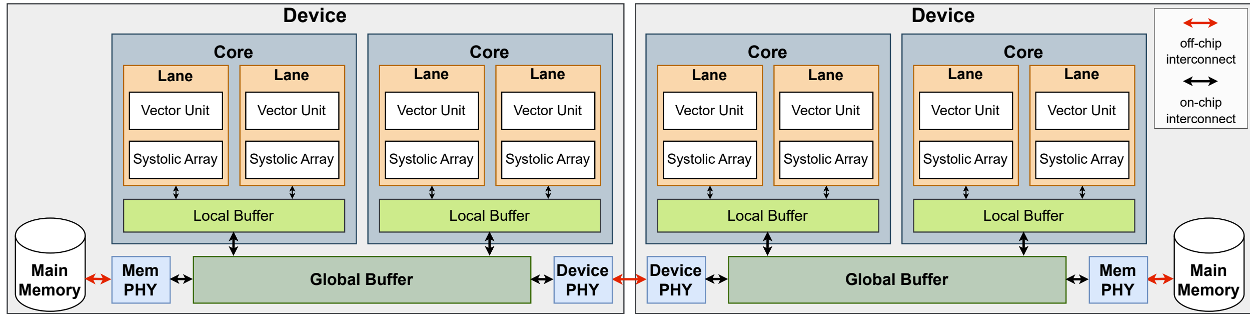


Figure 5.3: LLMCompass Hardware Description Template – Adapted from [ZNPW24].

class. The operator can be called with an input with any valid shape. The operators within the multi-head attention operator automatically deduce the intermediate shapes and stored these information to be used by the performance model.

Hardware Description – Figure 5.3 shows the hardware description template of LLMCompass. The hardware description template describes a hardware component as follows:

- A *system* (e.g., a DGX node) contains multiple compute devices interconnected through an inter-device network, such as NVLink.
- A *device* (e.g., a GPU) contains multiple compute cores, a shared global buffer, and off-chip main memory. The global buffer is essentially a shared last-level cache accessible by all cores.
- A *core* (e.g. a streaming multiprocessor in NVIDIA GPUs) contains multiple lanes and a local buffer. The local buffer is essentially a shared L1 cache accessible by all lanes.
- A *lane* contains the basic compute resources such as a vector unit and a systolic array.

The hardware description template is flexible enough to describe a wide range of existing computing platforms such as NVIDIA GPUs, AMD GPUs, and Google TPUs.

Performance Model – The performance model of LLMCompass contains a mapper and an architecture simulator. The mapper partitions the computation of an operator into smaller sub-tasks that can fit into the global buffer of a device. The sub-tasks are further partitioned into smaller operations that can fit in each core's local buffer. The mapper then provides a scheduling scheme and invokes the architecture simulator to get the execution time for this mapping. The mapper will search through different configurations to find the optimal mapping and scheduling that minimizes the execution time of the operator to fully demonstrate the hardware capability.

```python
from LLMCompass.software_model.operators import Operator, Reshape, Concat, Transpose
from LLMCompass.software_model.matmul import Matmul, BatchedMatmul
...

class MultiHeadAttention(Operator):
def __init__(self, d_model, n_heads, data_type):
  super().__init__(0, 0, 0, 0, data_type)
  self.d_model = d = d_model
  self.n_heads = n_heads
  # parameters
  self.Wq = Tensor([d, d], data_type)
  self.Wk = Tensor([d, d], data_type)
  self.Wv = Tensor([d, d], data_type)
  self.W0 = Tensor([d, d], data_type)
  self.W1 = Tensor([d, 4 * d], data_type)
  self.W2 = Tensor([4 * d, d], data_type)
  # operators
  self.Q_proj = Matmul(data_type)
  self.K_proj = Matmul(data_type)
  self.V_proj = Matmul(data_type)
  self.Q_reshape = Reshape(data_type)
  self.K_reshape = Reshape(data_type)
  self.V_reshape = Reshape(data_type)
  self.Q_transpose = Transpose(data_type)
  self.K_transpose = Transpose(data_type)
  self.V_transpose = Transpose(data_type)
  self.Q_mul_K = BatchedMatmul(data_type)
  self.A_softmax = Softmax(data_type)
  self.A_mul_V = BatchedMatmul(data_type)
  self.H_transpose = Transpose(data_type)
  self.H_reshape = Reshape(data_type)
  self.H_matmul = Matmul(data_type)

def __call__(self, X: Tensor) -> Tensor:
  # b: batch size, s: sequence length, d: hidden dimension, d_h: dimension per head
  b, s, d = X.shape
  h = self.n_heads
  d_h = d // h

  # multi-head attention
  Q = self.Q_proj(X, self.Wq)  # [b, s, d ]
  K = self.K_proj(X, self.Wk)  # [b, s, d ]
  V = self.V_proj(X, self.Wv)  # [b, s, d ]
  Q = self.Q_reshape(Q, [b, s, h, d_h])
  K = self.K_reshape(K, [b, s, h, d_h])
  V = self.V_reshape(V, [b, s, h, d_h])
  Q_T = self.Q_transpose(Q, [0, 2, 1, 3])  # [b, h, s, d_h]
  K_T = self.K_transpose(K, [0, 2, 3, 1])  # [b, h, d_h, s]
  V_T = self.V_transpose(V, [0, 2, 1, 3])  # [b, h, s, d_h]
  A = self.Q_mul_K(Q_T, K_T)  # [b, h, s, s]
  A_prob = self.A_softmax(A)
  H = self.A_mul_V(A_prob, V_T)  #  [b, h, s, d_h]
  H = self.H_transpose(H, [0, 2, 1, 3])  #  [b, s, h, d_h]
  H = self.H_reshape(H, [b, s, d])
  return self.H_matmul(H, self.W0)  #  [b, s, d]
```

Figure 5.4: LLM Model Description in LLMCompass– Software model of a multi-head attention operator in LLMCompass.

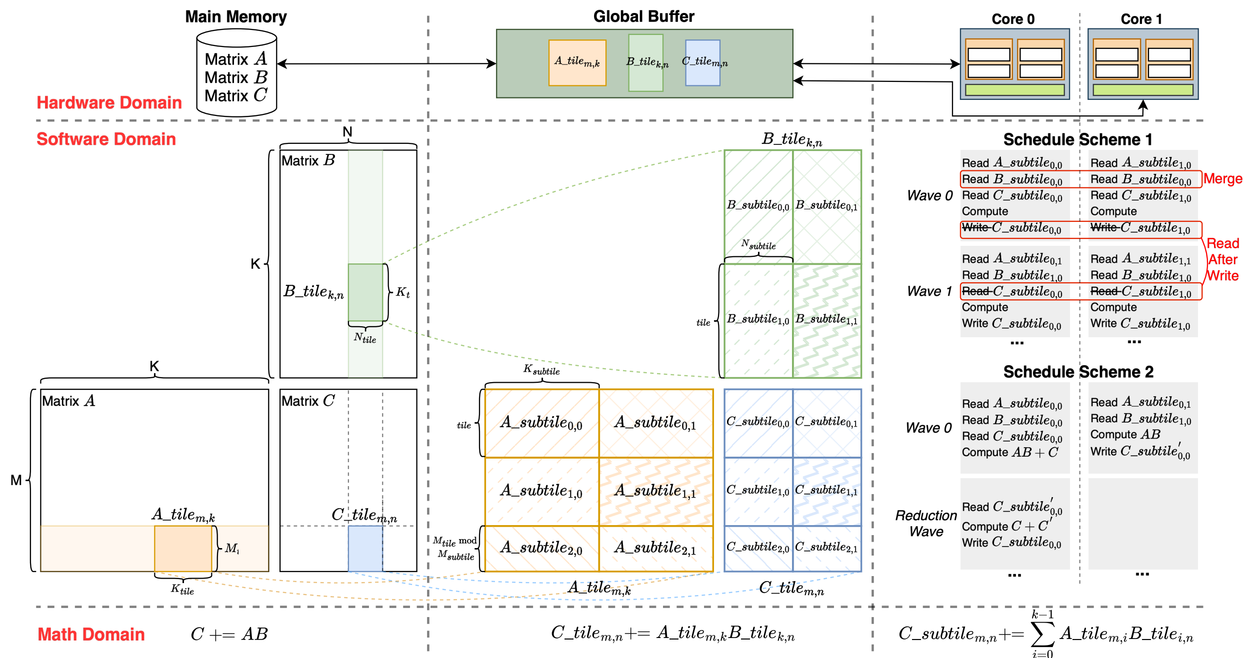


Figure 5.5: Matrix Multiplication Performance Modeling in LLMCompass – Adapted from [ZNPW24].

Figure 5.5 shows an example of modeling a matrix multiplication operation in LLMCompass. The performance model of LLMCompass is validated against real-world hardware platforms such as a 4-GPU node with NVIDIA A100 GPUs, a Google Cloud TPU node with 8 TPUv3 cores, and an AMD MI210 GPU. Compared to the actual hardware performance, the estimated performance by LLMCompass achieves an average error rate of 4.1% for LLM inference.

Area and Cost Model – LLMCompass uses parameters from existing designs, such as transistor counts and memory buffer area in open-source tape-outs and PHY area in publicly available die-photos, to empirically estimate the chip area. LLMCompass also integrates a supply chain model [NTW23] to estimate the wafer costs and per-die costs.

LLMCompass offers a fast, accurate, and flexible hardware evaluation platform specifically for LLM inference. However, it has several limitations that make it less suitable for modeling LLM training. First, LLMCompass primarily models the forward pass, requiring users to manually specify the backward pass, a process that can be complex and prone to errors. Additionally, It only models computation within a single compute node and only supports modeling symmetric collective communications such as all-reduce. It lacks capability to model data and pipeline parallelism across multiple nodes, which are essential for LLM training. These limitations motivate

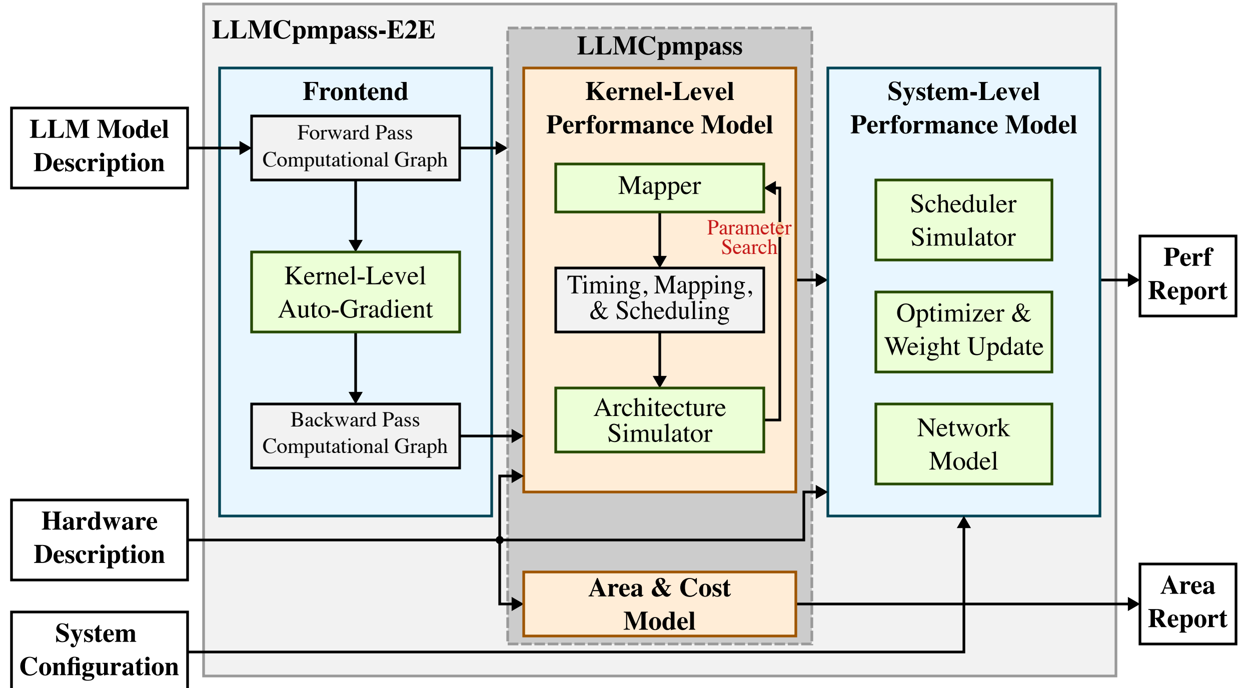


Figure 5.6: LLMCompass-E2E Overview

the need for an extended framework that can support the unique requirements of LLM training workloads at a larger scale.

5.3 LLMCompass-E2E

To address the methodology challenges in modeling large-scale LLM training and inference, we introduce LLMCompass-E2E, a comprehensive modeling framework that enables end-to-end modeling of LLM workloads. LLMCompass-E2E builds upon the original LLMCompass framework, extending its capabilities with a frontend (Section 5.3.1) and a system-level performance model (Section 5.3.2), as is illustrated in Figure 5.6.

The Front End processes the input LLM model description into a computational graph intermediate representation (IR) and performs auto-differentiation to generate the backward pass computational graph. These forward and backward computational graphs, along with the hardware description, are then fed into the kernel-level performance model to simulate the performance of individual computational kernels within each layer. Following this, the kernel-level performance results, hardware description, and an additional system configuration are used by the system-level

```python
from LLMCompassE2E.software_model.modules import MultiHeadAttention
from LLMCompassE2E.software_model.operators import Lossfunction
from LLMCompassE2E.hardware_model import nvidia_a100
from LLMCompassE2E.simulation.utils import simulate_graph

b, s, d = 8, 2048, 12288
num_heads, head_size = 96, 128
mha = MultiHeadAttention(num_heads, head_size, d, fp16)

x = Tensor([b, s, d], fp16, label='x')
y = mha(x)

loss = LossFunction.apply(y)
loss.backward()

fwd_perf_report = simulate_graph(loss.forward_graph(), nvidia_a100)
bwd_perf_report = simulate_graph(loss.backward_graph(), nvidia_a100)
```

Figure 5.7: LLMCompass-E2E Code Example – The code shows how to simulate the forward and backward passes of a multi-head attention block on an NVIDIA A100 model using LLMCompass-E2E.

performance model to simulate the entire training or inference process, producing end-to-end performance results. The system configuration specifies the number of compute nodes, the inter-node and intra-node interconnects, and the parallelization strategy (i.e., degree of data, tensor, and pipeline parallelism). Through these enhancements, LLMCompass-E2E is capable of modeling the entire LLM training and inference process across complex, distributed hardware systems, providing a detailed, holistic view of performance across diverse configurations.

These additions allow LLMCompass-E2E to simulate the entire training and inference process across complex, distributed hardware systems.

5.3.1 Frontend

The LLMCompass-E2E frontend serves as the initial processing stage. This stage processes the **LLM model description**, creating a **computational graph IR** that captures the model's computation and communication operations in a format suitable for performance modeling. An important feature of the Front End is the **kernel-level auto-gradient** module, which automatically generates the computational graph for the backward pass, eliminating the need for manual specification. Figure 5.7 shows an example of the frontend code. Below, we delve into the details of the key components of the frontend: an improved LLM model description, a computational graph IR, and the kernel-level auto-gradient.

```python
from LLMCompassE2E.software_model import Tensor, Operator

class Mul(Operator):
  def forward(self, input1: Tensor, input2: Tensor) -> Tensor:
    # Forward pass of an element-wise multiplication operator generates a new tensor
    # with the same shape and data type as the input tensors.
    assert input1.shape == input2.shape and input1.dtype == input2.dtype
    return Tensor(input1.shape, input1.dtype)

  def backward(self, local_grad: Tensor) -> Tuple[Tensor, Tensor]:
    input1, input2 = self.input_tensors
    # Apply chain rule for multiplication, where y = x1 * x2; dy/dx1 = x2; dy/dx2 = x1
    return input2 * local_grad, input1 * local_grad
```

Figure 5.8: LLMCompass-E2E Operator Example – The code shows how to define a new elementwise multiplication operator in LLMCompass-E2E. The `forward` method computes the shape and data type of the output tensor, while the `backward` method specifies how to compute the gradients of the input tensors given a local gradient.

LLM Model Description – LLMCompass-E2E provides a natural way of describing LLM models that is very similar to PyTorch, as is demonstrated in Figure 5.9, which shows an example code snippet that describes a multi-head attention block in LLMCompass-E2E. Instead of manually registering each operator in the model like in LLMCompass, LLMCompass-E2E implicitly keeps track of the operators that are called in the `forward` method and builds the computational graph for the forward pass.

Computational Graph IR – Figure 5.10(a) shows the computational graph generated by the frontend for the forward pass of the multi-head attention block in Figure 5.9. In the computational graph, each node stores a tensor and the operator that produces the tensor. Each edge represents the data dependency between the tensors. Unlike LLMCompass, where the user needs to manually invoke the simulator on each operator, LLMCompass-E2E automatically processes each operator in the computational graph in topological order. The computational graph IR also potentially makes LLMCompass-E2E more flexible, as it can also be instrumented before sending to the kernel-level performance model. For example, the frontend can fuse the operators in the computational graph to reduce the number of kernel invocations. This can be useful for exploring compiler-level optimizations for LLM workloads.

Kernel-Level Auto-Gradient – The backward pass is typically more complex and contains more operators than the forward pass. For example, the backward operation of an element-wise multiplication involves two element-wise multiplications. The backward operation of a batched matrix multiplication involves two batched-matrix multiplications with a number of broadcast and reduction operations. This makes it challenging and less straightforward to manually specify the

operations in the backward pass. To address this, LLMCompass-E2E provides an auto-gradient module that can automatically generate the computational graph for the backward pass based on the forward pass computational graph. Figure 5.8 shows an example of how to define a new operator in LLMCompass-E2E. An operator needs to be defined with a `forward` method that computes the shape and data type of the output tensor, and a `backward` method that specifies how to compute the gradients of the input tensors given a local gradient. The auto-gradient starts from the final output of the forward pass (e.g., the output of the loss function), iteratively calls the *backward* method of operators, and builds up the computational graph for the backward pass in a breadth-first search manner. Figure 5.10(b) shows the generated computational graph generated for the backward pass of the multi-head attention block in Figure 5.9.

5.3.2 System-Level Performance Model

LLMCompass-E2E also extends the original LLMCompass with a system-level performance model that simulates the entire training or inference process across a number of compute nodes. The core component of the system-level performance model is a scheduler simulator, an event-driven simulator that takes in the system configuration and kernel-level performance results. It is able to generate a pipeline schedule based on hardware constraints such as memory capacity per GPU. This is essential for modeling pipeline parallelism.

The scheduler simulator is capable of generating different pipeline parallelism schedules, such as Gpipe [HCB+19] and PipeDream [NHP+19, NPS+21], to allow exploration of trade-offs in these different schedules. In pipeline parallelism, each compute node (i.e. pipeline stage) holds a subset of layers of the LLM, which reduces the memory capacity requirement for each compute device. However, pipeline parallelism can also introduce additional memory footprint in order to achieve full throughput in the steady state. For each pipeline stage, after computing the forward pass of a microbatch, the activations (i.e. the intermediate results) need to be stored in memory for the corresponding backward pass. For a N-stage pipeline, activations of N microbatches need to be stored in memory in order to achieve full throughput. If the compute device does not have enough memory to store all the activations, we can either (1) stop issuing the microbatches into the pipeline until the backward pass is completed, or (2) adopt activation recomputation to reduce the memory footprint [KCL+23]. The first option can lead to a suboptimal utilization of the pipeline, while the second option can introduce additional computation overhead. The scheduler simulator

supports simulating both options. The scheduler simulator also supports modeling various optimization techniques, such as overlapping the gradient computation of the input with the gradient computation of the model parameters across different microbatches, overlapping the backward pass computation with data parallelism communication, and scatter-reduce optimization to reduce the communication overhead between pipeline stages [NSC+21]. The scheduler simulator can also be used for modeling LLM inference with pipeline and tensor parallelism.

To model the communication overhead between pipeline stages, LLMCompass-E2E assumes a fat tree topology for the inter-node interconnect, which is widely adopted in today's supercomputers. It is possible to implement more detailed network models for other interconnect topologies in LLMCompass-E2E or integrate with existing network simulators such as ASTRA-SIM [RSSK20]. In practice, the peer-to-peer communication between pipeline stages can be slightly misaligned. For example, the sender node may send the data slightly earlier than the receiver node is ready to receive the data, due to the fluctuations in clock frequency, network latency, bandwidth, etc. LLMCompass-E2E supports adding a randomized delay to the peer-to-peer communications to model such synchronization overhead.

5.4 Evaluation

I evaluate the LLMCompass-E2E framework by comparing its simulation results against the reported end-to-end training results of the Megatron-LM models [NSC+21]. I created a detailed model of an 8-GPU A100 node, along with an InfiniBand interconnect model, to replicate the hardware environment used in the Megatron-LM experiments. Using these models, I conduct end-to-end simulations with the same training configurations, including model size, degree of parallelism, batch size, and sequence length, to closely mirror the actual system setup.

Table 5.2 presents a comparison between the reported Megatron-LM results on real hardware and the simulated results from LLMCompass-E2E. The table covers a range of LLMs, from smaller models with 1.7 billion parameters to extremely large models with up to a trillion parameters. Two sets of simulation results are presented: one without synchronization overhead and one with a randomized synchronization overhead (with an average delay of 5ms) added to peer-to-peer communications. As shown in Table 5.2, the simulated hardware utilization with synchronization overhead aligns more closely with the reported Megatron-LM results, especially for larger models

```python
from LLMCompassE2E.software_model import Tensor, Module
from LLMCompassE2E.software_model.operators import Softmax, MegatronF, MegatronG

class MultiHeadAttention(Module):
  def __init__(self, num_heads: int, head_size: int, embed_size: int, dtype: DataType,
               tensor_parallelism: int = 1):
    super().__init__()
    self.embed_size = embed_size
    self.head_size = head_size
    self.num_heads = num_heads
    self.tensor_parallelism = t = tensor_parallelism

    self.key = Linear(embed_size, embed_size // t, dtype)
    self.query = Linear(embed_size, embed_size // t, dtype)
    self.value = Linear(embed_size, embed_size // t, dtype)
    self.proj = Linear(embed_size // t, embed_size, dtype)

  def forward(self, x: Tensor) -> Tensor:
    # b - batch size, s - sequence length, d - embed size (hidden size)
    # h - num_heads, d_h - head size
    # input shape: [b, s, d]
    b, s, d = x.shape
    h = self.num_heads
    d_h = self.head_size
    t = self.tensor_parallelism

    # Apply tensor parallelism
    d = d // t
    h = h // t

    # If tensor parallelism is used, apply the `f` operator as is described
    # in the Megatron paper.
    if t > 1:
      x = MegatronF.apply(x)

    q = self.query(x) # [b, s, d]
    k = self.key(x)   # [b, s, d]
    q = q.reshape([b, s, h, d_h]) # [b, s, h, d_h]
    k = k.reshape([b, s, h, d_h]) # [b, s, h, d_h]
    q_t = q.transpose([0, 2, 1, 3]) # [b, h, s, d_h]
    k_t = k.transpose([0, 2, 3, 1]) # [b, h, d_h, s]
    a = q_t @ k_t                # [b, h, s, s]
    a_prob = Softmax.apply(a) # [b, h, s, s]

    # Attention over values
    v = self.value(x)                      # [b, s, d]
    v = v.reshape([b, s, h, d_h])          # [b, s, h, d_h]
    v_t = v.transpose([0, 2, 1, 3])        # [b, h, s, d_h]
    heads = a_prob @ v_t                   # [b, h, s, d_h]
    heads = heads.transpose([0, 2, 1, 3]) # [b, s, h, d_h]
    heads = heads.reshape([b, s, d])       # [b, s, d]
    out = self.proj(heads)                 # [b, s, d]

    # If tensor parallelism is used, apply the `g` operator as is described
    # in the Megatron paper.
    if t > 1:
      out = MegatronG.apply(out)
    return out
```

Figure 5.9: LLM Model Description in LLMCompassE2E – Software model of a multi-head attention block with support for tensor parallelism in LLMCompassE2E.

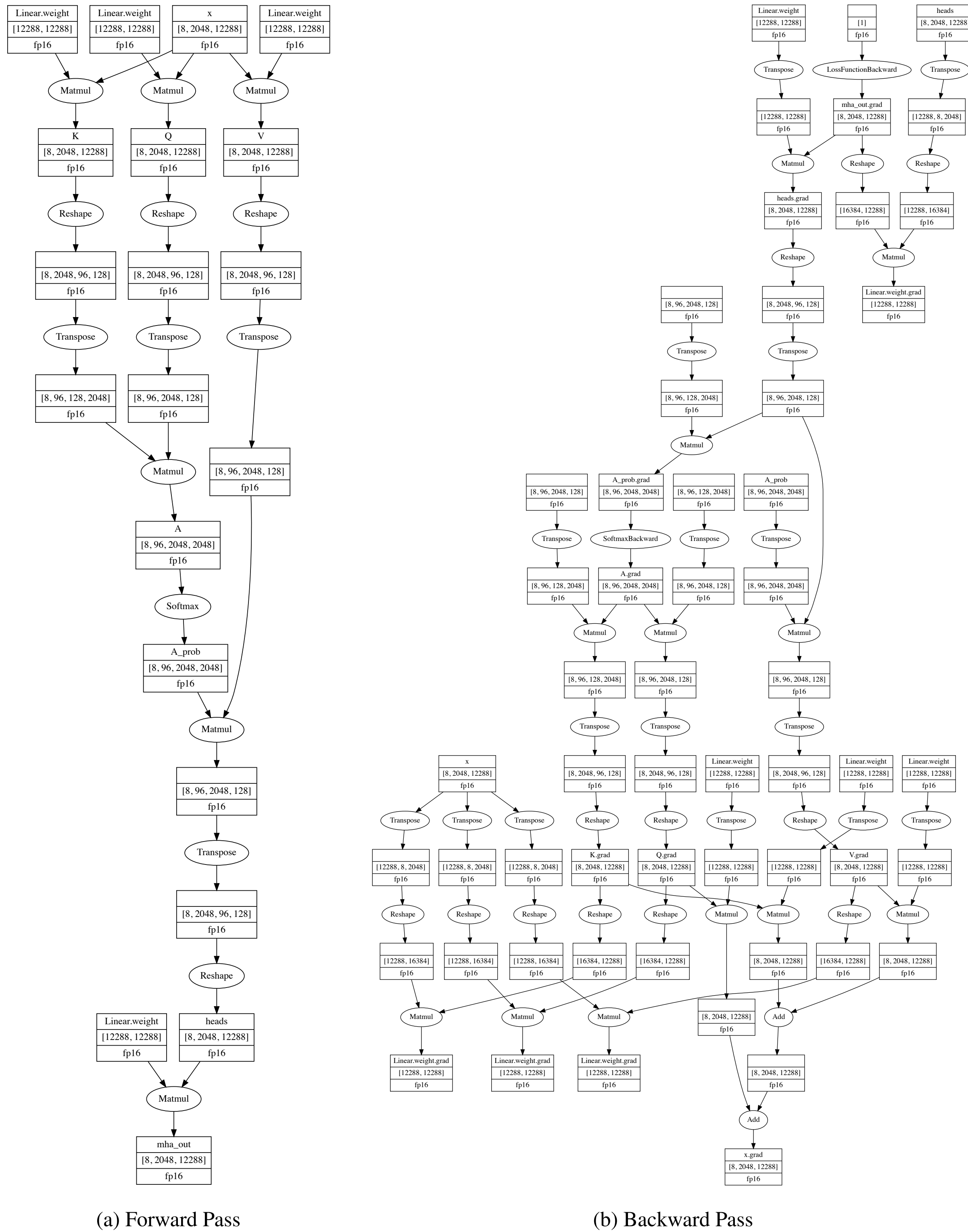


Figure 5.10: Computational Graphs of A Multi-Head Attention Block – Computational graphs generated by the frontend for the forward and backward passes of a multi-head attention block.

#Param. (Billion)	Attention Heads	Hidden Size	#Layers	TP	PP	DP	#GPUs	Batch Size	Reported HW Util. in [NSC+21]	Sim. HW Util.	Sim. HW Util. with Sync. Overhead	Sim. Time (Min)
1.7	24	2304	24	1	1	32	32	512	44%	44.5%	44.5%	11.3
3.6	32	3072	30	2	1	32	64	512	44%	42.8%	42.8%	14.1
7.5	32	4096	36	4	1	32	128	512	44%	41.3%	41.3%	7.48
18.4	48	6144	40	8	1	32	256	1024	43%	38.5%	38.5%	12.1
39.1	64	8192	48	8	2	32	512	1536	44%	48.3%	46.3%	9.9
76.1	80	10240	60	8	4	32	1024	1792	45%	50.3%	47.5%	13.8
145.6	96	12288	80	8	8	24	1536	2304	47%	54.9%	50.9%	13.0
310.1	128	16384	96	8	16	15	1920	2160	50%	61.1%	53.4%	13.5
529.6	128	20480	105	8	35	9	2520	2520	52%	62.0%	53.9%	17.7
1008	160	25600	128	8	64	6	3072	3072	52%	65.0%	54.8%	25.8

Table 5.2: Comparison Against Real Hardware Results – This table shows comparison between the reported end-to-end training results in [NSC+21] and the LLMCompass-E2E simulation results. **TP** = degree of tensor parallelism; **PP** = degree of pipeline parallelism; **DP** = degree of data parallelism. The simulated hardware utilization with synchronization overhead adds a randomized synchronization overhead (5ms on average) to peer-to-peer communications. All experiments are run with a sequence length of 2048.

that rely on extensive pipeline parallelism. Without this adjustment, the simulated results assume perfectly aligned computations across all GPUs at all times, leading to an overoptimistic estimation of the end-to-end performance.

Table 5.2 also shows the simulation time for each experiment. The majority of the simulation time is spent on the kernel-level performance model. For most of the models, the end-to-end simulation time is less than 15 minutes, demonstrating the efficiency of the LLMCompass-E2E framework in providing quick feedback on the performance of LLM training workloads.

5.5 Conclusion

This chapter introduces LLMCompass-E2E, a comprehensive framework for end-to-end performance estimation of large-scale distributed training of LLMs. Building upon the capabilities of the original LLMCompass framework, LLMCompass-E2E incorporates essential extensions such as the front end for generating computational graphs of forward and backward passes, as well as a system-level performance model that considers multi-node configurations, inter-node communication, and various parallelism strategies. Through validation against real-world results from large-scale systems, LLMCompass-E2E demonstrates high accuracy in performance modeling. This framework addresses the unique challenges of modeling the full training and inference pipeline

of LLMs, allowing researchers to evaluate architectural decisions in the context of complex, distributed hardware systems.

CHAPTER 6
ARCHITECTURE: OPTICALLY CONNECTED MULTI-STACK HBM MODULE

This chapter explores the potential of co-packaged optics in memory interconnects. Large language models (LLMs) have grown exponentially in size, presenting significant challenges to traditional memory architectures. Current high bandwidth memory (HBM) systems are constrained by chiplet I/O bandwidth and the limited number of HBM stacks that can be integrated due to packaging constraints. In this chapter, we propose a novel memory system architecture that leverages silicon photonic interconnects to increase memory capacity and bandwidth for compute devices. By introducing optically connected multi-stack HBM modules, we extend the HBM memory system off the compute chip, significantly increasing the number of HBM stacks. Our evaluations through LLMCompass-E2E show that this architecture can improve training efficiency for a trillion-parameter model by 1.4× compared to a modeled A100 baseline, while also enhancing inference performance by 4.2× if the L2 is modified to provide sufficient bandwidth.

6.1 Introduction

In recent years, modern high-performance compute devices such as GPUs and TPUs have largely shifted to using high bandwidth memory (HBM) due to its superior memory bandwidth compared to DDR and GDDR. For instance, in 2017, NVIDIA's Tesla V100 GPU introduced HBM2 with 900 GB/s bandwidth and 16 GB capacity. More recently, HBM3e has further pushed these limits, offering 8 TB/s bandwidth and 192 GB capacity in the NVIDIA GB200 GPU. Fig. 6.1 shows the trend of HBM capacity per device over time. While the capacity per HBM stack has grown from 4 GB to 24 GB over the past seven years, the aggregated HBM capacity remains fundamentally limited by the number of stacks per compute chip. The HBM stacks are connected to the compute chip via high-density short-reach chiplet I/Os, and the number of HBM stacks is ultimately limited by the perimeter of the compute chip. Other memory expansion solutions, such as CXL memory modules [PKS+24, LBH+24], come at the cost of reduced memory bandwidth.

Meanwhile, large language models (LLMs) have demonstrated remarkable capabilities across a broad range of applications [BMR+20, DCLT19]. Driven by the need for higher accuracy and the ability to perform more sophisticated tasks, the size of LLMs continues to increase and has recently

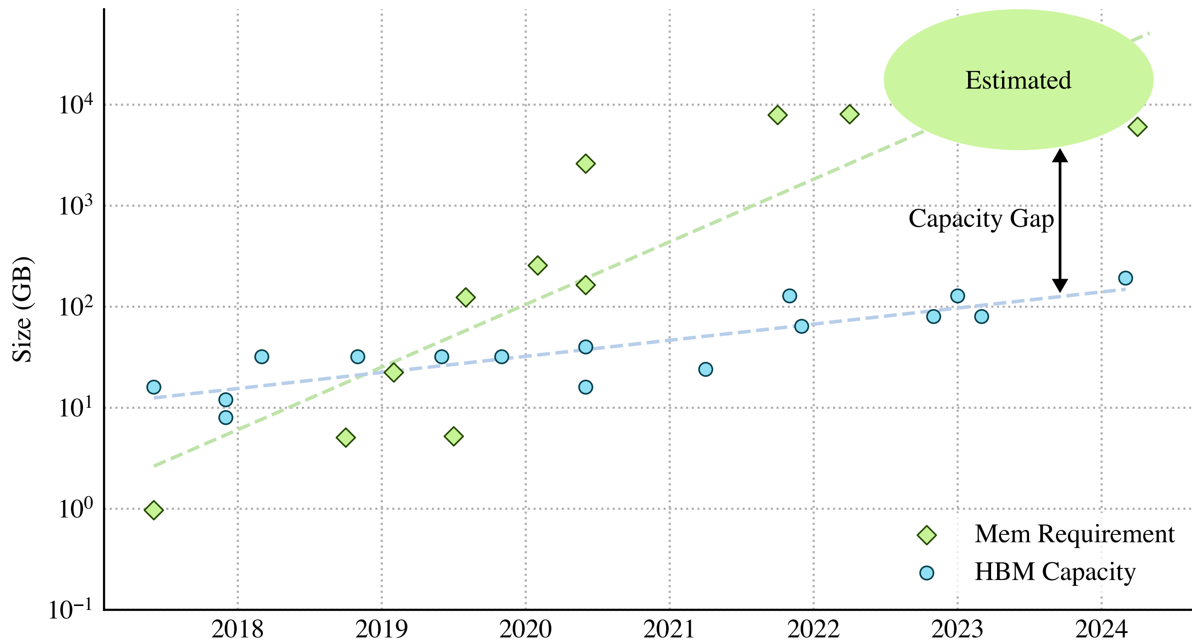


Figure 6.1: Memory Requirement for Training LLMs and HBM Capacity Per Device Over Time – We assume 16-bit precision for LLM parameters and gradients and 32-bit precision for optimizer state. HBM capacity data is adapted from [tec24b].

surged into trillions of parameters [FZS22] (see Fig. 6.1). This poses significant challenges for the memory capacity of compute devices. For instance, training the GPT-3 model with 175 billion parameters [BMR+20], requires approximately 3 TB of memory for storing the model parameters, gradients, and optimizer state. Today's larger models, like Switch Transformer [FZS22], can have 1–2 trillion parameters and require up to 32 TB of memory to train.

Early LLMs fit within a single compute device, enabling efficient scaling of training through *data parallelism*. However, today's largest models far exceed single device memory capacity, and thus require the use of *model parallelism*, including tensor parallelism [SPP+19] and pipeline parallelism [NHP+19, NPS+21]. Tensor parallelism splits the computation of each layer across multiple devices, while pipeline parallelism segments the model into different stages processed sequentially. However, these parallelism strategies introduce new complications. With tensor parallelism, the frequent all-reduce communication between devices can reduce overall efficiency. Similarly, pipeline parallelism suffers from bubble inefficiencies where stages of the pipeline are underutilized, particularly during the ramp-up and ramp-down phases of the pipeline. This leads to the key observation that motivates this work: **Efficiently exploiting model parallelism for LLM training is largely limited by per-device memory capacity.**

In this chapter, we propose a novel memory architecture using silicon photonic interconnects to expand the memory capacity and bandwidth of compute devices. We introduce optically connected multi-stack HBM modules, a separate chip package with multiple HBM stacks and connected to the compute chip via co-packaged optics. With co-packaged optics, we extend the HBM memory system off the compute interposer, circumventing the chip packaging constraint and allowing more HBM stacks to be connected to the compute chip. In an augmented A100 system, we achieve 576 GB of memory capacity and 12 TB/s of bandwidth using the same HBM technology. Our system improves model FLOPs utilization (MFU) by up to 1.4× for trillion-parameter LLM training. In addition, the increased bandwidth of our system can also benefit LLM inference, improving decoding performance by up to 4.2× with sufficient L2 bandwidth.

6.2 Background

Silicon photonics is an emerging technology that enables the integration of optical components, such as ring resonators, photodetectors, and microdisk modulators onto silicon-based substrates using CMOS-compatible fabrication processes. While optics offer several advantages over electrical interconnects, including higher bandwidth density, higher energy efficiency, lower loss, and reduced latency over a long distance, integrating optics with electronics presents challenges. This section introduces three integration approaches for silicon photonics and discusses their advantages and challenges.

Optical interconnects up to date have mainly taken the form of pluggable optical transceivers, where standardized, removable optical modules can be plugged into a system board or a switch chassis, as is shown in Figure 6.2(a). Each pluggable optical transceiver is a separate board that houses an optical chip and an electrical transceiver chip. This approach offers flexibility and cost-effectiveness compared to more integrated approaches, as these pluggable transceivers can be independently replaced or upgraded without altering the system board. However, the separation between the compute chip and the optical module introduces critical drawbacks. Data going to the compute chip must traverse many steps of electrical traces, such as connectors, PCB traces, PCB vias, and package traces, which contributes to signal loss and limits the achievable bandwidth and energy efficiency. The final electrical pathways ultimately constrain the overall performance of a pluggable optical transceiver.

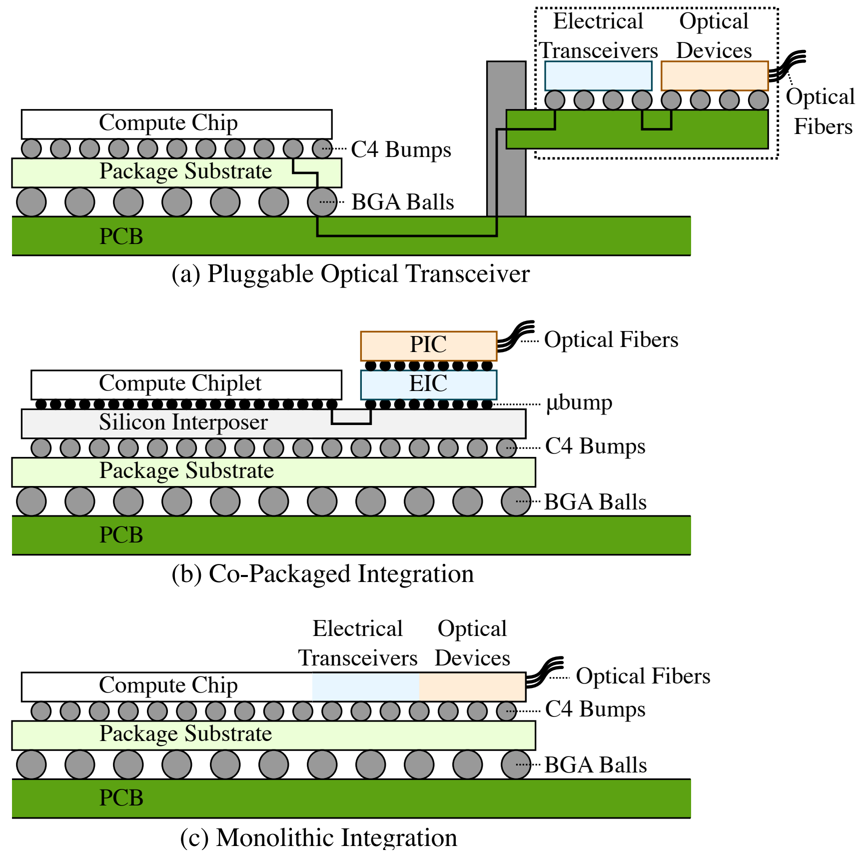


Figure 6.2: Silicon Photonics Integration Technologies

Monolithic integration, illustrated in Figure 6.2(c), refers to the fabrication of both electrical and optical components directly on the same silicon substrate. This approach provides the closest possible integration of electrical and optical components, potentially offering the highest bandwidth density and lowest power consumption by eliminating the interconnect losses between the electrical and optical domains. Early work explored the potential of this approach [BJO+09, OMS+12, SWL+15, BZL+12, RGP+13], but it has yet to see widespread adoption. The major challenge of monolithic integration is that the fabrication process for advanced electrical and optical components is inherently incompatible. The optimal process node for electronics is typically not applicable for optics, and vice versa. While modern compute chips require highly aggressive process nodes (e.g., 7nm or 5nm), optical components work more reliably with older, less dense nodes. This mismatch limits the use of cutting-edge electronics and optics in monolithic integration, which can lead to suboptimal performance and yield.

Co-packaged integration, illustrated in Figure 6.2(b), has emerged as a promising technology for silicon photonics integration since the chiplet-based packaging has become increasingly popular in the semiconductor industry [CKT+15, CSY+23, DLK+23, DRA+21, DRL+23, KLO+24, RDN+23a]. This approach involves integrating multiple chiplets, such as an electrical interface chiplet (EIC) and an optical interface chiplet (PIC) within the same chip package as the compute chiplet. The chiplets are connected using high-density microbumps and/or through-silicon vias (TSVs) which offer comparable bandwidth and energy efficiency to on-chip wires. Moreover, this approach also enables fabricating the compute chiplet, EIC and PIC using their respective optimal process nodes, which can lead to higher yield and better performance for all components. Co-packaged integration strikes a balance between the pluggable optical transceivers and monolithic integration. It eliminates the long electrical traces between the compute chip and the optical module, while also overcoming the process node mismatch by decoupling the fabrication of the electrical and optical components. This approach has the potential to unleash the full benefits of silicon photonics to satisfy the communication demand of modern highly distributed workloads.

6.3 System Architecture

Fig. 6.3 depicts an example system architecture with optically connected multi-stack HBM modules. The proposed system consists of a compute multi-chiplet module (MCM) and multiple multi-stack HBM modules. The compute MCM incorporates six electrical interface chiplets (EICs) and six photonic interface chiplets (PICs), which are co-packaged using 3D integration. The multi-stack HBM module includes an EIC-PIC pair and is connected to the compute MCM directly via optical fibers. The system design is based on an A100-sized compute chip. Key design parameters, such as chip dimensions, optical fiber pitch, and optical bandwidth are adapted or derived from recent works on opto-electronic transceivers and MCMs [KLO+24, WWP+24, WNP+23]. Given that the width of the EIC-PIC pair is close to that of an HBM stack, the six HBM stacks in the A100 GPU chip can be replaced with six EIC-PIC pairs. Each EIC-PIC pair has a total of 48 optical fibers with a pitch of 127 µm, constituting 16 optical channels. Each optical channel is comprised of three fibers: one for unmodulated comb lines, one for transmitter (TX) signals and one for receiver (RX) signals. Each signal fiber carries 64 wavelengths modulated at 16 Gb/s, resulting in a total unidirectional bandwidth of 1 Tb/s. In total, our design provides 12 TB/s of

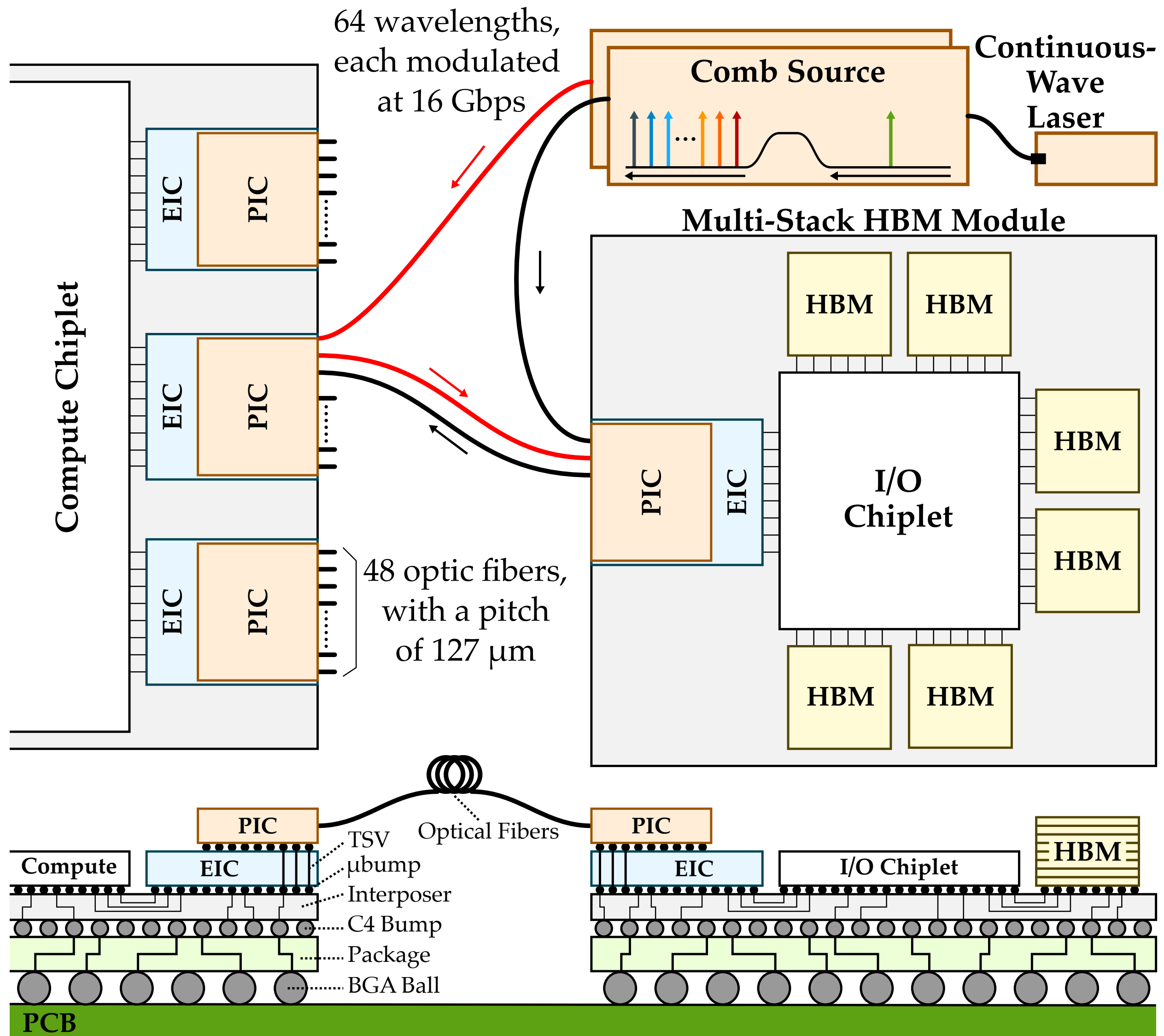


Figure 6.3: Example System Architecture

bandwidth and 576 GB of capacity using the same HBM2e technology. The system shown in Fig. 6.3 is just one conceptual example, and the proposed architecture can also be adapted to other packaging strategies such as embedded PIC [WNP+23], EIC on top of PIC [DRL+23, DLK+23], and monolithic EIC-PIC [SJZ+20, HKS+21, HKS+22].

Fig. 6.4 shows the detailed structure of the optical datapath highlighted in Fig. 6.3. The TX array is driven by a comb source that generates hundreds of low-noise frequency channels (comb lines) from a continuous-wave (CW) laser. The comb lines are subdivided by two stages of de-interleavers into four buses, each containing 16 wavelengths. Each wavelength is modulated by a microdisk modulator in the TX array. The modulated wavelengths are combined by two stages of

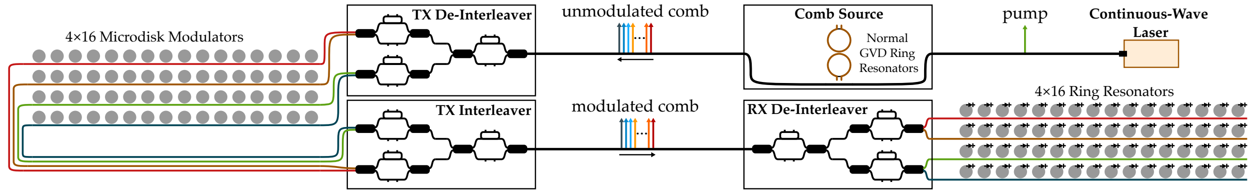
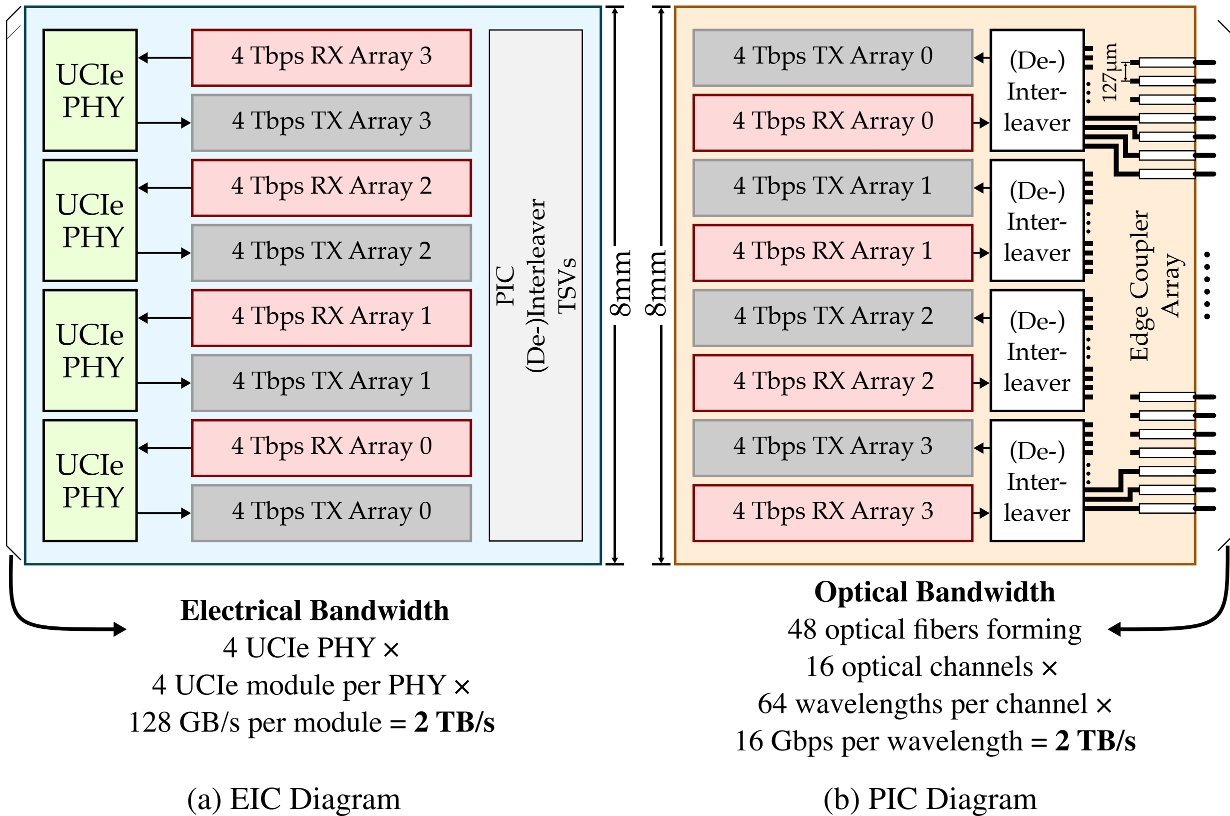


Figure 6.4: Optical Channel Datapath



(a) EIC Diagram (b) PIC Diagram

Figure 6.5: EIC and PIC Architecture – Mixed-signal transceivers in the EIC is based on [KLO+24]. PIC design is adapted from [WNP+23, WNP+23].

interleavers and transmitted via a single fiber. At the RX side, the modulated wavelengths are de-interleaved into four buses and sent to arrays of cascaded ring resonators that drop each wavelength onto a photodetector to generate electrical signals.

Fig. 6.5(a) shows the EIC structure. The EIC is connected to the compute or I/O chiplet via Universal Chiplet Interconnect express (UCIe) PHYs. We derive the design parameters from the 16 GT/s UCIe PHY from the UCIe 1.0 specification [uci24]. Each UCIe PHY contains four 128 GB/s UCIe modules. The EIC includes four UCIe PHYs to provide a total bandwidth of 2 TB/s, matching the optical bandwidth. The EIC also includes mixed-signal transceiver arrays

to communicate with the PIC via high density microbumps and through-silicon vias (TSVs). Fig. 6.5(b) illustrates the PIC structure, adapted from [WNP+23]. It includes an edge coupler array, de-interleavers and interleavers, and transceiver arrays. The edge coupler array is used for attaching optical fibers. Each TX array contains 256 microdisk modulators and each RX array contains 256 ring resonators, corresponding to four optical channels.

Our design addresses two key constraints limiting the capacity and bandwidth of current HBM-based memory systems.

Constraint #1: The number of HBM stacks is restricted by the perimeter of the compute chiplet. The number of HBM stacks is limited by the perimeter of the compute chiplet, which is ultimately constrained by the reticle size limit in lithography. Our proposed design overcomes this by extending the HBM stacks into separate chip packages using co-packaged optics, with slight overhead in latency and energy. Instead of direct microbump connections to HBM, the compute chiplet connects optically to multiple I/O chiplets, and each I/O chiplet is connected to multiple HBM stacks. As a result, without increasing the chip perimeter, significantly more HBM stacks can be connected to the compute chiplet.

Constraint #2: The HBM interface operates below the maximum possible data rate of chiplet I/O. Current compute chiplets use HBM PHYs to communicate with the HBM stacks at up to 9.6 Gb/s per microbump (HBM3e). However, state-of-the-art chiplet interconnects like UCIe can reach 16 Gb/s or even 32 Gb/s under similar bump pitch. The bandwidth of HBM PHYs are constrained by the DRAM speed rather than the data rate of microbumps. By replacing the HBM stacks with EIC-PIC pairs, our design leverages faster UCIe PHYs to achieve a higher bandwidth. In the multi-stack HBM module, multiple HBM stacks can be accessed in parallel to match the optical bandwidth.

6.4 Evaluation

We evaluate our proposed system using the LLMCompass-E2E framework presented in Chapter 5 for both training and inference. We model an 8-GPU A100 compute node with LLMCompass-E2E as the baseline system. We create a model of our proposed memory system in LLMCompass-E2E and integrate it into the A100 model. The memory system, as is detailed in Section 6.3, offers 12 TB/s of memory bandwidth and 576 GB of memory capacity.

Fig 6.6 shows the evaluation results for modeling training of a 175-billion and a 1-trillion parameter LLM. We use the kernel-level performance model to simulate the execution time of the forward and backward pass. We generate different pipeline schedules based on the memory capacity constraints and create an InfiniBand network model in LLMCompass to simulate the communication time for data parallelism. We compare the achieved MFU of three different systems: the baseline A100 model, the A100 model with only the bandwidth enhancement of our design, and the A100 model with optically connected multi-stack HBM modules. We sweep the number of GPUs from 1 to 4096, and each point in the plot represents a possible mapping with certain degree of tensor, pipeline, and data parallelism. We can see that the improvements in MFU mainly benefit from the capacity enhancement as opposed to bandwidth. This is because the main operations in the training process, matrix-matrix multiplications, are compute-bound. Our design particularly benefited the training of the 1-trillion parameter LLM by offering higher memory capacity. At 4096 GPUs, the best mapping with our design achieves a 1.4× improvement in MFU. The baseline system, constrained by the memory capacity, has to use more model parallelism to partition the model parameters, gradients, and optimizer state. It also requires activation recomputation during the backward pass since it does not have sufficient memory to hold the activations, which leads to reduced MFU.

We compare the per-layer latency of the prefill and decode stages for the 175-billion and 1-trillion parameter LLM. Since our design exhibits bandwidth inversion, offering 12 TB/s bandwidth which is well above the 7 TB/s L2 bandwidth of the baseline A100, we also evaluate a system with an aggressive 24 TB/s L2 bandwidth to fully utilize the optical bandwidth. As is shown in Fig. 6.7, for prefill, our design with aggressive L2 bandwidth achieves an average 1.14× speedup across different sequence lengths for both the 175B and 1T model. For decoding, our design with aggressive L2 achieves 3.17× speedup for the 175B model and 4.23× speedup for the 1T model on average across different context lengths. Without the L2 modification, the speedup is 1.53× and 1.67× respectively, indicating that our design requires rethinking the memory hierarchy design to fully utilize the massive memory bandwidth offered by the optically connected multi-stack HBM modules. The capacity enhancement of our design can also potentially benefit the latency and throughput of LLM inference by allowing larger sequence length, batch size, KV cache size, as well as enabling more optimal parallelization strategies. Future work will evaluate the impact of our design on end-to-end LLM inference systems.

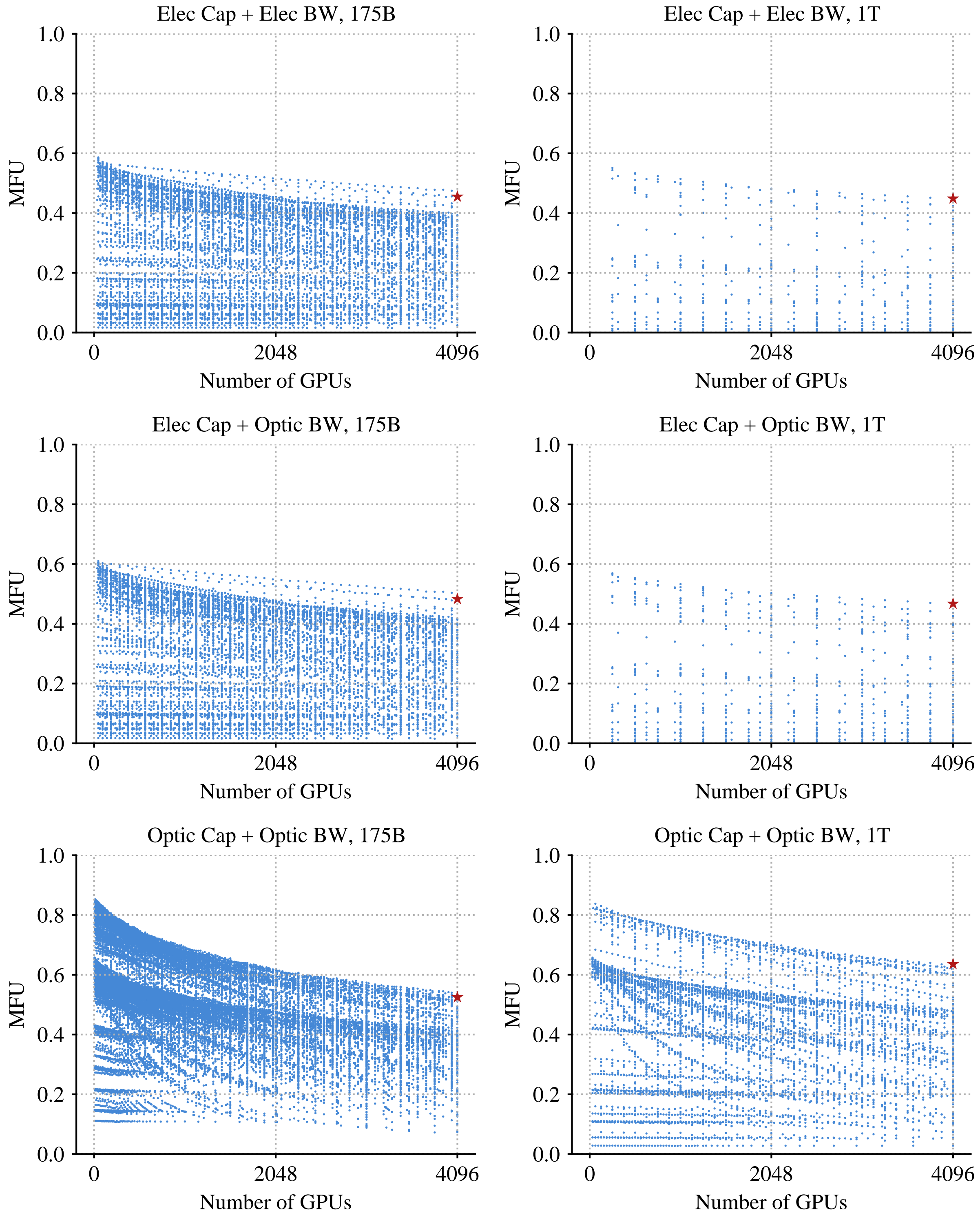


Figure 6.6: Evaluation Results for Training – Each dot represents a valid training configuration for a given number of GPUs. Optimal configuration at 4096 GPUs are highlighted. Elec Cap = 80 GB capacity, Elec BW = 2 TB/s bandwidth, Optic Cap = 576 GB capacity, Optic BW = 12 TB/s bandwidth. 175B = 96 layers, 96 attention heads, and an embedding size of 12288, 1T = 128 layers, 160 attention heads, and an embedding size of 25600. TP = tensor parallelism, PP = pipeline parallelism, DP = data parallelism, act. recomp. = activation recomputation. All experiments are performed using a sequence length of 2048 and a batch size of 4096.

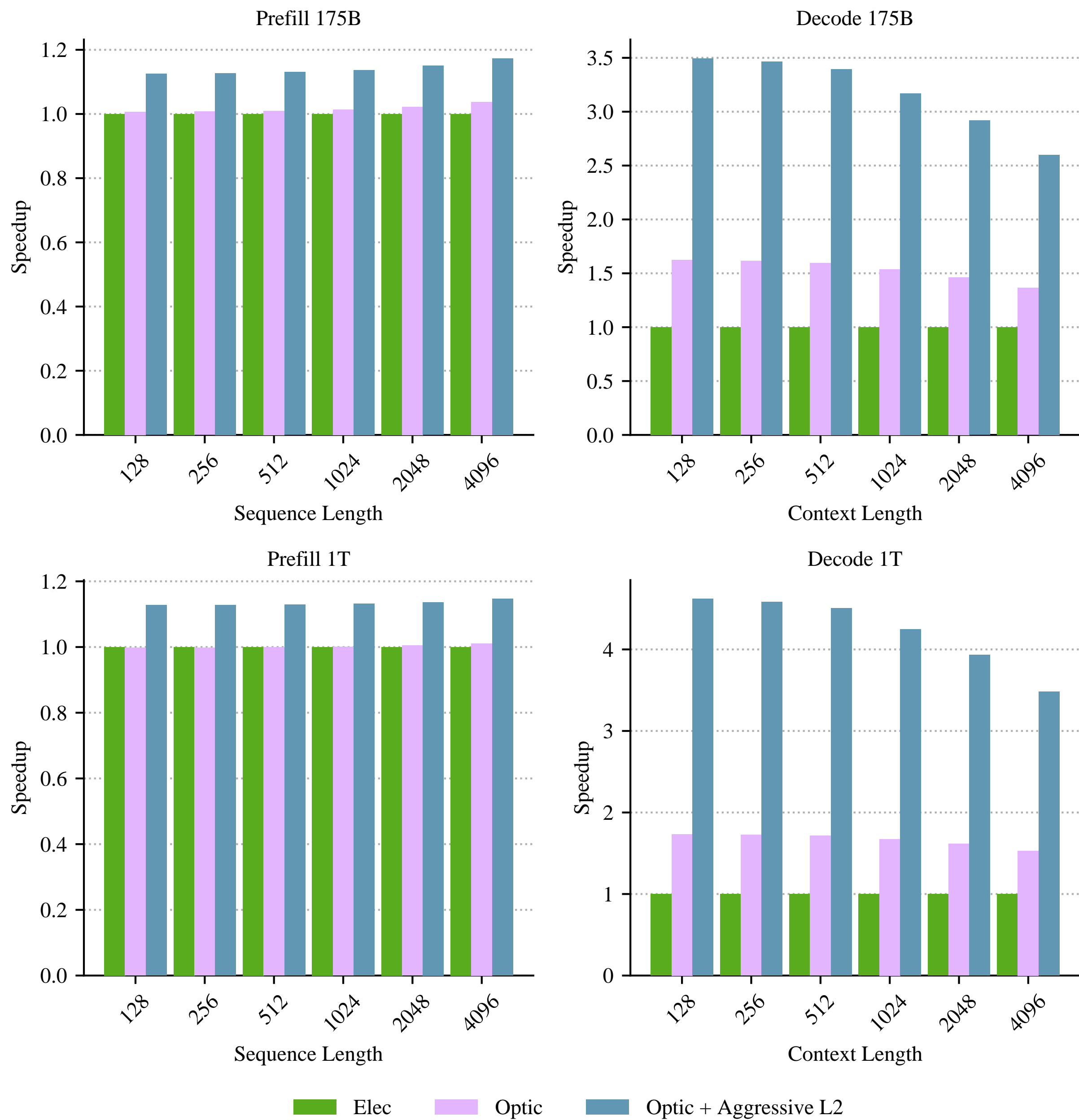


Figure 6.7: Per Layer Speedup for Inference – Elec = A100 GPU, Optic = A100 GPU with our design, Optic + Aggressive L2 = A100 GPU with our design and 24 TB/s L2 bandwidth. Both prefill and decode is simulated with batch size = 8.

6.5 Related Work

Beamer et al. propose PIDRAM, a photonically interconnected DRAM architecture that uses monolithically integrated silicon photonics to address the bandwidth and power limitations of electrical DDR-based memory systems [BSK+10]. Our work is distinct from PIDRAM in that we use 3D integrated silicon photonics to augment the HBM-based memory system.

Khani et al. introduce SiP-ML, which uses silicon photonics to create a flat network topology for inter-GPU communication [KGA+21]. Wu et al. propose SiPAC, co-designing a inter-GPU silicon photonic interconnect and a collective communication algorithm to accelerate distributed deep learning. Our work is complementary to SiP-ML and SiPAC in that our work explores optical interconnects between the compute chip and the HBM-based main memory.

Gonzalez et al. propose an optically connected memory architecture for disaggregated data centers, using silicon photonics to create high-bandwidth, low-latency optical links between different resource pools [GGH+22]. The GPU memory system is not modified. In contrast, our work leverages co-packaged optics to augment the HBM-based memory for compute devices.

6.6 Conclusions

We propose optically connected multi-stack HBM modules to enhance the capacity and bandwidth of HBM-based memory systems. Utilizing co-packaged optics, our design connects compute devices to multiple off-chip HBM stacks. Our evaluations show significant improvements in memory capacity and bandwidth, enhancing the training and inference efficiency for large-scale LLMs. The results also suggest that current memory hierarchy designs need to be reconsidered to fully exploit the advantages of optical interconnects. Future work will explore memory hierarchy designs optimized for optical interconnects and assess the impact of our proposed architecture on end-to-end inference systems.

CHAPTER 7
PROTOTYPE: PIPES CHIP TAPE-OUT

While the previous chapter discusses the potential of co-packaged optics for memory interconnects, this chapter explores its application for system interconnects by presenting the PIPES[1] tape-out. The PIPES tape-out is a silicon prototype that demonstrates the potential of co-packaged optical interconnects in meeting future demands for scalable, high-performance, and energy efficient interconnects. This chapter describes the PIPES multi-chip module (MCM), a novel system which uses hybrid 2.5D/3D integration to compose a state-of-the-art FPGA compute chiplet, three electrical interface chiplets, and three photonic interface chiplets. We use register-transfer-, gate-, transistor-, and device-level simulations to demonstrate the potential for this system to achieve 96 Tb/s of bi-directional bandwidth, and we experimentally demonstrate key components including a complete opto-electrical channel. Our results provide a strong case for hybrid 2.5D/3D integration as the key enabler for scaling co-packaged optical interconnects. Evaluation through LLMCompass-E2E shows that the PIPES system can potentially improve the This chapter thus contributes to the thesis's exploration of advanced interconnect architectures, providing a practical demonstration of how photonic interconnects can break existing trade-offs in bandwidth and scalability for high-performance systems.

7.1 Introduction

Modern data-center and high-performance computing workloads are increasingly limited by inter-node communication overheads. This has motivated the use of inter-node optical interconnects to enable longer reach, higher bandwidth, and lower energy compared to equivalent electrical interconnects [CBG$^+$18]. State-of-the-art systems use fiber-optic cables that are connected to compute boards through pluggable optical transceiver modules, which are then connected to the compute package through board-level electrical interconnects. Unfortunately, this final step of the electrical interconnect is a significant bandwidth and energy bottleneck. Tightly integrated optical interconnects promise to overcome this bottleneck by directly attaching fiber-optic cables to the compute package itself.

[1] PIPES stands for Photonic in Package for Extreme Scalability.

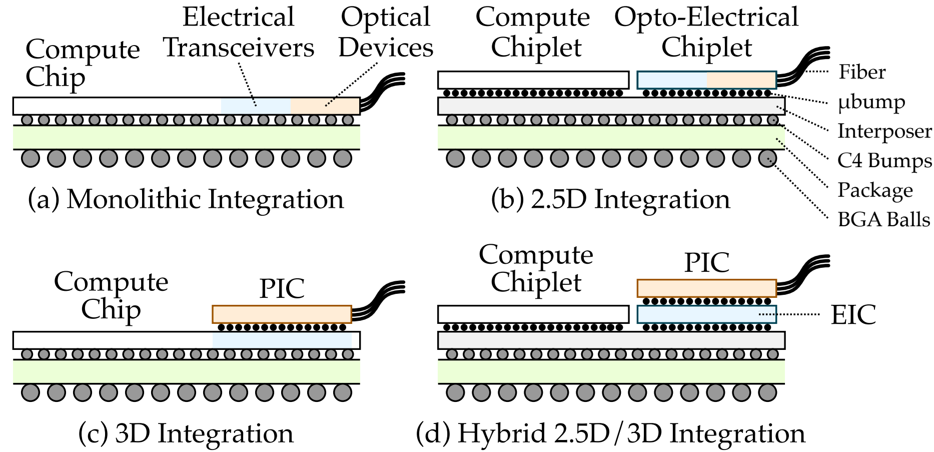


Figure 7.1: Approaches to Tightly Integrated Optical Interconnect

Early work proposed *monolithic integration* where optical devices are directly integrated into the compute die [BJO+09, OMS+12, SWL+15, BZL+12, RGP+13] (see Figure 7.1(a)). However, this approach has yet to see widespread adoption as the optimal process for electronics is often sub-optimal for optics and vice-versa. An alternative approach uses *2.5D integration* where an electrical compute chiplet and opto-electrical chiplet (with electrical transceivers and optical devices) are co-packaged on an interposer or with an embedded silicon bridge [HKS+21, HKS+22] (see Figure 7.1(b)). This approach allows the electrical compute chiplet to be fabricated in an advanced technology node, but designers again face a difficult trade-off in optimizing the opto-electrical chiplet. *3D integration* involves an electrical compute chiplet (with compute logic and electrical transceivers) and an optical device chiplet directly integrated into a 3D stack [CKT+15, CSY+23, DLK+23, DRA+21, DRL+23, KLO+24, RDN+23a] (see Figure 7.1(c)). This approach not only requires more sophisticated packaging, but also requires fixing the integration of compute logic and transceivers at design time. *Hybrid 2.5D/3D integration* offers a compelling compromise: 3D integration stacks an optimized optical device chiplet with an optimized electrical transceiver chiplet and then 2.5D integration packages this 3D stack with an optimized electrical compute chiplet (see Figure 7.1(d)). While prior work has proposed hybrid 2.5D/3D integration for co-packaged optics [CKT+15, CSY+23], no prior work has experimentally demonstrated a complete system combining a state-of-the-art electrical compute chiplet, electrical transceiver chiplet, and optical device chiplet using hybrid 2.5D/3D integration.

This chapter describes a novel system which uses hybrid 2.5D/3D integration to compose an Intel Stratix 10 FPGA chiplet fabricated on an advanced technology node, three electrical inter-

face chiplets (EIC) fabricated on Intel 16 nm, and three photonic interface chiplets (PIC) fabricated through AIM Photonics (see Figure 7.1). An EIC and PIC are stacked using 3D integration based on 55 µm-pitch µbumps, and each EIC/PIC stack is integrated with the FPGA chiplet using using 2.5D integration based on 45 µm-pitch µbumps and Intel's Embedded Multi-Die Interconnect Bridge (EMIB) technology. Unlike Figure 7.1(d), the system presented in this paper positions the EIC on top of the PIC. The underlying design philosophy is scaling to many more channels at modest data rates will be particularly advantageous for achieving both high aggregate bandwidth and high energy efficiency [TPN+24, Ton24]. Given this motivation, each EIC/PIC stack implements 1,024 optical channels in each direction with 64 channels wavelength-division multiplexed (WDM) onto a single fiber. The channels are designed to potentially operate at up to 16 Gb/s leading to a peak bi-directional optical bandwidth of 32 Tb/s per EIC/PIC stack and 96 Tb/s aggregated across the system. We have conducted a rigorous simulation-based study using register-transfer-, gate-, transistor-, and device-level models, and we have conducted a functional demonstration of the system in the lab including validating a complete opto-electrical channel from the EIC through the PIC.

Our main contributions are: (1) a detailed description of a novel FPGA/EIC/PIC system which uses hybrid 2.5D/3D integration including discussion of key techniques for overcoming scaling challenges in both the EIC and PIC; (2) simulation-based evaluation demonstrating the potential for this system to achieve 96 Tb/s of bi-directional optical bandwidth; and (3) experimental demonstration of the key components of this system including a complete opto-electrical channel.

7.2 PIPES System Architecture

Figure 7.2 illustrates the three types of chiplets in the system. We use an Intel Stratix 10 FPGA chiplet with 1.3M logic elements, 2.5K digital signal processing (DSP) blocks, and 114 Mb on-chip memory. The FPGA chiplet connects to three separate EICs through Intel's EMIB and an Advanced Interface Bus (AIB) interface. Each EIC is 8×8 mm with 1.4K C4 bumps and 13K µbumps. Each EIC is flip-chip bonded to a PIC. Each PIC is 8.6×8.1 mm with 10K µbumps and is directly attached to an optical fiber array.

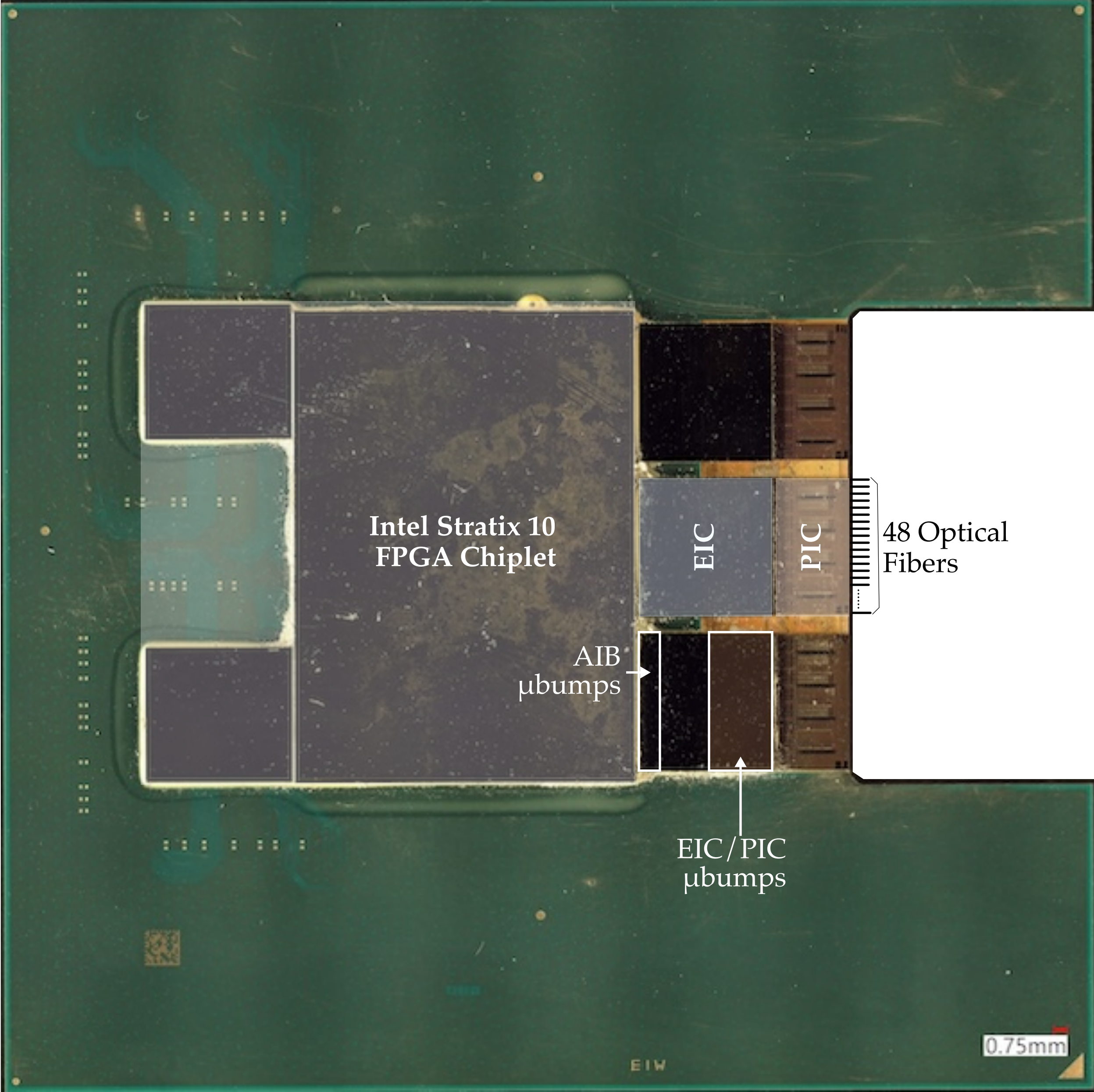


Figure 7.2: PIPES Package Photo – Photo taken at Cornell NanoScale Facility.

7.2.1 EIC Architecture

Figure 7.3 shows the EIC die photo, and Figure 7.4 illustrates the three major EIC blocks: AIB, crossbar, and transceivers (TRX).

1) AIB: The AIB is a commercially available physical-layer IP that implements the AIB protocol [int]. The AIB connects 24 channels to a crossbar which operates at 500 MHz. Each AIB channel can potentially support up to 40 Gb/s in AIB 1.0 mode or 160 Gb/s in AIB 2.0 mode, although only AIB 1.0 mode is enabled in this system for a total potential AIB bandwidth of 960 Gb/s per EIC. The AIB is configured through an Avalon Memory Mapped (AVMM) interface, which is an address-based read/write interface.

2) Crossbar: The crossbar routes the AIB channels to the TRX channels and hosts a majority of the control and test infrastructure for the EIC. The crossbar integrates 48 AIB interface units, 1024 TRX interface units, and 48 32-bit channels at 500 MHz which connect the AIB interface macros to 48 of the TRX interface units. Each interface macro is equipped with programmable scan chains, pseudorandom binary sequence (PRBS) generators and verifiers, and fixed pattern generators and verifiers. Both AIB and TRX interface units can be configured to either pass through or loop back the received data.

3) TRX: The TRX is designed for high-density 3D integration with the PIC and is similar to the architecture from Khilwani et al. [KLO+24]. The TRX comprises four TRX groups, each containing 256 TRX cells. Each TRX cell includes a transmitter (TX) path that transmits data from the crossbar to the PIC, and a receiver (RX) path that receives data from the PIC and forwards it to the crossbar. Both paths are designed to provide 16 Gb/s bandwidth per channel, resulting in an aggregate TRX bandwidth of 16 Tb/s in each direction.

The TX path consists of a 128-bit buffer, serializer, and level-shifting driver. Similarly, the RX path includes an offset DAC, analog front end (AFE), deserializer, and 128-bit buffer. The 128-bit buffers form a mesochronous interface for synchronization between the TRX and crossbar. The TRX/crossbar interface supports 32 bits per channel at 500 MHz and two test modes. The TX test mode provides a scan chain interface to the 128-bit buffer, and the RX test mode enables alternating 32-bit patterns in place of streaming data from the PIC.

Unlike prior work [KLO+24], our architecture: (1) uses a heater DAC with linear power output to enable simplified tuning; and (2) integrates active TX/RX re-calibration circuitry for long-term operation. By measuring the amplitude of the AFE signal, we create a control feedback loop to

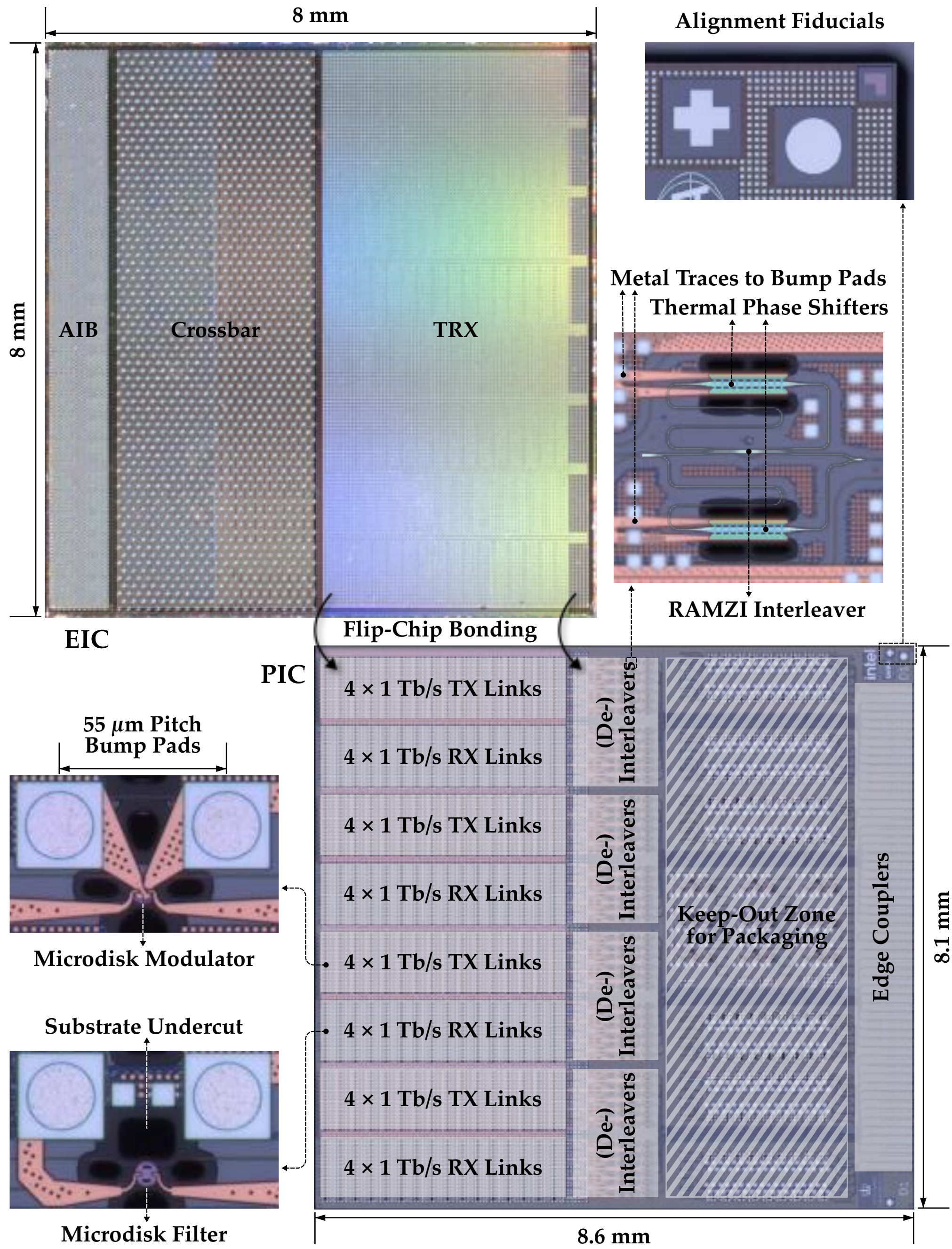


Figure 7.3: EIC and PIC Die Photos – PIC die photo adapted from [WWP+24].

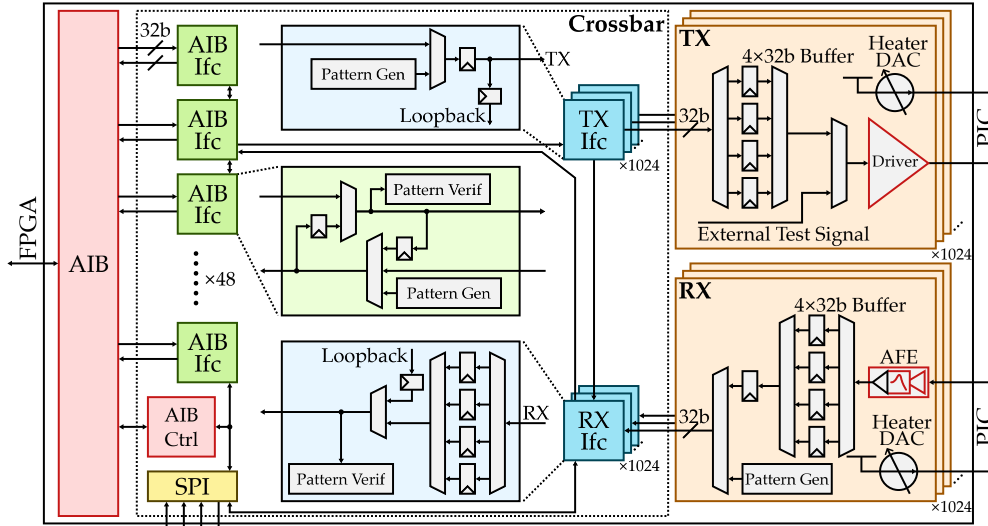


Figure 7.4: EIC Architecture Block Diagram

the heater DACs which keeps the modulators and filters at the appropriate resonance. In the case of the RX, the calibration also adjusts the offset DAC which keeps the received signal centered at mid-rail.

7.2.2 EIC Scaling Challenges

Although prior EIC implementations have demonstrated up to 100 optical channels [DLK+23, DRA+21, DRL+23, KLO+24, HKS+21, HKS+22, CKT+15, CSY+23], scaling to 1024 optical channels raises new physical design challenges with respect to routing and clocking.

1) Routing distance: Routing to the crossbar from 16 TRX cells in each TRX row was particularly challenging. The TRX is ≈4 mm wide resulting in the latency from the rightmost TRX cells exceeding the 2 ns clock period of the crossbar. Although only some cells violated timing, we inserted pipeline registers on all signals eight cells away from the signal's originating cell.

2) Routing congestion: Each TRX cell interface consists of 64 data bits and 7 control bits, resulting in 1,136 wires at the left edge of each 16-cell TRX row. This caused near-100% horizontal routing track usage and limited scalability of the row size. We used a "swizzle" layout with one signal buffered at a time and then shifted down one position in the bus. This style allowed for regular buffering of a small number of signals at a time due to limited space for downward vias, and it also allowed for a modular layout pattern. The crossbar then effectively needed to mux over tens of thousands of signals from the TRX to thousands of pins on the AIB. Given the large number

of signals, we used hierarchical design with AIB interface and TRX interface macros. Routing by abutment was used to simplify top-level routing and timing closure.

3) Clock distribution: Although the crossbar and the TRX digital interface could operate on the same clock, the TRX layout was too dense to allow for clock tree balancing across the array. We instead used mesochronous buffers at the TRX-crossbar interface [KPND20]. No handshake mechanism was needed because both sides operate at the same frequency and there is no back-pressure in the physical-level interface.

7.2.3 PIC Architecture

The PIC is co-designed with the EIC for flip-chip bonding. Each TRX cell modulates (TX) and detects (RX) a single wavelength; 64 wavelengths are wavelength-division multiplexed (WDM) onto a single optical link resulting in a total of 16 TX links and 16 RX links arranged into four groups on the PIC (see Figure 7.3). Figure 7.5 shows the link architecture based on [WNP+23, WWP+24]. At the TX side, we use ring-assisted Mach-Zehnder interferometer RAMZI-based interleavers to subdivide the incoming wavelength channels onto separate buses [WWM+24]. Data is modulated onto each wavelength by separate banks of cascaded microdisk modulators, driven by the EIC through the µbumps. The 64 modulated channels are then recombined into a single fiber output. At the RX side, a similar interleaving structure sends the wavelengths onto four buses of cascaded microdisk filters for sensing. The optical devices are designed to support each wavelength operating at 16 Gb/s, achieving an aggregate bandwidth of 1 Tb/s per fiber, and thus 16 Tb/s per PIC with a 2 Tb/s/mm shoreline bandwidth density.

7.2.4 PIC Scaling Challenges

Just as for the EIC, scaling to 1024 optical channels raises new PIC design challenges, including managing optical bandwidth, optical losses, process variations, and thermal control.

1) Optical bandwidth: Conventional single-bus link architectures struggle to accommodate massive WDM due to the limited free spectral range (FSR) of the microresonators. We adopt a multi-bus link architecture that de-interleaves WDM channels onto multiple buses, as proposed in [RDN+23a, JNR+23]. Since each stage of de-interleaving doubles the channel spacing, unwanted resonances are placed between channels with minimal crosstalk. This enables 64 WDM

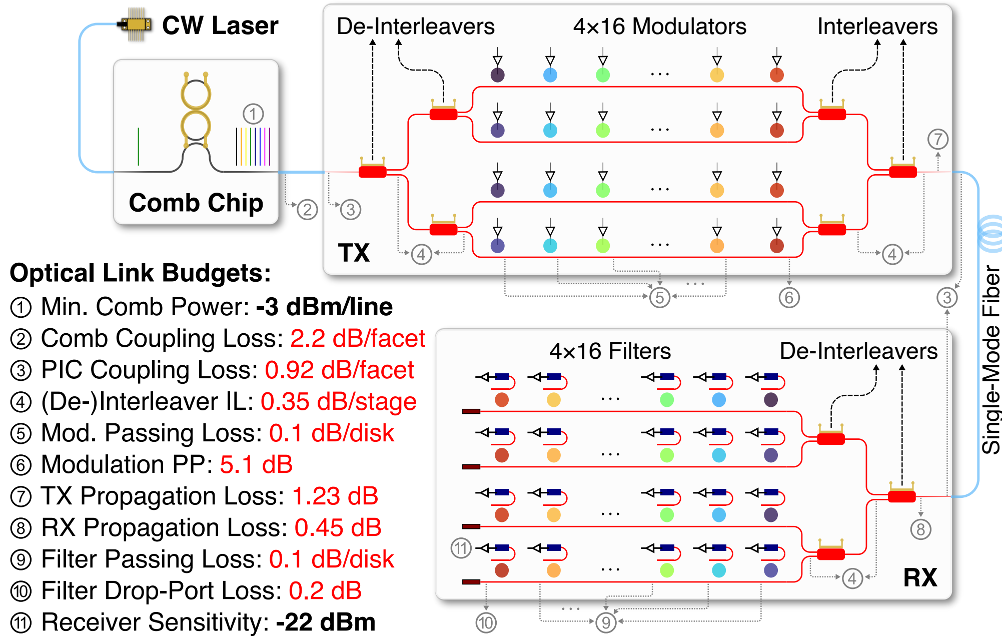


Figure 7.5: Photonic Link Architecture – Measured link budgets satisfying simulated RX sensitivity.

channels spaced at 100 GHz (spanning > 50 nm in C-band) with modulators and filters of a moderate 25.69 nm FSR.

2) Optical losses: Massive WDM scaling also reduces the optical power budget per channel. Silicon nonlinearities limit the total optical power per waveguide, and optical losses must be minimized to meet the target receiver sensitivity. We carefully optimize the number of interleaver stages to balance interleaver insertion loss vs. accumulated modulator/filter passing losses. We also adopt custom vertical-junction (VJ) microdisk modulators similar to [NWR+24]. The improved depletion response of VJ modulators compared to lateral-junction modulators allows using < 0.8 V CMOS-compatible voltage swings, while still achieving high extinction ratios and thus low power penalties [NJD+23]. Chip-fiber coupling loss can also be reduced to < 1 dB per facet by using optimized edge coupler designs, such as [DRL+23].

3) Process variations and thermal management: Thermal control is challenging in so many optical channels. On-chip thermal control is desirable, but faces limited area due to dense packaging requirements. Off-chip thermal control has more area available, but suffers from limited I/O count and bandwidth to the many devices requiring tuning. We adopt a fabrication-robust platform where wide waveguides are used in certain sections to reduce the sensitivity to process variations [RDN+23b], while maintaining single-mode operation with specially designed bend ge-

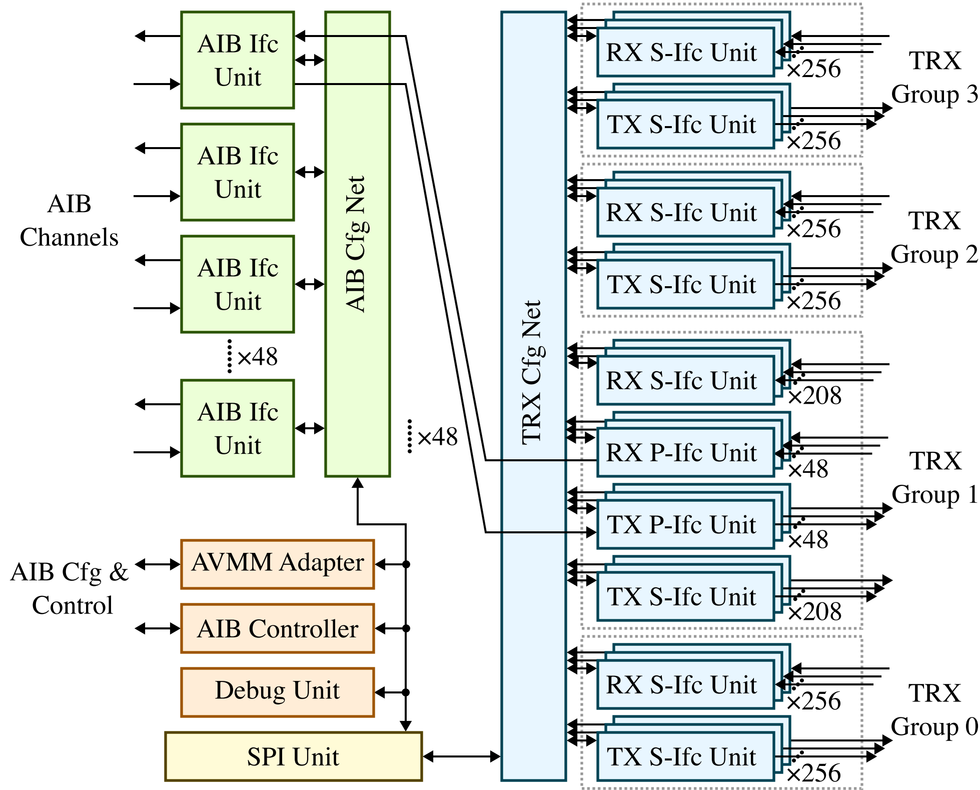


Figure 7.6: Crossbar Block Diagram

ometries. We also explore the use of substrate undercut around thermally tuned devices, as shown in Figure 7.3, which can improve thermal tuning efficiency by at least 5× [RDvN+23].

7.3 EIC Design, Implementation, and Verification

The EIC is a key component of the PIPES system, responsible for bridging the compute chiplet and the photonic interface chiplet. This section details the design, implementation, and verification of the EIC.

7.3.1 Crossbar Design

As is described in Section 7.2.1, the EIC is composed of three main components, AIB, crossbar, and TRX. The AIB is a commercial PHY IP. The TRX is designed by a group of students at Cornell using a mixed-signal design flow. I designed the crossbar and conducted the top-level integration of the three components. In addition to transferring data from AIB to TRX, the crossbar also hosts

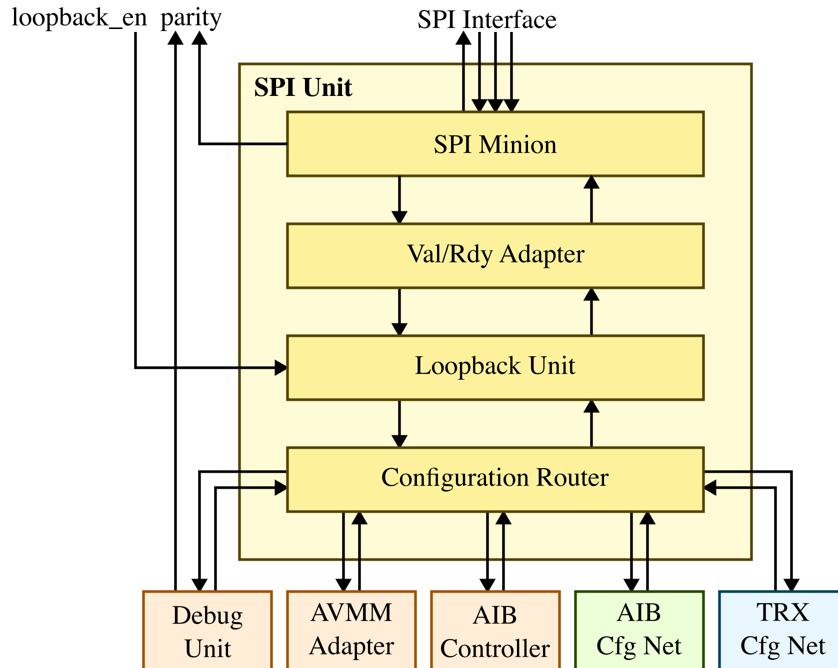


Figure 7.7: SPI Configuration Stack – The `loopback_en`, `parity`, and SPI interface signals are all external external I/O signals.

a number of control, configuration, and testing infrastructures. Figure 7.6 illustrates the crossbar design, which consists of an SPI unit, a configuration network, 48 AIB interface units, and 1024 TRX interface units, including 48 primary and 976 secondary TRX interface units.

SPI Unit – The EIC interfaces with the outside world through an SPI interface. I designed the SPI unit and implemented a custom protocol to handle SPI transactions. The SPI unit (minion) is driven by an external SPI driver (master). The SPI driver initiates a transaction by setting the chip select signal (`CS`) to 0 and ends it by setting it back to 1. During each transaction, the external SPI driver sends a request packet through the serial input (`MOSI`), synchronized with the positive edge of a clock signal(`SCLK`). In the meantime, the SPI driver samples the serial output (`MISO`) signal at the negative edge of `SCLK` to read the response packet from the SPI unit. Both the request and response packet shares the same format, including a `valid` bit to indicate if the current packet is valid, a `stall` bit to instruct the receiver to pause sending a valid packet for the next transaction, and a 54-bit payload that contains the configuration packet to or from the configuration network.

The SPI unit is structured as a stack of components (see Figure 7.7), including an SPI minion, a val/rdy adapter, a loopback unit, and a configuration router. The SPI minion interfaces with the external SPI driver. Instead of treating the `SCLK` signal as a conventional clock signal, which

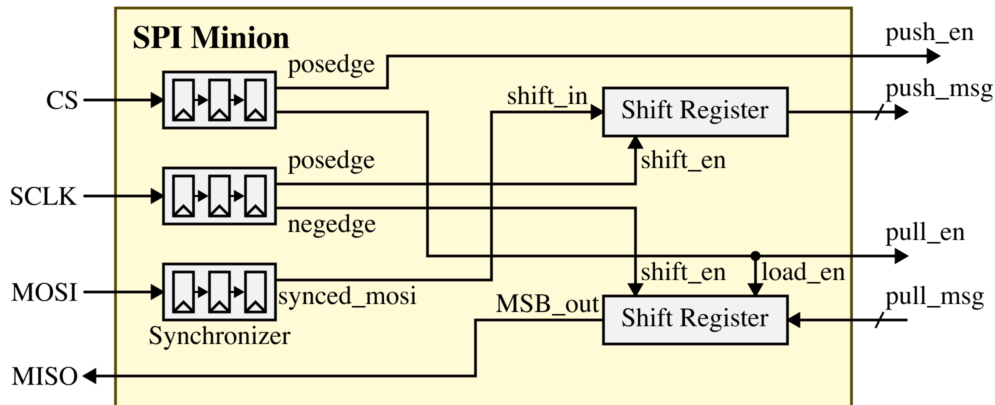


Figure 7.8: SPI Minion Diagram

would require managing clock domain crossing, the SPI minion treats all input signals (`CS`, `SCLK`, and `MOSI`) as data signals. Each of the input signals passes through a synchronizer which consists of three back-to-back flip-flops (see Figure 7.8), which effectively down-sampling them with the crossbar clock and detects rising and falling edges. The SPI minion then deserializes the input data and sends it to the val/rdy adapter. The val/rdy adapter converts the push/pull interface of the SPI minion into a val/rdy interface. It is also responsible for setting the valid and stall bit of the response packet based on the previous request packet and its remaining buffer space. The request packet is then passed to the loopback unit, which, when enabled by an external signal, loops the request packet back to the SPI minion. This is useful for testing if the SPI interface is operating correctly. Finally, the request packet arrives at the configuration router, which forwards the request packet to the appropriate destination based on its address. Possible destinations include the debug unit, the AVMM adapter, the AIB controller, the AIB interface unit configuration network, or the TRX interface unit configuration network based on the address of the request packet.

Configuration Network – The AIB interface units and TRX interface units are both connected to a configuration network. The configuration network is responsible for distributing configuration packets to the appropriate interface units based on the address of the packet. The configuration network also collects responses from these interface units and forwards them back to the SPI unit. The configuration network is implemented as a chain of configuration routers designed with PyOCN.

AIB Interface Unit – Figure 7.9 shows the datapath of the AIB interface unit. By default, the AIB Interface Unit forwards data directly from the AIB to the TRX. It includes several key

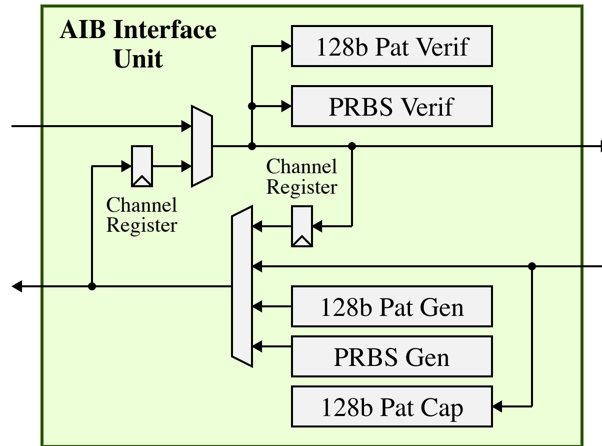


Figure 7.9: AIB Interface Unit Datapath Diagram – Pat Verif = pattern verifier, Pat Gen = pattern generator, Pat Cap = pattern capture unit.

components to support testing and verification, such as a 128-bit fixed pattern generator, a PRBS pattern generator, a 128-bit fixed pattern generator, a PRBS pattern verifier, and a 128-bit pattern capture unit to enable multiple test paths. The fixed pattern generator can be configured to produce any 128-bit pattern, while the PRBS generator can generate various pseudo-random patterns based on configurable seed values. The pattern verifiers compare incoming data against the expected patterns, with the results available for readout through the configuration network. The pattern capture unit captures received data, which can also be accessed via the configuration network. The AIB Interface Unit supports two loopback paths for testing purposes: one from the AIB input to the AIB output, and another from the TRX input to the TRX output. Channel registers on these loopback paths are accessible for readout, allowing for further verification and debugging.

TRX Interface Unit – There are four types of TRX interface units, including TX primary, TX secondary, RX primary, and RX secondary interface units. The 48 TX primary interface units and 48 RX primary interface units are connected to both the AIB interface units and the TRX, while the secondary interface units are only connected to the TRX. Figure 7.10 illustrates the datapath of these interface units. The TX primary interface unit includes a 64-bit fixed pattern generator and an interface test unit, which drives the testing interface of the TX and can shift out the 128-bit data in the TX buffer. The TX primary interface unit also includes a loopback path that goes to the RX primary interface unit. The RX primary interface unit includes a mesochronous buffer that synchronizes the incoming data with the local clock domain. It also includes a 64-bit pattern capture unit that can be accessed through the configuration network. The TX secondary units

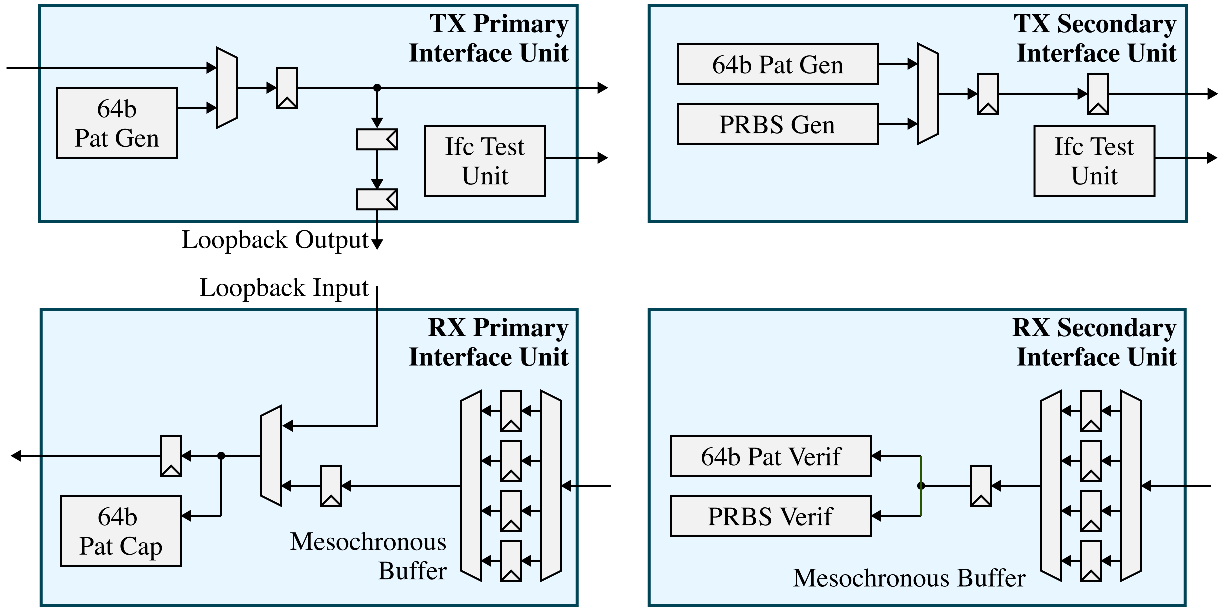


Figure 7.10: TRX Interface Unit Datapath Diagram

include a 64-bit fixed pattern generator, a PRBS pattern generator, and an interface test unit, while the RX secondary units include a mesochronous buffer, a 64-bit fixed pattern verifier, and a PRBS pattern verifier.

7.3.2 EIC Implementation

Figure 7.11 shows the ASIC flow of the PIPES tape-out. The AIB is a commercial IP that comes with both timing and physical models, allowing it to be integrated into the ASIC flow using standard steps. The TRX, however, is a mixed-signal block and it does not come with a timing model required by the synthesis tool, i.e., it only has a physical (layout) view but lacks a timing view. To overcome this, we co-designed the TRX with a standard-cell-based digital interface and I implemented a mixed-signal characterization flow to extract the timing model of the TRX macro. This process involved using custom scripts to process the TRX netlist and DEF file to remove the non-standard-cell components. The post-processed DEF file along with the TRX LEF file was then used in a parasitic extraction step to generate parasitic data (SPEF file) of the interface. Finally, The post-processed netlist and the SPEF file were then used to perform extracted timing model (ETM) to produce the timing model of the TRX digital interface, allowing it to integrate smoothly with the rest of the ASIC flow.

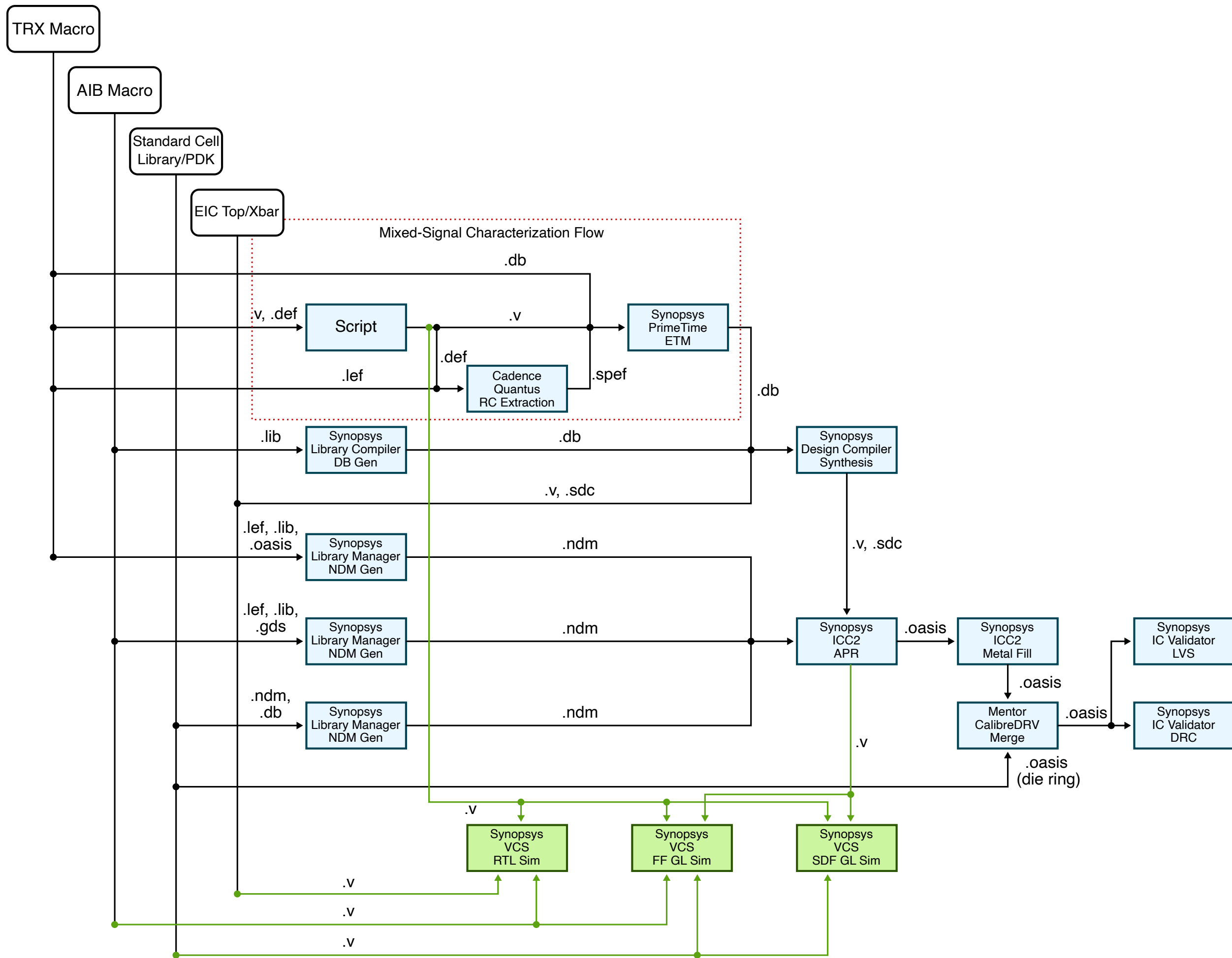
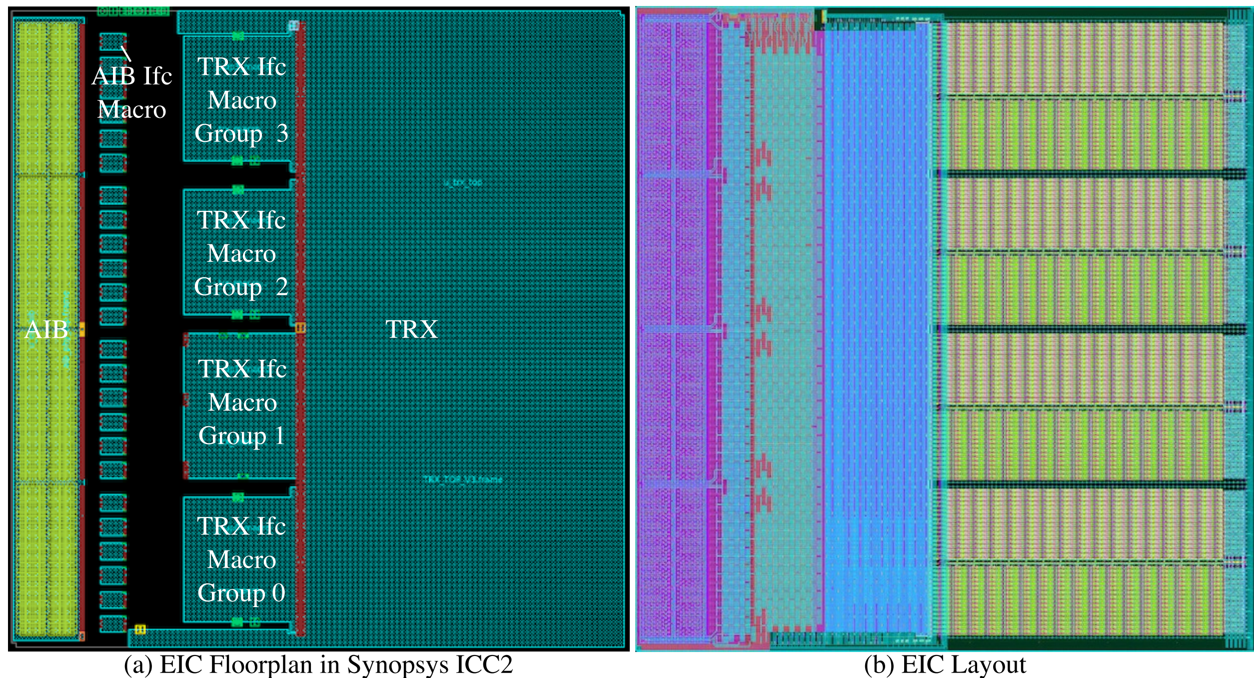


Figure 7.11: PIPES ASIC Flow

I implemented the RTL of the crossbar in PyMTL and integrated the generated Verilog code into the Verilog code of the EIC top-level module. I took a hierarchical design approach to implement the crossbar in the ASIC flow. The 48 AIB interface units, along with their configuration routers, were grouped into 24 AIB interface macro, each corresponding to an AIB channel. The 1024 TX and 1024 RX interface units are grouped into four TRX interface macros. Two types of TRX interface macro were created, one for group 0, 2, and 3 with just secondary interface units and one for group 1 with both primary and secondary interface units Figure 7.12(a) shows the EIC floorplan with these macros, while Figure 7.12(b) displays the final layout.

The EIC incorporates three types of bumps, including the AIB μbump, C4 bump, and TRX μbump. Since the AIB and TRX macros already include their respective bumps, I just needed to precisely overlap the top-level bumps with the AIB and TRX bumps to prevent the tool from attempting to perform any routing. The C4 bumps were mainly for power and crossbar I/Os and



(a) EIC Floorplan in Synopsys ICC2

(b) EIC Layout

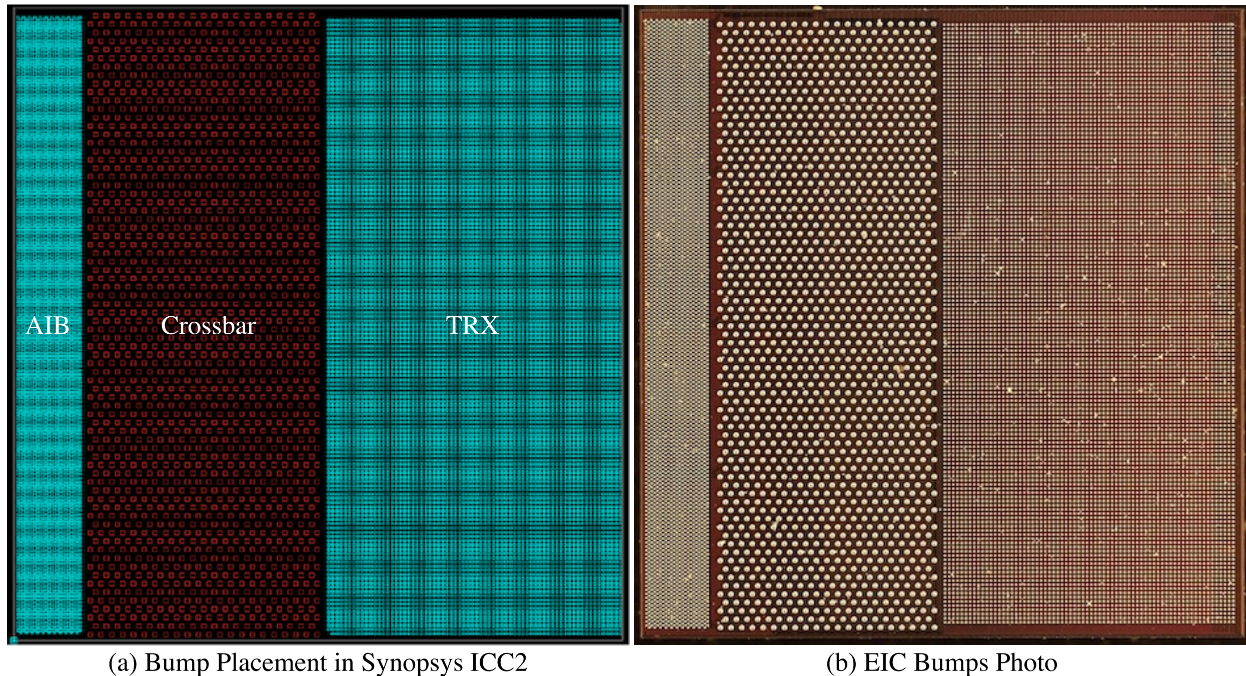
Figure 7.12: EIC Floorplan and Layout

required RDL routing. Figure 7.13 shows the bump placement of the EIC and an actual photo of the EIC bumps.

7.3.3 Pre-Silicon Verification

The pre-silicon testbench for the EIC included an RTL model of the FPGA AIB, an RTL model of the EMIB, and the EIC top-level module, which integrated RTL models of the AIB and crossbar along with a gate-level model of the TRX. Interaction with the EIC model was achieved through direct toggling of I/O signals, sending requests over the SPI interface, or sending data from the FPGA AIB. Various test suites, including unit tests and integration tests, were implemented to verify different data paths and functionalities within the EIC. These tests are detailed as follows.

- **Crossbar Bring-Up Tests –** This test suite verified the most basic functionality of the crossbar, ensuring correct clock injection, proper operation of the SPI interface, and successful read/write access to all configuration registers.
- **AIB Bring-Up Tests –** This test suite verified if the AIB can be properly configured and started up by the AVMM adapter and the AIB controller in the crossbar.



(a) Bump Placement in Synopsys ICC2 (b) EIC Bumps Photo

Figure 7.13: EIC Bumps – The photo of EIC is taken at Cornell NanoScale Facility.

- **Crossbar AIB Interface Unit Tests –** This test suite tested the AIB interface units in isolation by configuring each unit to send data from its pattern generator to its own pattern verifier.
- **Crossbar TRX Primary Interface Unit Tests –** This test suite verified the TX and RX primary interface units by enabling the loopback path, sending data from the pattern generator, and checking the channel registers and pattern capture unit.
- **Crossbar AIB Interface to TRX Interface Integration Tests –** This test suite verified the data paths between the AIB interface units and the TRX interface units. It included sending data from an AIB interface unit to a TX primary interface unit, and vice versa, with data looping back to a RX primary interface unit.
- **Crossbar TRX Interface Integration Tests –** This test suite verified the integration of the TRX interface units and the TRX digital interface. For the TX path, data was sent into the TRX using pattern generators and then shifted out through the interface test unit. For the RX path, RX test mode was enabled, and data capture was verified through the pattern capture unit of the RX interface unit.

- **AIB Crossbar Integration Tests –** This test suite verified the integration of the AIB and the crossbar by testing fixed and PRBS patterns sent from the FPGA AIB to the EIC, checking the pattern verifier in the AIB interface unit, and testing data transfers from the AIB interface to the FPGA AIB. Loopback tests were also included, where data was either looped back in the AIB interface unit or on the FPGA AIB side.
- **FPGA to TRX Interface Integration Tests –** This test suite verified data paths between the FPGA and TRX interfaces in both directions. For the TX path, data was sent from the FPGA AIB to the TRX and shifted out through the interface test unit. For the RX path, RX test mode was enabled, and the pattern was checked at the FPGA AIB.
- **Crossbar TRX Integration tests –** This test suite further verified the integration of the crossbar with the TRX by configuring the TRX for digital or analog loopback, sending data from the crossbar to the TRX, and checking that the crossbar received the expected pattern.
- **Full System Tests –** This test suite verified the end-to-end datapath of the EIC by sending data from the FPGA AIB and checking the TRX output, or sending data from the FPGA AIB, looping back the TRX output, and checking if the same pattern was received by the FPGA AIB.

Many of these test cases involved sending SPI requests to the crossbar to configure certain functionalities or to read out the status of the interface units. The testbench supported dumping out the trace of the SPI transactions, which could be reused to replay the test cases in the post-silicon verification stage. The EIC passed all the pre-silicon test suites before tape-out.

7.3.4 Post-Silicon Verification

For post-silicon verification of the EIC, I used a commercial SPI driver to drive the SPI interface of the EIC. To facilitate testing, I developed a frontend program capable of reading the SPI transaction trace from pre-silicon verification and replaying these transactions on the post-silicon EIC. This frontend also included a library of utility functions for reading and writing configuration registers within the crossbar, enabling direct development of new test cases specifically for post-silicon verification.

Apart from unit testing the crossbar, I worked with the TRX team to conduct integration tests. Data patterns generated by the crossbar were observed at the TX test point, confirming correct data

flow. Additionally, I worked with the PIC team to conduct integration tests between the EIC and PIC, where data patterns generated by the crossbar were successfully observed on the PIC side.

7.4 Link-Level Evaluation

We evaluate the opto-electrical links using simulation-based and experimental results to demonstrate the potential for hybrid 2.5D/3D integration to enable co-packaged optical interconnects.

7.4.1 Simulation-Based Evaluation

We use bare-die measurement of photonic devices to characterize the optical link losses, parasitic resistance/capacitance of the optical modulator and photodetector, drive voltage of the photodetector, and thermal characteristics of the heaters. We build compact models of the PIC devices and use transistor-level modeling of the TRX on the EIC to characterize the EIC to PIC bandwidth, latency, and energy. We build register-transfer-level (RTL) behaviorial models of the TRX and use RTL modeling of the AIB and crossbar along with a target FPGA design to characterize the FPGA to EIC bandwidth, latency, and energy.

Our end-to-end simulations, from the FPGA to the TRX microbumps, validate that the FPGA can sustain 768 Gb/s over the AIB, crossbar, TRX, and PIC by using 48 of the optical channels each operating at 16 Gb/s. This is below the peak AIB 1.0 bandwidth since two 32-bit EIC channels are mapped to each 80-bit AIB channel. Our simulations also validate that the EIC can sustain 32 Gb/s bi-directional bandwidth over the crossbar, TRX, and PIC by generating PRBS traffic on-chip for every TRX channel. Finally, our device-level characterization validates that we can meet the optical power limits and receiver sensitivity target (-22 dBm for 1E-12 bit error rate at 16 Gb/s). Our simulations show an end-to-end latency of 39 ns from the FPGA to PIC. Our simulation-based energy analysis suggests it should be possible to achieve sub-1 pJ/b including the TRX and PIC with another 1 pJ/b for the AIB and crossbar.

7.4.2 Experimental Evaluation

Figure 7.14 illustrates our experimental setup. The optical path has a tunable laser source followed by a thulium-doped fiber amplifier (TDFA) and a polarization controller (PC) before en-

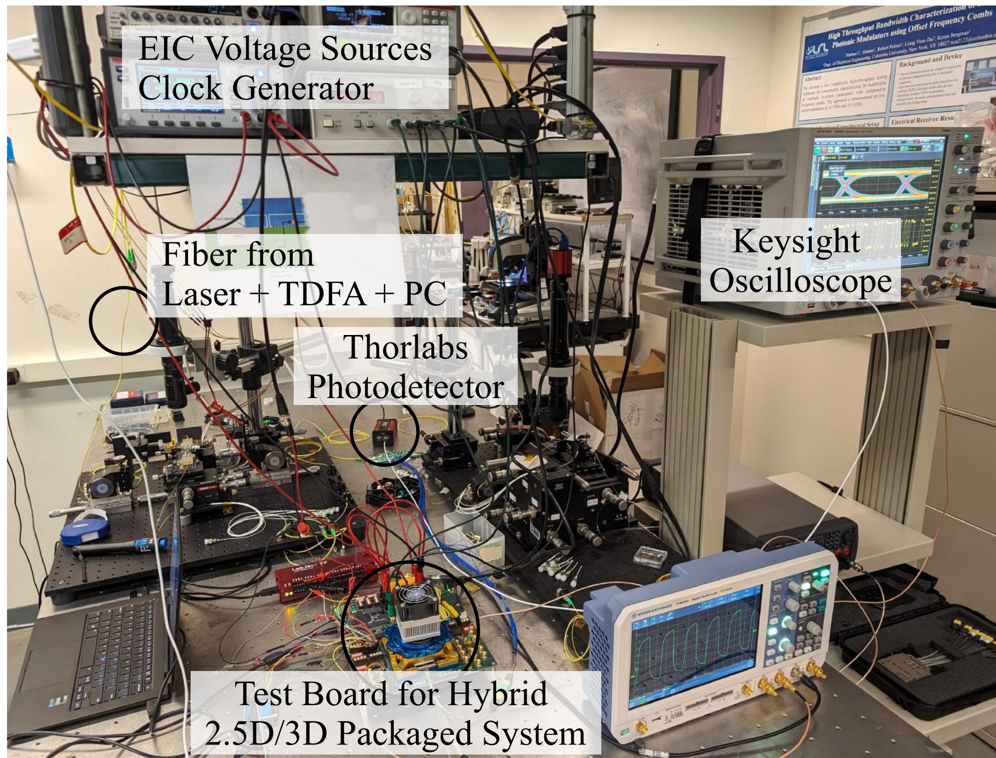


Figure 7.14: Experimental Setup

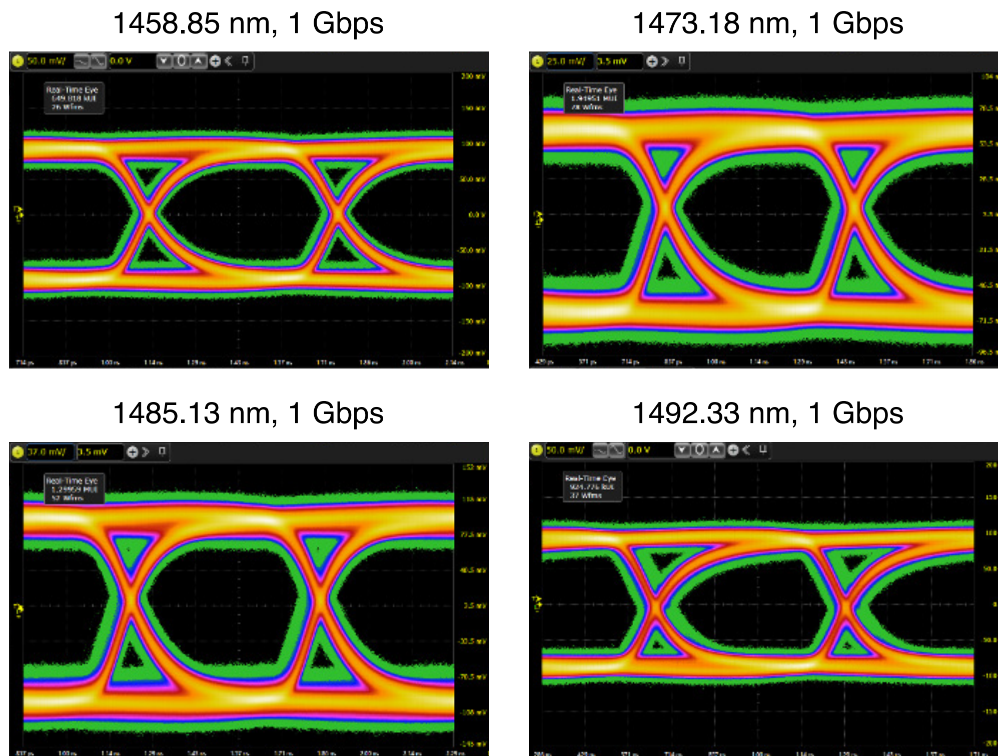


Figure 7.15: Eye Diagrams for Opto-Electrical EIC/PIC Channel

tering the PIC. The modulated optical carrier from the PIC is converted to an electrical signal by a Thorlabs photodetector and inspected by a Keysight oscilloscope. The EIC is configured through an SPI controller to generate a 128-bit repeating fixed pattern or pseudo-random pattern. The 128-bit pattern is sent from the crossbar through the TRX to the PIC for modulation. We successfully demonstrated the FPGA functioning in isolation and demonstrated multiple complete opto-electrical channels from the EIC through the PIC each running at 1 Gb/s (see Figure 7.15). However, several implementation oversights prevented demonstrating the system's full capabilities. While each optical channel is functional, we were unable to include delay-locked loops in the EIC meaning the RX clock to data skew is unknown. A timing analysis bug prevented the crossbar from running at the target 500 MHz, and insufficient power routing led to unreliable operation of several AIB interface units.

7.5 System-Level Evaluation

To evaluate the system-level performance benefits of PIPES optical links for LLM training, I used the LLMCompass-E2E framework to model and simulate various training configurations. Specifically, I incorporated a model of the PIPES optical links, based on the link-level evaluation results in 7.4.1, and compared it to baseline electrical interconnects. This evaluation was conducted using the same A100 model as in Chapter 6, with two LLM sizes: a 175-billion-parameter model and a 1-trillion-parameter model. For each model, I simulated all possible training configurations across 4096 GPUs, sweeping through different degrees of tensor, pipeline, and data parallelism. Figure 7.16 shows the execution time breakdown of the optimal LLM training mappings for various A100 systems with different interconnects and memory systems at 4096 GPUs. The configurations are denoted as either using the baseline electrically connected HBM memory (E-Mem) or the optically connected HBM memory (O-Mem) with either baseline electrical NVLink (E-Link) or PIPES optical link (O-Link). When simulating with the PIPES optical links, I assume more GPUs can be interconnected within a node (up to 64) due to its superior link reach, meaning that it can support a larger degree of tensor parallelism. For each setup, the normalized execution time is broken down into forward pass computation, backward pass computation, and the overheads from tensor parallelism, pipeline parallelism, and data parallelism.

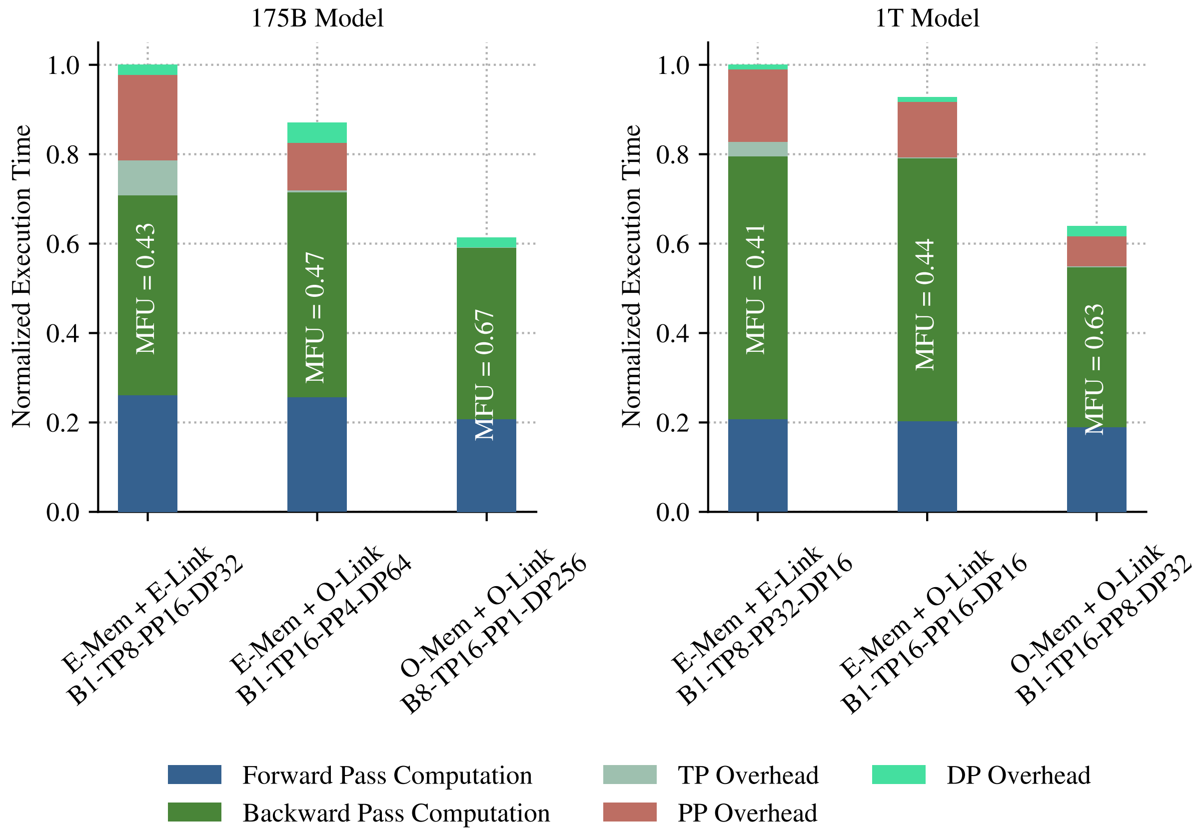


Figure 7.16: Optimal Mappings for LLM Training – The plot shows the execution time breakdown of the LLM Training mappings with the best MFU at 4096 GPUs for various A100 systems. The same 175B and 1T model as is described in Section 6.4 are explored. E-Mem = Baseline electrically connected HBM memory system, O-Mem = optically connected HBM memory system as described in Chapter 6, E-Link = baseline electrical NVLinks, O-Link = PIPES optical links. B = µbatch size, TP = degree of tensor parallelism, PP = degree of pipeline parallelism, DP = degree of data parallelism.

For the 175-billion-parameter model, adopting the PIPES optical link alone can improve the overall training performance by 1.09×. The benefit partially comes from reduced TP overhead, as PIPES optical link offers higher bandwidth compared to NVLink (2 TB/s vs. 300 GB/s). The PIPES optical link also enables a different mapping from the baseline with smaller pipeline parallelism, which reduces the pipeline parallelism overhead. Combining the PIPES optical link with the optically connected HBM memory system further improves the performance by 1.47×. The combination leads to a completely different mapping, with no pipeline parallelism and a larger microbatch size of 8. The larger batch size makes the computation of the forward and backward passes more efficient as it enables more reuse of the model parameters. Additionally, the extended processing time for each microbatch enables better overlap between data parallelism communica-

tion and backward pass computation. As a result, only the all-reduce operation for the last layer remains exposed, effectively minimizing data parallelism overhead.

For the 1-trillion parameter model, adopting the PIPES optical link alone can improve the overall training performance by 1.07×. The benefit mainly comes from reduced tensor parallelism overhead and reduced pipeline parallelism overhead due to a shorter pipeline. Combining the PIPES optical link with the optically connected HBM memory system further improves the performance by 1.43×. This improvement is mainly driven by reduced model parallelism overhead and the ability to avoid activation recomputation, which significantly shortens the execution time of the backward pass.

While adopting the PIPES optical link alone significantly reduces the all-reduce latency for tensor parallelism, the overall performance gain is largely limited by the overheads from pipeline parallelism and activation recomputation. Although I allow the system with PIPES optical link to use a larger degree of tensor parallelism, the optimal mapping tends not to use a large degree of tensor parallelism. This is because a large degree of tensor parallelism can lead to suboptimal partitioning of the matrix-matrix multiplication, reducing the compute efficiency. The enhanced scalability of the PIPES optical link, however, could be particularly advantageous for training mixture-of-experts (MoE) models, which require frequent all-reduce communication across different tensor-parallel groups. PIPES optical links can allow multiple tensor-parallel groups to fit in a single node and offer high bandwidth all-to-all communication between them. Future work will explore the performance of MoE models with PIPES optical links to further demonstrate the potential of co-packaged optical interconnect in the scaling LLM training.

7.6 Conclusion

In this chapter, we presented a novel system that uses hybrid 2.5D/3D integration to compose a state-of-the-art FGPA compute chiplet, three electrical interface chiplets, and three photonic interface chiplets. Our simulation-based evaluation demonstrates the potential for this system to achieve 96 Tb/s of bi-directional optical bandwidth, and our experimental demonstration functionally validates key components including a complete opto-electrical channel. While implementation oversights prevented our experimental system from demonstrating the target system-level bandwidth,

this work still shows the potential of hybrid 2.5D/3D integration and serves as an important next step towards scaling co-packaged optical interconnects for system-level communication.

CHAPTER 8
CONCLUSION

Focusing on scaling on-chip interconnects for manycore architecture and scaling off-chip interconnects for distributed LLM workloads, this thesis presents modeling frameworks, proposes innovative architectures, and validates these solutions through silicon prototypes. In this chapter, I summarize my thesis contributions and discuss possible future research directions.

8.1 Thesis Summary and Contributions

This thesis begins by presenting the methodology challenges and architecture challenges in scaling on- and off-chip interconnects. Since the looming end of Moore's Law and the end of Dennard scaling, modern computing systems must embrace parallelism, both within a single chip and across interconnected devices, to meet the growing computational requirements. The need for efficient data movement, both on-chip and off-chip, form the foundation of the two parts of this thesis. Scaling on-chip interconnects and scaling off-chip interconnects each presents unique challenges in both methodology and architecture.

Part I of this thesis focuses on scaling on-chip interconnects to support the growing parallelism in manycore architectures. First, it introduces PyOCN, a unified framework for modeling, testing, and evaluating on-chip networks. PyOCN enables rapid design-space exploration by providing a flexible, Python-based platform that supports multiple levels of abstraction. Next, I propose practical low-diameter OCN topologies that can be effectively implemented with a tiled physical design methodology, bridging the gap between theoretical advances and practical implementation. Finally, the CIFER chip tape-out demonstrates the real-world application of PyOCN.

Part II of this thesis shifts focus to off-chip interconnects, including memory interconnects that connect compute chip with external memory, and system interconnects that connect multiple compute devices. I first present LLMCompass-E2E, a comprehensive performance evaluation framework for distributed LLM training that captures the complex interactions between different hardware components as well as parallelization strategies. I then leverage LLMCompass-E2E to explore the potential of co-packaged silicon photonic interconnects in both memory and system interconnects. For memory interconnect, I propose an optically connected multi-stack HBM module to expand the memory capacity and bandwidth of state-of-the-art HBM-based memory systems.

For system interconnect, I present the PIPES silicon photonic tape-out, a practical demonstration of co-packaged optical interconnects that connects different compute devices. Through evaluation with LLMCompass-E2E, I show that the co-packaged optical interconnects can significantly improve the overall performance and efficiency of large-scale LLM training.

To reiterate the major contributions of this thesis:

- I developed PyOCN, a unified Python-based framework for modeling, testing, and evaluating on-chip networks that enables rapid design-space exploration.
- I proposed and evaluated practical OCN topologies that reduce the network diameter while remaining practical to implement using a tiled physical design methodology.
- I demonstrated the benefits of PyOCN through the CIFER chip tape-out, highlighting practical design trade-offs and optimizations.
- I developed LLMCompass-E2E, a performance evaluation framework for large-scale distributed training workloads.
- I proposed and evaluated the use of co-packaged silicon photonic interconnect to scale the memory capacity and bandwidth of HBM-based memory systems, which are essential for LLM workloads.
- I demonstrated a practical implementation of the proposed co-packaged silicon photonic interconnect through the PIPES tape-out.

8.2 Future Work

This thesis addresses key challenges in scaling on-chip and off-chip interconnects, but several open directions remain for future exploration. In this section, I outline several potential future research directions inspired by the work presented in this thesis.

8.2.1 Testing Methodology for On-Chip Networks

OCNs typically contain a massive number of stateful components, such as input and output buffers, virtual channels, arbiters, credit counters, etc. The stateful nature of these components

makes it challenging to thoroughly test the networks and trigger corner-case bugs, such as deadlocks and livelocks. While Chapter 2 demonstrates successful integration of PyOCN with Hypothesis, a property-based testing framework, the testing approach is based on generating random input packets and traffic patterns. Triggering bugs like deadlocks may require a combination of a very specific sequence of packets and back pressure pattern, which may not be easily generated by random testing. One possible approach to address this challenge is to develop a testing methodology tailored specifically for OCNs. Instead of generating random input packets and back pressure patterns, the testing methodology can directly generate a valid architecture state of the network, and then apply a sequence of operations to trigger corner-case bugs. For example, it can generate a state where all buffers are full and then apply a sequence of packets. This may be more likely to trigger deep stateful bugs like deadlocks. The testing methodology can also be integrated with Hypothesis to enable automated test case reduction, so that when a bug is detected, the testing methodology can automatically reduce the test case to a minimal set of operations with a simpler state that triggers the bug. Such a testing methodology for OCNs can be more effective in verifying OCN designs and reduce the time spent on debugging and verification.

8.2.2 LLMCompass-E2E

In Chapter 5, I present LLMCompass-E2E, a performance evaluation framework for distributed LLM training workloads. This section outlines several potential future research directions to further improve the framework.

Developing or Combining Detailed Network Simulator – The current LLMCompass-E2E framework uses simplified first-order network models calibrated with empirical hardware results and to estimate the communication overheads in distributed LLM training. It also makes assumptions about the network topologies. To make the framework more flexible and accurate, a more detailed network simulator can help better model the network latency, congestion, and other network dynamics across various configurations and topologies. One can leverage the event-driven scheduler simulator in LLMCompass-E2E to develop a detailed network simulator from scratch or combine it with existing network simulators like ASTRA-sim [RSSK20]. A detailed network simulator could more accurately capture the communication overhead in LLM training and enable the exploration of custom network topologies.

Seamless PyTorch Integration for LLMCompass-E2E – While LLMCompass-E2E provides a PyTorch-like interface for defining LLMs, users still need to convert their actual model to the LLMCompass model. To make LLMCompass-E2E more user-friendly, a seamless PyTorch integration can be developed. Instead of using our own computational graph intermediate representation, we can directly leverage the computational graph in PyTorch so that LLMCompass-E2E can just take a real LLM and estimate its performance. This may require re-engineering the kernel-level performance model to directly interface with PyTorch's computational graph to extract the necessary information for performance modeling. Furthermore, we can even interface the PyTorch Distributed library (`torch.distributed`) with our simulator to directly simulate the distributed training of LLMs in LLMCompass-E2E. Such seamless PyTorch integration will help LLMCompass-E2E keep up with the evolving model architectures and optimization techniques in the LLM domain.

8.2.3 Co-Packaged Optics for Memory and System Interconnects

In Chapter 6, I propose the optically connected multi-stack HBM module, which leverages co-packaged optics for memory interconnect, as a potential solution to close the memory capacity gap. In Chapter 7, I explore leveraging co-packaged optics for system interconnects to improve communication efficiency in distributed training. This section outlines several potential future research directions to further explore the potential of co-packaged optics for memory and system interconnects.

Accelerators in EIC – In Chapter 6 and Chapter 7, the EIC only contains communication components such as the UCIe PHY and the electrical transceiver arrays. However, our tape-out experience in Chapter 7 suggests that there are still plenty of area to use in the EIC, and that the EIC can be further extended to include accelerators that can offload computation from the host compute chip. For example, the EIC can include a reduction engine that can be used to accelerate all-reduce operations in distributed training. This can further increase the effective bandwidth of the optical links.

Alternative Optically Connected Multi-Stack HBM Module Design – In the proposed optically connected multi-stack HBM module, an I/O chiplet is used to interface the HBM module with the host compute chip. The memory requests would come into the I/O chiplet through the EIC, and the I/O chiplet would need to route the requests electrically to the appropriate HBM PHY. For an

aggressive I/O chiplet connecting many HBM stacks, this may require a large crossbar connecting the UCIe PHYs to the HBM PHYs and results in the I/O chiplet becoming a bandwidth and energy bottleneck. One potential future work is to explore scalable and energy-efficient designs for the I/O chiplet. Alternatively, we can explore directly integrating HBM stacks with PIC by changing the logic layer in the HBM stack to incorporate electrical transceivers. This would eliminate the need for the I/O chiplet but would require an optical bus and a more convoluted PIC design.

HBM Memory Pool – In the memory system proposed in Chapter 6, each compute chip has its own HBM stacks. Communicating with other compute chips would require going through the system interconnect. One potential future work is to explore a shared HBM memory pool that is connected to many compute chips. Such a shared memory architecture could enable more flexible and efficient memory utilization across different compute chips. Key research challenges include designing a low-latency, high-bandwidth interconnect to connect the shared HBM pool with compute chips. Novel opto-electrical devices can be explored, such as the bi-modal wavelength-selective switch that can enable either one-to-one or all-to-all communication patterns, which can be a great fit for LLM workloads.

Evaluating Co-Packaged Optical Interconnect for Mixture of Experts LLMs – Mixture of Experts (MoE) architectures have become increasingly popular in the domain of LLMs due to its ability to efficiently handle an enormous number of parameters. MoE models adopt a selective routing mechanism that activates only a subset of specialized feed-forward networks, i.e. ”experts”, for each input. However, the use of MoE introduces additional communication overheads in both training and inference. Distributed MoE models require efficient interconnects to handle frequent data exchanges between experts located on separate devices or nodes. This makes scalable, high-performance interconnect solutions, such as the PIPES optical links presented in Chapter 7, especially relevant for future MoE-based LLMs, as they provide the necessary scalability and communication efficiency. Future work could evaluate the performance of MoE models with co-packaged optical system interconnects to further demonstrate how optical interconnects can improve the training and inference efficiency of MoE models. By offering a more scalable and efficient communication infrastructure, co-packaged optical interconnects could make the training of large-scale MoE models more efficient. LLMcompass-E2E could be leveraged to simulate the training of various MoE configurations across hardware systems with different memory systems and interconnects. Such evaluations could show the potential of co-packaged optical intercon-

nects as a cornerstone technology for MoE architectures, paving the way for more scalable and resource-efficient AI systems.

New Deep Learning Model Architecture – Another potential research direction is to explore new deep learning model architectures that can be enabled by proposed high-bandwidth and high-capacity memory systems. Traditional model architectures are often constrained by the limitations of current memory technologies, such as limited bandwidth, capacity, or scalability. By addressing these bottlenecks, the proposed memory systems open the door to designing models with larger parameter sizes, deeper layers, or more intricate interconnections, pushing the boundaries of what is computationally feasible.

BIBLIOGRAPHY

[ADL+13] N. Abeyratne, R. Das, Q. Li, K. Sewell, B. Giridhar, R. G. Dreslinksi, D. Blaauw, and T. Mudge. Scaling Towards Kilo-Core Processors with Asymmetric High-Radix Topologies. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2013.

[AKPJ09] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2009.

[AM03] S. N. Adya and I. L. Markov. Fixed-outline floorplanning: enabling hierarchical design. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, Dec 2003.

[APM+12] P. Abad, P. Prieto, L. G. Menezo, A. Colaso, V. Puente, and J.-Á. Gregorio. Topaz: An Open-Source Interconnection Network Simulator for Chip Multiprocessors and Supercomputers. *Int'l Symp. on Networks-on-Chip (NOCS)*, May 2012.

[BB04] D. Bertozzi and L. Benini. Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip. *IEEE Circuits and Systems Magazine*, Sep 2004.

[BBB+11] N. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 Simulator. *SIGARCH Computer Architecture News (CAN)*, 39(2):1–7, Aug 2011.

[BD06] J. Balfour and W. Dally. Design Tradeoffs for Tiled CMP On-Chip Networks. *Int'l Symp. on Supercomputing (ICS)*, Jun 2006.

[BEA+08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 Processor: A 64-Core SoC with Mesh Interconnect. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2008.

[BHY+18] M. Besta, S. M. Hassan, S. Yalamanchili, R. Ausavarungnirun, O. Mutlu, and T. Hoefler. Slim NoC: A Low-Diameter On-Chip Network Topology for High Energy Efficiency and Scalability. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2018.

[BJM+05] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. D. Micheli. NoC Synthesis Flow for Customized Domain Specific Multiprocessor Systems-on-Chip. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 16(2):113–129, 2005.

[BJO+09] C. Batten, A. Joshi, J. S. Orcutt, A. Khilo, B. Moss, C. W. Holzwarth, M. A. Popović, H. Li, H. I. Smith, J. L. Hoyt, F. X. Kärtner, R. J. Ram, V. Stojanović,

and K. Asanović. Building Manycore Processor-to-DRAM Networks with Monolithic CMOS Silicon Photonics. *IEEE Micro*, 29(4):8–21, Jul/Aug 2009.

[BLS+20] J. Balkind, K. Lim, M. Schaffner, F. Gao, G. Chirkov, A. Li, A. Lavrov, T. M. Nguyen, Y. Fu, F. Zaruba, et al. BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2020.

[BMF+16] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff. OpenPiton: An Open Source Manycore Research Framework. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2016.

[BMR+20] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems (NeurIPS)*, Dec 2020.

[Bol12] J. Bolaria. Xeon Phi Targets Supercomputers. *Microprocessor Report, The Linley Group*, Sep 2012.

[BSK+10] S. Beamer, C. Sun, Y.-J. Kwon, A. Joshi, C. Batten, V. Stojanović, and K. Asanović. Re-Architecting DRAM Memory Systems with Monolithically Integrated Silicon Photonics. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2010.

[BSP+17] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M. Baas. KiloCore: A 32-nm 1000-Processor Computational Array. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(4):891–902, Apr 2017.

[BZL+12] J. F. Buckwalter, X. Zheng, G. Li, K. Raj, and A. V. Krishnamoorthy. A Monolithic 25-Gb/s Transceiver With Photonic Ring Modulators and Ge Detectors in a 130-nm CMOS SOI Process. *IEEE Journal of Solid-State Circuits (JSSC)*, 47(6):1309–1322, Apr 2012.

[CBG+18] Q. Cheng, M. Bahadori, M. Glick, S. Rumley, and K. Bergman. Recent Advances in Optical Technologies for Data Centers: A Review. *Optica*, 5(11):1354–1370, Oct 2018.

[CDS+14] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2014.

[CH11] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *ACM SIGPLAN Notices*, 46(4):53–64, 2011.

[CHB+10] J. Chan, G. Hendry, A. Biberman, K. Bergman, and L. P. Carloni. Phoenixsim: A Simulator for Physical-Layer Analysis of Chip-Scale Photonic Interconnection Networks. *Design, Automation, and Test in Europe (DATE)*, Mar 2010.

[CKES16] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2016.

[CKT+15] Y. Chen, M. Kibune, A. Toda, A. Hayakawa, T. Akiyama, S. Sekiguchi, H. Ebe, N. Imaizumi, T. Akahoshi, S. Akiyama, S. Tanaka, T. Simoyama, K. Morito, T. Yamamoto, T. Mori, Y. Koyanagi, and H. Tamura. A 25Gb/s Hybrid Integrated Silicon Photonic Transceiver in 28nm CMOS and SOI. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2015.

[CLG+23] T.-J. Chang, A. Li, F. Gao, T. Ta, G. Tziantzioulis, Y. Ou, M. Wang, J. Tu, K. Xu, P. J. Jackson, A. Ning, G. Chirkov, M. Orenes-Vera, S. Agwa, X. Yan, E. Tang, J. Balkind, C. Batten, and D. Wentzlaff. CIFER: A 12nm, 16mm2, 22-Core SoC with a 1541 LUT6/mm2 1.92 MOPS/LUT, Fully Synthesizable, CacheCoherent, Embedded FPGA. *Custom Integrated Circuits Conf. (CICC)*, Apr 2023.

[CMM+05] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti. Noxim: An Open, Extensible and Cycle-Accurate Network on Chip Simulator. *Int'l Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*, Jul 2005.

[CND+24] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, et al. PaLM: scaling language modeling with pathways. *ACM Journal of Machine Learning Research (JMLR)*, Mar 2024.

[cor19] CoreLink Interconnect. Online Webpage, accessed Sep 20, 2019. Https://developer.arm.com/ip-products/system-ip/corelink-interconnect.

[CP04] J. Chan and S. Parameswaran. NoCGEN: A Template based Reuse Methodology for Networks on Chip Architecture. *Int'l Conf. on VLSI Design*, Jan 2004.

[CPK+13] C.-H. O. Chen, S. Park, T. Krishna, S. Subramanian, A. P. Chandrakasan, and L.-S. Peh. SMART: A Single-Cycle Reconfigurable NoC for SoC Applications. *Design, Automation, and Test in Europe (DATE)*, Mar 2013.

[CSY+23] P.-H. Chang, A. Samanta, P. Yan, M. Fu, Y. Zhang, M. B. On, A. Kumar, H. Kang, I.-M. Yi, D. Annabattuni, D. Scott, R. Patti, Y.-H. Fan, Y. Zhu, S. Palermo, and S. J. B. Yoo. A 3D Integrated Energy-Efficient Transceiver Realized by Direct Bond Interconnect of Co-Designed 12nm FinFET and Silicon Photonic Integrated Circuits. *Journal of Lightwave Technology*, 41(21), Nov 2023.

[Dal91] W. J. Dally. Express Cubes: Improving the Performance of k-ary n-cube Interconnection Networks. *IEEE Trans. on Computers (TOC)*, 40(9):1016–1023, Sep 1991.

[DCLT19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *North American Chapter of the Association for Computational Linguistics (NAACL)*, Jun 2019.

[DLK+23] S. Daudlin, S. Lee, D. Kilwani, C. Ou, A. Rizzo, S. Wang, M. Cullen, A. Molnar, and K. Bergman. Ultra-dense 3D Integrated 5.3 Tb/s/mm2 80 Micro-Disk Modulator Transmitter. *Optical Fiber Communication Conf.*, Mar 2023.

[DRA+21] S. Daudlin, A. Rizzo, N. C. Abrams, S. Lee, D. Khilwani, V. Murthy, J. Robinson, T. Collier, A. Molnar, and K. Bergman. 3D-Integrated Multichip Module Transceiver for Terabit-Scale DWDM Interconnects. *Optical Fiber Communication Conf.*, Jun 2021.

[DRL+23] S. Daudlin, A. Rizzo, S. Lee, D. Khilwani, C. Ou, S. Wang, A. Novick, V. Gopal, M. Cullen, R. Parsons, A. Molnar, and K. Bergman. 3D Photonics for Ultra-Low Energy, High Bandwidth-Density Chip Data Links. *Computing Research Repository (CoRR)*, arXiv:2310.01615, Oct 2023.

[DT04] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[DXT+18] S. Davidson, S. Xie, C. Torng, K. Al-Hawaj, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. G. Dreslinski, C. Batten, and M. B. Taylor. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro*, Mar 2018.

[FCL+23] A. Firoozshahian, J. Coburn, R. Levenstein, R. Nattoji, A. Kamath, O. Wu, G. Grewal, H. Aepala, B. Jakka, B. Dreyer, A. Hutchin, U. Diril, K. Nair, E. K. Aredestani, M. Schatz, Y. Hao, R. Komuravelli, K. Ho, S. Abu Asal, J. Shajrawi, K. Quinn, N. Sreedhara, P. Kansal, W. Wei, D. Jayaraman, L. Cheng, P. Chopda, E. Wang, A. Bikumandla, A. Karthik Sengottuvel, K. Thottempudi, A. Narasimha, B. Dodds, C. Gao, J. Zhang, M. Al-Sanabani, A. Zehtabioskuie, J. Fix, H. Yu, R. Li, K. Gondkar, J. Montgomery, M. Tsai, S. Dwarakapuram, S. Desai, N. Avidan, P. Ramani, K. Narayanan, A. Mathews, S. Gopal, M. Naumov, V. Rao, K. Noru, H. Reddy, P. Venkatapuram, and A. Bjorlin. MTIA: First Generation Silicon Targeting Meta's Recommendation Systems. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2023.

[FFDMS14] F. Fatollahi-Fard, D. Donofrio, G. Michelogiannakis, and J. Shalf. Opensoc Fabric: On-Chip Network Generator: Using Chisel to Generate a Aarameterizable On-Chip Interconnect Fabric. *Int'l Workshop on Network on Chip Architectures*, Dec 2014.

[fle19] Arteris FlexNoC Interconnect IP. Online Webpage, accessed Sep 20, 2019. `http://www.arteris.com/flexnoc`.

[FZS22] W. Fedus, B. Zoph, and N. Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *ACM Journal of Machine Learning Research (JMLR)*, Apr 2022.

[GGH+22] J. Gonzalez, M. G. Palma, M. Hattink, R. Rubio-Noriega, L. Orosa, O. Mutlu, K. Bergman, and R. Azevedo. Optically Connected Memory for Disaggregated Data Centers. *Journal of Parallel and Distributed Computing*, May 2022.

[GHKM09] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Express Cube Topologies for On-Chip Interconnects. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2009.

[GHKM11] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Kilo-NOC: A Heterogeneous Network-on-Chip Architecture for Scalability and Service Guarantees. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[GLB+24] P. Georgiev, V. I. Lei, R. Burnell, L. Bai, A. Gulati, et al. Gemini 1.5: Unlocking Multimodal Understanding Across Millions of Tokens of Context. *Computing Research Repository (CoRR)*, arXiv:2403.05530, Mar 2024.

[Gre11] P. Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *EE Times*, Oct 2011.

[Hal20] T. R. Halfhill. ThunderX3's Cloudburst of Threads: Marvell Previews 96-core 384-thread Arm Server Processor. *Microprocessor Report, The Linley Group*, Apr 2020.

[HCB+19] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in Neural Information Processing Systems (NeurIPS)*, Dec 2019.

[HDH+10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2010.

[HKS+21] K. Hosseini, E. Kok, S. Y. Shumarayev, C.-P. Chiu, A. Sarkar, A. Toda, Y. Ke, A. Chan, D. Jeong, M. Zhang, S. Raman, T. Tran, K. A. Singh, P. Bhargava, C. Zhang, H. Lu, R. Mahajan, X. Li, N. Deshpande, C. O'Keeffe, T. T. Hoang, U. Krishnamoorthy, C. Sun, R. Meade, V. Stojanović, and M. Wade. 8 Tbps Co-

Packaged FPGA and Silicon Photonics Optical IO. *Optical Fiber Communication Conf.*, Feb 2021.

[HKS+22] K. Hosseini, E. Kok, S. Y. Shumarayev, D. Jeong, A. Chan, A. Katzin, S. Liu, R. Roucka, M. Raval, M. Mac, C.-P. Chiu, T. Tran, K. A. Singh, S. Raman, Y. Ke, C. Li, L.-F. Yang, P. Chao, H. Lu, F. Luna, X. Li, T. T. Hoang, A. Sarkar, A. Toda, R. Mahajan, N. Deshpande, C. O'Keeffe, U. Krishnamoorthy, V. Stojanović, C. Madden, C. Zhang, M. Sysak, P. Bhargava, C. Sun, and M. Wade. 5.12 Tbps Co-Packaged FPGA and Silicon Photonics Interconnect I/O. *Symp. on VLSI Technology and Circuits (VLSI)*, Jun 2022.

[HVS+07] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-GHz Mesh Interconnect for a Teraflops Processor. *IEEE Micro*, 27(5):51–61, Sep/Oct 2007.

[hyp19] Most testing is ineffective - Hypothesis. Online Webpage, accessed Sep 20, 2019. Https://hypothesis.works.

[inf24] Need for Speed – InfiniBand Network Bandwidth Evolution. Online Webpage, Jan 2024. `https://community.fs.com/article/need-for-speed-%E2%80%93-infiniband-network-bandwidth-evolution.html`.

[int] Accelerating Innovation Through A Standard Chiplet Interface: The Advanced Interface Bus (AIB). Intel Whitepaper. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/accelerating-innovation-through-aib-whitepaper.pdf`.

[JBM+13] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally. A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2013.

[JDZ+20] T. Jung, S. Davidson, C. Zhao, D. Richmond, and M. B. Taylor. Ruche Networks: Wire-Maximal No-Fuss NoCs. *Int'l Symp. on Networks-on-Chip (NOCS)*, Sep 2020.

[JIB18] S. Jiang, B. Ilbeyi, and C. Batten. Mamba: closing the performance gap in productive hardware development frameworks. *Design Automation Conf. (DAC)*, Jun 2018.

[JKL+23] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. A. Patterson. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2023.

[JMBM04] A. Jalabert, S. Murali, L. Benini, and G. D. Micheli. xpipesCompiler: A Tool for Instantiating Application Specific Networks on Chip. *Design, Automation, and Test in Europe (DATE)*, Feb 2004.

[JNR+23] A. James, A. Novick, A. Rizzo, R. Parsons, K. Jang, M. Hattink, and K. Bergman. Scaling Comb-Driven Resonator-Based DWDM Silicon Photonic Links to Multi-Tb/s in the Multi-FSR Regime. *Optica*, Jul 2023.

[JOP+20] S. Jiang, Y. Ou, P. Pan, K. Cheng, Y. Zhang, and C. Batten. PyH2: Using PyMTL3 to Create Productive and Open-Source Hardware Testing Methodologies. *IEEE Design & Test*, 40(4):58–66, Jul/Aug 2020.

[JPOB20] S. Jiang, P. Pan, Y. Ou, and C. Batten. PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification. *IEEE Micro*, 40(4):58–66, Jul/Aug 2020.

[kal24a] Kalray MPPA Products. Online Webpage, 2024 (accessed Oct 2024). `https://www.kalrayinc.com/products/mppa-technology/`.

[kal24b] Kalray MPPA Products. Online Webpage, 2024 (accessed Oct 2024). `https://tenstorrent.com/hardware/grayskull`.

[KB15] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *Int'l Conf. on Learning Representations (ICLR)*, May 2015.

[KBD07] J. Kim, J. Balfour, and W. Dally. Flattened Butterfly Topology for On-Chip Networks. *Int'l Symp. on Microarchitecture (MICRO)*, Aug 2007.

[KCKP14] T. Krishna, C.-H. O. Chen, W.-C. Kwon, and L.-S. Peh. SMART: Single-Cycle Multihop Traversals over A Shared Network on Chip. *IEEE Micro*, 34(3):43–56, 2014.

[KCL+23] V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro. Reducing Activation Recomputation in Large Transformer Models. *Annual Conf. on Machine Learning and Systems (MLSys)*, Jun 2023.

[KGA+21] M. Khani, M. Ghobadi, M. Alizadeh, Z. Zhu, M. Glick, K. Bergman, A. Vahdat, B. Klenk, and E. Ebrahimi. SiP-ML: High-Bandwidth Optical Network Interconnects for Machine Learning Training. *SIGCOMM*, Aug 2021.

[KK17] H. Kwon and T. Krishna. Opensmart: Single-Cycle Multi-Hop NoC Generator in BSV and Chisel. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2017.

[KLN12] A. B. Kahng, B. Lin, and S. Nath. Explicit Modeling of Control and Data for Improved NoC Router Estimation. *Design Automation Conf. (DAC)*, Jun 2012.

[KLN15] A. B. Kahng, B. Lin, and S. Nath. ORION3.0: A Comprehensive NoC Router Estimation Tool. *IEEE Embedded Systems Letters (ESL)*, Feb 2015.

[KLO+24] D. Khilwani, S. Lee, C. Ou, S. Daudlin, A. Rizzo, S. Wang, M. Cullen, K. Bergman, and A. Molnar. 3D-Integrated, Low Power, High Bandwidth Density Opto-Electronic Transceiver. *Int'l Conf. on Circuits and Systems (ISCAS)*, May 2024.

[KLPS09] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration. *Design, Automation, and Test in Europe (DATE)*, Apr 2009.

[KMH+20] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling Laws for Neural Language Models. *Computing Research Repository (CoRR)*, arXiv:2001.08361, Jan 2020.

[KPK+09] P. Kumar, Y. Pan, J. Kim, G. Memik, and A. Choudhary. Exploring Concentration and Channel Slicing in On-Chip Network Router. *Int'l Symp. on Networks-on-Chip (NOCS)*, May 2009.

[KPKJ07] A. Kumar, L.-S. Peh, P. Kundu, and N. Jha. Express Virtual Channels: Towards the Ideal Interconnection Fabric. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2007.

[KPND20] D. Konstantinou, A. Psarras, C. Nicopoulos, and G. Dimitrakopoulos. The Mesochronous Dual-Clock FIFO Buffer. *Symp. on VLSI Technology and Circuits (VLSI)*, Jun 2020.

[KS08] B. Kim and V. Stojanović. Characterization of Equalized and Repeated Interconnects for NoC Applications. *IEEE Design and Test of Computers*, 25(5):430–439, Sep 2008.

[KSK18] H. Kwon, A. Samajdar, and T. Krishna. Maeri: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2018.

[KTK+18] M. Karunaratne, C. Tan, A. Kulkarni, T. Mitra, and L.-S. Peh. Dnestmap: Mapping Deeply-Nested Loops on Ultra-Low Power CGRAs. *Design Automation Conf. (DAC)*, Jun 2018.

[KYAC11] Y.-H. Kao, M. Yang, N. S. Artan, and H. J. Chao. CNoC: High-Radix Clos Network-on-Chip. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(12):1897–1910, Dec 2011.

[LBH+24] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond:

CXL-Based Memory Pooling Systems for Cloud Platforms. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2024.

[LCG+23] A. Li, T.-J. Chang, F. Gao, T. Ta, G. Tziantzioulis, Y. Ou, M. Wang, J. Tu, K. Xu, P. Jackson, A. Ning, G. Chirkov, M. Orenes-Vera, S. Agwa, X. Yan, E. Tang, J. Balkind, C. Batten, and D. Wentzlaff. CIFER: A Cache-Coherent 12-nm 16-mm2 SoC With Four 64-Bit RISC-V Application Cores, 18 32-Bit RISC-V Compute Cores, and a 1541 LUT6/mm2 Synthesizable eFPGA. *IEEE Solid-State Circuits Letters (SSCL)*, Aug 2023.

[LFF+18] L. Li, J. Fang, H. Fu, J. Jiang, W. Zhao, C. He, X. You, and G. Yang. swCaffe: A Parallel Framework for Accelerating Deep Learning Applications on Sunway TaihuLight. *Int'l Conf. on Cluster Computing*, Sep 2018.

[LSC+10] M. Lis, K. S. Shim, M. H. Cho, P. Ren, O. Khan, and S. Devadas. DARSIM: A Parallel Cycle-Level NoC Simulator. *Workshop on Modeling, Benchmarking and Simulation (MOBS)*, Jun 2010.

[LSC+13] M. Lis, K. S. Shim, M. H. Cho, I. Lebedev, and S. Devadas. Hardware-Level Thread Migration in a 110-Core Shared-Memory Multiprocessor. Technical Report 512, MIT CSAIL CSG, Nov 2013.

[LW21] A. Li and D. Wentzlaff. PRGA: An Open-Source FPGA Research and Prototyping Framework. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2021.

[LZB14] D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[MBD09] G. Michelogiannakis, J. Balfour, and W. J. Dally. Elastic-Buffer Flow Control for On-Chip Networks. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2009.

[met24] Introducing Meta Llama 3: The most capable openly available LLM to date. Online Webpage, Apr 2024. `https://ai.meta.com/blog/meta-llama-3/`.

[MFN+17] M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, J. Balkind, A. Lavrov, M. Shahrad, S. Payne, and D. Wentzlaff. Piton: A Manycore Processor for Multitenant Clouds. *IEEE Micro*, Mar 2017.

[MM04] S. Murali and G. D. Micheli. SUNMAP: A Tool for Automatic Topology Selection and Generation for NoCs. *Design Automation Conf. (DAC)*, Jul 2004.

[MPZ+20] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. Walker, et al. VTR 8: High-Performance CAD and

Customizable FPGA Architecture Modelling. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(2):1–55, 2020.

[MWM04] R. Mullins, A. West, and S. Moore. Low-Latency Router Microarchitecture for Intra-Chip Networks. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[net19] NetmakerWiki. Online Webpage, accessed Sep 20, 2019. Http://www-dyn.cl.cam.ac.uk/ rdm34/wiki/index.php.

[NHP+19] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN training. *Symp. on Operating Systems Principles (SOSP)*, Oct 2019.

[Nik04] N. Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High-Level Specifications. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, Jun 2004.

[NJD+23] A. Novick, A. James, L. Y. Dai, Z. Wu, A. Rizzo, S. Wang, Y. Wang, M. Hattink, V. Gopal, K. Jang, R. Parsons, and K. Bergman. High-Bandwidth Density Silicon Photonic Resonators for Energy-Efficient Optical Interconnects. *Applied Physics Letters*, 10(4):041306, Dec 2023.

[NPS+21] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia. Memory-Efficient Pipeline-Parallel DNN Training. *Int'l Conf. on Machine Learning (ICML)*, Jul 2021.

[NSC+21] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia. Efficient large-scale language model training on GPU clusters using megatron-LM. *Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, Nov 2021.

[NTW23] A. Ning, G. Tziantzioulis, and D. Wentzlaff. Supply Chain Aware Computer Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2023.

[nvi20] NVIDIA A100 Tensor Core GPU Architecture. NVIDIA Whitepaper, 2020. `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`.

[nvi23] NVIDIA H100 Tensor Core GPU Architecture. NVIDIA Whitepaper, 2023. `https://resources.nvidia.com/en-us-tensor-core`.

[nvl24] Unveiling The Evolution of NVLink. Online Webpage, Feb 2024. `https://www.naddod.com/blog/unveiling-the-evolution-of-nvlink`.

[NWR+24] A. Novick, S. Wang, A. Rizzo, V. Gopal, and K. Bergman. Ultra-Efficient Interleaved Vertical-Junction Microdisk Modulator with Integrated Heater. *Optical Fiber Communication Conf.*, Mar 2024.

[OAB20] Y. Ou, S. Agwa, and C. Batten. Implementing Low-Diameter On-Chip Networks for Manycore Processors Using a Tiled Physical Design Methodology. *Int'l Symp. on Networks-on-Chip (NOCS)*, Sep 2020.

[Olo16] A. Olofsson. Epiphany-V: A 1024-processor 64-bit RISC System-On-Chip. *CoRR arXiv:1610.01832*, Aug 2016.

[OMS+12] J. S. Orcutt, B. Moss, C. Sun, J. Leu, M. Georgas, J. Shainline, E. Zgraggen, H. Li, J. Sun, M. Weaver, S. Urošević, M. Popović, R. J. Ram, and V. Stojanović. Open Foundry Platform for High-Performance Electronic-Photonic Integration. *Optics Express*, 20(11):12222–12232, May 2012.

[ope16] OpenPiton Microarchitecture Specification. OpenPiton Manual, 2016. `https://parallel.princeton.edu/openpiton/docs/micro_arch.pdf`.

[PCSV08] A. Pinto, L. P. Carloni, and A. Sangiovanni-Vincentelli. COSI: A Framework for the Design of Interconnection Networks. *Design, Automation, and Test in Europe (DATE)*, Oct 2008.

[PD01] L.-S. Peh and W. J. Dally. A Delay Model and Speculative Architecture for Pipelined Routers. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Jan 2001.

[PH13] M. K. Papamichael and J. C. Hoe. CONNECT: Re-examining Conventional Wisdom for Designing NoCs in the Context of FPGAs. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2013.

[PH18] A. Pathania and J. Henkel. HotSniper: Sniper-Based Toolchain for Many-Core Thermal Simulations in Open Systems. *IEEE Embedded Systems Letters (ESL)*, Aug 2018.

[PKS+24] S.-S. Park, K. Kim, J. So, J. Jung, J. Lee, K. Woo, N. Kim, Y. Lee, H. Kim, Y. Kwon, J. Kim, J. Lee, Y. Cho, Y. Tai, J. Cho, H. Song, J. H. Ahn, and N. S. Kim. An LPDDR-based CXL-PNM Platform for TCO-efficient Inference of Transformer-based Large Language Models. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar 2024.

[pyt14] PyTest. Online Webpage, 2014 (accessed Oct 1, 2014). `http://www.pytest.org`.

[PZD+20] D. Petrisko, C. Zhao, S. Davidson, P. Gao, D. Richmond, and M. B. Taylor. NoC Symbiosis (Special Session Paper). *Int'l Symp. on Networks-on-Chip (NOCS)*, Nov 2020.

[Ram11] C. Ramey. TILE-Gx100 ManyCore Processor: Acceleration Interfaces and Architecture. *Symp. on High Performance Chips (Hot Chips)*, Aug 2011.

[RDN+23a] A. Rizzo, S. Daudlin, A. Novick, A. James, V. Gopal, V. Murthy, Q. Cheng, B. Y. Kim, X. Ji, Y. Okawachi, M. van Niekerk, V. Deenadayalan, G. Leake, M. Fanto, S. Preble, M. Lipson, A. Gaeta, and K. Bergman. Petabit-Scale Silicon Photonic Interconnects With Integrated Kerr Frequency Combs. *Journal of Selected Topics in Quantum Electronics*, 29(1):3700120, Jan/Feb 2023.

[RDN+23b] A. Rizzo, U. Dave, A. Novick, A. Freitas, S. P. Roberts, A. James, M. Lipson, and K. Bergman. Fabrication-Robust Silicon Photonic Devices in Standard Sub-Micron Silicon-on-Insulator Processes. *Optics Letters*, 48(2):215, Jan 2023.

[RDvN+23] A. Rizzo, V. Deenadayalan, M. van Niekerk, G. Leake, C. Tison, A. Novick, D. Coleman, K. Bergman, S. Preble, and M. Fanto. Ultra-Efficient Foundry-Fabricated Resonant Modulators with Thermal Undercut. *Conf. on Lasers and Electro-Optics (CLEO)*, May 2023.

[RGP+13] J. C. Rosenberg, W. M. J. Green, J. Proesel, S. Assefa, D. M. Gill, T. Barwicz, S. M. Shank, C. Reinholm, M. Khater, E. Kiewra, S. Kamlapurkar, and Y. A. Vlasov. A Monolithic Microring Transmitter in 90nm SOI CMOS Technology. *Photonics Conference*, Sep 2013.

[RLC+12] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, N. Zheng, and S. Devadas. Hornet: A Cycle-Level Multicore Simulator. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 31(6):890–903, 2012.

[RRA+21] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He. ZeRO-Offload: Democratizing Billion-Scale Model Training. *USENIX Annual Technical Conference (ATEC)*, Jan 2021.

[RSSK20] S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna. ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Aug 2020.

[Rup24] 42 Years of Microprocessor Trend Data. Online Webpage, 2024 (accessed Oct 2024). `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data`.

[RZAH+19a] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. A 1.4 GHz 695 Giga RISC-V Inst/s 496-core Manycore Processor with Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS. *Symp. on Very Large-Scale Integration Circuits (VLSIC)*, Jun 2019.

[RZAH+19b] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. Evaluating Celerity: A 16nm 695 Giga-RISC-V Instructions/s Manycore Processor with Synthesizable PLL. *IEEE Solid-State Circuits Letters (SSCL)*, 2(12):289–292, Dec 2019.

[SBM+19] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2019.

[SCK+12] C. Sun, C.-H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic. DSENT-A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. *Int'l Symp. on Networks-on-Chip (NOCS)*, May 2012.

[SCP+18] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman. Mesh-TensorFlow: deep learning for supercomputers. *Advances in Neural Information Processing Systems (NIPS)*, Dec 2018.

[SGC+16] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36:34–46, Mar/Apr 2016.

[Sha22] D. D. Sharma. The Evolution of the PCI Express Specification: On its Sixth Generation, Third Decade and Still Going Strong. PCI-SIG Blog, Jan 2022. `https://pcisig.com/blog/evolution-pci-express-specification-its-sixth-generation-third-decade-and-st`

[SJZ+20] C. Sun, D. Jeong, M. Zhang, W. Bae, C. Zhang, P. Bhargava, D. Van Orden, S. Ardalan, C. Ramamurthy, E. Anderson, A. Katzin, H. Lu, S. Buchbinder, B. Beheshtian, A. Khilo, M. Rust, C. Li, F. Sedgwick, J. Fini, R. Meade, V. Stojanović, and M. Wade. TeraPHY: An O-Band WDM Electro-Optic Platform for Low Power, Terabit/s Optical I/O. *2020 IEEE Symposium on VLSI Technology*, Jun 2020.

[SPP+19] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *Computing Research Repository (CoRR)*, arXiv:1909.08053, Mar 2019.

[SWL+15] C. Sun, M. T. Wade, Y. Lee, J. S. Orcutt, L. Alloatti, M. S. Georgas, A. S. Waterman, J. M. Shainline, R. R. Avizienis, S. Lin, B. R. Moss, R. Kumar, F. Pavanello, A. H. Atabaki, H. M. Cook, A. J. Ou, J. C. Leu, Y.-H. Chen, K. Asanović, R. J.

Ram, M. A. Popović, and V. M. Stojanović. Single-Chip Microprocessor that Communicates Directly Using Light. *Nature*, 528(7583):534–538, Dec 2015.

[TB12] A. Tran and B. Baas. NoCTweak: A Highly Parameterizable Simulator for Early Exploration of Performance and Energy of Networks On-Chip. Technical Report ECE-VCL-2012-2, VLSI Computation Lab, ECE Department, University of California, Davis, Jul 2012.

[tec24a] TechPowerUp CPU Database. Online Database, 2024. `https://www.techpowerup.com/cpu-specs`.

[tec24b] TechPowerUp GPU Database. Online Database, 2024. `https://www.techpowerup.com/gpu-specs`.

[TFZ+08] G. Tan, D. Fan, J. Zhang, A. Russo, and G. R. Gao. Experience on Optimizing Irregular Computation for Memory Hierarchy in Manycore Architecture. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Feb 2008.

[TKM+03] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, W. Lee, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, , and A. Agarwal. A 16-Issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2003.

[TKMP18] C. Tan, M. Karunaratne, T. Mitra, and L.-S. Peh. Stitch: Fusible Heterogeneous Accelerators Enmeshed with Many-Core Architecture for Wearables. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2018.

[TLI+23] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. LLaMA: Open and Efficient Foundation Language Models. *Computing Research Repository (CoRR)*, arXiv:2302.13971, Feb 2023.

[TLM+04] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[TOJ+19] C. Tan, Y. Ou, S. Jiang, P. Pan, C. Torng, S. Agwa, and C. Batten. PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks. *Int'l Conf. on Computer Design (ICCD)*, Nov 2019.

[Ton24] D. Tonietto. Connecting Switch to Fiber: The Energy Efficiency Challenge. *Optical Fiber Communication Conf.*, Mar 2024.

[TPN+24] W. J. Turner, J. W. Poulton, Y. Nishi, X. Chen, B. Zimmer, S. Song, W. John M, W. J. Dally, and C. T. Gray. Leveraging Micro-Bump Pitch Scaling to Accelerate Interposer Link Bandwidths for Future High-Performance Compute Applications. *Custom Integrated Circuits Conf. (CICC)*, Apr 2024.

[uci24] UCIe 1.0 Specification. Online Webpage, Sep 2024. `https://www.uciexpress.org/specifications`.

[VSP+17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention Is All You Need. *Advances in Neural Information Processing Systems (NeurIPS)*, Dec 2017.

[WGH+07] D. Wentzlaff, P. Griffin, H. Hoffman, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27:15–31, Sep/Oct 2007.

[Whe20] B. Wheeler. Ampere Maxes Out at 128 Cores. *Microprocessor Report, The Linley Group*, Jul 2020.

[WKP11] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 31(2):50–59, Mar/Apr 2011.

[WNP+23] Y. Wang, A. Novick, R. Parsons, S. Wang, K. Jang, A. James, M. Hattink, V. Gopal, A. Rizzo, C.-P. Chiu, K. Hosseini, T. T. Hoang, and K. Bergman. Scalable Architecture for Sub-pJ/b Multi-Tbps Comb-Driven DWDM Silicon Photonic Transceiver. *Next-Generation Optical Communication: Components, Sub-Systems, and Systems (SPIE OPTO)*, Mar 2023.

[Wol20] C. Wolf. Yosys Open SYnthesis Suite. Yosys Manual, 2020. `https://yosyshq.net/yosys/`.

[WPM02] H. Wang, L.-S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. *Int'l Symp. on Microarchitecture (MICRO)*, Nov 2002.

[WWM+24] S. Wang, Y. Wang, X. Meng, K. Hosseini, T. T. Hoang, and K. Bergman. Automated Tuning of Ring-Assisted MZI–Based Interleaver for DWDM Systems. *Optical Fiber Communication Conf.*, May 2024.

[WWP09] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, Apr 2009.

[WWP+24] Y. Wang, S. Wang, R. Parsons, A. Novick, V. Gopal, K. Jang, A. Rizzo, C.-P. Chiu, K. Hosseini, T. T. Hoang, S. Shumarayev, and K. Bergman. Silicon Photonics Chip I/O for Ultra High-Bandwidth and Energy-Efficient Die-to-Die Connectivity. *Custom Integrated Circuits Conf. (CICC)*, Apr 2024.

[ZB19] F. Zaruba and L. Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 27(11):2629–2640, 2019.

[ZNPW24] H. Zhang, A. Ning, R. B. Prabhakar, and D. Wentzlaff. LLMCompass: Enabling Efficient Hardware Design for Large Language Model Inference. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2024.

[ZSB21] F. Zaruba, F. Schuiki, and L. Benini. Manticore: A 4096-Core RISC-V Chiplet Architecture for Ultraefficient Floating-Point Computing. *IEEE Micro*, Mar/Apr 2021.