©Copyright 2026
Dai Cheol Jung

A Manycore Architecture for Scalability, Programmability, and Compute Density

Dai Cheol Jung

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2026

Reading Committee:

Michael Taylor, Chair

Mark Oskin

Ang Li

Alvaro Chavarria

Program Authorized to Offer Degree:
Department of Electrical and Computer Engineering

University of Washington

Abstract

A Manycore Architecture for Scalability, Programmability, and Compute Density

Dai Cheol Jung

Chair of the Supervisory Committee:
Professor Michael Taylor
Department of Electrical and Computer Engineering

Manycore architectures have been proposed as a way to transform abundant silicon resources into highly programmable parallel processors with exceptional compute density. However, many architectural mechanisms inherited from large, complex cores – designed to optimize single-thread performance and support cache coherence – have limited their ability to achieve the compute density and scalability needed to compete with other parallel architectures. This thesis presents HammerBlade Manycore, which integrates numerous novel ideas spanning multiple areas of parallel architecture, such as VLSI resource organization, on-chip networks, memory hierarchy, memory-level parallelism, address mapping and translation, synchronization, and thread management. Its flexibility has been demonstrated with a parallel benchmark suite representing a broad range of computational and communication patterns.

Network-on-Chip (NoC) has become one of the most critical components in modern, parallel architectures. This thesis presents Ruche Networks, which leverage unused wiring resources cost-effectively to reduce network diameter and increase bisection bandwidth by augmenting the 2-D mesh with uniform long-range physical links. Ruche Networks are tileable, physically scalable, and energy efficient. A comprehensive evaluation is provided, comparing Ruche Networks with other NoCs, such as 2-D mesh, torus, and multi-mesh, in terms of area, power, network performance, and scalability.

Finally, this thesis presents the 12 nm implementation of the 2048-core HammerBlade

ASIC. This chip achieves several milestones: it is the first to implement Ruche Networks in a large-scale and high-frequency manycore processor, and it sets a record peak RISC-V instruction throughput and CoreMark scores. Building a chip at this scale presents significant challenges, including timing closure and excessively long CAD tool runtimes. This thesis documents the technical lessons learned during tapeout.

TABLE OF CONTENTS

Page

List of Figures iv

List of Tables vi

Chapter 1: Introduction 1
- 1.1 Motivation for Scalable Manycore Architecture 1
- 1.2 Motivation for Scalable Network-on-Chip 4
- 1.3 Contributions 8
- 1.4 Thesis Outline 8
- 1.5 Relation to Published Works 9
- 1.6 Open-source Research Artifacts 9
- 1.7 Acknowledgments 10

Chapter 2: HammerBlade Manycore 11
- 2.1 Architecture 11
  - 2.1.1 Cellular Manycore 11
  - 2.1.2 HammerBlade RISC-V Core 14
  - 2.1.3 HB Network-on-Chip 16
- 2.2 Parallel Programming Model 19
  - 2.2.1 HB Programming Model 20
  - 2.2.2 Parallel Benchmark Suite 24
- 2.3 Evaluation 28
  - 2.3.1 Experimental Setup 28
  - 2.3.2 Incremental Feature Analysis 32
  - 2.3.3 Core and HBM2 Utilization 34
  - 2.3.4 Scaling Irregular Workloads using Tile Groups 37
  - 2.3.5 NoC Bisection Utilization 38
  - 2.3.6 Doubling HW Resources 39
  - 2.3.7 Comparison with Hierarchical Manycore 40

2.3.8 Energy Analysis 41
2.3.9 GPU Comparison 42
2.4 Performance debugging in HB 44
2.5 Related Work 48
2.5.1 Flat Manycore 48
2.5.2 Hierarchical Manycore 50

Chapter 3: Ruche Networks 51
3.1 NoC Architecture 51
3.1.1 Physically scalable Ruche topology 51
3.1.2 Ruche Router Architecture 54
3.2 Evaluation 58
3.2.1 Full Ruche - Synthetic Traffic 59
3.2.2 Full Ruche - Area and Cycle Time 62
3.2.3 Full Ruche - Energy 64
3.2.4 Full Ruche - Fairness 65
3.2.5 Half Ruche - Synthetic Traffic 66
3.2.6 Half Ruche - Benchmark Speedup 70
3.2.7 Half Ruche - Benchmark Scalability 73
3.2.8 Half Ruche - Remote Load Latency 75
3.2.9 Half Ruche - Energy 77
3.2.10 Half Ruche - Geomean Summary 79
3.3 Related Work 81

Chapter 4: Silicon Prototype 83
4.1 Implementation 83
4.1.1 Tile-based Hierarchical Design Methodology 84
4.1.2 Constraining Tile Interfaces 86
4.1.3 Building a Cell Row 91
4.1.4 Building Ruche Networks 92
4.2 Chip Measurement 95
4.2.1 Shmoo Plot 95
4.2.2 Packet Energy vs Distance 96
4.3 Chip Comparison 97
4.3.1 NoC Comparison 97
4.3.2 CoreMark Comparison 98

4.4 Related Work 99

Chapter 5: Conclusion and Future Work 101
5.1 Conclusion 101
5.2 Future Work 102

Bibliography 106

LIST OF FIGURES

Figure Number | Page

1.1 Data-level parallelism with highly irregular control flow and data access. . . . 3
1.2 A SoC based on a variety of fixed-function accelerator tiles vs. a flexible manycore array with a similar area. 4
1.3 Low-diameter NoC topologies with non-local links. 5
2.1 Resource organization in conventional manycore architectures. 12
2.2 Overview of HB Cellular manycore architecture. 13
2.3 HB RISC-V core pipeline schematic. 15
2.4 Instruction cache branch target buffer. 16
2.5 Bisection link utilization between two Cells over time. 18
2.6 HW Barrier Network and latency comparison with SW-implemented barriers. 20
2.7 HammerBlade PGAS Mapping. 22
2.8 IPOLY hashing table for 32 banks. 22
2.9 A host program launching producer-consumer kernels on two Cells. 24
2.10 A kernel routine for synchronization between a producer and a consumer tile-group. 25
2.11 Jacobi kernel snippet demonstrating the use of Group SPM pointers to remotely access nearby scratchpads. 30
2.12 BFS kernel snippet demonstrating the use of atomic add to implement a simple parallel for-loop. 31
2.13 Different strategies to double the hardware resources. 31
2.14 Speedup over Baseline Manycore. 33
2.15 Core and HBM2 utilization graphs. 36
2.16 Speedup and HBM2 utilization with varying tile group size. 37
2.17 Horizontal bisection link utilization. 38
2.18 Speedup using different strategies to double hardware resources. 39
2.19 Performance comparison between 32×8 HB and the hierarchical manycore model. 40
2.20 Comparison of "Energy per Instruction" (EPI) with Piton. 41
2.21 Comparison between simulated NVIDIA V100 and 64 16×8 HB Cells. 43

2.22 Core temporal utilization graph for all benchmarks. 46
2.23 Full-stack utilization graph for SGEMM. 47

3.1 A bitwise pattern of mapping Half Ruche (RF = 3) on a tile-based design. . . 52
3.2 Two approaches to combine 2x multi-mesh routers. 54
3.3 Two variants of a deadlock-free dimension-ordered routing algorithm for Full Ruche (RF = 3, X-Y order). 55
3.4 Full Ruche crossbar connectivity matrix (X-Y DOR). 56
3.5 VC Allocation Router. 57
3.6 Synthetic traffic analysis on 2-D mesh, 2-D torus, and various Full Ruche topologies. 61
3.7 Area vs. Cycle Time comparison of Mesh, Multi-mesh, Full Ruche and 2-D torus routers. 63
3.8 Distribution of average latency in 16×16 Uniform Random. 65
3.9 Synthetic traffic analysis on 16×8, 32×16, and 64×8. 68
3.10 Speedup over 2-D mesh on 32×16 and 16×8. 72
3.11 Scalability comparison between 64×8 and 32×16. 74
3.12 Average remote load latency for 32×16. 76
3.13 Total energy breakdown for 32×16 (normalized to 2-D mesh). 78

4.1 11×9 mm^2 layout of 2048-core HammerBlade ASIC. 83
4.2 An overview of automated VLSI CAD flow. 85
4.3 Decomposing an architecture using the tile-based design methodology. 86
4.4 Setup timing path between two tiles interfaced with NoC routers. 88
4.5 Final input and output delay constraints used for logic synthesis. 89
4.6 Final input and output delay constraints used for APR. 90
4.7 Final clock skew slack on HB compute tile interfaces. 91
4.8 Module hierarchy for Cell Row. 93
4.9 Cell Row floorplan and clock tree. 93
4.10 Floorplan view of Ruche feedthrough wire placement. 94
4.11 Cross-sectional view of Ruche feedthrough wire placement. 95
4.12 HammerBlade compute tile layout and its area breakdown. 96
4.13 Chip measurement: shmoo plot. 97
4.14 Chip measurement: energy per packet. 98

5.1 Small versus big tiles – illustrating the inefficient utilization of sense amplifiers. 103

LIST OF TABLES

Table Number Page

2.1 Ten parallel benchmarks used to demonstrate HB's parallel programmability. 26
2.2 HB machine configurations for evaluation. 28
2.3 List of core stall types. 34
2.4 Hardware configuration used for HammerBlade and NVIDIA V100 comparison. 42
2.5 Comparison of manycore designs on network topology, processor type, and compute density. 48

3.1 Comparing various NoC topologies based on physical scalability criteria. . . . 53
3.2 Multi-mesh, Full Ruche and 2-D torus Router Area Breakdown. 63
3.3 Full Ruche and 2-D torus Router Energy per Packet 64
3.4 Comparison of bandwidth ratio. 69
3.5 Benchmarks and input datasets for NoC evaluation. 71
3.6 Summary of Half Ruche eval using geomean scores. 80

4.1 NoC comparison table. 99
4.2 CoreMark score (CM) comparison. 100

ACKNOWLEDGMENTS

I would like to first thank some of the engineering professors I met at Brown University – Prof. Iris Bahar, Prof. William Patterson, and Prof. Jimmy Xu – who have inspired me to appreciate the wonders of the physical world and foster a lifelong love for learning.

I would like to extend my gratitude to Prof. Scott Hauck and Prof. Visvesh Sathe, who helped me navigate through grad school during my early years at UW. I would also like to thank the members of the thesis committee – Prof. Mark Oskin, Prof. Ang Li, and Prof. Alvaro Chavarria.

This thesis would not have been possible without the guidance of my PhD advisor, Prof. Michael Taylor, who has always challenged me with increasingly more difficult engineering problems. It was simply the fact that I liked the research and the idea that I am pushing the envelope that got me out of bed every morning during PhD. So I thank my advisor for the opportunity and inspiration.

I was very fortunate to have worked with the members of Bespoke Silicon Group, who dared to dive into one of the most herculean academic endeavors. My first recognition goes to two post-doc researchers, Dr. Chun Zhao and Dr. Shaolin Xie, who helped me gain knowledge by patiently answering my countless questions about building hardware, when I was just starting out at BSG. I remember going to the lab, and even on rainy, weekend days, I would often find them working in their respective spots. Throughout my PhD, it was probably during those quiet hours, around their supportive presence, when my productivity reached its peak. My second recognition goes to Paul Gao, whom I befriended both inside and outside of the lab. Although we come from different backgrounds, we related with each other on some common grounds, such as foreign student life, similar cultural values, and appreciation for local breweries along the Burke-Gilman Trail. I would also like to recognize and thank many other members who helped me complete this journey: Scott

Davidson, Max Ruttenberg, Dan Petrisko, Farzam Gilani, Huwan Peng, Yuan-Mao Chueh, Rico Li, Bandhav Veluri, and Dustin Richmond.

Finally, it was for my family in South Korea, who have put me through the best education one can possibly receive since high school, that I was able to afford this luxury to pursue knowledge. I would like to thank my loving wife Katherine, who joined me in Seattle from Utah for the second half of PhD as a guiding light and a moral compass.

Chapter 1

INTRODUCTION

1.1 Motivation for Scalable Manycore Architecture

An ideal, scalable architecture should be able to adapt to improvements in process technology, while delivering commensurate processing power and energy efficiency. Earlier single-core processors benefited from an increased gate density by dedicating more resources toward advanced microarchitectural features, such as out-of-order and speculative execution, to prevent stalls and tolerate latency. However, scaling up performance with these features faces two main limitations.

First, these features become less useful for programs with limited instruction-level parallelism [185]. Out-of-order execution may help improve single-threaded programs with moderately complex control flow and data dependencies, but accelerating compute-intensive, data-parallel workloads tends to be more limited by the number of functional units and instruction issue rate. Second, scaling the hardware structures that enable these features (e.g. instruction window, multi-ported register file, reservation station) becomes increasingly expensive in terms of area and logic complexity [133].

As the logic level deepens, additional pipeline stages must be added to keep critical path delay under control. However, pipelining introduces further side effects, such as increased execution latency and higher branch misprediction penalties [155]. Growing hardware complexity makes scaling in newer technology less profitable [8], as the distance that signals can traverse within a scaled clock cycle shrinks [80]. The percentage of area (and energy) devoted to functional units that perform useful computations (e.g. floating-point unit) continues to decline [76]. Furthermore, verification effort often takes more time than the design process itself [57].

Manycore architectures aim to reverse these trends by using a large number of simple, efficient cores for independent, parallel processing, rather than a small number of large,

complex cores optimized for single-thread performance. Such cores typically have single-issue and in-order execution pipelines to minimize hardware cost. Building a system from these cores involves replicating identically shaped tile blocks interconnected by Network-on-Chip (NoC) routers. Due to their simplicity, tiled manycore architectures can be designed, implemented, and verified by a small engineering team, reducing both the time and cost required to deploy a new system [11, 128].

An array of these scalar processors executing in the SPMD (Single-Program, Multiple-Data) model is generally easier to program for workloads with irregular control flow and data access (Figure 1.1) than vector or SIMD (Single-Instruction, Multiple-Data) architectures [111]. In contrast, SIMT-based (Single Instruction, Multiple Threads) GPUs – whose massive parallelism relies on a multithreaded frontend fetching instructions for multiple lanes of functional units – perform poorly on programs with irregular control flow and memory access patterns, primarily due to the restriction that all threads must progress in lockstep [37, 129]. Unsurprisingly, prior research on GPU acceleration of parallel algorithms has largely focused on workarounds for the programmability constraints imposed by this execution model (e.g. branch divergence, uncoalesced access) [38, 81, 113, 176].

Hardware accelerators represent the least flexible end of the programmability spectrum. Specialization is relatively straightforward when the workload on a chip is expected to remain largely unchanged throughout its product lifetime (e.g. a Bitcoin miner). While accelerators can boost performance and energy efficiency by orders of magnitude, they are less attractive for designing flexible, parallel architectures because they sacrifice programmability and can quickly become obsolete when new workloads appear. Although accelerators can be configurable to some extent, they may lose efficiency with even slight changes in input parameters and data formats.

As new applications become more complex and diverse, the fraction of workloads (f) that *cannot* be accelerated by a particular specialization grows over time. By Amdahl's Law [14], the maximum speedup that can be achieved through specialization is bounded by $\frac{1}{f}$, assuming an ideal speedup for the accelerator. If we also account for the fraction of chip area allocated to the accelerator (A) – effectively reducing the area available for general-

```
#define THRESHOLD 100
for (i = 0; i < N; i++) {
  found = false;
  j = B[i];
  while (!found && j < B[i+1]) {
    len = 0;
    curr = C[j];
    while (curr != -1 && len < THRESHOLD) {
      curr = C[curr];
      len++;
    }
    if (len >= THRESHOLD) {
      found = true;
      break;
    }
    j++;
  }
  result[i] = found;
}
```

Figure 1.1: Data-level parallelism with highly irregular control flow and data access.

purpose computation – the maximum attainable speedup is further diminished to $\frac{1-A}{f}$. Under this assumption, for accelerators to meaningfully impact end-to-end performance, the fraction of workload that *can* be accelerated ($f' = 1 - f$) must exceed A.

The trends unfavorable to specialization are already apparent in neural networks, where non-GEMM operators have become increasingly diverse and frequent [65]. Modern SoCs now integrate one or more fixed-function accelerators alongside a small number of general-purpose processors, with the accelerator tiles occupying significant chip area [53,60] (Figure 1.2 left). If these trends continue, the computational bottleneck may inevitably shift back to the non-accelerable portion of the application.

Manycore architectures, in contrast, align closely with the growing demand for flexible, rather than specialized, parallel hardware that can support innovation in next-gen AI, ML, and other emerging domains [49, 82, 91, 108, 124, 183]. A large array of flexible cores can be regrouped and reprogrammed as computational demands shift (Figure 1.2 right). A primary objective of this thesis is to extend the manycore concept to the extreme, building a scalable and programmable system with ASIC-like compute density.

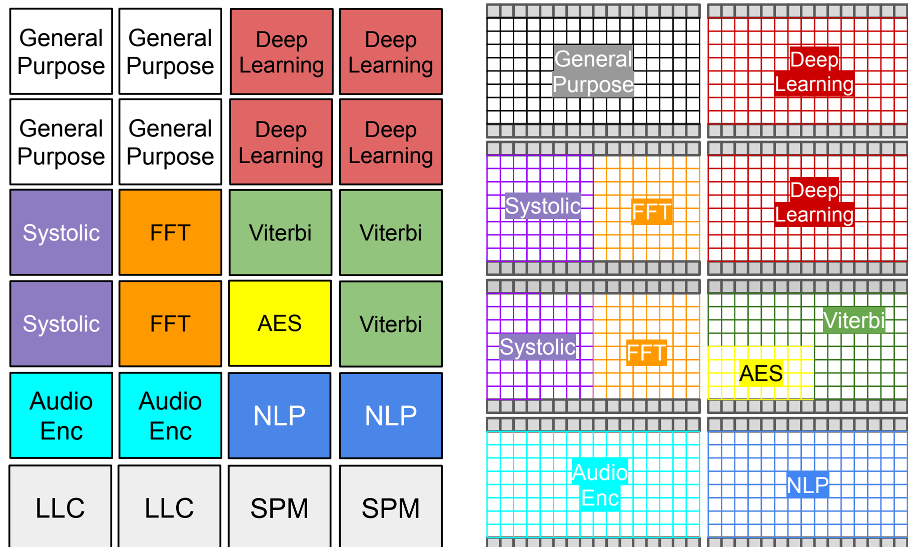


Figure 1.2: A SoC based on a variety of fixed-function accelerator tiles vs. a flexible manycore array with a similar area.

However, many architectural mechanisms inherited from large, complex cores – designed to optimize single-thread performance and support cache-coherent shared memory programming – have limited prior manycore designs from achieving the compute density and scalability required to compete with other parallel architectures, such as GPGPUs. This thesis presents *HammerBlade Manycore*, the result of integrating numerous novel ideas across multiple domains of parallel architecture, including VLSI resource organization, on-chip networks, memory hierarchy, memory-level parallelism, address mapping and translation, synchronization, and thread management.

1.2 Motivation for Scalable Network-on-Chip

Network-on-Chip (NoC) has become one of the most critical components in modern parallel architectures. Today's parallel processors and accelerators, built to handle data-intensive workloads, demand far more bandwidth than ever before. At the same time, the scale of on-chip networks has continued to grow – designs with over 100K cores on a single, full-reticle silicon die no longer seem far-fetched. This raises an important question: how should we

design large-scale NoCs for future chips? To answer this, we must first recognize that NoC concepts originally emerged *bottom-up*, driven by the physical-design challenges that chip engineers have faced, such as wire delay [161] and limitations with VLSI CAD tools [48]. Yet many of the prior NoC proposals appear to have overlooked these constraints. This thesis renews attention to physically motivated NoC designs by emphasizing two fundamental requirements for large-scale NoCs: physical scalability and tileability.

Physical scalability refers to an ability to increase network size without compromising cycle time or wire congestion. To achieve that end, certain aspects of NoCs must remain constant as the network scales. First, router complexity (e.g. router radix, crossbar, buffer size) should remain constant so that the area and critical path delay do not grow unbounded. Second, maximum wire lengths between nodes should not grow in order to keep the wire delay under control. Lastly, the number of links to route between nodes should not be fixed to avoid exhausting wiring tracks or having to narrow channel widths. Figure 1.3 shows examples of low-diameter NoC topologies with non-local links. High-radix topologies (Figure 1.3 c,d), such as MECS [71] and Flattened Butterfly [102], would be the examples of NoCs that have been regarded as the state-of-the-art, but not physically scalable.

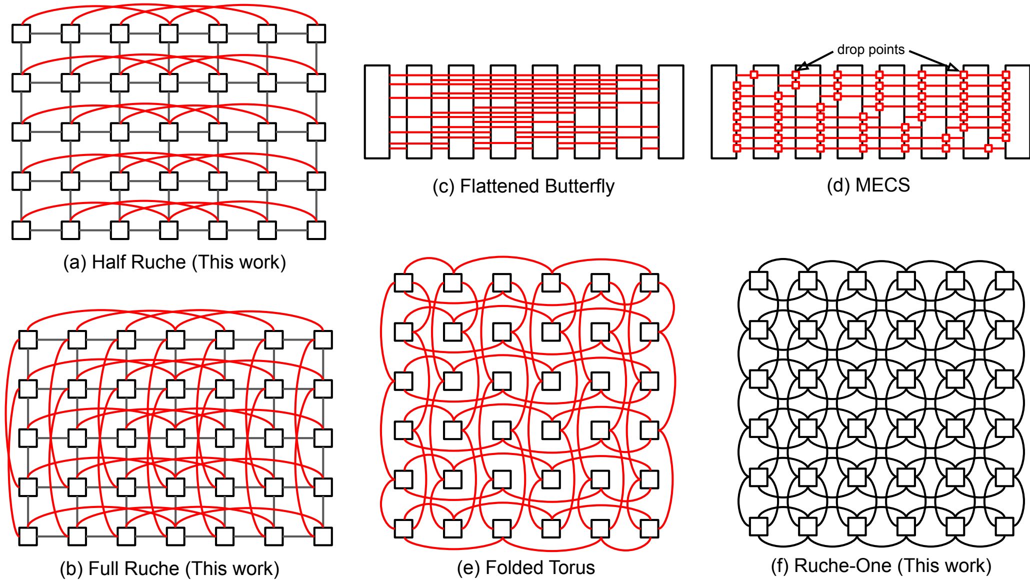


Figure 1.3: Low-diameter NoC topologies with non-local links.

The second important property for a large-scale NoC is *tileability*. The idea of replication, whether at a transistor-level or at a tile-level, is very fundamental to VLSI. For practical reasons, it is important to have an architecture that can be decomposed into replicable tiles. VLSI CAD algorithms (e.g. placement, routing, gate sizing) are inherently NP-hard, because they aim to optimize multiple conflicting objectives. When a CAD tool is given too large or too complex of a problem, it may fail to converge. Chip design using CAD tools is a highly iterative process, so if each iteration takes a day or even a week, it decimates productivity and the chances of meeting the tapeout deadline. Furthermore, these CAD tools are proprietary and very expensive, costing up to $300K per license. Therefore, uniform NoC routers that allow replicable tiles are desirable. High-radix topologies (Figure 1.3 c,d) require a unique tile for each core due to their non-uniform routers and irregular wiring.

Unsurprisingly, 2-D mesh has been the predominant on-chip network topology in most architectures, as it is both physically scalable and tileable. Unlike high-radix topologies [5, 71, 102], 2-D mesh features local and regular wire routing between tiles. This allows every tile to have an identical shape that can be replicated in an array of any size to fill the available silicon area – even non-power-of-2, as seen in 6×6 ET-SoC-1 [52] and 18×20 Tesla DOJO [159]. Additionally, 2-D mesh can be implemented using a standard-cell-based, automated CAD flow, making it relatively straightforward to adapt an existing design to a more advanced process technology.

Despite its low design complexity, 2-D mesh suffers from poor latency and throughput scaling, as noted by many prior studies [5, 23, 102, 116, 152]. As the network size grows, the network diameter becomes substantial, and the bisection bandwidth becomes a critical bottleneck, forcing cores to inject fewer packets [152]. Energy inefficiency also degrades with the increased number of routers and added hops [35]. Furthermore, packets travel at a speed significantly slower than the raw speed of wire.

Several approaches have been proposed in literature to address the scalability limitations of 2-D mesh. However, these solutions often trade one problem for another and are based on assumptions that are either outdated or no longer valid in modern architectures.

Concentration co-locates a number of cores (usually 2 to 4) within a single tile to share

a network router with a multiplexed input [23]. This approach assumes that the traffic injection rate of each core is low, so the probability of conflicts at the network input also remains low. While this assumption may hold for multi-core cache-coherent networks – where a processor issues a request and waits – it does not apply to modern parallel processors optimized for data-intensive kernels, where word-level packets are sent and received every cycle in bursty, continuous streams. Overall, the aggregate injection bandwidth is also reduced.

Concentration reduces the total number of networks nodes, thereby lowering network latency and overall energy consumption, but it also halves the bisection bandwidth. In order to compensate for this lost bisection bandwidth, a typical mitigation has been to double the network channel width. However, previous NoC studies incorrectly assume that the underlying core architecture can efficiently exploit a wider channel. In reality, a wider channel requires additional logic for serialization and deserialization to match the intrinsic ingress and egress bandwidth of the endpoint (e.g. a number of words that can be read from SRAM each cycle). This mismatch not only introduces serialization latency, which negates the latency reduction benefit of concentration, but also increases area overhead by necessitating more buffers to prevent network stalls. Alternatively, channel bandwidth could be matched by widening the SRAM and processor datapaths, but this enlarges the tile size and, ironically, does not improve the network bandwidth in absolute physical terms (e.g. Tera-bit/s/mm).

Even without using concentration, increasing the channel width is not a scalable method for boosting bandwidth in 2-D mesh, as it linearly increases the area required for crossbars and buffers. Furthermore, as observed in previous parallel processor designs, 2-D mesh networks still underutilize the available wiring tracks between tiles, achieving only about 24% utilization [140]. Using a concentrated router with a wide channel typically requires a larger, slower crossbar, which can increase the number of cycles per network hop or force a reduction in clock frequency to meet timing constraints. This, in turn, negates the latency benefits provided by concentration. Placing the NoC in a slower clock domain suffers from similar bandwidth-mismatch issues.

Ruche Networks (Figure 1.3 a,b) address the scalability challenges of 2-D mesh by aug-

menting it with regular, long-range links [93, 95, 131]. While retaining all the desirable properties of 2-D mesh to maintain physical scalability, Ruche Networks provide an architecturally flexible and cost-effective mechanism to scale bisection bandwidth without requiring an overhaul of the underlying processor architecture. This thesis presents a detailed evaluation of its network performance and energy efficiency compared to other NoC topologies.

1.3 Contributions

This thesis makes the following contributions:

- **HammerBlade (HB)**, a novel manycore architecture, achieves unprecedented compute density and scalability by replacing outdated, inefficient architectural mechanisms with simpler yet more effective alternatives. Its flexibility is demonstrated using a parallel benchmark suite that captures a wide range of computation and communication patterns.

- **Ruche Networks** are physically scalable and tileable NoCs, which cost-effectively leverage otherwise unused on-chip wiring resources to improve both network performance and energy efficiency.

- **The 12-nm, 2048-core, 99 mm^2 HammerBlade ASIC** sets multiple records. It is the first to demonstrate the feasibility of Ruche Networks in a large-scale, high-frequency manycore processor. It achieves the highest single-chip RISC-V instruction throughput and CoreMark score.

1.4 Thesis Outline

This thesis is organized as follows:

Chapter 2 describes the HammerBlade Manycore architecture and evaluates it using the parallel benchmark suite. Chapter 3 goes into an in-depth discussion and evaluation of Ruche Networks. Chapter 4 presents the 12 nm implementation of Ruche Networks in a 2048-core HammerBlade ASIC. Chapter 5 concludes the thesis and outlines future research.

1.5 Relation to Published Works

This thesis contains materials from published or submitted papers, where I am named as the first author or one of the co-first authors.

- "Scalable, Programmable and Dense: The HammerBlade Open-Source RISC-V Manycore" [94], *Published in ISCA 2024*.

- "Ruche Networks: Wire-maximal, No-fuss NoCs" [93], *Published in NOCS 2020*.

- "Evaluating Ruche Networks: Physically Scalable, Cost-Effective, Bandwidth-Flexible NoCs" [95], *Published in ISCA 2025*.

- "HammerBlade in Silicon: A 12-nm 2048-core RISC-V Manycore SoC with Ruche Networks" [61], *Published in TVLSI 2026*.

1.6 Open-source Research Artifacts

Research artifacts, created and used by this research, are contributed to the following public, open-source repositories.

- **BSG Manycore** – RTL and testing infrastructures for HammerBlade Manycore.
  `https://github.com/bespoke-silicon-group/bsg_manycore`

- **BaseJump STL** – RTL for basic hardware building blocks.
  `https://github.com/bespoke-silicon-group/basejump_stl`

- **HammerBench** – Parallel benchmarks used for HammerBlade evaluation.
  `https://github.com/bespoke-silicon-group/hb_hammerbench`

- **bsg_replicant** – C++/Verilog cosimulation infrastructure for HammerBlade.
  `https://github.com/bespoke-silicon-group/bsg_replicant`

- **hb_utilization** – Python scripts for parsing, summarizing, and visualizing performance metrics.
  `https://github.com/tommydcjung/hb_utilization`

- **hb_bigblade** – TCL scripts used to build the 12 nm HammerBlade ASIC.
  `https://github.com/bespoke-silicon-group/hb_bigblade`

1.7 Acknowledgments

Portions of this work were partially supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement numbers FA8650-18-2-7863 and FA8650-18-2-7856; NSF grants SaTC-1563767, SaTC-1565446 and NSF Award 2118628. This work intersects and leverages research and infrastructure created by the members of the Bespoke Silicon Group, spanning across accelerators ([15,19,31,36,59, 67,68,69,98,132,134,136,137,138,149,150,158,181,182,188,194,197]), ML ([180]), ASIC Clouds ([99, 100, 119, 165, 167, 170, 174, 189]), open source hardware ([55, 147, 171, 172]) RISC-V ([10, 46, 50, 92, 94, 112, 123, 139, 142, 143, 145, 146, 179, 196]), Network-on-Chips ([93,101,140,169,199]), security ([13,18,39,40,78,79]), benchmark suites ([21,103,175]), dark silicon ([32, 67, 70, 163, 164, 166, 181]), multicore ([20, 44, 45, 72, 73, 74, 101, 139, 160, 161,162,168,169,173,184]), compiler tools ([7,17,62,63,64,86,87,88,89,193]) and FPGAs ([21,66,83,148,198]).

Chapter 2
HAMMERBLADE MANYCORE

2.1 Architecture

This section describes the architectural features and mechanisms of HammerBlade Manycore.

2.1.1 Cellular Manycore

Figure 2.1 shows the resource organization used in conventional manycore architectures. Generally, a larger pool of cores and on-chip storage is desirable to enable greater parallelism and data reuse. *Logical scalability* refers to the degree to which network resource consumption and latency for a basic memory operation remain controlled as the core count grows.

Earlier manycore architectures were primarily *flat manycores* (Figure 2.1a), consisting of a large, uniform tile array interconnected by a 2-D mesh, with memory or I/O interfaces positioned at the edges of the chip. For nearest-neighbor communication, this design scales effectively; however, for random or all-to-edge communication, as typical in DRAM accesses, network resource consumption grows quickly. For $N \times N$ mesh, each tile can only inject packets at an average rate of $2/N$ per cycle before the edge network channels become fully saturated [152].

Later, *hierarchical manycores* (Figure 2.1b) were introduced to address this logical scalability limitation by grouping cores into clusters connected in a rigid network hierarchy. Within each cluster, cores are often connected via low-latency crossbars to share common resources. While crossbars provide uniform latency between all cores in a cluster, scaling cluster size is limited because the crossbar area grows quadratically with the number of inputs. These clusters are then interconnected by the network through another level of hierarchy, usually with wider channels. This approach works well for transferring cache-line

sized data but is inefficient for fine-grained, random access common in graph processing and sparse datasets. Sharing data between clusters also requires coordination between L1 and L2 caches, introducing additional cache-coherency complexity and overhead.

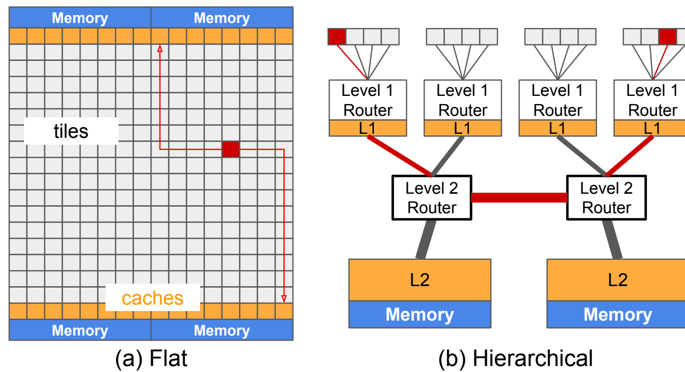


Figure 2.1: Resource organization in conventional manycore architectures.

HammerBlade adopts a distinctive resource organization, called *Cellular manycore*, consisting of an array of replicable macro units, referred to as *Cells*, as shown in Figure 2.2. Each HB Cell is a 2-D array of compute tiles and two 1-D arrays of cache banks. All compute and cache tiles are interconnected via the Ruche Network [93, 95, 131], a 2-D mesh augmented with uniform long-range links that pass through tiles. These additional links increase bisection bandwidth and reduce the network diameter without compromising the mesh's ability to map efficiently onto a silicon substrate. As the length of these links increases, more wiring passes through the tiles, effectively utilizing otherwise unused VLSI wiring resources with minimal hardware overhead. In the HB implementation, Ruche links that skip three tiles horizontally boost the peak bisection bandwidth by 4×, compared to conventional 2-D mesh.

HammerBlade employs a flat, uniform network hierarchy, with the Ruche Network extending beyond individual Cell boundaries to connect adjacent Cells. This design allows network packets to reach any location on the chip without altering the packet format, avoiding additional hardware for serialization and deserialization and eliminating arbitrary network bottlenecks. By default, the cores within a single Cell cooperate on a common prob-

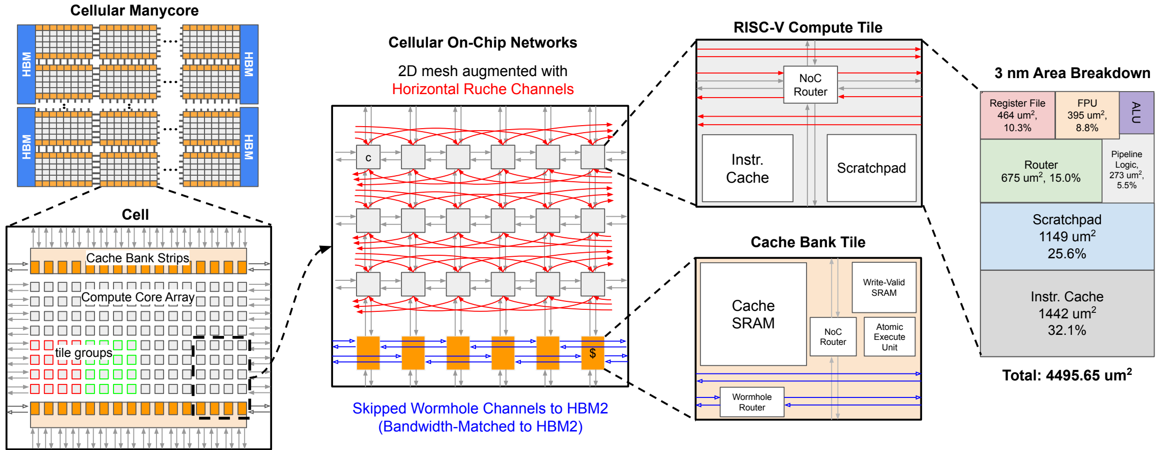


Figure 2.2: Overview of HB Cellular manycore architecture.

lem using the Cell's shared cache banks. In Section 2.2, we describe how PGAS (Partitioned Global Address Space) is configured to support this affinity.

HammerBlade's cache hierarchy is also flat. The cache banks serve as the last-level cache before DRAM, with each bank independent and mapped to an exclusive range of DRAM address space to avoid coherence issues. The cache banks implement a *write-validate* policy [90], as commonly used in GPUs [97], which eliminates unnecessary DRAM reads for cache write misses. This is a useful feature for workloads that write results in large blocks. The cache banks are also *non-blocking* and provide sufficient buffering to ensure that primary and secondary misses are drained out of the network, allowing other hit requests to proceed without delay. All miss status holding registers (MSHR) are consolidated at the last level of cache hierarchy and shared among all tiles for better utilization, rather than being scattered throughout the hierarchy. Additionally, RISC-V cores can perform remote atomic operations (e.g. atomic-swap, OR, add) on these cache banks with acquire/release semantics [186], enabling the synchronization primitives required for efficient parallel programming.

Each strip of cache banks incorporates 1-D wormhole flow-controlled channels that are used exclusively for cache refill and eviction traffic. As shown in Figure 2.2, each cache bank strip includes multiple pairs of *skipped* channels, which improve fairness and reduce latency for banks located near the middle of the strip. The skip distance and channel width can be

tuned to match the available HBM2 bandwidth.

2.1.2 HammerBlade RISC-V Core

Each HB core is a high-frequency, area-optimized, single-issue, in-order, 5-stage, RISC-V processor with atomic and floating-point extensions. Each core contains 4 KB scratchpad memory (SPM) and 4 KB instruction cache (icache) (Figure 2.3). Smaller SRAMs enable a greater number of cores at the cost of reduced bit density. We select the size that maximizes core count while remaining sufficient to hold sizable kernel inner loops. This design choice is a major contributor to HB's improved compute density. In contrast, a Piton tile contains 16 KB of instruction cache and 80 KB of multi-level data cache, yet still provides only single-issue throughput [121].

SPMD's independent program execution model is a key advantage over SIMT, in which threads must execute in lockstep. HB cores minimize the associated cost in their frontend to support this model efficiently. The core employs a simple branch predictor with a two-cycle misprediction penalty. The core predicts 'taken' for a backward branch and 'not taken' for a forward branch, which is sufficiently accurate for a wide range of data-parallel kernels and inexpensive to implement. The icache is *direct-mapped* with four-instruction cache lines and 12-bit tags. This provides 16 MB of program space, which is practically 'unlimited' for many data-parallel kernels. Tag bits and instructions are combined in a single SRAM block, incurring only a 25% area increase compared to the 4 KB instruction memory used in Celerity [50]. Furthermore, for branch and jump instructions stored in the icache, the lower bits of the target address are precomputed and embedded in the immediate field, effectively serving as a zero-area branch target buffer (Figure 2.4).

HammerBlade uses Berkeley HardFloat [77] for floating-point arithmetic with some modification. The fused multiply-add (FMA) unit was modified to share its multiplier with the integer multiply unit to reduce area. We also optimized the count-leading-zero (CLZ) logic used in the FMA and conversion units to shorten the critical path and improve timing.

HB provides more efficient mechanisms for memory-level parallelism (MLP) than other throughput-oriented architectures. HB cores are *explicitly interfaced* with the on-chip net-

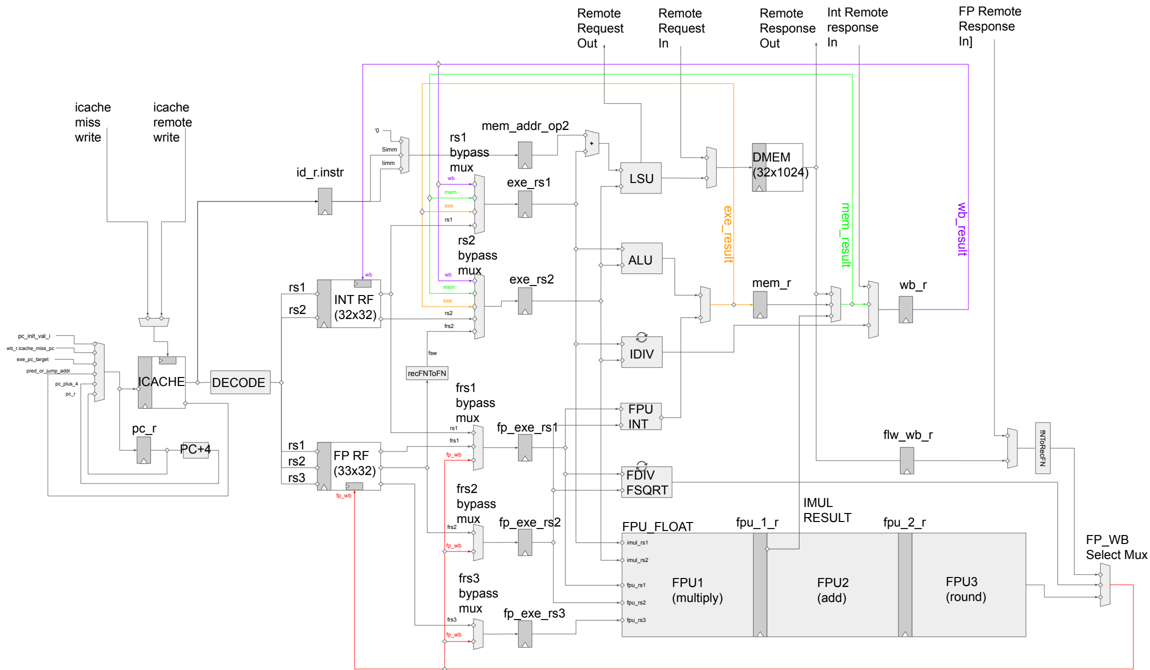


Figure 2.3: HB RISC-V core pipeline schematic.

work, such that each RISC-V memory operation targeting a remote resource is directly converted into a network packet. In most architectures, data movement is handled implicitly through cache refill and evict mechanisms at block-level granularity. For fine-grained accesses, this may waste on-chip bandwidth and storage by transferring unnecessary data.

HB's remote memory operations are *non-blocking*, enabled by a bit-vector-based scoreboard that occupies less than 1% of tile area. In contrast, GPU multithreading relies on a massive register file to keep every thread context readily available, which amounts to a significant fraction of GPU area and energy [109]. A single HB tile can launch as many as 63 outstanding load requests (equal to the number of FP and integer registers), each of which may generate a cache miss and a DRAM request at the shared LLC banks, providing an ample source of MLP. While store requests can be injected without any restrictions, a configuration register is available to throttle the number of outstanding requests and prevent network overload.

Remote loads can be pipelined through the network, creating a separation between when the load requests are launched and when their results are consumed, thereby hiding latency (Figure 2.11). In contrast, SIMD execution models impose the restriction that all memory requests generated by an instruction must complete before execution can proceed. Uncoalesced GPU memory accesses must be replayed over multiple cycles, further degrading pipeline utilization [37, 129].

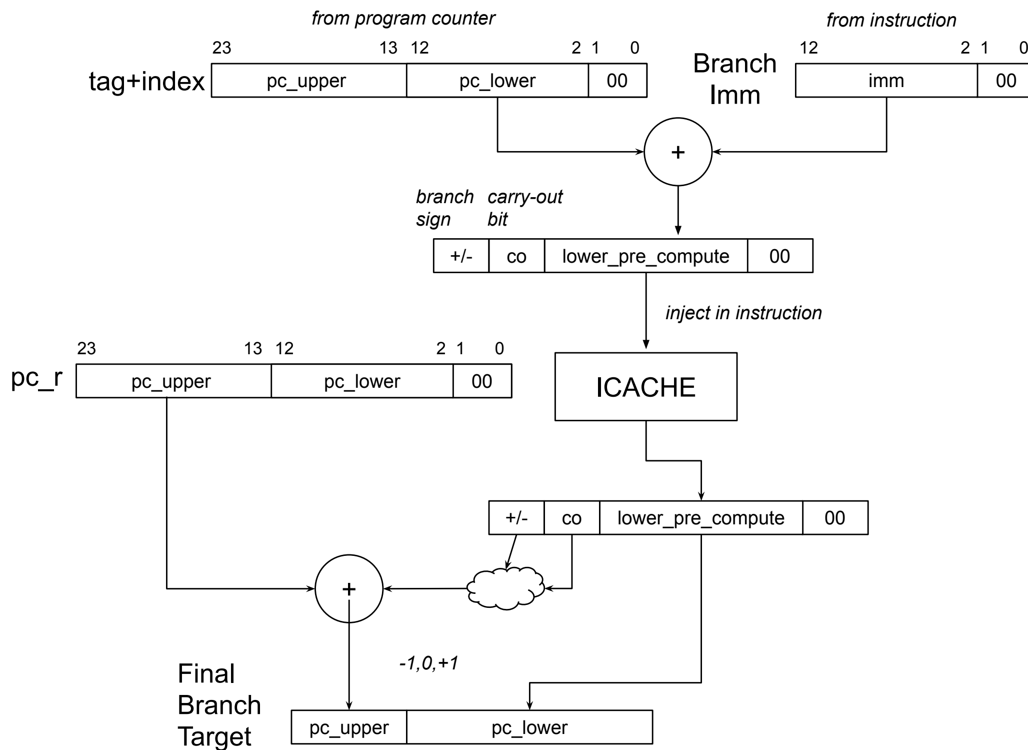


Figure 2.4: Instruction cache branch target buffer.

2.1.3 HB Network-on-Chip

HB's simplified NoC design plays a crucial role in maximizing compute density and network scalability. It represents a major departure from prior manycore designs, which often integrated multiple NoCs tailored to various traffic types. For example, Epiphany-V tiles [127] include routers for both on-chip communication and off-chip (DRAM) traffic. In contrast,

HammerBlade separates off-chip routers from compute tiles, keeping tile area low.

Raw [173] and Tilera [30] implement fast register-to-register scalar transfer networks, but their routing rules must be pre-configured at compile-time in a large instruction buffer, using a specialized compiler [110]. Raw also includes a dynamic wormhole-routed message network; however, because it is prone to deadlock, an additional identical network is required for deadlock recovery [161]. Tilera further provides dedicated networks reserved exclusively for I/O and kernel-level traffic to isolate them from user code [187]. Since many workloads do not fully utilize all of these specialized networks, substantial network resources often remain unused.

HammerBlade addresses these issues by adopting a NoC design that provides only the fundamental functionalities required for scalable, efficient communication, eliminating unnecessary complexity and underutilized resources.

2.1.3.1 HB Global Network

All core traffic travels on two physically separate Half Ruche networks: one for requests, and the other for responses. Each RISC-V memory operation is injected into the network as a single-flit packet containing the destination (X,Y) coordinate and an offset address, which is translated from the memory address using the PGAS mapping described in Section 2.2. Each request packet also contains the source coordinates and opcodes.

Because the frequently accessed cache banks are located along the top and bottom edges of each Cell, static dimension-ordered routing (DOR) using X→Y for the request network and Y→X order for the response network maximizes the network throughput [6]. As the size of the Cell grows, horizontal channels that cross the bisection become the primary bottleneck. The Half Ruche network [93], which augments the mesh with additional horizontal channels, alleviates this limitation.

Unlike the channels that interconnect clusters in hierarchical manycores, the Cellular manycore's globally uniform network efficiently transfers sparse, random data between Cells. In HB, the wiring density (e.g. *bits per mm*) is 21.6× higher horizontally and 7.0× higher vertically than in ET-SoC-1, a representative hierarchical manycore that employs a 1024-

bit-wide 2-D mesh network [151].

Figure 2.5 plots the utilization of bisection links between two Cells over time during the transfer of 1 MB of sparse data to random locations in the cache banks of an adjacent Cell, in both the vertical and horizontal directions, assuming no cache miss. The Cellular manycore's globally uniform network and word-access per packet enable high bandwidth utilization (80∼90%) even for completely sparse, random data transfer between Cells – performance that is unattainable with the 1024-bit wide channels used in hierarchical manycores.

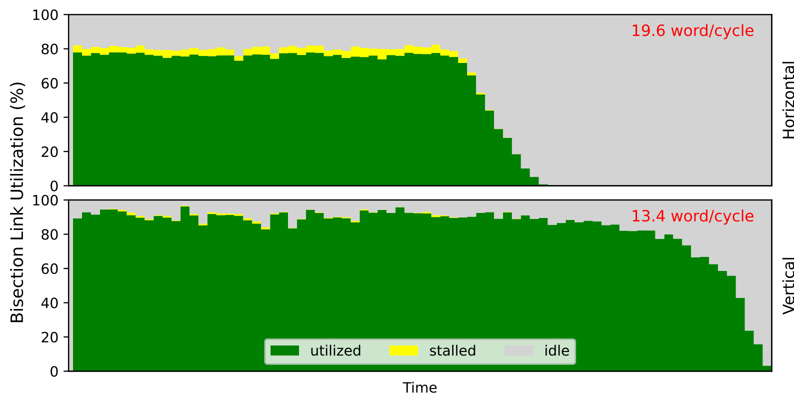


Figure 2.5: Bisection link utilization between two Cells over time.

While *word-access per packet* is highly effective for sparse, random data transfers, it can easily overwhelm the network when many tiles simultaneously read large sequential blocks. To address this, HB implements *Load Packet Compression* to utilize on-chip bandwidth more efficiently. When the processor detects consecutive remote load instructions in the instruction stream that access sequential addresses and target the same network destination, it coalesces them into a single packet. Instead of transmitting four separate packets —- each carrying a register ID and address —- the core sends one packet containing four register IDs and a single base address. Combined with the Ruche Network, this mechanism significantly alleviates the bisection bottleneck and enables scalable operation with larger Cells.

2.1.3.2 Hardware (HW) Barrier

HammerBlade adopts the Bulk-Synchronous Parallel (BSP) programming model [177], in which data ownership (e.g. between readers and writers) is exchanged in *bulk* between successive program phases, separated by memory fence and barrier synchronization. Unlike cache-coherent shared-memory models, the BSP model scales better and enables more hardware-efficient implementations by avoiding fine-grained coherence.

HB Compute tiles can synchronize with low latency using a 1-bit-wide network that shares the same Half Ruche topology (Figure 2.6). The HW barrier is controlled by two configuration registers: (1) the input directions (if any) from which a tile must receive barrier signals before forwarding its own signal, and, (2) the output direction to which the tile sends its barrier signal upon joining the barrier. Using these configurations, a group of cores can form a tree-like structure [191], with barrier signals converging at a single root node.

Once the signal reaches the root, a wake-up signal is propagated back to the leaf nodes. With the Ruche links that skip three tiles horizontally, a barrier signal from the most distant tile can reach the root in only eight cycles. The HW barrier costs extremely low area overhead (less than 1% of tile area), and its latency scales far better with core count than the software-implemented barriers (Figure 2.6).

Invoking the HW barrier from software requires only two custom RISC-V instructions: one to activate the barrier signal (`barsend`) and one to wait for synchronization (`barrecv`). In contrast, software barriers can require tens to hundreds of instructions. Given HB's small 4 KB icache, reducing instruction footprint is a major win.

2.2 Parallel Programming Model

This section describes HB's programming model and the PGAS mapping that impose a logically defined hierarchy on flat hardware. We present the basic programming primitives upon which higher-level abstractions can be built. We introduce the parallel benchmark suite used for evaluation.

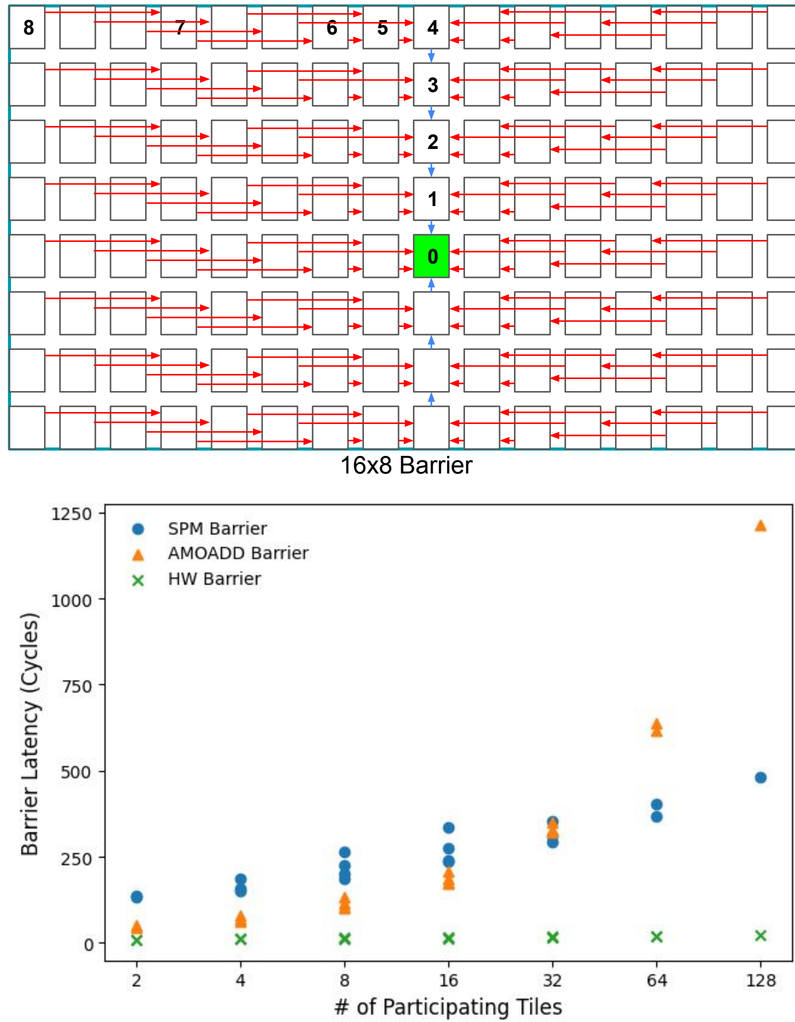


Figure 2.6: HW Barrier Network and latency comparison with SW-implemented barriers.

2.2.1 HB Programming Model

At the Cell-level, HB's programming model follows SPMD, which is well-suited for irregular data access and control flow. Within a Cell, the tile array can be logically divided into smaller rectangular units called *tile groups* (Figure 2.2), enabling finer-grained thread management compared to SIMT architecture, where threads are organized into coarse-grained units such as warps or wavefronts.

Choosing a tile group size involves a trade-off between latency and throughput. For

certain workloads, latency can be reduced by assigning more tiles to a single group. If latency is less critical, multiple smaller tile groups can execute independent tasks in parallel, improving overall throughput. However, some workloads, such as graph and sparse applications, may not benefit linearly from larger tile groups. In such cases, smaller tile groups enable task-level parallelism.

In this context, a *task* refers to a coarse-grained unit of computation with data-dependent control flow that operates on a shared data structure and can execute in parallel. Examples include executing different queries on a common graph or multiplying a stationary sparse weight matrix with multiple activation matrices. Tile groups can leverage the reconfigurable HW Barrier to synchronize efficiently within the groups. At the chip-level, each Cell can either run the same kernel on a portion of a large problem or execute different kernels in a producer-consumer pipeline.

Kernel code can be written in C/C++ and compiled with RISC-V GNU/LLVM toolchain [2], with optional support from domain-specific languages [36, 44, 45]. The host runtime is responsible for memory management and data movement, and it launches kernels on the Cells by passing pointers to the relevant data. Tiles are assigned unique IDs, analogous to block and thread IDs in CUDA [126].

Kernels execute within the PGAS, which is divided into five major address spaces to reflect the logical memory hierarchy in the HB system (Figure 2.7). This organization allows programmers to place data in the most appropriate memory level to maximize *physical locality*, keeping data close to the processing tiles. PGAS mapping also defines how tiles collaborate and share data within HB's physically flat cache and network hierarchy.

Translation from a virtual address to a network address is performed using low-cost combinational logic, avoiding expensive hardware such as TLBs (translation lookaside buffers). A few of the upper address bits determine the major address space to which an address belongs. Within each address space, the X and Y coordinates of the destination tiles are either directly encoded in the address or generated via an address-hashing scheme.

1. *Local SPM* provides access to a tile's local 4 KB scratchpad. Addresses (`0x0` ∼ `0xfff`) are private to a tile. A program stack is allocated here. A tile can copy blocks of data

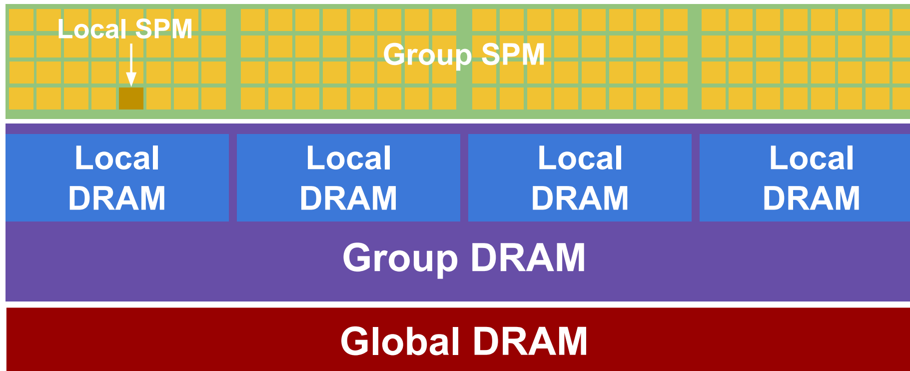


Figure 2.7: HammerBlade PGAS Mapping.

	b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]	b[10]	b[11]	b[12]	b[13]	...
a[0]	1			1		1	1			1	1	1	1	1	
a[1]		1			1		1	1			1	1	1	1	
a[2]	1		1	1			1	1	1	1	1				
a[3]		1		1	1			1	1	1	1	1			
a[4]			1		1	1			1	1	1	1	1		

Figure 2.8: IPOLY hashing table for 32 banks.

from DRAM into Local SPM for fast, low-latency processing.

2. *Group SPM* provides a shared address space that allows tiles to remotely access other tiles' scratchpads. The address encodes both the relative X,Y coordinates of the target tile and the SPM address offset. This space is particularly useful for spatially distributing data when communication patterns are structured and predictable, such as nearest-neighbor or systolic-array computations.

Setting up a tile group is straightforward. Each tile has a configuration register specifying the *tile group origin* coordinates. All tiles in the same group set this register to the coordinates of the top-left tile. Using this register with the relative coordinates encoded in the address, the absolute coordinates of any tile can be calculated. This approach allows tile groups to be placed anywhere in the manycore array without

requiring kernel recompilation.

3. *Local DRAM* provides access to the portion of DRAM exclusively allocated to each Cell, where most computation occurs. This address space is private to the Cell but shared among its constituent tiles. Accesses are mediated through the Cell's local cache banks. *Regional IPOLY hashing* [144] pseudo-randomly distributes addresses across cache banks at cache-line granularity. This prevents the partition camping problem caused by 2^n-stride access patterns, which are common in many parallel applications [12, 97].

   Figure 2.8 illustrates a simple method to generate an IPOLY hashing table. For 2^N banks, if N is odd, the seed in the first column is set to `101`; if N is even, it is set to `11`. Each subsequent column is generated by shifting the previous column down by one row. If the most significant bit (MSB) of the current column is 1, the shifted result is XORed with the first column. This process is repeated until enough columns are generated to cover the size of the target address space.

4. *Group DRAM* provides access to the Local DRAM of another Cell, enabling efficient broadcasting or gathering of results across Cells before the next program phase. As shown in Figure 2.5, Cellular manycore's inter-Cell connections support high-bandwidth transfers of both sparse and sequential data. Figure 2.9 illustrates host code for a producer-consumer model, where a producer Cell uses a Group DRAM pointer to write results directly into a consumer Cell's Local DRAM space, minimizing unnecessary data movement.

   Figure 2.10 shows a kernel routine that synchronizes a producer and a consumer after a data transfer completes. It uses custom load-reservation instructions, where `lr` loads and creates a reservation on a local SPM, and `lr_aq` stalls until the reservation is broken by a remote store sent from another tile to the reserved address.

5. *Global DRAM* is shared by all tiles on the chip, and is distributed across all cache banks using a custom hash function. This address space provides a convenient location

for Cells to combine partial results at the end of kernel execution. The host can use this space to transfer large blocks of data to the chip, taking advantage of the full DRAM bandwidth. On very large chips, such as a 100K-core design, all-to-all communication can become unsustainable. In such cases, the chip can be divided into *grids*, a rectangular group of Cells, to introduce an additional layer of locality. The most significant bits of an address select the grid, while the remaining bits are used for hashing within the grid.

```
// HOST PROGRAM;
// Instantiate two Cells at (X,Y) location (0,0) and (1,0)
auto cell0 = new Cell(0,0);
auto cell1 = new Cell(1,0);
// Load different kernels on two Cells;
cell0.load_kernel("producer.riscv");
cell1.load_kernel("consumer.riscv");
// Allocate memory on Cells' Local DRAM;
float* input0, input1, output2;
cell0.malloc(4096*sizeof(float), &input0);
cell1.malloc(4096*sizeof(float), &input1);
cell1.malloc(4096*sizeof(float), &output2);
// Launch kernels;
// group_dram() produces a pointer to
// Cell1's Local DRAM in Group DRAM space;
cell0.launch(input0, cell0.group_dram(cell1,input1));
cell1.launch(input1, output2);
```

Figure 2.9: A host program launching producer-consumer kernels on two Cells.

2.2.2 Parallel Benchmark Suite

To demonstrate the parallel programmability of HB, we introduce a parallel benchmark suite, inspired by Berkeley's parallel computing dwarfs [16]. *Dwarfs* represent a broad set of parallel computation and communication patterns that have persisted through time and are expected to remain relevant in the foreseeable future. Covering these bases lends

```
// 'alert' is an address in local scratchpad, initialized to 0;
void tile_group_wait (int* alert) {
  // tile0 waits for a remote store to 'alert' from another Cell,
  // before joining the barrier with other tiles;
  if (__tile_id == 0) {
    while (1) {
      // loads and creates reservation;
      int tmp = lr(alert);
      if (tmp) {
        // if it's 1, remote store has already arrived;
        break;
      } else {
        // tile stalls on lr_aq() until the reservation is broken by a remote store;
        tmp = lr_aq(alert);
        if (tmp) {
          break;
        }
      }
    }
  }
  hw_barrier(); // hardware barrier sync;
}
```

Figure 2.10: A kernel routine for synchronization between a producer and a consumer tile-group.

confidence that HB can adapt to rapidly evolving workloads as a general-purpose parallel architecture. Table 2.1 summarizes the benchmarks and the corresponding dwarfs. These benchmarks generally fall into one of three categories:

(1) *Compute-intensive, Low-communication*: AES, BS, and SW kernels exhibit high operational intensity and require minimal memory access. In these workloads, efficiently using the local scratchpad for frequently accessed data is critical. For example, in AES, each tile stores its own copy of the S-box in Local SPM. BS is dominated by floating-point division and square-root operations, both of which are supported in every HB tile. SW is an example of dynamic programming, which tends to have a high branch-miss rate. In general,

Benchmarks (Abbrev.)	***Dwarfs***	**Input Data**
AES (AES)	Combinational Logic	16384×1KB messages
Barnes-Hut (BH)	N-Body	16K, 32K, 64K bodies
Black-Scholes (BS)	MapReduce	10M options
Breadth-First Search (BFS)	Graph Traversal	See Table 2.1b
2-D FFT (FFT)	Spectral Method	16K/32K points (64/512×)
Jacobi (Jacobi)	Structured Grid	256/512×512×64
PageRank (PR)	Graphical Model	See Table 2.1b
Smith-Waterman (SW)	Dynamic Programm.	64K sequences
MatMult (SGEMM)	Dense LA	512×512×512 (256×)
Sparse MatMult (SpGEMM)	Sparse LA	See Table 2.1b

(a) List of benchmarks and their corresponding *Dwarfs* from [16].

Name (Abbrev.)	**Type**	**Edges**	**Vertices**
wiki-Vote (WV)	Social	103689	8297
offshore (OS)	Scientific	4242673	259789
roadNet-CA (CA)	Road	5533214	1971281
road-central (RC)	Road	33866826	14081816
road-usa (US)	Road	57708624	23947347
ljournal-2008 (LJ)	Social	79023142	5363260
hollywood-2009 (HW)	Social	113891327	1139905
soc-Pokec (PK)	Social	30622564	1632803

(b) List of sparse matrix, graphs in Compressed Sparse Row format used in the evaluation (Source: [51])

Table 2.1: Ten parallel benchmarks used to demonstrate HB's parallel programmability.

these kernels scale well and can be accelerated simply by adding more cores.

(2) *Compute-intensive, Sequential-access*: SGEMM, FFT, and Jacobi kernels are characterized by distinct phases in which tiles initially load large, sequential blocks of data, perform extended computations, and then write back the results. Load Packet Compression accelerates the initial loading, while the write-validate cache policy optimizes the write back. For workloads with well-defined communication patterns, such as Jacobi (nearest-neighbor), data can be spatially distributed in Group SPM for fast access and persistent storage, ensuring it is not evicted as it would be in a conventional cache.

Figure 2.11 shows a Jacobi kernel that leverages Group SPM pointers. Each tile loads

$1 \times 1 \times 512$ vector into its Local SPM and synchronizes by using the memory fence followed by a HW barrier. Tiles can access neighboring pixels with low latency using Group SPM pointers. This can be difficult to achieve in hierarchical manycores, where inter-cluster bandwidth is limited and accessing another cluster may involve additional network or cache hierarchy levels.

This kernel also demonstrates how non-blocking remote loads can be pipelined in the network, creating a temporal separation between when the loads are issued and when their results are used, thereby hiding latency. Subsequent local loads further extend this separation, improving overall performance.

(3) *Memory-intensive, Irregular-access*: SpGEMM, PR, BFS, and BH operate on sparse and irregular data structures that are difficult to partition. Each Cell replicates all or part of the data structure in its Local DRAM for faster access. These algorithms generally involve multiple iterations, requiring Cells to synchronize at the end of each iteration to exchange partial results for the next iteration. Data exchange can be performed either by broadcasting via Group DRAM pointers or by combining results in the Global DRAM space.

Workload imbalance is a major challenge for memory-intensive, irregular-access kernels [117, 153, 154, 157, 190]. Although addressing this issue is beyond the scope of this thesis, we describe the basic mechanisms used in our evaluation, which more advanced methods, such as work-stealing runtime [45], can build upon.

To distribute work among Cells, nodes or tasks are first statically partitioned. For example, in the direction-switching BFS [26], each Cell is assigned a subset of frontier nodes in the forward direction or a subset of unvisited nodes in the backward direction. In SpGEMM, following Gustavson's algorithm [75], each Cell is assigned a subset of output rows to compute. Within each Cell, tiles can use atomic-add operations to implement parallel for-loops (Figure 2.12).

The runtime of each subtask can vary significantly depending on the input graph. Here, the SPMD model's independent thread execution is a key advantage, making it well-suited for kernels with highly irregular data access and control flow.

2.3 Evaluation

In this section, we evaluate the HB architecture using the parallel kernels, described in Section 2.2.

2.3.1 Experimental Setup

HB's performance has been evaluated by cycle-accurate simulation of its silicon-validated open-source RTL. We simulate four 16 GB stacks of HBM2 [85] operating at 1.0 GHz, providing a peak bandwidth of 1 TB/s. DRAMSim3 [114] is integrated with the RTL model via the SystemVerilog DPI interface to accurately model HBM2 pseudo-channel timing.

From the 2048-core ASIC, measurements indicate a peak frequency of 1.35 GHz at nominal voltage under passive air cooling. Key parameters are summarized in Table 2.2. Due to prohibitively long simulation time for multi-Cell simulations, we model multi-Cell performance by executing multiple single-Cell simulations in parallel and conservatively estimating inter-phase data transfer times based on the data size and network bandwidth.

Configurations	**16×8**	**16×16**	**32×8**	**2×16×8**
Area (mm^2)	311	539	620	620
Cell Array	8×8	8×8	8×8	16×8
Core Array	16×8	16×16	32×8	16×8
Scratchpad Size (KB)	4			
Cache Sets	64			
Cache Ways	8			
Cache Block Size (B)	64			
Cell Cache Banks	32	32	64	32
Cell Cache Size (MB)	1	1	2	1
Total On-chip Storage (MB)	96	128	192	192
Core Frequency	1.35 GHz			
Memory Frequency	1.0 GHz			
Core / mm^2	26.4	30.3	26.4	26.4

Table 2.2: HB machine configurations for evaluation.

Our baseline HB design consists of an 8×8 Cell array, with each Cell containing a 16×8

core array, for a total of 8192 cores. In this setup, each Cell is mapped to a single HBM2 pseudo-channel. Based on our 12 nm implementation, the Baseline HB occupies 310 mm^2, which is less than half the die area of contemporary GPGPUs [125]. We explore different strategies to double compute resources, while keeping the HBM2 bandwidth constant (Figure 2.13):

(1) *Doubling the size of each Cell vertically* (16×16): This approach doubles the number of compute tiles within a Cell, but reduces the cache capacity per tile by half. Larger Cells generally increase the average hop latency to the cache banks.

(2) *Doubling the size of each Cell horizontally* (32×8): This strategy doubles the number of compute tiles, the cache bank capacity, and the aggregate cache bandwidth. The larger cache allows more efficient processing of bigger datasets by reducing the cache miss rate. However, a wider Cell dimension puts more pressure on the bisection bandwidth.

(3) *Doubling the number of Cells* (2×16×8): This achieves a similar increase in compute resources as the 32×8 approach, but creates two distinct Local Cell address spaces, alleviating pressure on the bisection bandwidth. This is well-suited for embarrassingly-parallel kernels, where data can be easily split. However, data structures that are difficult to partition, such as graphs or octrees, may need to be duplicated in the Local DRAM of each Cell.

```
// Each tile has 1x1x514 buffer on local SPM;
#define Z 512
float data[Z+2];
// Pointers to neighbor SPMs using group SPM pointer;
// __tile_x,y is this tile's coordinate;
float *p_left  = group_spm(__tile_x-1,__tile_y,&data[1]);
float *p_right = group_spm(__tile_x+1,__tile_y,&data[1]);
float *p_up    = group_spm(__tile_x,__tile_y+1,&data[1]);
float *p_down  = group_spm(__tile_x,__tile_y-1,&data[1]);
for (int i = 0; i < Z; i+=4) {
  // Load 22-points in register file;
  register float self[6];
  register float left[4];
  register float right[4];
  register float up[4];
  register float down[4];
  // Remote loads; unrolled by compiler...
  for (int j = 0; j < 4; j++) {
    left[j] = p_left[i+j];
    right[j] = p_right[i+j];
    up[j] = p_up[i+j];
    down[j] = p_down[i+j];
  }
  // Local loads; unrolled by compiler...
  // Also helps hide remote load latency;
  for (int j = 0; j < 6; j++) {
    self[j] = data[i-1+j];
  }
  // Compute and store 1x1x4 output...
}
```

Figure 2.11: Jacobi kernel snippet demonstrating the use of Group SPM pointers to remotely access nearby scratchpads.

```
#define NUM_FRONTIER 1000000
// These point to buffers in Local DRAM;
int* q0; // Initialized to 0 by host;
int* offset, nonzeros, distance;
int* curr_frontier, next_dense_frontier;
// Parallel for-loop using amoadd;
for (int i=amoadd(1,q0); i<NUM_FRONTIER; i=amoadd(1,q0)) {
  int src = curr_frontier[i];
  int start = offset[src];
  int end = offset[src+1];
  for (int j = start; j < end; j++) {
    int nz = nonzeros[j];
    if (distance[nz] == -1) {
      // node hasn't been visited;
      int word_idx = nz / 32;
      int bit_idx = nz % 32;
      amoor(1<<bit_idx, &next_dense_frontier[word_idx]);
    }
  }
}
```

Figure 2.12: BFS kernel snippet demonstrating the use of atomic add to implement a simple parallel for-loop.

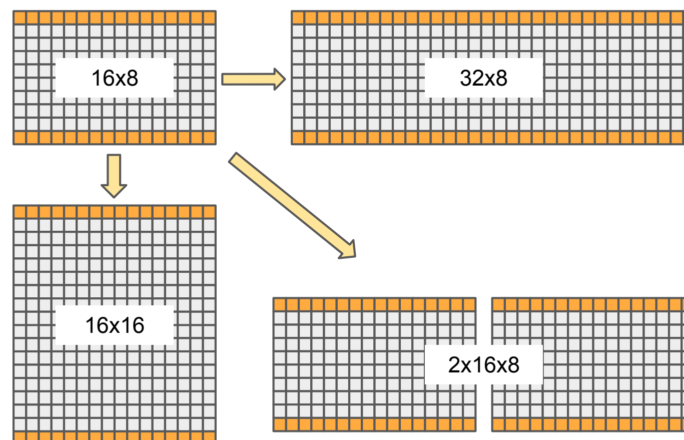


Figure 2.13: Different strategies to double the hardware resources.

2.3.2 Incremental Feature Analysis

In Figure 2.14, we evaluate the performance of a 16×8 single Cell as hardware features are incrementally applied to our baseline Cellular HB architecture. We also compare against the "Baseline Manycore", which has cache capacity, core density, and NoC router bandwidth normalized to a conventional flat manycore [30]. Each parameter – router, cache, and density optimization – is gradually improved to match the Cellular baseline.

Hardware features are then added in the following order: non-blocking loads, Ruche Networks, write-validate cache policy, Load Packet Compression, Regional IPOLY, and finally non-blocking cache. Figure 2.14 shows that these optimizations improve performance across a wide variety of kernels without significantly degrading any single kernel. Applying all optimizations yields 5.2× geomean speedup compared to the Baseline Manycore.

Examining the progression of the geomean speedup, it is clear that higher core density benefits all kernels the most. This motivates HB's focus on tile area optimization to maximize compute density. For memory-intensive kernels, having enough cores is essential to generate sufficient memory requests and keep the HBM2 channels saturated. Compute-intensive kernels, on the other hand, scale easily with additional cores, further supporting the design choice of area-optimized RISC-V cores to maximize compute density.

BH benefits the most from Regional IPOLY, as each tile is allocated 4 KB of private stack space in Local DRAM to track nodes to visit during the tree-traversal. Without this hashing scheme, all tiles would initially contend for the same cache bank, creating a massive traffic jam. Regional IPOLY not only alleviates this contention but also improves programmability by removing the need to manually balance network traffic. Additionally, the Jacobi kernel shows a performance improvement of 17–48×, demonstrating the effectiveness of using Group SPM for well-defined communication patterns (e.g. nearest-neighbor, systolic-array).

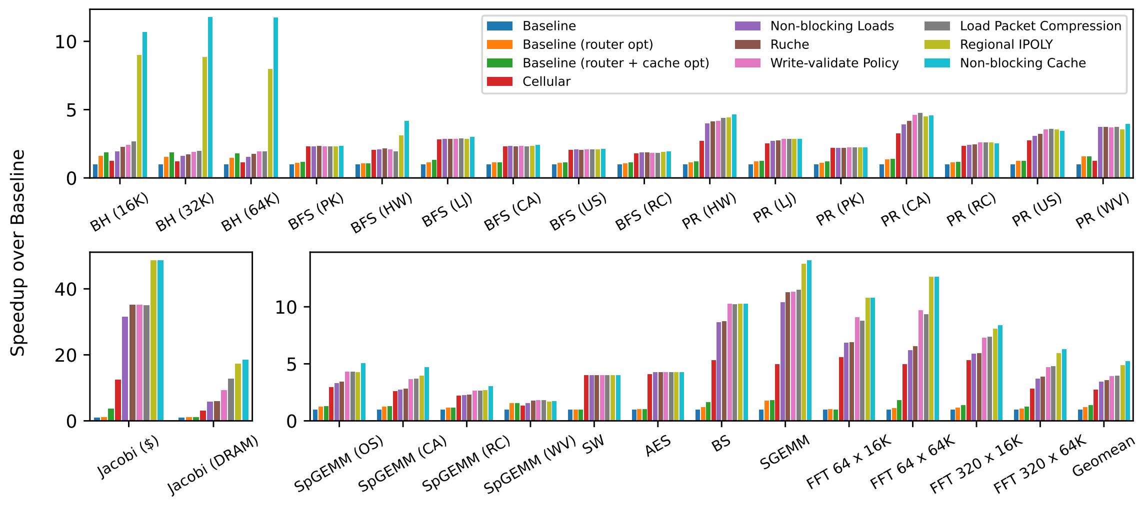


Figure 2.14: Speedup over Baseline Manycore.

2.3.3 Core and HBM2 Utilization

Figure 2.15 analyzes the core and HBM2 utilization of the most optimized HB Cell from Figure 2.14. The core utilization graph shows the percentage of cycles during which cores are either actively executing (integer or FP instructions) or stalling. Note that integer instructions also include memory-access and control instructions. The stall percentages are further broken down by stall types in Table 2.3.

Stall Type	**Description**
MemorySysStall	Stalled on remote load responses from DRAM
NetworkStall	Outgoing request packet is stalled due to network congestion
BypassStall	Pipeline interlock due to back-to-back dependent instructions
BranchMiss	Bubble cycles incurred by branch miss
DivStall	Stalled on the iterative FP divide and square-root unit
FenceStall	Stalled on all outstanding remote memory operations to complete
BarrierStall	Stalled on the HW barrier to complete

Table 2.3: List of core stall types.

The HBM2 utilization graph shows the percentage of cycles during which an HBM2 channel is reading, writing, busy, or idle. A channel is considered busy when one or more requests are queued but cannot be served due to DRAM timing constraints, and idle when the request queue is empty. Refresh cycles are excluded from the denominator to provide a more accurate measure of effective utilization.

In Figure 2.15, kernels are ordered from memory-intensive to compute-intensive. The wide spectrum of utilization indicates that different parallel kernels stress different architectural bottlenecks, highlighting the need for a balanced parallel architecture that performs efficiently across diverse workloads.

PR, BFS, and SpGEMM are memory-bound kernels, spending most of their time waiting for remote load responses from HBM2. High HBM2 utilization is generally a positive indicator, as performance cannot improve without additional memory bandwidth. Kernels with low HBM2 utilization can often benefit from increasing the number of memory requests per core, for example by loop unrolling.

Graph workloads with small size or high degree variance, such as wiki-Vote (WV), tend to perform poorly. BFS on road networks exhibits relatively lower HBM2 utilization because the frontier size remains small throughout the search. One remedy is to exploit task-level parallelism by dividing the Cell into multiple tile groups, with each group independently running different graph queries on the same data structure to generate more memory requests. Finally, high barrier stall often indicates high tail-latency issues, which can potentially be mitigated through better load balancing.

AES, SW, SGEMM, and BS are compute-bound kernels that can benefit from adding more tiles. SW suffers from a high branch-miss rate, which could be mitigated with the RISC-V integer min-max extension [1] or the fused add min-max functions recently introduced in GPUs [54]. BH and BS could gain from a faster iterative FP divide and square-root unit, especially for back-to-back inverse square-root operation. SGEMM achieves high core utilization but does not fully utilize the HBM2 bandwidth, indicating that adding more cores would further improve performance. BS experiences high bypass stalls due to its the FP polynomial calculation.

Some kernels (FFT, Jacobi, SGEMM) are stalled due to the network congestion. The Ruche Networks and Load Packet Compression help by increasing the bisection bandwidth and reducing network load (Figure 2.17). Additionally, Regional IPOLY also helps by distributing the network traffic more randomly, further reducing congestion.

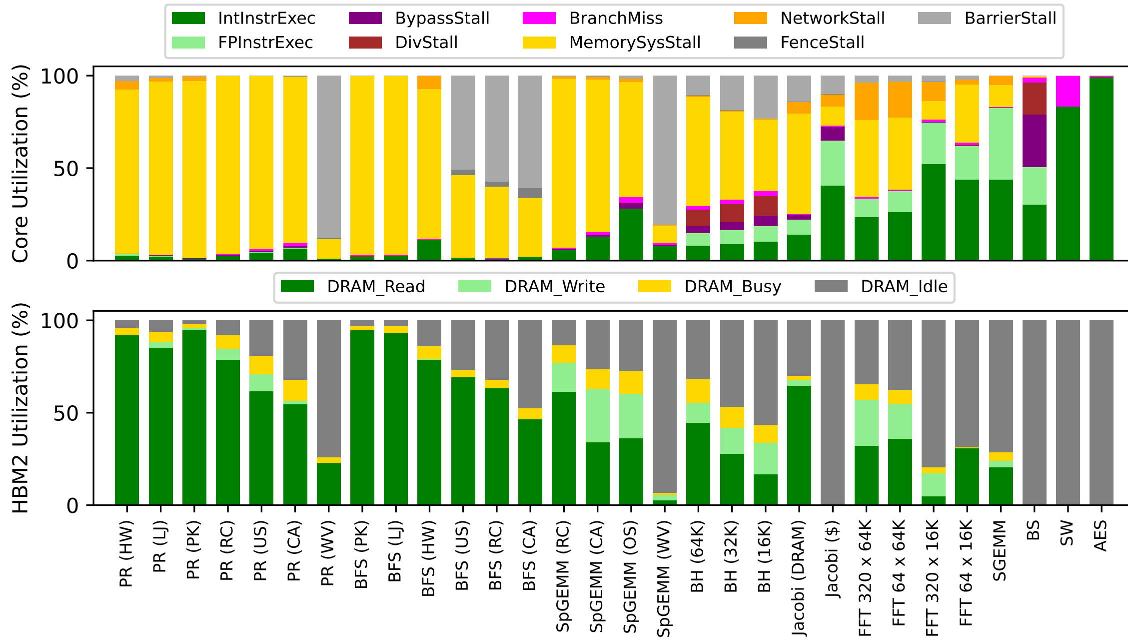


Figure 2.15: Core and HBM2 utilization graphs.

2.3.4 Scaling Irregular Workloads using Tile Groups

Figure 2.16 illustrates how tile groups can be used to scale highly irregular workloads that otherwise exhibit poor resource utilization. A 16×8 Cell can be divided into smaller tile groups to improve both throughput and HBM2 utilization for irregular applications. HB uses tile groups to manage threads in smaller groups, each working on different tasks while sharing a common data structure.

For example, using eight 4×4 tile groups (instead of a single 16×8 tile group) improves the throughput of SpGEMM (WV) by 4.0× and HBM2 utilization by 7.8×. Dividing into even smaller tile groups yields diminishing returns, as the increased working set size leads to more network traffic and higher cache miss rates.

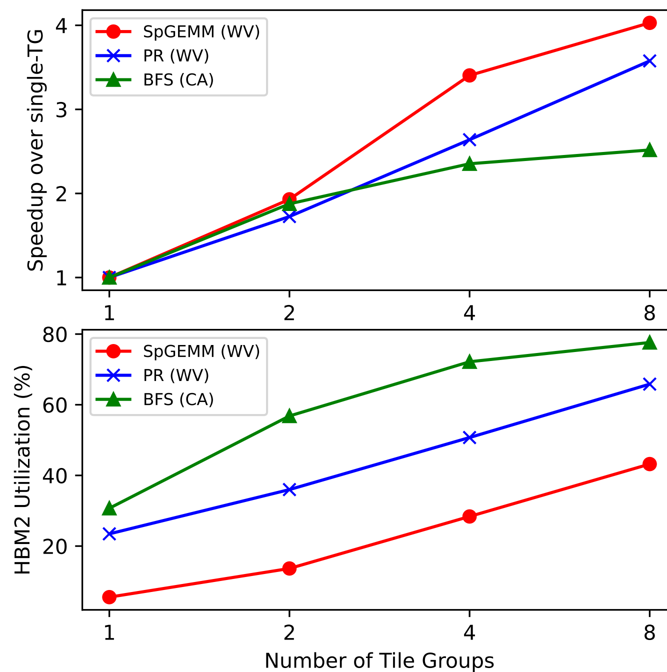


Figure 2.16: Speedup and HBM2 utilization with varying tile group size.

2.3.5 NoC Bisection Utilization

Figure 2.17 shows the utilization of horizontal channels that cross the bisection of a 16×8 Cell. Because IPOLY hashing evenly distributes network traffic among all cache banks, the bisection bandwidth becomes the limiting factor. Even in a modestly sized 16×8 Cell, the bisection links on a plain 2-D mesh can stall up to 50% of the time.

HB employs Ruche Networks and Load Packet Compression to mitigate this problem. Ruche Networks boost the bisection bandwidth by creating extra channels with unused wires. Load Packet Compression reduces the network load by combining sequential accesses into single packets. These features enable larger Cells, providing both increased cache capacity and a larger thread pool.

Figure 2.17 compares the bisection link utilization of a 16×8 Cell with (1) 2-D mesh, (2) Ruche Network, and (3) Ruche Network + Packet Compression. For 2-D mesh, PR (HW), Jacobi (DRAM), FFT (64K) have particularly high stall percentages. Ruche Networks significantly reduces the time packets are stalled across almost all kernels, except Jacobi ($), where most traffic is nearest-neighbor. Load Packet Compression helps with most kernels, although it is less effective for SpGEMM, which contains fewer sequential accesses.

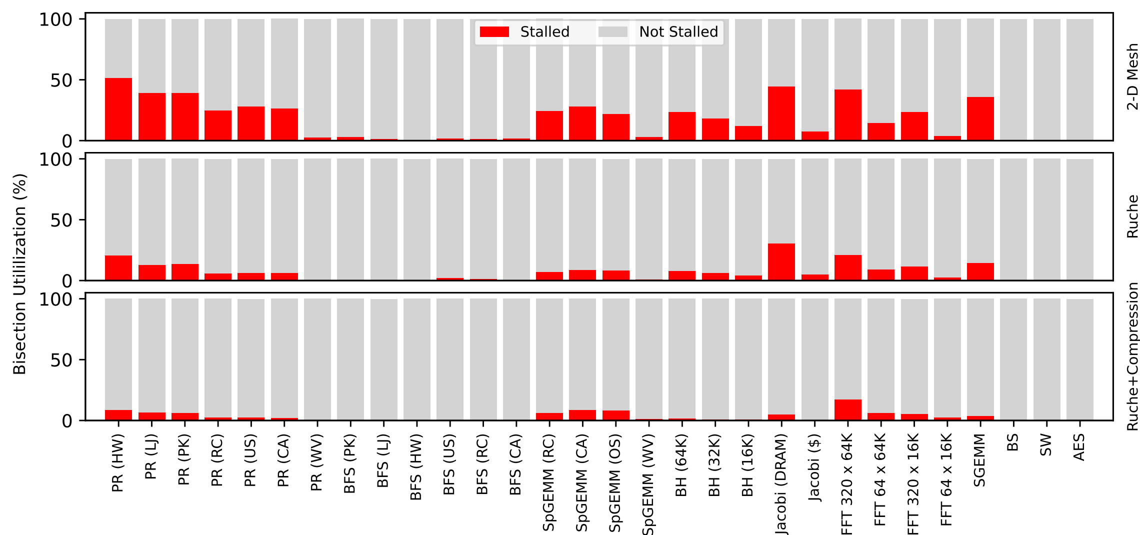


Figure 2.17: Horizontal bisection link utilization.

2.3.6 Doubling HW Resources

Figure 2.18 evaluates three strategies for doubling the number of cores while keeping HBM2 bandwidth constant (as described in Figure 2.13). The configurations 16×16, 32×8, and 2×16×8 achieve geomean speedups of 1.25×, 1.39×, 1.34×, respectively, over the Baseline HB.

Compute-intensive kernels are generally easy to accelerate with more cores. However, doubling core without increasing cache capacity and bandwidth is less effective. The advantages of larger Cells versus more Cells are particularly evident in BH. In 2×16×8, the octree structure, which exhibits good temporal and spatial locality, is duplicated in the Local DRAM space of both Cells, wasting HBM2 bandwidth and cache capacity. For data structures with less locality, such as graphs in BFS or sparse matrices in SpGEMM, duplicating the data does not significantly impact performance.

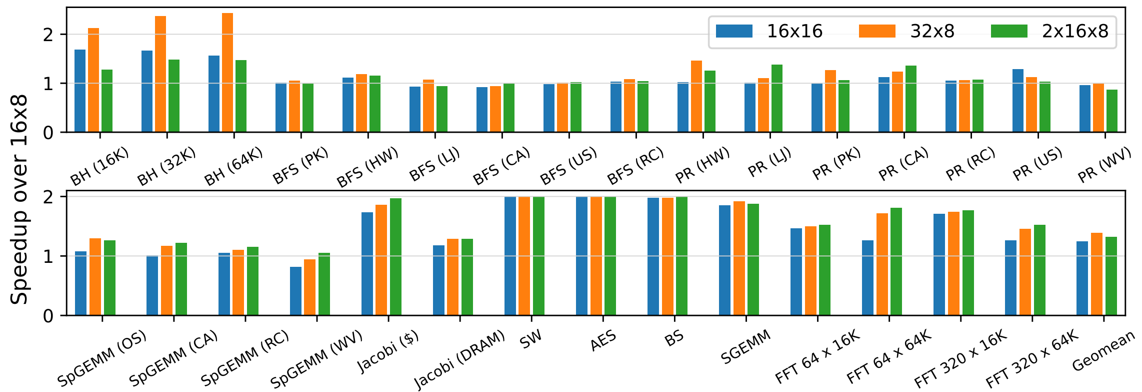


Figure 2.18: Speedup using different strategies to double hardware resources.

2.3.7 Comparison with Hierarchical Manycore

Figure 2.19 compares 32×8 HB Cells against a hierarchical manycore model (ET), with thread density, cache capacity, and network bandwidth configured according to [52]. Both designs are constrained by equal HBM2 bandwidth. The comparison is done on irregular workloads that require data transfers between clusters, highlighting the impact of the on-chip network on the overall performance.

Figure 2.19 breaks down the total runtime into execution time and data transfer time between program phases. In some cases, ET's higher L2 cache slightly improves execution time. However, HB's higher independent thread density provides a clearer advantage for irregular workloads overall. The figure also shows that transferring large volumes of sparse data over wide 2-D mesh channels is inefficient, which negatively impacts the overall program runtime.

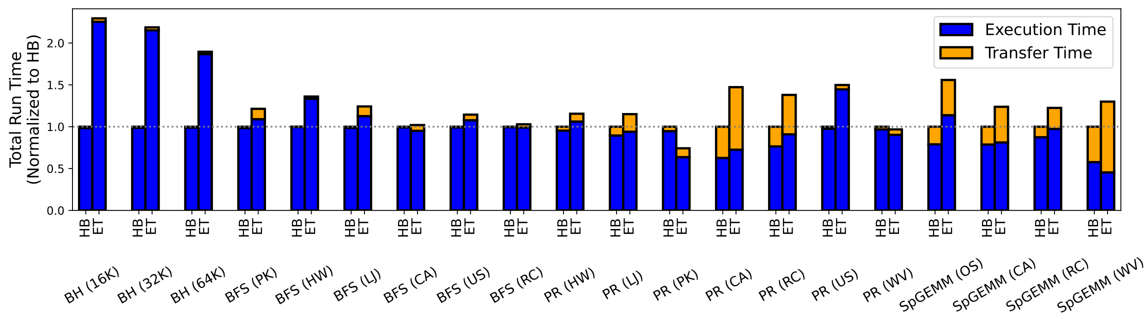


Figure 2.19: Performance comparison between 32×8 HB and the hierarchical manycore model.

2.3.8 Energy Analysis

Figure 2.20 compares “Energy per Instruction” (EPI) against Piton [121], with Piton’s figures normalized to the same 12 nm node using CV^2 scaling. For HB, EPI was measured by collecting switching activities with random operands on a post-layout gate-level netlist, followed by power analysis in Synopsys PrimeTime with extracted parasitics. The EPI has been broken down by hardware components for HB to highlight the contribution of each part.

The result shows that HB’s EPI is 3.6–15.1× more efficient. The observed efficiency can be attributed to several factors. The smaller icache (4 KB) reduces the instruction fetch energy. Using scratchpad instead of L1/L1.5 data caches reduces the energy overhead of memory operations. Instruction latencies are generally shorter in HB (3 cycles for fma, 2 cycles for mul and load/store). Furthermore, process-independent wire capacitance (0.2 pF/mm) favors the smaller HB tiles over Piton tiles (16.6×), as clock, control, and data wires travel much shorter distance within the tile boundary.

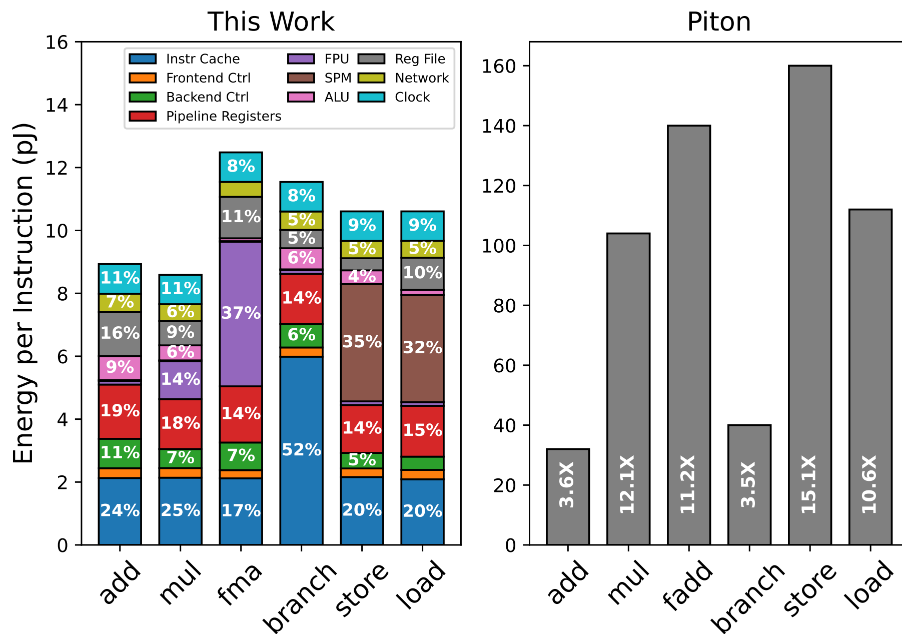


Figure 2.20: Comparison of “Energy per Instruction” (EPI) with Piton.

2.3.9 GPU Comparison

Table 2.4 compares HammerBlade and NVIDIA V100 [125], a commercial GPGPU manufactured in a similar technology node. In this comparison, HB has a 8×8 Cell array of 16×8 cores. HB occupies less than half the area of V100 (0.46×). It is important to note that V100's core area also contains extra hardware units, such as double-precision FPUs, tensor cores, and support for backward-compatibility, which are not part of HB.

In terms of thread counts, V100 has much higher thread-level parallelism (163840 threads per chip), compared to 8192 single-issue cores in HB. However, these are SIMT threads, scheduled in coarse-grained warps (32 threads per warp) that must execute in lockstep, resulting in only 5120 warps on V100. This massive thread count requires a register file 10× larger than HB's. Total SRAM storage in HB (scratchpad and LLC) is 96 MB, which is 6× larger than that of V100 (16 MB including L1 and L2 cache). GPU SM Frequency (non-boost) is taken from [120].

HammerBlade	**Value**	**NVIDIA V100**	**Value**
Process	GF 12 nm	Process	TSMC 12 nm
Cell Frequency	1.35 GHz	SM Frequency	1.2 GHz
Core Area	310 mm^2	Core Area	676 mm^2
Cell Array	8×8	Number of SMs	80
Core Array	16×8	Max Warps/Threads per SM	64/2048
RISC-V Core per chip	8192	Max Warps/Threads per chip	5120/163840
Total 32-bit FPU	8192	Total 32-bit FPU	5120
Total 32-bit Register File Entries	524288	Total 32-bit Register File Entries	5242880
Scratchpad Size per core	4 KB	L1 Cache Size per SM	128 KB
LLC Size per Cell	1 MB	L2 Cache Size per chip	6 MB
Total SRAM Storage	96 MB	Total SRAM Storage	16 MB

Table 2.4: Hardware configuration used for HammerBlade and NVIDIA V100 comparison.

We simulated V100 using AccelSim [97] for the performance comparison. We use the following open-source CUDA libraries for the GPU simulation: SGEMM using Cutlass [96], AES using [22], BFS and PageRank using Graphit [36], Barnes-Hut using Lonestar [106], Black-Scholes using [141], FFT using VkFFT [176], Jacobi using Parboil [156], Smith-

Waterman using GASAL2 [9], and finally SpGEMM using [115]. GPU simulations by AccelSim does not make use of FP64 units or tensor cores.

Figure 2.21 compares the performance of the two architectures described in Table 2.4. Overall, HB has promising 2.9× geomean speedup over V100. The primary reasons for this performance advantage are: (1) HB's scalar cores handle irregular control flow and data access more efficiently than GPU SIMT cores, (2) larger on-chip LLC and fast local scratchpads in HB reduce memory access latency, (3) HB has more floating-point units, (4) HB operates at a slightly higher frequency.

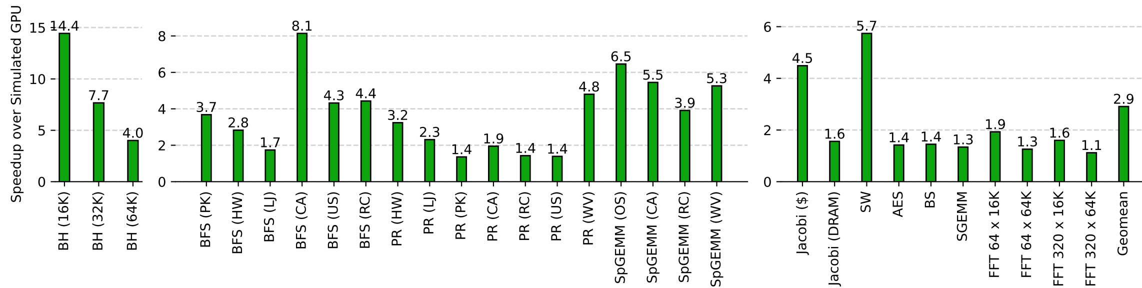


Figure 2.21: Comparison between simulated NVIDIA V100 and 64 16×8 HB Cells.

2.4 Performance debugging in HB

HB provides an extensive set of performance debugging and visualization tools that analyze where and why cores spend time during kernel executions. These tools also measures the utilization of key hardware resources – DRAM, caches, processors, and network routers – to help identify system bottlenecks. HB's profiling infrastructure includes traditional reporting formats, such as text-based summaries and PC-aggregated (program counter) reports. While these formats are useful for quickly assessing overall performance or identifying performance-critical code regions, they do not capture the temporal behavior of kernel execution. To address this limitation, HB introduces a new visualization format called the *temporal utilization graph*, which preserves and exposes the time-varying behavior of hardware resource utilization during execution.

Figure 2.22 presents the core temporal utilization graphs for ten parallel benchmarks. The x-axis represents time, and each narrow vertical column is a stacked percentage bar that aggregates profiling events across all cores over a fixed time window; each bar corresponds to 256 cycles. Note that time scale differs across benchmarks.

The x-ticks appear every 3900 clock cycles, corresponding to one HBM2 refresh interval, with each refresh lasting approximately 260 clock cycles. Instruction execution counts are shown in different shades of green, where each shade represents a distinct instruction type (integer, FP, memory). More green generally indicates that cores are performing useful work without stalling.

These graphs reveal the highly variable phase behavior across benchmarks. Notably, a rush-hour traffic behavior emerges even in regular workloads such as SGEMM, which does not have any explicit inter-core synchronization. In addition to software-induced phase behavior, the graphs clearly expose periodic stalls caused by HBM2 refresh cycles, during which all cache banks temporarily halt, creating a predictable, repeating pattern in core utilization.

Figure 2.23 uses SGEMM as an example to illustrate a full system-wide temporal utilization graph. In the core utilization view, most stalls are related to memory access latency and network congestion, which occur when the cores are copying blocks from DRAM. We

can correlate these core stalls with the cache and DRAM utilization graphs. We observe that even when there is a rush-hour behavior to the memory system, there are still some idle cycles in the cache and DRAM.

The NoC utilization graph visualizes the utilization of network channels that cross various NoC bisections: tile-to-cache, horizontal, vertical. We notice a heavy stall at the tile-to-cache bisection. In fact, the entire FWD network is heavily congested at both vertical and horizontal bisections. This suggests that when the caches are handling misses, requests get backed up and cause congestion, which then blocks other requests heading to idle caches. This congestion can be mitigated by either limiting the maximum number of outstanding requests per tile or increasing the size of the request buffer in each cache bank, so that all the requests are drained from the network and do not block others.

After each data transfer phase, the cores enter a compute-intensive phase, operating at nearly 99% utilization. During this period, there is virtually no cache or DRAM activity. Since SGEMM does not require data synchronization among cores, one possible remedy is to divide the core array into two or more groups and stagger their access to the memory system access. This approach could reduce rush-hour congestion and improve overall DRAM utilization.

Lastly, the REV vertical shows a non-negligible amount of stalls (about 6% of total cycles). These stalls also show up in the cache utilization graph as “resp stall”, indicating that the cache is stalled because it cannot send responses due to congestion. These response stalls also propagate back to the FWD network and the cores, slowing down the entire system.

As previously mentioned, the REV network uses the Y-X dimension-ordered routing. The REV horizontal graph shows significantly fewer stalls, suggesting that once a response packet makes a turn, it stalls less frequently (possibly due to the horizontal Ruche channels). Currently, the arbitration policy in the network routers is round-robin with equal weights. A modified policy that prioritizes the packet coming from the vertical directions (i.e. coming out from the cache) may improve the throughput of the overall system.

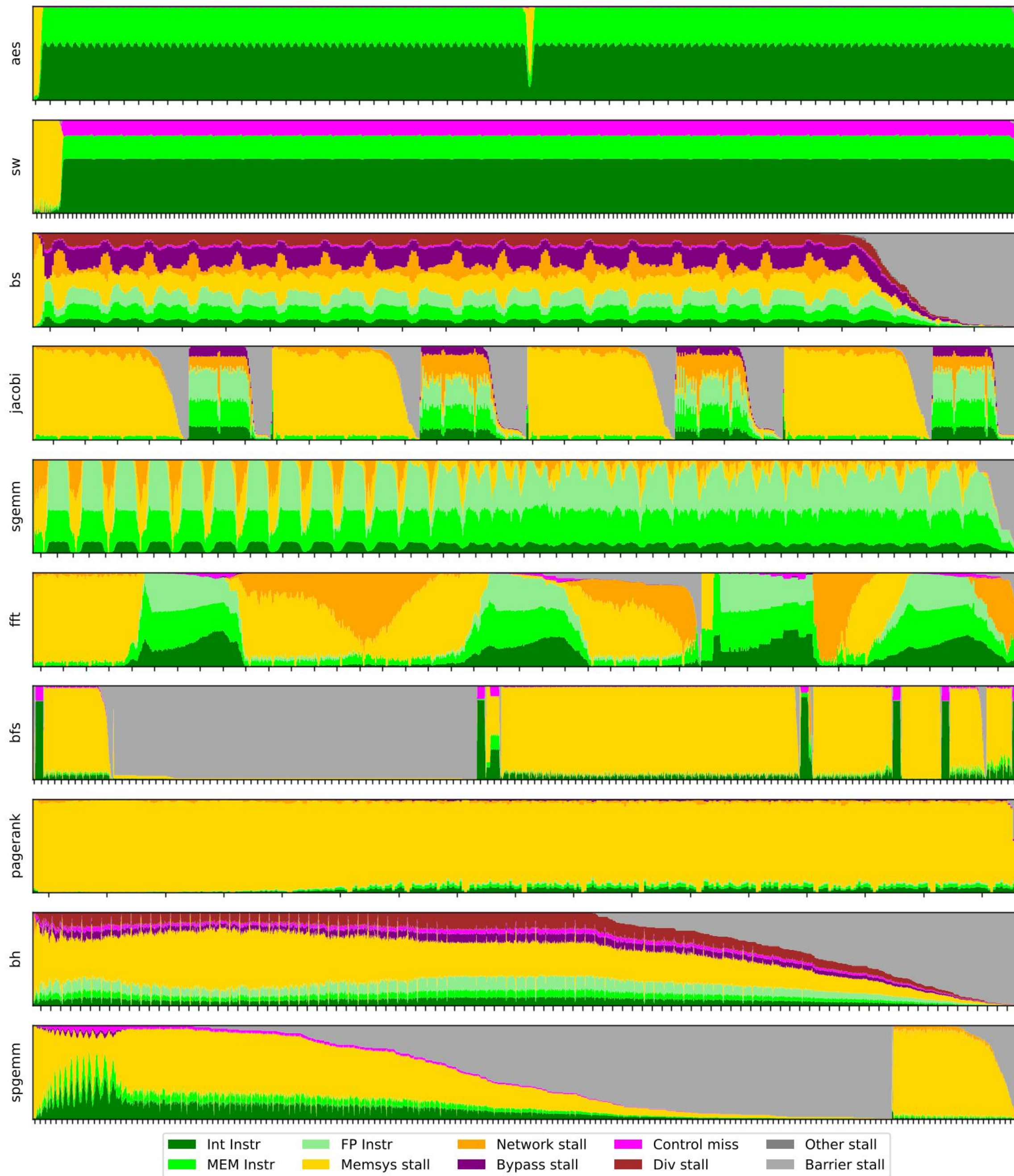
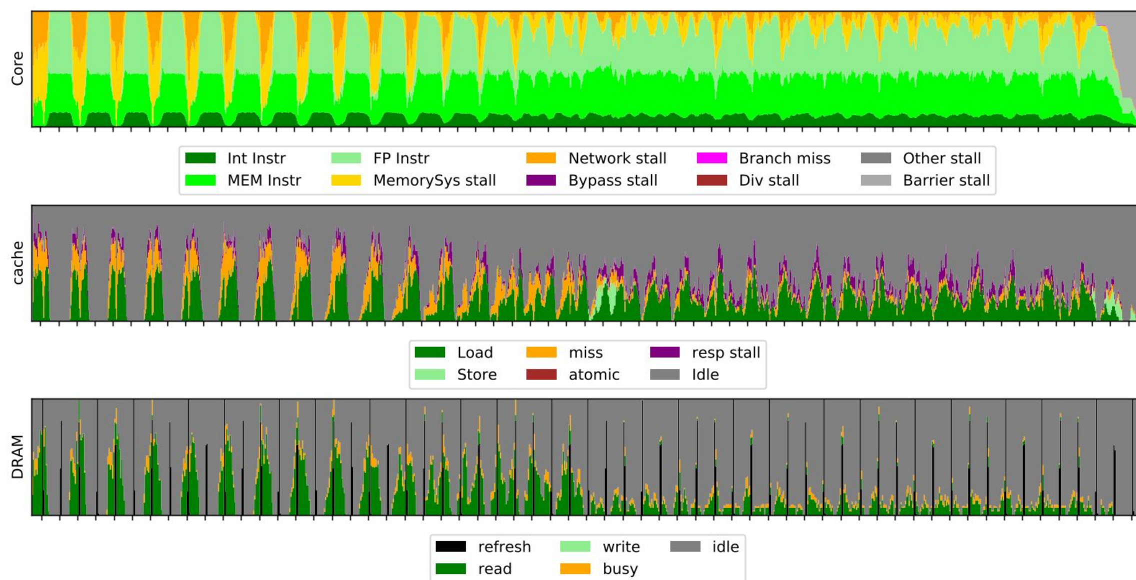
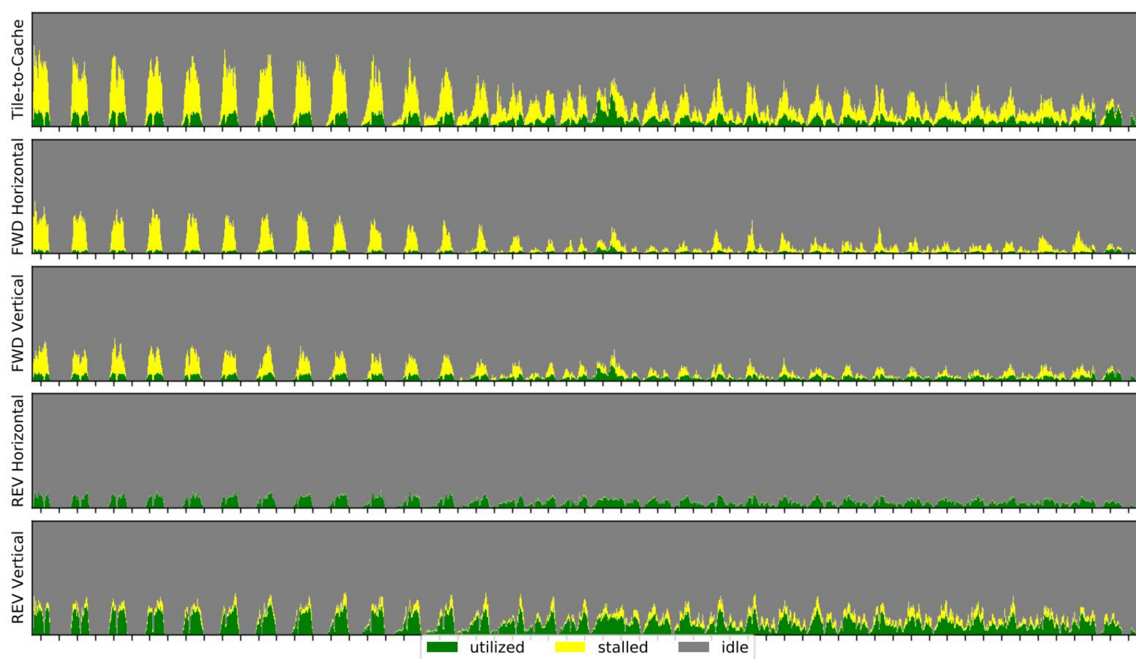


Figure 2.22: Core temporal utilization graph for all benchmarks.



(a) Core, cache, DRAM Utilization



(b) NoC Bisection Utilization

Figure 2.23: Full-stack utilization graph for SGEMM.

2.5 Related Work

Related manycore architectures are shown in Table 2.5. The area of each manycore chip has been scaled to the 12 nm node for comparison. HB's area-optimized RISC-V cores and simplified NoCs significantly improve the network scalability and compute density. HB provides as much as 41.4× higher core density and 5.2× higher 32-bit FPU density compared to previous manycore designs. We summarize these results and highlight the key differences.

Related Work	Category	Networks	Processor	Cores	FPUs	Scaled Area	$\frac{\text{Cores}}{\text{mm2}}$	Our×	$\frac{\text{FPUs}}{\text{mm2}}$	Our×
HammerBlade	Cellular	2 × 2-D Ruche	Single-issue	2048	2048	77.5 mm^2	26.4	1.0×	26.4	1.0×
TILE64 [30]	Flat	5 × 2-D Mesh	VLIW	64	0	19.4 mm^2	3.3	8.0×	0.0	–
RAW [173]	Flat	4 × 2-D Mesh	Single-issue	16	16	2.6 mm^2	6.2	4.3×	6.2	4.3×
Celerity [50, 146]	Flat	2 × 2-D Mesh	Single-issue	496	0	15.3 mm^2	32.4	0.8×	0.0	–
Epiphany V [127]	Flat	3 × 2-D Mesh	Dual-issue	1024	2048	117 mm^2	8.8	3.0×	17.5	1.5×
OpenPiton [24]	Flat	3 × 2-D Mesh	Single-issue	25	25	11.1 mm^2	2.3	11.7×	2.3	11.7×
ET-SoC-1 [52]	Hierarchical	Crossbar, 2 × 2-D CMesh	Vector	1088	8704	1710 mm^2	0.6	41.4×	5.1	5.2×
MemPool [41]	Hierarchical	Crossbar, Radix-4 Butterfly	Single-issue	256	0	8.6 mm^2	29.9	0.9×	0.0	–

Table 2.5: Comparison of manycore designs on network topology, processor type, and compute density.

2.5.1 Flat Manycore

Raw [173] is a 16-core, general-purpose, 32-bit manycore architecture based on a RISC ISA. Although Raw supports a global address space, it does not provide load and store instructions that can directly access other cores' memories. Instead, explicit dynamic messages must be sent in software, and the receiving core services the request either by triggering an interrupt or by explicitly handling the memory request. HB achieves much higher compute density by removing the expensive scalar operand network [20] and by providing simplified mechanisms for accessing remote memory.

TILE64 [30] is a 3-wide VLIW 64-core, Linux-capable manycore architecture. A DMA

engine in each tile facilitates block-oriented data transfers between caches and memory interfaces in the background. Message-based communication between tiles can be prone to head-of-line blocking. In order to support out-of-order processing of messages (i.e. processing in an order different from their arrival order), messages has to be tagged by the sender and sorted into one of multiple receiving queues in each tile, incurring large area overhead.

Epiphany-V [127] is a 1024-core, 64-bit, dual-issue RISC processor. It features three 136-bit wide 2-D mesh networks for read requests, on-chip writes, and off-chip writes. The off-chip traffic is separated from on-chip traffic to make on-chip latency more deterministic. Epiphany-V does not include L1 or L2 caches; instead, each processor has a multi-ported scratchpad that can service instruction fetch, local load/store, and remote load/store simultaneously. Dual-issue cores are effective at keeping multiple hardware units busy (e.g. scratchpad and FPU every cycle), but the heavily-ported register file and complex bypass paths can be area-consuming.

Celerity [50] is a 496-core RV32I manycore processor. Similar to HB, all scratchpads in Celerity are globally addressable over a 2-D mesh network using a PGAS system. However, Celerity lacks remote load support, cache banks, an instruction cache, and an FPU, which makes it significantly less programmable.

OpenPiton [24] is a cache-coherent manycore architecture featuring three 64-bit-wide 2-D mesh networks that implement a distributed, directory-based MESI protocol. Like TILE64, OpenPiton emphasizes on Linux capability, resulting in low compute density. Each tile contains private L1 and L1.5 caches, and a shared L2 cache with inclusive policy at every level, which leads to duplication of data across the hierarchy. By default, cache lines are distributed among all L2 caches in the system. Although Coherence Domain Restriction [58] can be used to map a page to a specific subset of tiles to enable nearest-neighbor or consumer-producer communication, this approach adds complexity, as it requires additional storage and hardware to maintain a software-defined mapping from virtual pages to physical L2 caches.

2.5.2 Hierarchical Manycore

ET-SoC-1 [52] contains 1088 RV64IMAF *minion* cores, each with configurable L1 data cache/scratchpad and FP vector units. Eight minion cores are grouped into *neighborhoods* that share a single large instruction cache. Four neighborhoods are connected via a crossbar to form *shires*. The shires communicate over a mesh network with 1024-bit links. Communication outside the shire is block-structured, which limits the ability to perform fine-grained random accesses. In constrast, all cores in HB have global, word-level access to every scratchpad and cache bank on the chip via a globally uniform network.

MemPool [41] is a proposed architecture with 256 single-cycle *Snitch* [192] cores implementing RV32IMA. Eight cores are aggregated into a *Tile* that shares a 2 KB L1 instruction cache, and a 16 KB data scratchpad via two fully-connected crossbars. 16 Tiles are aggregated into *Groups* using two 16×16 crossbars, and four Groups are interconnected by a radix-4 butterfly network. In HB, the network routers are co-located with the processor logic within each tile using NoC Symbiosis [140], allowing seamless integration of thousands of tiles. In MemPool, however, the routers are placed between Tiles, which results in inefficient area utilization and routing congestion.

Chapter 3

RUCHE NETWORKS

3.1 NoC Architecture

This section describes the Ruche networks – its tileable, physically scalable network topology, routing algorithm, and router architecture.

3.1.1 Physically scalable Ruche topology

Ruche networks [93] augment 2-D mesh with equidistant, long-range, physical channels (a.k.a *Ruche channels*) between remote tiles in the same row or column. As these channels pass through the tiles, they consume available VLSI wiring tracks. Increasing the Ruche Factor (RF) – the skip distance of the Ruche channels – consumes more wires and reduces the network diameter. By adjusting the Ruche Factor, architects can scale bisection bandwidth without widening the channel width. Figure 1.3a and 1.3b show the Half and Full Ruche topologies with Ruche Factor of 3. Full Ruche adds Ruche channels in both vertical and horizontal axes, while Half Ruche adds channels in only one axis. Ruche-One (Figure 1.3f) is a special case where the Ruche Factor is one, topologically equivalent to having two mesh networks in parallel (Figure 3.2a). In this work, we consider networks that provide in-order packet delivery, as is commonly required for streams and ordered memory traffic.

Figure 3.1 illustrates the bitwise mapping of a Half Ruche network (RF = 3) using the tile-based method. Long-range wires cross each tile in straight lines, utilizing less resistive upper-mid metal layers. Repeaters are placed between tiles to drive these long wires. Note that all tiles share identical VLSI layouts. Local and Ruche channels are bit-interleaved, with the i-th bit of all links routed together, so that swizzling only shifts signals by a few wiring tracks.

Figure 1.3 shows several low-diameter topologies with non-local (express) links proposed in the past. A key differentiator of Ruche Networks is that the router radix remains constant

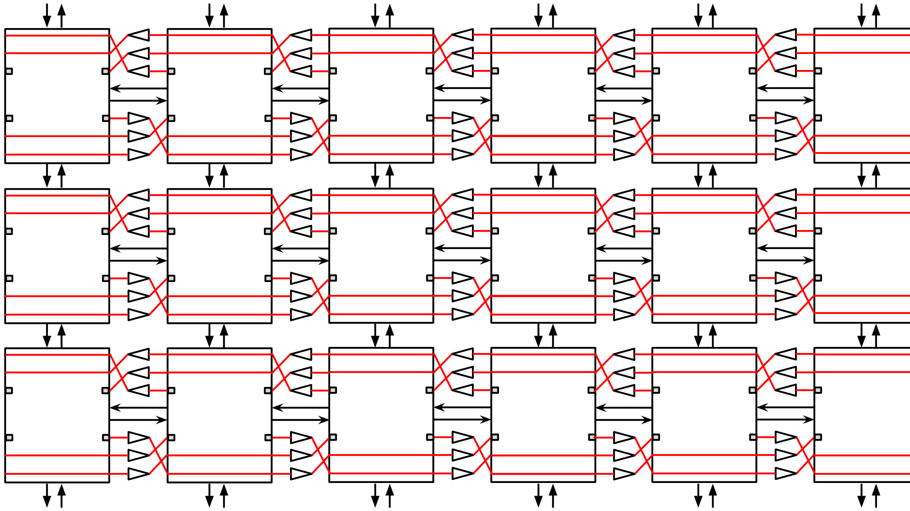


Figure 3.1: A bitwise pattern of mapping Half Ruche (RF = 3) on a tile-based design.

as the network size grows. In contrast, in high-radix topologies (e.g. MECS, flattened butterfly) (Figure 1.3c, 1.3d), router radix grows linearly with the network size in each dimension. As network size increases, MECS must continually reduce channel width to avoid exhausting wire tracks and to maintain cycle time and area constraints. Narrower channels, however, limit injection bandwidth at each node and add serialization latency.

Another differentiator is that the physical distance of Ruche channels remains constant as the network size grows. In MECS, the longest channel distance (and its wire delay) increases with the network size. Eventually, wire delay dominates, so either the cycle time must be increased or the wires need to be pipelined, increasing hop latency. These properties make it difficult to efficiently tile the architecture

. Timing closure is also more difficult for MECS. Normally, tile input and output ports are constrained with estimated external delays for setup and hold timing. Unlike the Ruche channels, where each channel has a single sender and receiver, multi-drop channels have multiple receivers each with varying external delays. This makes it hard to create a consistent set of timing constraints that can be uniformly applied across all tiles. Table 3.1 compares these low-diameter NoC topologies based on physical scalability criteria.

Folded 2-D torus (Figure 1.3e) is an interesting alternative because it maintains the

Topology	Regular Tile Shape	Regular Wire Routing	Constant Router Radix	Standard-Cell Based	Non-power-of-2 Tiling	Long-range Links	Constant Link Distance
Ruche (This paper)	✓	✓	✓	✓	✓	✓	✓
2-D Folded Torus	✓	✓	✓	✓	✓	✓	✓
2-D Mesh	✓	✓	✓	✓	✓		✓
Multi-mesh	✓	✓	✓	✓	✓		✓
Flattened Butterfly [102]				✓		✓	
MECS [71]				✓	✓	✓	
Swizzle-Switch [5]					✓	∼	

Table 3.1: Comparing various NoC topologies based on physical scalability criteria.

same router radix as 2-D mesh while reducing the network diameter by half and doubling the bisection bandwidth. It satisfies the physical scalability criteria, similar to 2-D mesh and Ruche networks (Table 3.1). Folded 2-D torus uses a tiling technique similar to that shown in Figure 3.1, as implemented in recent ML accelerators by Tenstorrent [84, 178], so its physical-design complexity is comparable to Ruche networks.

Ruche routers, however, have a higher router radix, making 2-D torus appear to have lower area cost. A key disadvantage of 2-D torus is that it is deadlock-prone due to cyclic channel dependencies. A standard solution to achieve deadlock freedom involve adding virtual channels (VC) and dateline logic to break the dependency cycles [47]. In this light, the area overhead of the 2-D torus becomes comparable to that of Ruche networks. However, VC routers require complex *allocation* logic [29], which adds significant cycle time, area, and power overhead, unlike the simple logic used in Ruche routers. Fairness and matching quality, which affect utilization, are additional concerns for VC routers [29].

Figure 3.2a shows a 2x multi-mesh, where a second parallel mesh router is introduced to double the NoC bandwidth. Figure 3.2b and 3.2c illustrate two approaches to combining two parallel routers in a 2x multi-mesh. In terms of input FIFOs, both VC and Full Ruche routers have the same capacity. Full Ruche routers (Figure 3.2b) combines two

mesh crossbars into a single larger crossbar, maintaining the peak router bandwidth. VC routers, on the other hand, (Figure 3.2c) discard one of the mesh crossbars, halving the peak bandwidth. VC routers expend extra area merely to virtualize physical links (with VC mux) but removing a crossbar does not increase router bandwidth. In this thesis, we compare 2-D torus and Ruche networks in terms of power, cycle time, area, and network throughput and latency.

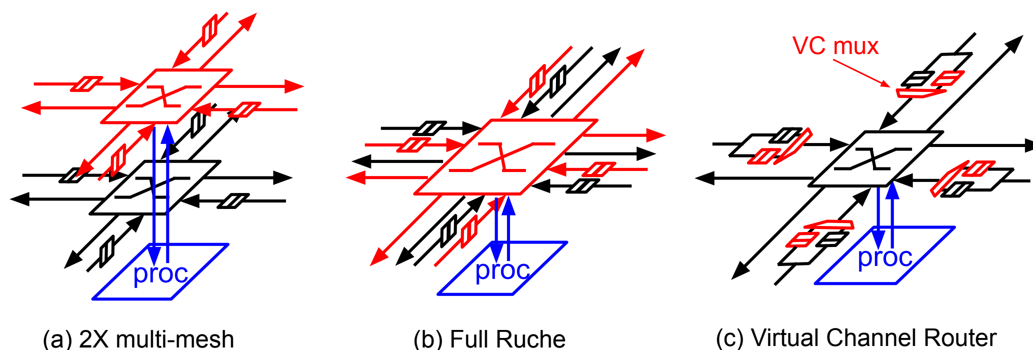


Figure 3.2: Two approaches to combine 2x multi-mesh routers.

3.1.2 Ruche Router Architecture

Ruche Networks augment 2-D mesh with extra directions – Ruche East/West/South/North (RE/RW/RS/RN) – whose channels can be extended to connect distant tiles. Ruche routers provide two variants of routing algorithms to trade off crossbar area and network latency: *fully-populated* and *depopulated* (Figure 3.3). In both variants, packets are routed using a modified dimension-ordered routing (DOR) that is deadlock-free without virtual channels.

Generally, the routing in the first dimension uses "Ruche-first", where a packet initially travels on a Ruche link – like taking a highway for the majority of the distance – before transitioning to local links for the remaining distance. Routing in the second dimension uses "local-first", where a packet travels on local links until the remaining distance modulo the Ruche Factor is zero, then switches to Ruche links to reach the destination.

In fully-populated, packets can make a direct turn from the lower to higher dimension using Ruche links (e.g. from RE/RW to S/N/P/RS/RN). Depopulated routing is non-

minimal in the sense that packets must switch from Ruche links to local links before making a turn. Fully-populated optimizes X-Y turns at the expense of larger crossbar area, but it provides greater routing flexibility, which may perform better in smaller networks or with large Ruche Factors. Depopulated can perform better in some case because it balances traffic more evenly between Ruche and local links. In large networks, local links tend to be less utilized.

Figure 3.4 shows the additional crossbar connectivities added to a minimal 2-D mesh router using DOR as employed in the Celerity RISC-V manycore [50]. By restricting certain crossbar paths, depopulated routers significantly reduce crossbar area. Fully-populated connectivities (red x) are added on top of the depopulated (blue triangle). Depopulated routers reduce the total connectivities by 16. The output port with the highest number of inputs (e.g. P output) has its inputs reduced from 9 to 7.

Ruche-One is a special case (Figure 1.3f), where both local and Ruche links connect only to nearest neighbors. In this configuration, packets take Ruche links for the entire path if the total distance is even (or local links if odd) to balance traffic. Ruche-One operates only on fully-populated routers.

The connectivity matrix for Y-X order is simply the transpose of the X-Y order matrix. The matrix for Half Ruche can be obtained by excluding the rows and columns corresponding to RS and RN.

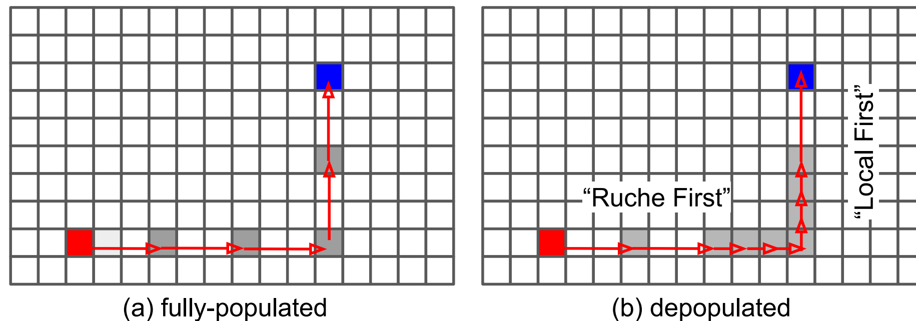


Figure 3.3: Two variants of a deadlock-free dimension-ordered routing algorithm for Full Ruche (RF = 3, X-Y order).

Input Ports

Output Ports

	RS	RN	RE	RW	S	N	E	W	P
RS		△	x	x		△	x	x	x
RN	△		x	x	△		x	x	x
RE				△					△
RW			△						△
S			x	x		o	o	o	o
N			x	x	o		o	o	o
E				△				o	o
W			△				o		o
P	△	△	x	x	o	o	o	o	o

o = Original 2-D mesh connection
△ = Added by depopulated router
x = Added by fully-populated router

Figure 3.4: Full Ruche crossbar connectivity matrix (X-Y DOR).

By default, the input ports of Ruche routers are minimally buffered with two-element FIFOs. Packets are arbitrated using a simple round-robin policy at each output direction. In high compute-density manycore processors [50, 94], tiles are small, so wire delay across a tile is relatively low, and the crossbar gate delay dominates the critical path between tiles. In this case, packets can traverse the network at a rate of one cycle per hop.

As tile size or the Ruche Factor increases, wire delay begins to dominate. In such cases, both the router and physical links must be pipelined using credit-based flow control, and input FIFO capacity must be increased to hide the credit-return latency. This challenge is not unique to Ruche Networks; other router architectures face the same requirement. However, the issue is more pronounced in routers that use virtual channels. In Ruche routers, the generation of request signals going to arbiter is independent of the ready signal from the output side in terms of combinational logic (e.g. ready-valid-and [171]). In VC

routers, the generation of request signals going to the allocator must depend on the credit availability signal of the destination VC (e.g. ready-then-valid [171]), otherwise the allocator could grant access to VCs that could not have moved forward and block the progress of other VCs.

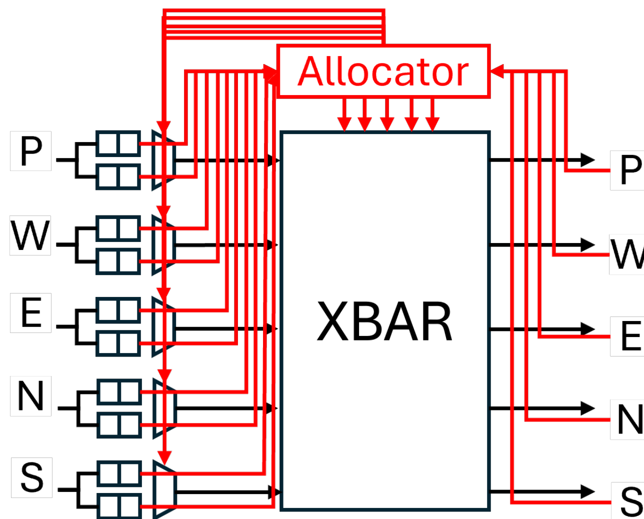


Figure 3.5: VC Allocation Router.

For these reasons, VC routers generally have longer critical paths and need to be pipelined. canonical VC routers typically have four sequential pipeline stages (e.g. routing, VC allocation, SW allocation, SW traversal). Speculative VC routers [135] reduce the number of stages to three by allowing VC and SW allocation to take place in parallel. This relies on speculating that VC allocation will be successful while giving priority to non-speculative SW requests over the speculative ones. However, this approach requires replicating SW allocators for speculative and non-speculative requests.

Mullins et al. [122] proposes a single-cycle router architecture in which switch traversal and speculation on allocation decisions for the next cycle are overlapped. The key idea is that, using a least-recently-granted priority scheme of matrix arbiters, the router can predict the next grant signals, knowing that the currently granted input will have the lowest priority on the following cycle. However, due to its complex control logic, speculative VC routers

add significant power overheads over wormhole and non-speculative VC routers [25].

Another important consideration is that network throughput is highly sensitive to the router latency as a side-effect of credit-return latency, which reduces the buffer utilization [135]. Thus, Ruche routers that can achieve lower cycle time at lower area than VC routers under the same conditions (e.g. tile size, channel width) without any form of speculation are more advantageous.

3.2 Evaluation

In this section, we evaluate Ruche NoCs against 2-D mesh and torus under two common scenarios. First, we consider generic all-to-all communication patterns in square networks (8×8 and 16×16). In this case, both vertical and horizontal bisection can become bottlenecks. Therefore, we compare Full Ruche, which adds Ruche channels in both dimensions, against 2-D torus. Second, using the HammerBlade manycore simulator, we evaluate *all-to-edge* communication patterns, where most network traffic flows to memory ports on the northern and southern edges. In this arrangement, using X-Y DOR for request traffic and Y-X DOR for response provides the best network throughput [6]. For all-to-edge, horizontal bisections are the primary bottleneck, as packets first try to reach their destination column. We therefore evaluate Half Ruche and half-torus, which add long-range channels in the horizontal directions to alleviate this bottleneck. Vertical long-range links are excluded in this scenario, since vertical channel bandwidth already matches memory port bandwidth 1:1, offering little benefit under high network load. Although two scenarios are not mutually exclusive, they demonstrate that Ruche NoCs can be specialized for common traffic patterns found in target architectures.

3.2.1 Full Ruche - Synthetic Traffic

Figure 3.6 shows synthetic traffic analysis for 2-D mesh, 2-D torus, multi-mesh, and various Full Ruche topologies. These results are based on cycle-accurate simulations of RTL-level implementations. 2-D mesh and Full Ruche are without virtual channels, whereas 2-D torus has two VCs per input. We assume using single-flit packets, with each FIFO or VC holding up to two packets. Packets traverse the network at one cycle per hop in all configurations. The 2-D torus uses dateline VC partitioning to ensure deadlock freedom [47].

The routing algorithm used for 2-D torus is DOR, which produces one output VC direction. SW allocation in the 2-D torus is performed using an acyclic implementation of wavefront allocator for maximal matching quality [28]. Evaluation is conducted on 8×8 and 16×16 networks. Four synthetic traffic patterns are used: uniform random, bit complement, transpose, and tornado. Five Full Ruche configurations are evaluated, varying Ruche Factors from 1 to 3 (ruche1-3) and considering both fully-populated (pop) and depopulated (depop) crossbars.

In uniform random 8×8, 2-D mesh has saturation throughput around 28%. Although 2-D torus doubles the bisection bandwidth, its saturation throughput peaks at approximately 42%. The ruche1-pop configuration, which does not reduce network diameter but provides the same bisection bandwidth as 2-D torus, outperforms 2-D torus with a throughput of roughly 48%. As explained in Figure 3.2, VC routers halve the peak crossbar bandwidth, whereas Ruche routers retain the bandwidth of 2x multi-mesh.

This difference is even more pronounced in the case of uniform random 16×16. 2-D mesh throughput saturates around 15%, whereas 2-D torus throughput only reaches up to 19%, far below the throughput expected from doubling the bisection bandwidth. By maintaining the crossbar bandwidth, ruche1-pop is able to reach the saturation bandwidth of 28%, much closer to the expected value. 2x multi-mesh closely follows the ruche1-pop curve.

As the Ruche Factor increases, zero-load latency generally decreases, and saturation throughput improves. In relatively smaller networks (8×8), however, a very large Ruche Factor (ruche3-depop) can reduce the likelihood of Ruche links being utilized, causing the performance to drop. In such cases, fully-populated routers (ruche2-pop and ruche3-pop)

help by providing greater routing flexibility. The benefits of larger Ruche Factors are more visible in larger networks (16×16).

Under adversarial traffic patterns (bit complement, transpose, tornado), Ruche-One performs similarly to 2-D mesh, while the 2-D torus performs better. With Ruche Factor of 2 or 3, Full Ruche generally outperforms 2-D torus, with some exception. ruche3-pop in bit complement 8×8 is an exceptional case where there are zero conflicts between network traffic, so latency does not degrade. 2-D mesh, 2x multi-mesh, and ruche1-pop perform very similarly, except in the transpose pattern, where ruche1-pop performs better.

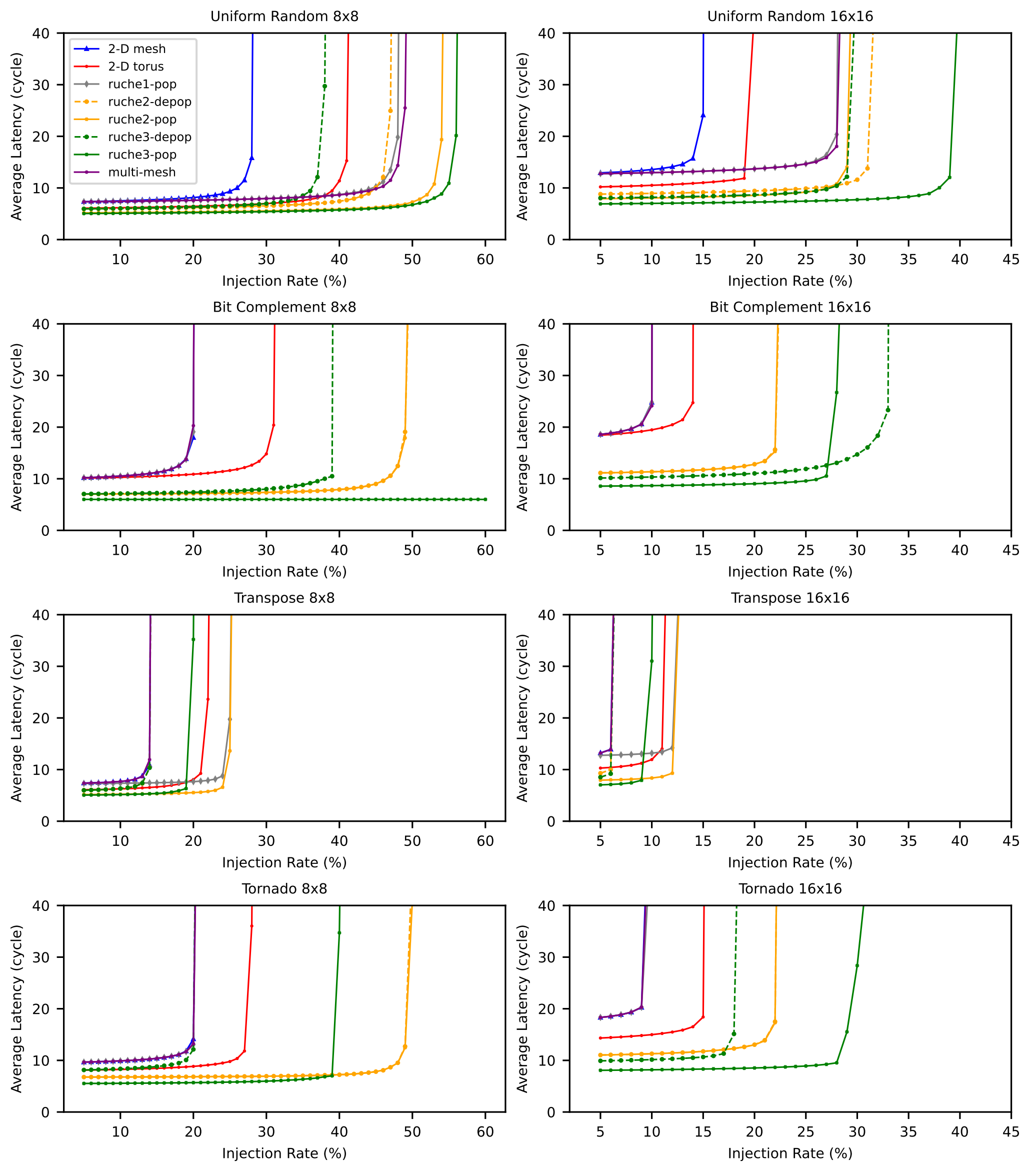


Figure 3.6: Synthetic traffic analysis on 2-D mesh, 2-D torus, and various Full Ruche topologies.

3.2.2 Full Ruche - Area and Cycle Time

We evaluate the area and cycle time of 2x multi-mesh, 2-D torus and Full Ruche routers following the methodology used in [28]. Figure 3.7 reports cell area as a function of target cycle time used during synthesis. Synthesis is performed using Synopsys Design Compiler with a 12 nm regular-Vt standard-cell library. Cycle time is normalized to the fanout-of-four (FO4) delay of the target library. For each router, the cycle time is decreased in fixed decrements until a timing violation occurs. All routers use 128-bit channel widths and X-Y DOR crossbars.

Figure 3.7 shows that Full Ruche routers can achieve much lower cycle times than 2-D torus routers, primarily because the wavefront allocator in the torus is significantly more complex than the decentralized round-robin logic used in Ruche routers. Depopulated Full Ruche routers have much lower area than fully-populated Ruche and multi-mesh routers due to significant crossbar area savings. When the cycle time is relaxed (∼100 FO4), the fully-populated has slightly higher area than 2-D torus. However, it can reach much lower cycle times than 2-D torus without timing violation.

Both fully-populated and depopulated reach about the same minimum cycle time, slightly higher than that of 2-D mesh. Multi-mesh has comparable minimum cycle times to Ruche, because of the higher fanout at the P-input port and the additional route compute logic to decide between two meshes when injecting packets (using mesh0 for even Manhattan distances, mesh1 for odd). The maximum number of crossbar mux input is 7 for depopulated and 9 for fully-populated (Figure 3.4), which is only a few gate delays.

A key advantage of Ruche routers is that it can achieve competitive cycle times without pipelining. In contrast, VC routers require pipelining, which either increases pipeline latency or demands complex speculation schemes that raise power consumption [135].

Table 3.2 shows the router area breakdown when the target cycle time is most relaxed (∼98 FO4). The depopulated router significantly reduces crossbar area by 40%, which is considerably less than the double 2-D mesh crossbars used in multi-mesh. In 2-D torus, route computation (decode), allocation logic, and virtual channels contribute to greater area overhead. Overall, the depopulated Full Ruche occupies 12% less area than 2-D torus.

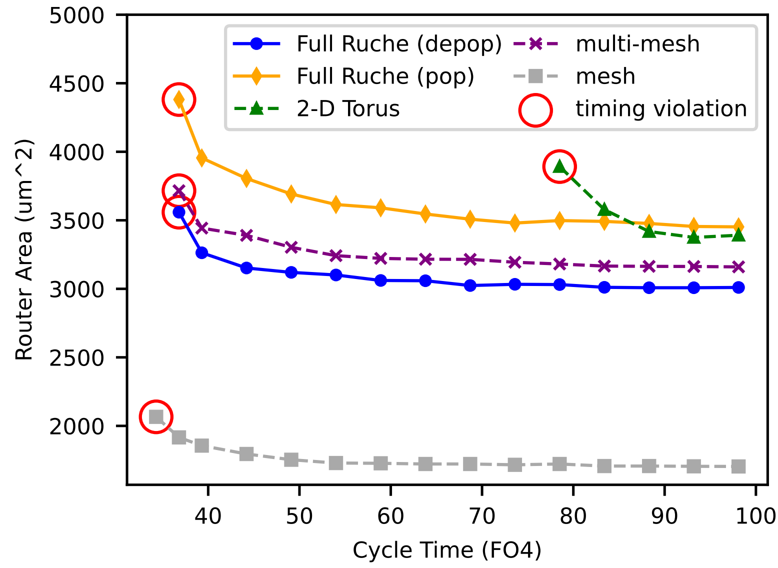


Figure 3.7: Area vs. Cycle Time comparison of Mesh, Multi-mesh, Full Ruche and 2-D torus routers.

	Multi-mesh (um^2)	**Full Ruche (depop)** (um^2)	**Full Ruche (pop)** (um^2)		**2-D Torus** (um^2)
Crossbar	791	599	986	Crossbar	410
Decode	96	99	100	Decode	349
FIFO	2250	2250	2250	VC	2435
Arbiter	53	42	74	Allocator	194
TOTAL	**3190**	**2991**	**3411**	**TOTAL**	**3388**

Table 3.2: Multi-mesh, Full Ruche and 2-D torus Router Area Breakdown.

3.2.3 Full Ruche - Energy

Table 3.3 reports the energy required to transmit a single packet across the routers. We place and route the routers with 128-bit-wide channels using Synopsys IC Compiler 2. The tile region is 187 um × 187 um, roughly 1.3× the size of a dense RISC-V core [94]. After place and route, wire parasitics were extracted by StarRC. In gate-level simulations, we collected switching activities for each output direction while streaming packets from the corresponding valid inputs. We assume that packet payloads have an activity factor of 0.25 (i.e. half of bits switch every cycle). With extracted parasitics and switching activities, we used Synopsys PrimeTime to calculate the average energy per packet. These results do not include energy dissipated by long-range links outside the tile area.

The result shows that Full Ruche routers are generally more energy efficient than 2-D torus. Depopulating the crossbar further reduces energy consumption, particularly for the Ruche directions, which have fewer crossbar inputs. As shown in Figure 3.4, depopulation reduces the number of mux inputs for RS and RN by 5.

Direction	**Full Ruche (depop)** (pJ)	**Full Ruche (pop)** (pJ)	**2-D Torus** (pJ)
Horizontal	1.66	1.95	2.41
Vertical	1.82	2.01	3.35
Ruche Horizontal	1.40	1.81	–
Ruche Vertical	1.49	2.00	–

Table 3.3: Full Ruche and 2-D torus Router Energy per Packet

3.2.4 Full Ruche - Fairness

In 2-D mesh, a tile's average latency strongly depends on its position. Tiles near the edges of the network experience higher average latencies than those near the center, and this positional unfairness increases with network size. 2-D torus, in contrast, is theoretically the fairest network topology, as it is symmetric from every node.

Figure 3.8 shows the distribution of average latency for each tile in a 16×16 network under uniform random traffic and low network load. As expected, 2-D mesh exhibits a much higher stdev ($\sigma = 1.67$) and mean ($\mu = 10.6$) than 2-D torus. Adding Full Ruche reduces both the stdev and the mean. Although Full Ruche does not achieve the ideal fairness of 2-D torus, Ruche2 and Ruche3 reduce the stdev by 2.0× and 2.93×, respectively, compared to 2-D mesh. The mean latency is also reduced relative to 2-D torus by 1.18× for Ruche2 and 1.34× for Ruche3.

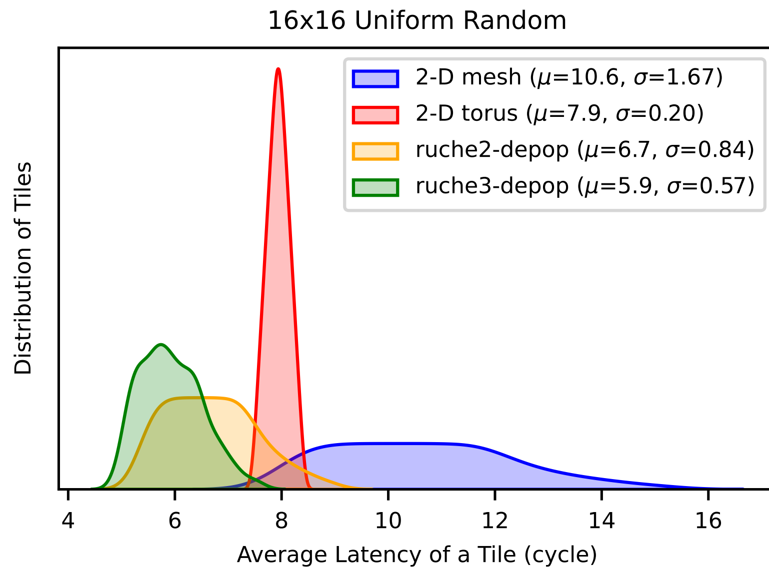


Figure 3.8: Distribution of average latency in 16×16 Uniform Random.

3.2.5 Half Ruche - Synthetic Traffic

We first evaluate Half Ruche networks using synthetic traffic patterns to study how network size, aspect ratio, and Ruche Factor affect performance. We evaluate 16×8 as a baseline, and scale up the network by 4× to evaluate scalability. Wide, rectangular aspect ratios are motivated by the desire to provide more bandwidth at the network edges. 32×16 maintains the aspect ratio of 16×8, but halves the compute-to-memory-tile ratio. In 64×8, the compute-to-memory-tile ratio is preserved, but the wider aspect ratio further stresses the bisection bandwidth. We consider two types of uniform random traffic that are common in cellular manycore [94]. *tile-to-tile* represents all-to-all communication among tiles. *tile-to-memory* represents a traffic directed to the top and bottom network edges for memory access.

Figure 3.9 shows the analysis of two traffic patterns across three network sizes. In general, Half Ruche topologies consistently outperform 2-D mesh in all cases. Saturation throughput of half-torus always falls in between that of 2-D mesh and Ruche2, supporting our earlier hypothesis that torus networks are limited by peak crossbar bandwidth. There is no significant difference between depopulated and fully-populated routers with the same Ruche Factor. In 16×8, the network size is too small to distinguish the performance benefits between Ruche2 and Ruche3. Even in 32×16, Ruche3 has only a slightly lower zero-load latency than Ruche2, while throughput saturates at a similar injection rate.

In tile-to-tile, adding Half Ruche channels nearly doubles saturation throughput. However, the benefit of Half Ruche is less noticeable in tile-to-memory. Ideally, the saturation throughput for tile-to-memory should match the compute-to-memory tile ratio. For example, in 16×8, the ratio of compute to memory tiles is 4:1, so each compute tile can ideally inject a packet every 4 cycles before saturating the memory bandwidth. However, due to X-Y DOR, the horizontal bisection bandwidth, which has the capacity to cross 16 packets per cycle in both direction (in case of 2-D mesh), is likely to saturate before saturating the memory tile bandwidth (32 packets per cycle). In 16×8, 2-D mesh saturates around 16∼17%, whereas Half Ruche brings it closer (∼21%) to the theoretical limit by breaking the bisection bottleneck.

In 32×16 tile-to-memory, Half Ruche appears to perform worse in terms of the saturation throughput. However, the compute-to-memory ratio is increased to 8:1 (12.5%), so the saturation throughput of 11% is actually pretty close to this theoretical limit.

64×8 is somewhat extreme in terms of aspect ratio. Typically, a 1:1 aspect ratio is preferred in 2-D mesh or Full Ruche to minimize network diameter. Nevertheless, 64×8 is an interesting option to consider, since it preserves the 4:1 compute-to-memory ratio. By adding more hops horizontally rather than vertically, it creates more opportunities to utilize horizontal Ruche channels to reduce latency. However, the wide aspect ratio exacerbates the bisection bottleneck. For 2-D mesh, the zero-load latency almost goes off the chart even at the lowest injection rate measured (5%).

In 64×8, performance difference between Ruche2 and Ruche3 is more distinguishable. Although Ruche3 does not reach the theoretical maximum saturation throughput (∼25%) expected from maintaining the 4:1 compute-to-memory tile ratio, it still improves performance significantly. We also explore Ruche4 for 64×8. Saturation throughput continues to improve with Ruche4, substantially higher than the maximum throughput measured in 32×16.

Table 3.4 summarizes these trends. In 16×8, the bisection bandwidth can easily exceed the memory-tile bandwidth by adding Ruche channels – with Ruche3, the bisection bandwidth becomes twice that of the memory-tile bandwidth. Maintaining the same 2:1 aspect ratio, 32×16 can do the same with Ruche channels; however, the compute-to-memory ratio is halved, which ultimately limits the per-core bandwidth.

64×8 preserves this compute-memory ratio, but at the expense of a more skewed aspect ratio (8:1). As a result, even with Ruche channels, matching bisection and memory-tile bandwidth becomes more difficult; in fact, a Ruche Factor as high as seven would be required to fully match them. Although not evaluated in this work, 32×8 with Ruche3 appears to be an attractive design point, as it can balance the bisection and memory-tile bandwidth 1:1.

In summary, the compute-memory tile ratio should first be selected based on application characteristics, as 4:1 may be too aggressive for some workloads. Given this ratio, the array aspect ratio and Ruche Factor can then be chosen to appropriately balance bisection and

memory-tile bandwidth. Ideally, the bisection bandwidth should be greater than or equal to the memory-tile bandwidth to avoid network bottlenecks. This analysis, however, considers bandwidth in isolation and does not account for the effects of latency, area, and power. The remainder of this section examines these additional metrics in greater detail.

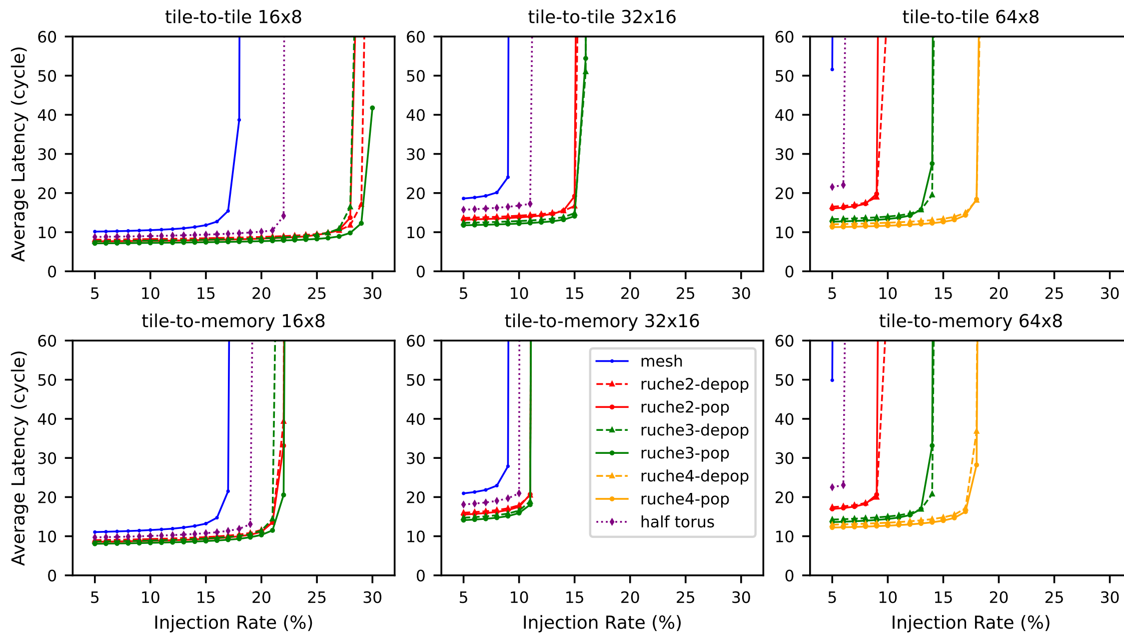


Figure 3.9: Synthetic traffic analysis on 16×8, 32×16, and 64×8.

Network Size	Aspect Ratio	NoC	Bisection BW	Memory Tile BW	Compute-Memory Ratio
16×8	2:1	mesh	16	32	4:1
		ruche2	**48**	32	
		ruche3	**64**	32	
32×16	2:1	mesh	32	64	8:1
		ruche2	**96**	64	
		ruche3	**128**	64	
64×8	8:1	mesh	16	128	4:1
		ruche2	48	128	
		ruche3	64	128	
32×8	4:1	mesh	16	64	4:1
		ruche2	48	64	
		ruche3	**64**	**64**	

Highlighted are where Bisection BW $\geq$ Memory Tile BW.

Table 3.4: Comparison of bandwidth ratio.

3.2.6 Half Ruche - Benchmark Speedup

In Section 3.2.6–3.2.9, we use the HammerBlade manycore simulator [94] to evaluate various network topologies using parallel workloads. The evaluation is based on full-system, RTL-level, cycle-accurate simulations of RISC-V core execution with various NoCs. Unlike *trace-driven* simulations, our *execution-driven* methodology preserves the dynamic feedback effect of network backpressure and remote load latency on core execution, providing a more realistic assessment of NoC behavior. The parallel workloads and datasets used in this study are summarized in Table 3.5. Our evaluation focuses on the compute phases of each program, during which the working set of data resides in the LLCs, and excludes data transfer phases between compute phases. During these transfer phases, the memory system dominates overall latency, obscuring the performance impact of the NoC and thus providing limited insight for network evaluation.

Figure 3.10 shows the parallel benchmark speedup relative to 2-D mesh and half-torus for both 16×8 and 32×16. Overall, Half Ruche NoCs provide consistent performance improvements over both half-torus and 2-D mesh across all benchmarks. Generally, fully-populated routers (pop) performs better than depopulated (depop), and Ruche3 performs better than Ruche2. This result appears to contradict the synthetic traffic analysis (Figure 3.9), which suggests there should not be much difference in performance.

This discrepancy can be explained by the nature of traffic patterns in real applications, which tend to be more bursty and structured than the randomized traffic used in synthetic simulations. Moreover, because our simulation methodology is execution-driven, packet injection timing is dynamically influenced by the changes in network congestion and latency. In contrast, the synthetic traffic testbench injects packets according to a fixed probability, independent of network state. As a result, the benefits of Ruche networks – particularly their ability to reduce congestion under bursty traffic – are more pronounced in realistic workloads.

The performance gains of Half Ruche are also more significant in 32×16 than in 16×8 networks, reflecting the increasing impact of bisection bandwidth and network diameter at larger scales. One exception is SpGEMM (US,RC) in 32×16, which shows limited improve-

ment. This behavior is attributed to heavy contention on an atomic add variable for dynamic linked-list node allocation, creating a persistent network hotspot that is exacerbated as the core array scales. With appropriate algorithmic changes, SpGEMM performance could likely be improved.

Another exceptional case is Jacobi in 32×16 half-torus, where performance drops by 20%. The Jacobi kernel frequently accesses the scratchpads of nearest-neighbor tiles. However, since folded torus topology skips every other tile, packets must take the longest route around the network to reach the nearest tiles. This problem exacerbates as the network size increases.

Benchmarks	**Input Dataset**
Jacobi	512×512×64 FP32
SGEMM	512×512×512 FP32
2-D FFT	16K/32K FP32
Barnes-Hut (BH)	16K/32K/64K bodies
BFS	See Graphs
PageRank (PR)	See Graphs
SpGEMM	See Graphs

Graph Name	**Type**	**Edges / Nodes**
offshore (OS)	Scientific	4.2M / 260K
roadNet-CA (CA)	Road	5.5M / 1.9M
road-central (RC)	Road	33.8M / 14.1M
road-usa (US)	Road	57.7M / 23.9M
ljournal-2008 (LJ)	Social	79.0M / 5.3M
hollywood-2009 (HW)	Social	113.9M / 1.1M
soc-Pokec (PK)	Social	30.6M / 1.6M

Table 3.5: Benchmarks and input datasets for NoC evaluation.

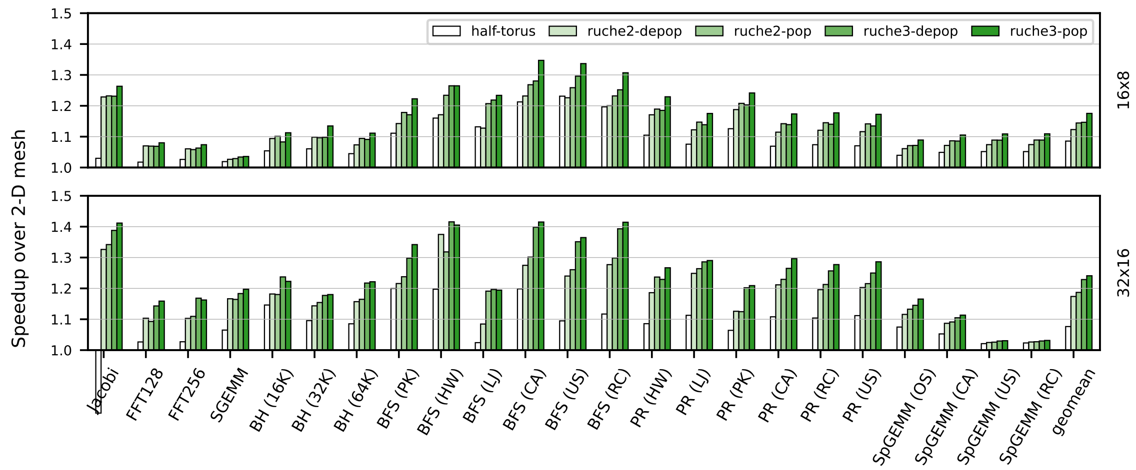


Figure 3.10: Speedup over 2-D mesh on 32×16 and 16×8.

3.2.7 Half Ruche - Benchmark Scalability

Here, we define scalability as the application speedup obtained by adding more compute nodes. Since we are quadrupling the number of cores, the most speedup we can expect is 4×. Figure 3.11 reports the speedup of 32×16 and 64×8 relative to 16×8 mesh across different network topologies.

Across all benchmarks, Ruche Networks consistently provide better scalability. Half torus always scales worse than Ruche and sometimes worse than even mesh. 64×8 mesh really struggles to make use of additional cores, because of the massive pressure on the horizontal bisection bottleneck.

At Ruche2, 32×16 scales better than 64×8, reflecting its more balanced aspect ratio and lower bisection pressure. Once Ruche3 is introduced, 64×8 performs better than 32×16 by exploiting its higher compute-to-memory tile ratio. For benchmarks with sequential and block-sized memory access patterns (e.g. Jacobi, FFT, SGEMM), Ruche3 with fully-populated could almost reach the ideal 4× scalability.

In contrast, BFS with social graphs (HW, LJ) demonstrates limited scalability despite the increase in core count. This behavior is primarily due to load imbalance and irregular memory access patterns, which reduce the effectiveness of additional compute resources.

Overall, the 32×16 configuration represents a more favorable design point than 64×8 when considering area and routing complexity. It delivers strong scalability with a relatively modest increase in Ruche channels, whereas, as noted in Table 3.4, the 64×8 configuration requires a substantially higher Ruche Factor to fully exploit its additional memory-tile bandwidth.

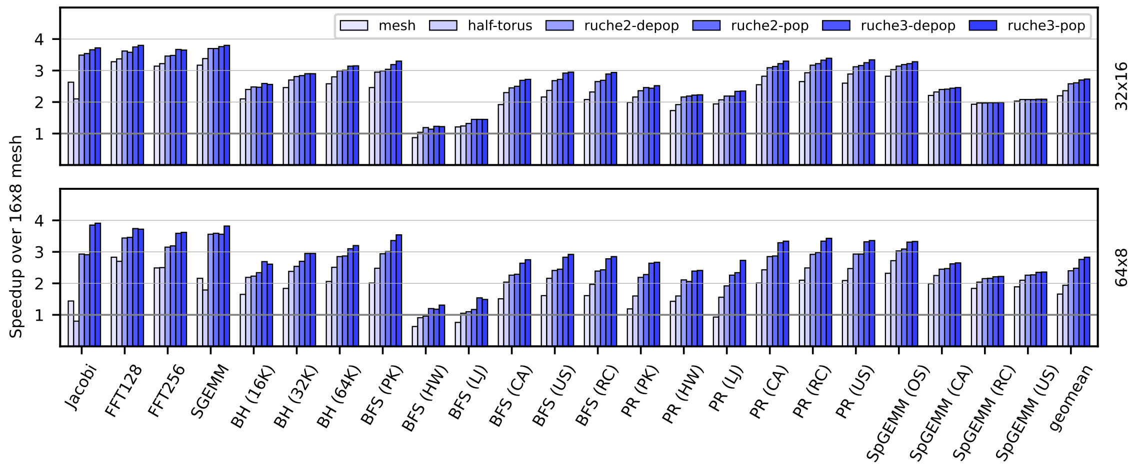


Figure 3.11: Scalability comparison between 64×8 and 32×16.

3.2.8 Half Ruche - Remote Load Latency

The average remote load latency is an important metric, because reducing it lowers both the software effort and the hardware resources required for latency hiding. We decompose the total remote load latency into two components: (1) *intrinsic* latency is the minimum latency under zero network load. (2) *congestion* latency is any additional latency caused by network contention.

Figure 3.12 reports the average remote load latency for the 32×16 array. The intrinsic delay is nearly identical across all benchmarks, suggesting that the IPOLY hashing [144], which interleaves the address space across LLC banks, effectively balances network traffic. For all workloads, both half-torus and Half Ruche reduce intrinsic delay (except for half-torus in Jacobi, as explained earlier). ruche2-depop reduces intrinsic latency by ∼27%, while more expensive routers provide only marginal incremental benefits.

Congestion-induced latency is especially high for regular workloads and PageRank on social graphs, where packet injection rates are very high. In contrast, SpGEMM (US, RC) are characterized as being latency-bound and pointer-chasing, because road networks in general have low degrees, and the SpGEMM kernels are implemented using linked lists. Due to latency reduction by Ruche links, cores stall less often and consequently inject packets more frequently. As a result, congestion latency increased slightly.

Nvertheless, Half Ruche effectively reduce congestion by providing more network links that cross the bisection. For some workloads, higher Ruche Factor or populated routers does not significantly reduce congestion-induced latency. However, congestion latency is never made worse by adding Ruche channels, so when the reduction of intrinsic latency is taken into account, the total remote load latency is almost always reduced.

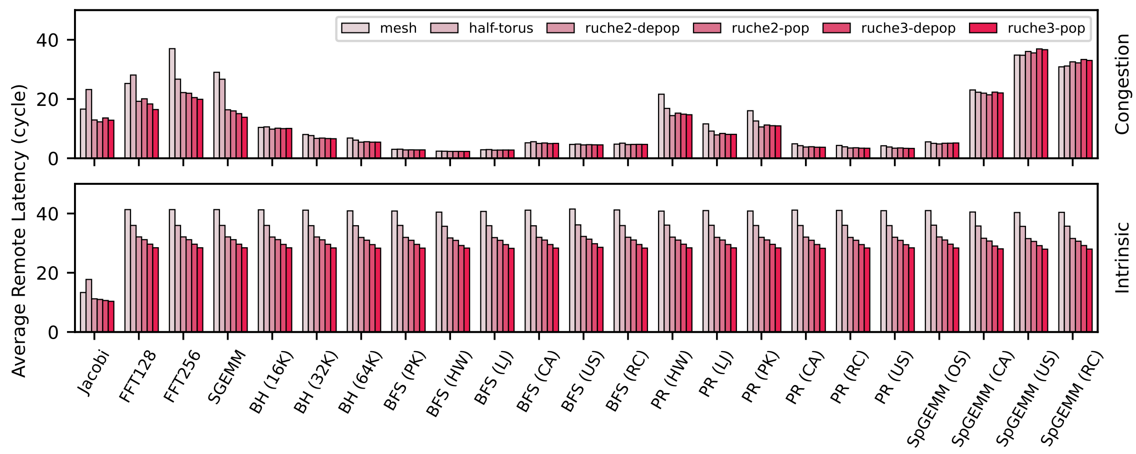


Figure 3.12: Average remote load latency for 32×16.

3.2.9 Half Ruche - Energy

We divide the total system energy into four components: (1) *core energy* accounts for the dynamic energy dissipated by the cores during instruction execution. The per-instruction energy comes from the measurement taken in [94]. (2) *stall energy* captures a leakage component of both core and router and ungated dynamic clock-tree energy, when the cores and NoC are idle. (3) *router energy* represents the dynamic energy consumed by NoC routers during packet transmission. This component is estimated using the same methodology to generate Table 3.3. (4) *wire energy* corresponds to the dynamic energy dissipated by long-range Ruche links. The wire energy of Ruche links is estimated by using a first-order repeater model [80] together with a process-independent, per-unit-length wire capacitance (0.2 pF/mm). The diffusion and gate capacitances of the repeaters driving the Ruche links are obtained from the 12 nm standard-cell library. We made the same assumption about the activity factor on each bit and the length of the Ruche wires as made in Table 3.3.

Figure 3.13 presents the total energy breakdown for 32×8 (normalized to energy spent by 2-D mesh). As expected, the core energy remains constant, since the total instruction count is not affected by changes in the network configurations. However, by reducing the remote load latency, the stall energy is reduced. For memory-bound workloads such as BFS, stall energy can constitute a large fraction of the total, even though the per-cycle stall energy is relatively small compared to the per-instruction energy .

Except for a few compute-intensive workloads (e.g. FFT and SGEMM), routers dissipate energy as much as cores do. In half-torus, we observe that energy savings from long wire is not enough to compensate for the increased router energy. In almost all benchmarks, half-torus ends up spending more total energy than 2-D mesh. Again, while ruche2-depop results in the sharpest reduction in energy, the fully-populated routers and higher Ruche Factors provide marginal additional benefits. Notably, even for Ruche Factor of 3, wire energy is only a small fraction of the total system energy.

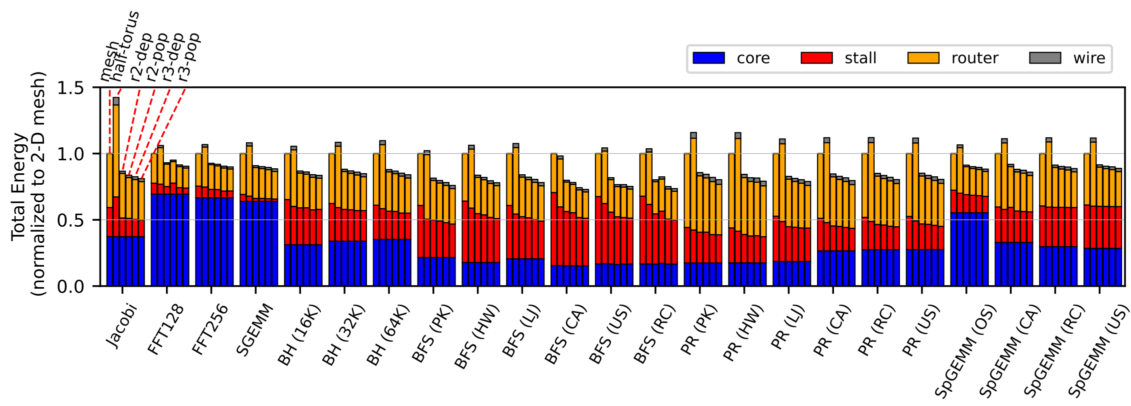


Figure 3.13: Total energy breakdown for 32×16 (normalized to 2-D mesh).

3.2.10 Half Ruche - Geomean Summary

Table 3.6 summarizes the Half Ruche evaluation with geomean scores. Across the board, ruche3-pop outperforms all other network configurations. As mentioned earlier, the general trend is that most gains come from ruche2-depop initially. This is an encouraging result, since most designs that already use 2-D mesh or torus can easily reap most of the benefits with a Ruche router with the least complexity. Looking at the area-normalized speedup, depopulated routers perform better than the fully-populated. Increasing Ruche Factor is a cost-effective means to increase performance. On the other hand, there is hardly any area-normalized speedup for half-torus (1.01×). We can summarize the results from Half Ruche evaluation with a following guideline: *save area with a depopulated router, and use longer wires for more cost-effective performance gains.*

Although Ruche NoC reduces the total NoC energy, the question still remains whether *NoC power* also decreases. If we define 'NoC power' as NoC energy divided by total runtime, as long as the NoC energy efficiency does not fall behind the speedup, NoC power should remain constant. Table 3.6 shows that NoC energy efficiency for Ruche NoCs is greater than speedup vs mesh for 32×16 by at least 0.07×; therefore, NoC power does indeed decrease. Similarly, *total power*, defined as total energy divided by total runtime, does not increase significantly for Ruche NoCs, since the difference between total energy efficiency and speedup vs mesh is very small (0.01-0.03×).

Metric	mesh	ruche2 depop	ruche2 pop	ruche3 depop	ruche3 pop	half torus
16×8 Speedup vs mesh	1.00×	1.12×	1.14×	1.15×	1.18×	1.09×
32×16 Speedup vs mesh	1.00×	1.17×	1.19×	1.23×	**1.24**×	1.08×
32×16 Scalability (vs 16×8 mesh)	2.20×	2.58×	2.61×	2.70×	**2.73**×	2.36×
64×8 Scalability (vs 16×8 mesh)	1.66×	2.40×	2.48×	2.76×	**2.83**×	1.94×
Load Latency Reduction (32×16, Intrinsic)	1.00×	1.28×	1.32×	1.38×	**1.44**×	1.12×
Load Latency Reduction (32×16, Congestion)	1.00×	1.21×	1.19×	1.20×	**1.22**×	1.05×
Load Latency Reduction (32×16, Total)	1.00×	1.27×	1.30×	1.35×	**1.40**×	1.11×
Energy Efficiency (32×16, Compute)	1.00×	1.12×	1.12×	1.15×	**1.16**×	1.06×
Energy Efficiency (32×16, NoC)	1.00×	1.28×	1.28×	1.30×	**1.35**×	0.75×
Energy Efficiency (32×16, Total)	1.00×	1.18×	1.18×	1.20×	**1.22**×	0.91×
Tile Area Increase	1.000×	**1.058**×	1.085×	1.063×	1.090×	1.071×
32×16 Speedup vs mesh (area normalized)	1.00×	1.11 ×	1.10×	**1.16** ×	1.14×	1.01×

Table 3.6: Summary of Half Ruche eval using geomean scores.

3.3 Related Work

This paper discusses in detail comparisons of Ruche networks with a wide variety of other proposed NoCs [5, 23, 29, 71, 102, 135, 178].

Generalized Express Cube (GEC) framework [71] proposes to express existing NoC topologies using the 6-tuple $\langle n, k, c, o, d, x \rangle$. However, these parameters are not general enough to express Ruche networks. The parameters $\langle n, k, c, x \rangle$ are orthogonal to describing the connectivity between nodes in each dimension. The parameters $\langle o, d \rangle$, which describe the router radix per dimension and sinks per channel, would be $\langle 4, 1 \rangle$ for Ruche networks. However, it lacks a parameter to describe the Ruche Factor (the skip distance of Ruche channels), so even the most broad generalization of express link topologies does not include Ruche networks.

Express Virtual Channel (EVC) [107] is a *flow control technique* that allows packets to bypass some of the pipeline stages in intermediate routers to reduce latency. EVC uses the same 2-D mesh topology without adding any new physical links; hence, there is no improvement in bisection bandwidth, and the throughput will converge to that of 2-D mesh at higher load or larger network. Instead, the mesh links are *virtualized* by adding express VCs that are prioritized over local ones. This prioritization enables EVC flits to skip some of the pipeline stages (e.g. buffer write, allocation) in intermediate routers, thereby reducing their latencies. However, EVC flits still need to be latched, and go through each crossbar, in every router on their way. In contrast, Ruche packets use the long-range physical links that skip the intermediate routers entirely. Fundamentally, while previous NoC techniques, such as speculative VC routers and EVC, sought to improve network performance by spending more area on *control logic*, usually with a tradeoff between performance and energy, Ruche NoC finds a way to efficiently utilize unused interconnects to improve *both* performance and energy.

EVC has two major flaws that limit its scalability for larger networks [104, 105]. First, EVC must conservatively guarantee that there is enough buffer space at the destination before it can safely inject EVC flits, and this leads to overprovisioning and underutilization of buffer spaces in average case. EVC uses *on-off signaling*, where the destination node sends

a token to the remote source to stop injecting more flits when the buffer space goes below a certain threshold. Since it takes multiple cycles for the token to reach the source, the total buffer space must account for the *worst-case*, which is calculated based on the number of cycles for this signal to reach the source and the number of flits that could already be in flight during that time. In other words, the minimum buffer space has to grow with the maximum EVC distance. In contrast, Ruche routers are minimally buffered with two-element FIFOs, which remain constant with the network size and Ruche Factor. Second, EVC is restricted to assign one express VC for each k-hop express paths, because control latency overhead required for dynamic assignment makes it impractical. Static assigning creates a problem where the source node may only send a limited number of packets for a particular distance, even though there may be other free VCs available. Furthermore, the source nodes that are further away take longer to learn from the destination node that the k-hop VC became free, thereby lowering the VC utilization. For these reasons, the maximum EVC distance is practically limited to only 3 to 4 hops.

Flit bubble flow control (FBFC) [118] proposes an alternative scheme for deadlock freedom in torus networks without using virtual channels. The main insight is that, even if cyclic channel dependency exists, as long as there is one bubble per each ring, packets can make forward progress without deadlock. FBFC imposes a restriction on when packets can be injected; the receiving FIFO must have at least one more free buffer slot than the packet length.

Dalorex [130], a manycore architecture that accelerates workloads with irregular memory access patterns by remote task invocation, purports to have done a comparison between Ruche and torus networks using C++ simulation. While no exact details were given about which Ruche Factor or crossbar schemes were used, it reports that Ruche has a larger area overhead and lower performance than torus. Figure 3.7 shows that 2-D torus has a larger area with virtual channels and depopulated Ruche routers in consideration. Dalorex distributes its data array using low-order index bits so all-to-all uniform random best reflects its traffic pattern. Figure 3.6 shows that Ruche networks achieve much higher throughput than 2-D torus under uniform random traffic.

Chapter 4
SILICON PROTOTYPE

This chapter presents the 12 nm implementation of Ruche Networks on 2048-core HammerBlade ASIC.

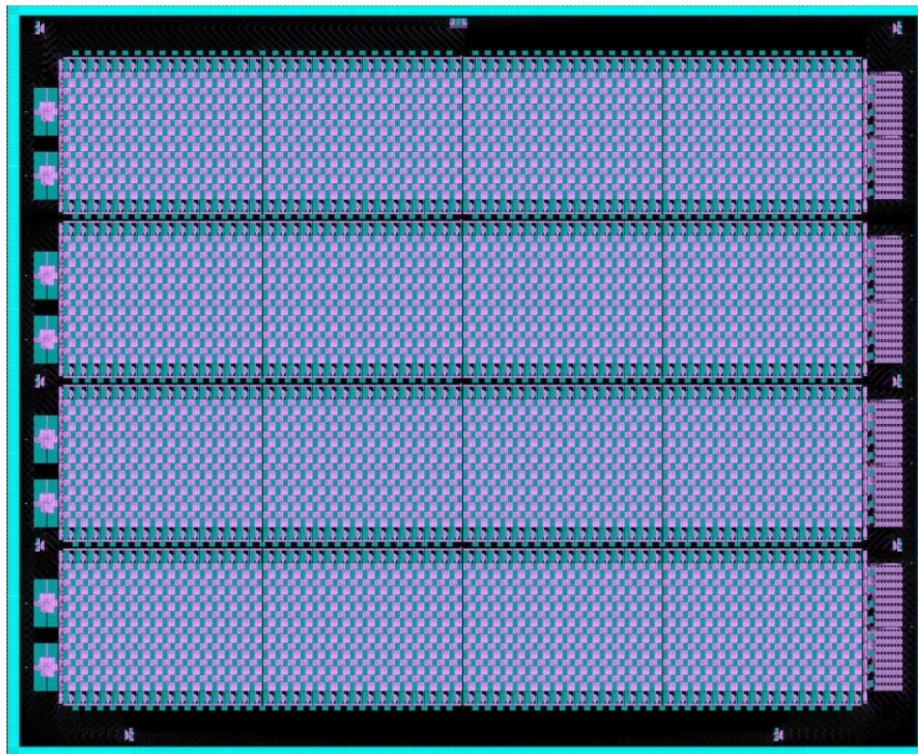


Figure 4.1: 11×9 mm^2 layout of 2048-core HammerBlade ASIC.

4.1 Implementation

This section describes the key technical considerations related to the hierarchical design methodology and the semi-automated process for the construction of HB Cells with Ruche Networks.

4.1.1 Tile-based Hierarchical Design Methodology

Figure 4.1 shows the 99 mm^2 chip with roughly 3.3 billion transistors. The chip is partitioned into four clock domains. Each clock domain contains a *Cell Row*, which is organized as a 4×1 array of HB Cells. Each Cell comprises a 16×8 array of HB compute tiles, with two 16×1 cache tile arrays located along the top and bottom sides.

Each Cell Row is padded with source-synchronous, single-data-rate interface blocks that facilitate clock-domain crossings between adjacent Cell Rows and other IP blocks on the sides (e.g. off-chip IO interface, OS-capable processors [139]).

Building a chip at this scale presented many serious technical challenges. Multi-project wafer (MPW) shuttle runs, which aggregate multiple projects onto a shared wafer, impose strict and inflexible deadlines. As it turns out, excessive CAD tool runtime, which can be as long as several days or even weeks if not carefully managed, proved to be one of the most critical risks to meeting these deadlines.

Figure 4.2 summarizes the CAD-based chip implementation flow, which is highly iterative and consists of multiple dependent steps. Errors introduced at any stage of the flow may not manifest until much later, often appearing as timing violations or DRC/LVS failures. Once such an issue is detected, the designer must trace it back through the flow, apply a corrective fix, and re-run the affected stages.

In some cases, the root cause and fix are straightforward. In others, it may involve sifting through thousands of lines of cryptic warning and error messages to determine where the bug was introduced. Compounding this challenge, user support for proprietary CAD tools is often limited, making systematic debugging both time-consuming and error-prone.

Next to each step in Figure 4.2 is the corresponding CAD tool runtime required to place and route a single HB compute tile. Even for this relatively small block, the total end-to-end runtime already reaches *2 hours and 48 minutes.* A simple linear extrapolation suggests that implementing a 2048-core manycore array using a flat, non-hierarchical design approach could require *up to 238 days* of CAD runtime. In practice, CAD runtime does not scale linearly with design size; global placement, routing congestion, and timing closure typically introduce superlinear behavior, making flat implementations even less tractable at

scale.

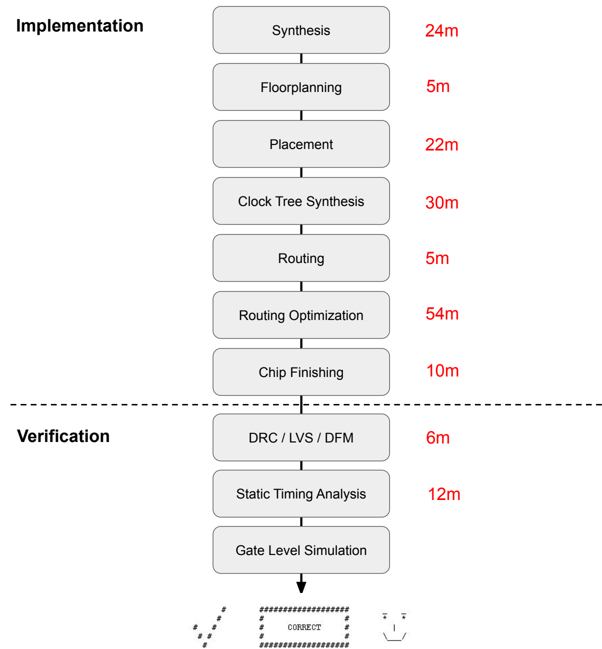


Figure 4.2: An overview of automated VLSI CAD flow.

Therefore, it was crucial to exploit a *tile-based hierarchical design methodology* to the greatest extent possible. The core idea is to construct the architecture from identical, rectangular tiles interfaced through NoC routers. Each tile encapsulates a manageable chunk of logic that can go through the CAD flow (Figure 4.2) in a reasonable amount of time. Once validated, these tiles can be replicated across the chip at essentially no additional CAD cost. There may be several kinds of tiles with different functionalities, which can go through the CAD flow in parallel.

Importantly, these tiles collectively encapsulate the entire architecture – including storage, processing and networking logic – such that no standalone logic remains between tiles that would otherwise require placement and routing. At the top level, only short inter-tile connections and interfaces to the pad rings are required. Figure 4.3 illustrates the transformation of a flat design into a tile-based hierarchical implementation.

This approach eliminates a substantial portion of algorithmic complexity in the CAD flow (Figure 4.2). For example, logic synthesis is simplified because the top-level netlist consists primarily of tile instantiations and their connections. Floorplanning and placement are also greatly simplified, as pre-placed tile macros replace millions of individual standard cells. Short inter-tile wires can be routed efficiently, reducing both routing runtime and optimization effort.

Nevertheless, significant challenges remain – notably clock tree synthesis and inter-tile timing closure. Unlike tile arrays, clock trees are not uniform structures that can be trivially replicated. Moreover, timing closure across tile boundaries may fail if excessive clock skew is introduced. The following sections describe how these challenges are managed.

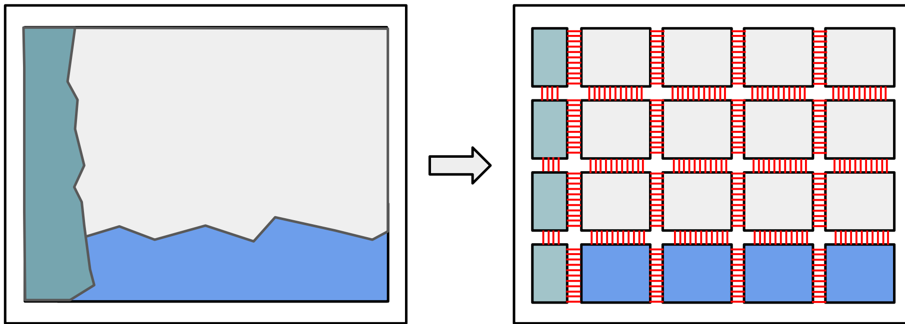


Figure 4.3: Decomposing an architecture using the tile-based design methodology.

4.1.2 Constraining Tile Interfaces

The approach I adopted focused less on eliminating clock skew through a "perfect" clock tree, but more on making individual tiles *tolerant* to small clock skews. The APR (Auto Place and Route) tool already produces a reasonably balanced clock tree in reasonable time, provided the tile array is not excessively large.

Figure 4.4 shows the setup timing path between two adjacent tiles. Two types of clock skew must be considered. First, external clock propagation delay – the difference in arrival time of the clock signal from the source to the sender and receiver tiles, denoted as $t_{clock,ext,send}$ and $t_{clock,ext,recv}$. Second, internal clock propagation delay – the delay from the

tile's clock port to the specific sender and receiver flip-flops, denoted as $t_{clock,int,send}$ and $t_{clock,int,recv}$. In some cases, if the sender and receiver flops are the same, this internal skew is zero.

In addition, the sender flop has a clk-to-q delay ($t_{clk-to-q}$) that depends on clock slew. Similarly, the receiver flop has a setup time (t_{setup}), which also depends on clock slew. Finally, there is the data propagation time ($t_{data-prop}$) between the sender and receiver flops.

To meet the setup timing, the following inequality must hold:

$$\begin{aligned} t_{clk-period} - t_{clk-uncertainty} > & t_{clk-to-q} + t_{data-prop} + t_{setup} \\ & - t_{clk,ext,recv} + t_{clk,ext,send} - t_{clk,int,recv} + t_{clk,int,send} \end{aligned} \tag{4.1}$$

By setting the clock period to 1000 ns and the clock uncertainty to 20 ns, and rearranging the timing terms, the equation now looks like:

$$\underbrace{980 - t_{clk-to-q} - t_{data-prop} - t_{setup} + t_{clk,int,recv} - t_{clk,int,send}}_{\text{clock skew slack (setup)}} > \underbrace{t_{clk,ext,send} - t_{clk,ext,recv}}_{\text{external clock skew}} \tag{4.2}$$

In this equation, the right-hand side represents the external clock skew, while the left-hand side corresponds to the *clock skew slack* – the maximum allowable clock skew between two tiles. Importantly, the individual components of the clock skew slack can be directly extracted from timing reports generated by the APR tool. After APR completes, chip designers can parse these reports to verify that sufficient clock skew slack exists along every inter-tile link. This verification process has been automated using a combination of Tcl and Python scripts.

How can we set the timing constraints to maximize this clock skew slack? The most straightforward way is to minimize $t_{data-prop}$, which typically dominates the critical path. The internal clock propagation delays ($t_{clk,int,send}$ and $t_{clk,int,send}$) are likely to cancel each other out, as APR tends to generate a balanced clock tree within a tile. Finally, the clk-to-q delay and setup time depend on the resulting clock slew, which APR also optimizes to maintain robust timing.

The HB tiles are interfaced with bi-directional NoC channels using ready-valid handshaking in six directions. The routers have a single pipeline stage, where the input ports feed

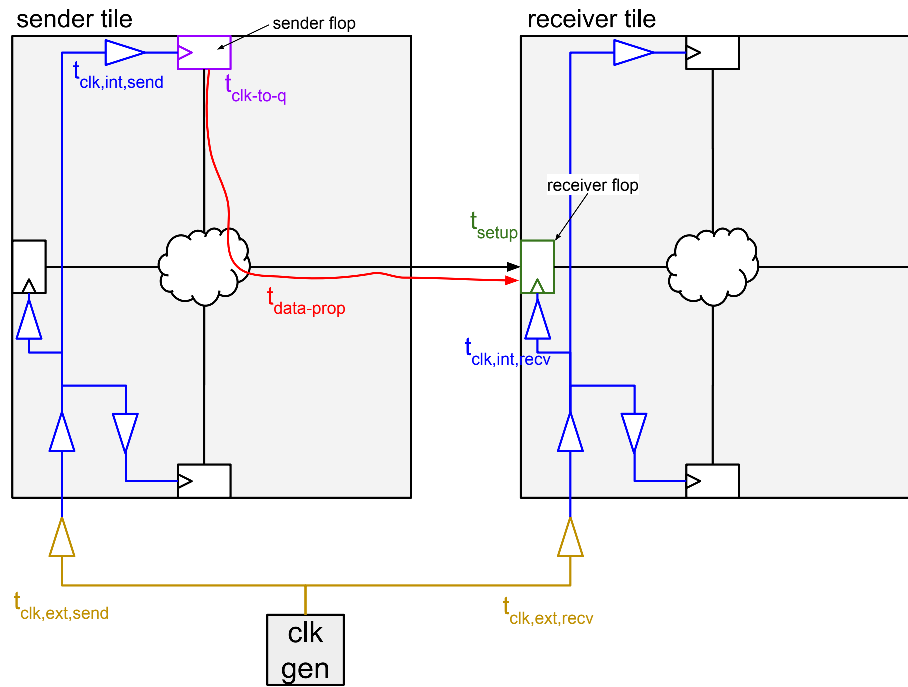


Figure 4.4: Setup timing path between two tiles interfaced with NoC routers.

directly into two-element FIFOs, and the outputs of the FIFOs pass through the crossbar before reaching the output ports.

Conventionally, input and output delays between two IPs are set using *time budgeting*. For example, each port might be allocated 50% of the clock period, or sometimes 40% per port, leaving a safety margin so the sum does not exceed the full clock period. The approach I adopted differs from this conventional method. As shown in Eq 4.2, the most effective way to maximize clock skew slack is to minimize $t_{data-prop}$. Therefore, it makes sense to set the maximum possible input and output delays on the tile interfaces, even if the sum exceed the clock period, to produce the fastest possible circuit.

Currently, finding these optimal delays is a manual and iterative process. First, start with some large delay values (e.g. 900 ns) that are likely to fail. For each iteration, check the design for timing correctness. There should be no timing violation in reg-to-reg paths. There should be sufficient clock skew slack on every network port (verified using the automated

Tcl/Python scripts). There should be no physical design errors (DRC, LVS, etc). If any violation exists, gradually reduce the delays until the design converges to a solution free of violations.

Figure 4.5 summarizes the final delay values for each network type, signal type, and network direction used during logic synthesis. Only the max delays are shown, because logic synthesis does not account for hold constraints in our setup. All input ports are constrained with the same delays, as they all connect to identical two-element FIFOs. Output delays, however, differ because each output has unique crossbar connectivity (Figure 3.4).

Input Side

Network	Direction	Signal Type	Max Delay (ns)
REV	ALL	Data (in)	935
REV	ALL	Valid (in)	850
REV	ALL	Ready (out)	910
FWD	ALL	Data (in)	935
FWD	ALL	Valid (in)	845
FWD	ALL	Ready (out)	890

Output Side

Network	Direction	Signal Type	Max Delay (ns)
REV	W/E	Data (out)	610
REV	W/E	Valid (out)	655
REV	W/E	Ready (in)	885
FWD	W/E	Data (out)	570
FWD	W/E	Valid (out)	685
FWD	W/E	Ready (in)	850
REV	N/S	Data (out)	665
REV	N/S	Valid (out)	740
REV	N/S	Ready (in)	890
FWD	N/S	Data (out)	570
FWD	N/S	Valid (out)	710
FWD	N/S	Ready (in)	860
REV	RE/RW	Data (out)	605
REV	RE/RW	Valid (out)	660
REV	RE/RW	Ready (in)	855
FWD	RE/RW	Data (out)	605
FWD	RE/RW	Valid (out)	700
FWD	RE/RW	Ready (in)	860

Figure 4.5: Final input and output delay constraints used for logic synthesis.

Similarly, we need to ensure that the clock skew slack for hold timing is sufficient. Similar to 4.1, the following inequality must be satisfied to meet the hold timing:

$$\begin{split} t_{clk-to-q} + t_{data-prop} + t_{clk,ext,recv} - t_{clk,ext,send} \\ + t_{clk,int,recv} - t_{clk,int,send} > t_{hold} + t_{clock-uncertainty} \end{split} \tag{4.3}$$

Again, with some rearranging of terms:

$$\underbrace{t_{clk-to-q} + t_{data-prop} + t_{clk,int,recv} - t_{clk,int,send} - t_{hold} - 20}_{\text{clock skew slack (hold)}} > \underbrace{t_{clk,ext,send} - t_{clk,ext,recv}}_{\text{external clock skew}} \tag{4.4}$$

Input Side						Output Side				
Network	Direction	Signal Type	Max Delay (ns)	Min Delay (ns)		Network	Direction	Signal Type	Max Delay (ns)	Min Delay (ns)
REV	ALL	Data (in)	870	200		REV	W/E	Data (out)	400	-60
REV	ALL	Valid (in)	680	200		REV	W/E	Valid (out)	290	-50
REV	ALL	Ready (out)	820	200		REV	W/E	Ready (in)	470	-70
FWD	ALL	Data (in)	870	200		FWD	W/E	Data (out)	435	-60
FWD	ALL	Valid (in)	665	200		FWD	W/E	Valid (out)	290	-50
FWD	ALL	Ready (out)	800	200		FWD	W/E	Ready (in)	470	-70
						REV	N/S	Data (out)	480	-60
						REV	N/S	Valid (out)	250	-60
						REV	N/S	Ready (in)	515	-70
						FWD	N/S	Data (out)	450	-60
						FWD	N/S	Valid (out)	235	-60
						FWD	N/S	Ready (in)	490	-70
						REV	RE/RW	Data (out)	405	-40
						REV	RE/RW	Valid (out)	280	-55
						REV	RE/RW	Ready (in)	670	-30
						FWD	RE/RW	Data (out)	450	-40
						FWD	RE/RW	Valid (out)	295	-55
						FWD	RE/RW	Ready (in)	575	-30

Figure 4.6: Final input and output delay constraints used for APR.

A standard way to fix hold timing is to make the cells weaker or insert buffers to lengthen the path. As mentioned previously, there is more logic after the NoC FIFOs than before, giving the APR tool greater flexibility to resolve hold timing on the output side rather than the input side.

Figure 4.6 shows the final delay values used during APR, which include min delays for hold timing. The max delays are slightly lower than those in Figure 4.5 because wire capacitance is now considered during APR. The input ports are assigned a min delay of 200 ns to guide the tool not to modify these paths – the wires from the input port should go directly to the FIFO flops. The output side are assigned negative min delay, encouraging the tool to account for clock skews and add necessary buffering. Ruche directions (RE/RW) have slightly smaller min delays, because the physical Ruche links naturally introduce additional delays.

Finally, Figure 4.7 shows the distribution of clock skew slack, measured on the final HB manycore tile, for both setup and hold. It shows that, on every link and in every direction, there is a sufficient slack to tolerate a minor external clock skew.

Providing a slightly pessimistic estimate of clock slew was another important constraint. By default, the CAD tools assume that the clock signal arriving at a tile has perfect edges (i.e. zero transition time), which is never the case in practice. Often, static timing analysis

may appear clean at the individual tile level but fail once tiles are arranged in an array due to non-ideal clock edges. Therefore, it is essential to explicitly set the clock transition time, as shown below.

```
set_input_transition 10 -min [get_ports clk_i]
set_input_transition 50 -max [get_ports clk_i]
```

Here, the max transition time (used for setup analysis) is set to 50 ns, slightly slower than the actual, while the min transition time (used for hold analysis) is set to 10 ns, slightly faster than the actual. This approach helps ensure that timing analysis accounts for real-world clock slew variations.

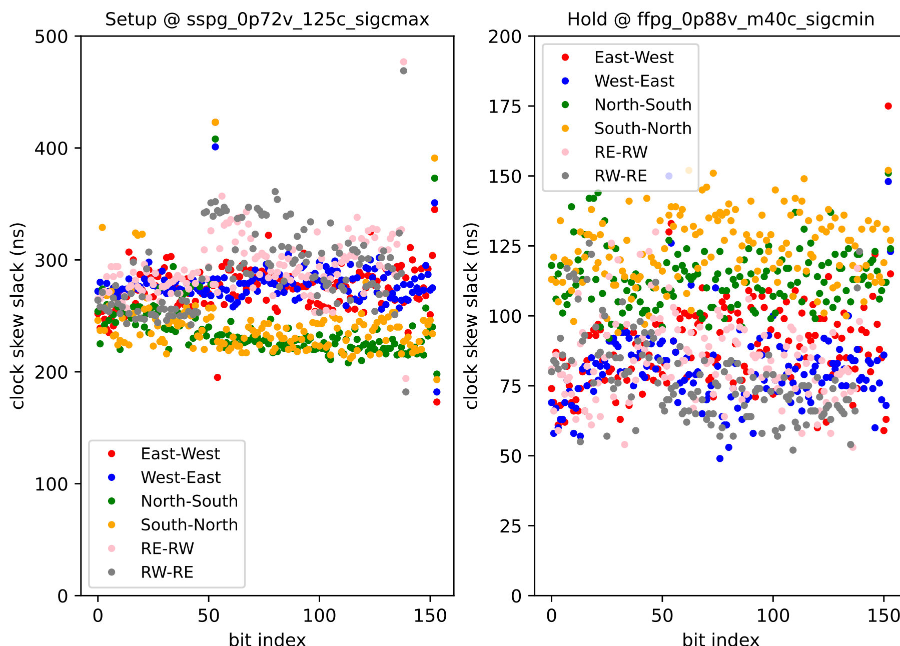


Figure 4.7: Final clock skew slack on HB compute tile interfaces.

4.1.3 Building a Cell Row

Figure 4.8 shows the module hierarchy for HB Cell Rows, which are composed of 10 distinct leaf-level tiles. As mentioned before, the Cell Row is surrounded by interfacing blocks that facilitate the crossing of network packets between clock domains. Although these blocks

essentially perform the same function, eight different versions were required – four for the sides (N,S,E,W) and four for the corners (NE, NW, SE, SW). This is because the tiles cannot be simply flipped or mirrored due to constraints on power grid geometry and port locations. However, since the leaf-level tiles can be placed and routed in parallel, there is no additional runtime cost.

Building the clock tree for a Cell row is done in two level. At the mid-level, the clock tree is constructed using the regular EDA flow, which effectively keeps the global clock skew under control. For instance, in a 16×2 compute tile array, the maximum clock skew between the clock pins of all tiles was less than 50 ns – well below the clock skew slacks shown in Figure 4.7.

Figure 4.9 shows the final floorplan for the Cell Row and its clock tree. The clock tree is manually generated by a custom Tcl script, which runs before routing data signals. The script takes the coordinates of each clock inverter and the desired tree topology as input. The clock pin for the Cell row is located at the top center, with the tree branching down to the middle row and then out in two directions. Each inverter has 16× drive strength and super-low threshold voltage (SLVT). The number of inverter stages should be kept even to avoid 180 degree phase shift.

Whenever a fanout occurs in the clock tree, the next-stage inverters must be placed close together to the driver; otherwise, the driver must drive multiple long wire segments, which degrades clock slew. Once the slew begins to degrade at a given stage, it is difficult to recover in subsequent stages. Fortunately, the timing reports indicate where the slowdown occurs, so debugging this problem was not so painful. After completing the clock tree and data signal routing, the entire Cell Row undergoes static timing analysis across all 17 process corners, taking roughly 8 hours (done in parallel). Debugging and iterating on the manual clock tree was arguably the most time-consuming part of the process.

4.1.4 Building Ruche Networks

Figure 4.10 shows a detailed layout view of how physical Ruche links are mapped on the 2-D silicon. The pins for the local link and three Ruche links (one originating from the tile

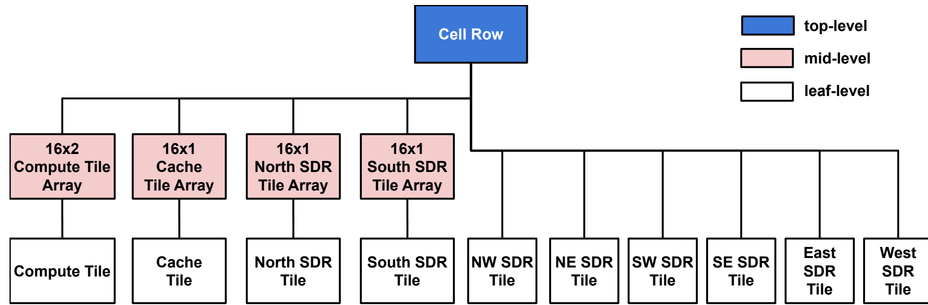


Figure 4.8: Module hierarchy for Cell Row.

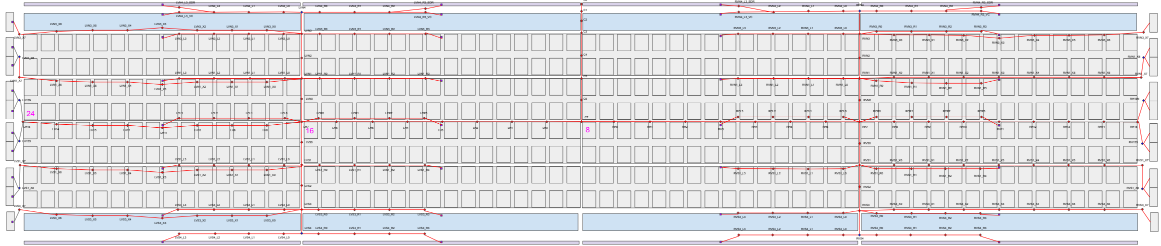


Figure 4.9: Cell Row floorplan and clock tree.

and two feedthroughs) are interleaved bitwise along the edge of tile macro. The i-th bit of each link is placed close together, so that inter-tile wires need to cross over only a few wiring tracks to connect to the Ruche buffers and the neighboring tiles. These feedthrough wires are placed and frozen (i.e. subsequent steps cannot modify them) before routing clock and other signals during the CAD flow.

Ruche buffers driving the long wires are positioned in a narrow gap between tiles (5.7 um), with equal drive strength to ensure one wire does not adversely affect the others. Although neighboring feedthrough wires may have large coupling capacitance, the signal arrival times are slightly skewed (e.g. signals crossing the first tile versus the second tile) to avoid the worse-case Miller effect (e.g. signals switching in opposite directions simultaneously). The zoomed-in view shows the Ruche buffers arranged in columns of three, with flylines indicating the connections between the buffers than the tile pins.

Figure 4.11 illustrates a cross-sectional view of the metal layer stack and the placement

of Ruche feedthrough wires. Long-range Ruche links feedthrough the tiles in straight lines using less resistive, mid-level (2X) metal layers – these layers are above (and not occupied by) the SRAM macros. The feedthrough wires are placed on two separate horizontal layers (note that there is an additional vertical layer between them), to reduce capacitive coupling. Within each layer, the feedthrough wires are evenly distributed every four tracks, ensuring a uniform distribution of free routing tracks for other signals.

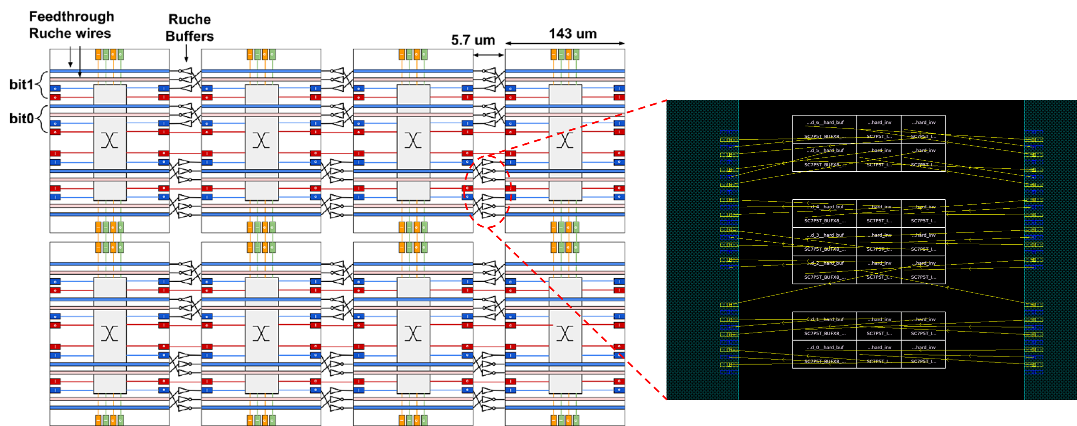


Figure 4.10: Floorplan view of Ruche feedthrough wire placement.

Ruche Networks on this chip demonstrate the high wiring density that can be achieved. Across the 187 um span of a tile, there are 2924 routing tracks in the two mid-level horizontal metal layers, of which roughly 2416 tracks (∼82%) are not blocked by power grids. Each tile side contains 1148 pins, occupying 47.5% of the available unblocked tracks. Figure 4.12 shows the compute tile layout and its area breakdown. The SRAM macros for the icache and scratchpad are placed in opposite corners. Leveraging NoC symbiosis [140], the router (red) and core logic (pink) cells are co-placed within the tile. Since Ruche routers are input-buffered, the input pins are placed near regions free of SRAM macros, allowing input buffers to be placed nearby. This was a subtle design decision that significantly improved tile routability. FWD and REV routers together occupy 13.5% of the total area. Despite the high routing density, Ruche Networks achieve a high area utilization of 85.3%, with physical-only cells (e.g. tap, filler cells) occupying the rest. This result suggests that a

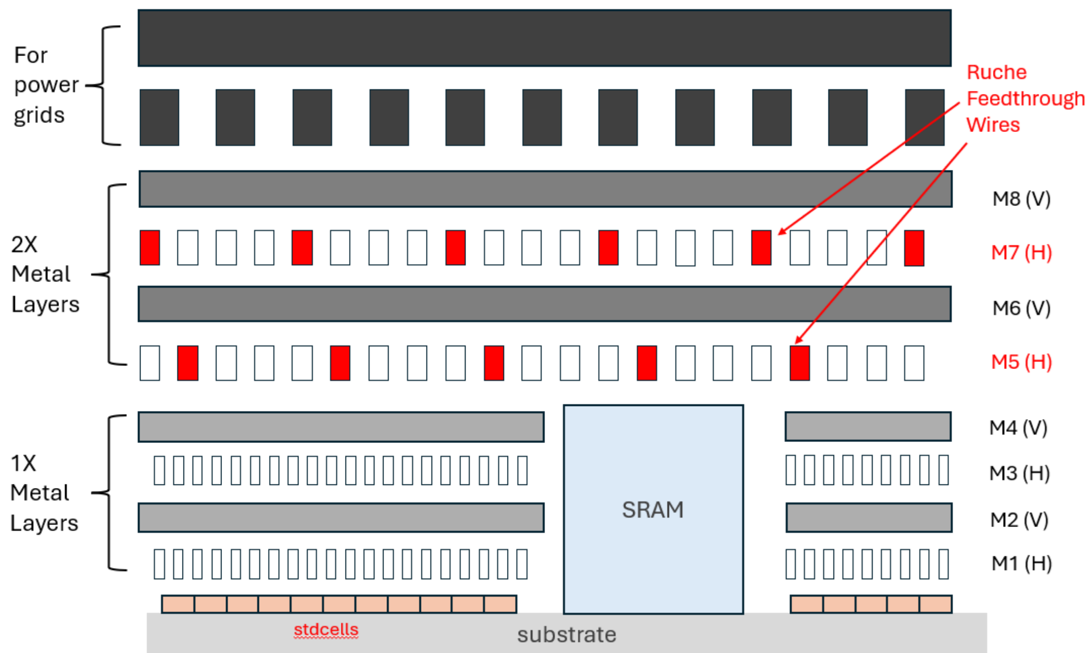


Figure 4.11: Cross-sectional view of Ruche feedthrough wire placement.

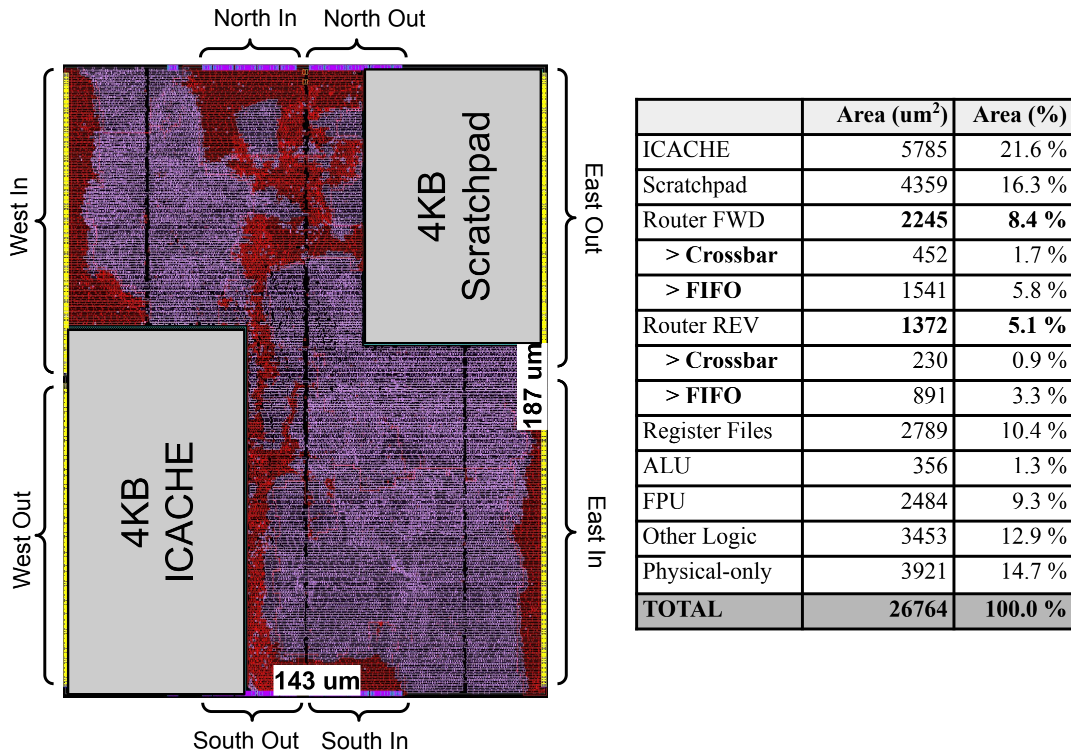
substantial portion of previously unused wiring resouces can be effectively leveraged by Ruche Networks.

4.2 Chip Measurement

This section presents selected measurements collected from the fabricated chip.

4.2.1 Shmoo Plot

Figure 4.13 shows the shmoo plot obtained from testing Ruche Networks on the chip. These measurements were collected under heavy network load using a uniform random traffic pattern, with payloads generated by a Pseudo-Random Binary Sequence (PRBS). Each core performed remote loads and stores to all other tiles to verify correct operations across every network path. At the nominal 0.8 V, the chip operates at 1480 MHz. The chip attains its maximum operating frequency of 1780 MHz at 1.0 V.



	Area (um^2)	Area (%)
ICACHE	5785	21.6 %
Scratchpad	4359	16.3 %
Router FWD	**2245**	**8.4 %**
> Crossbar	452	1.7 %
> FIFO	1541	5.8 %
Router REV	**1372**	**5.1 %**
> Crossbar	230	0.9 %
> FIFO	891	3.3 %
Register Files	2789	10.4 %
ALU	356	1.3 %
FPU	2484	9.3 %
Other Logic	3453	12.9 %
Physical-only	3921	14.7 %
TOTAL	**26764**	**100.0 %**

Figure 4.12: HammerBlade compute tile layout and its area breakdown.

4.2.2 Packet Energy vs Distance

Figure 4.14 shows a trend in packet energy required to send a request and receive a response from remote tiles, measured on a per-bit, per-length basis (pJ/bit/mm). The methodology follows that of [121]. A subset of tiles executed an unrolled kernel loop that launched multiple remote requests in the absence of network conflicts, and the current consumption was measured. The difference in current measurements between sending packets to itself versus to remote tiles, combined with the packet injection rate, was used to derive the energy per packet as a function of distance. Figure 4.14 exhibits characteristic dips in Ruche packet energy every three hops. The dips occur because the long-range Ruche links skip 3 hops in one cycle, consuming less energy than traversing the same distance through multiple

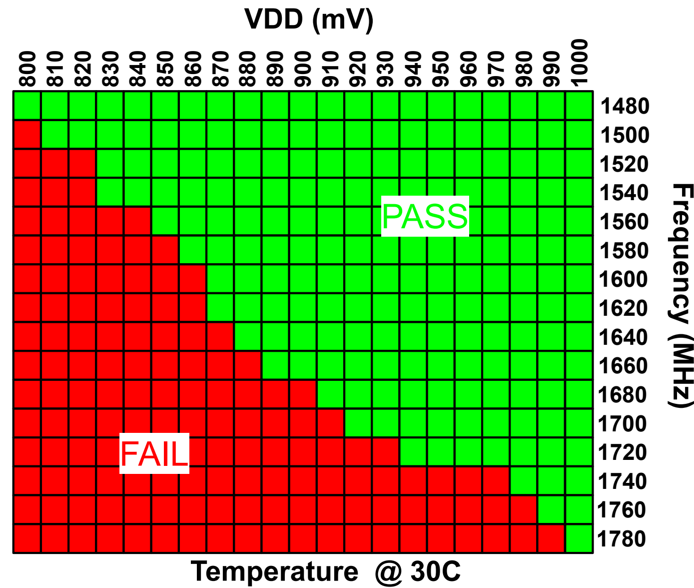


Figure 4.13: Chip measurement: shmoo plot.

local links. 2-D mesh energy is extrapolated from the first three data points. This result experimentally validates the simulation results reported in [95]. On average, Ruche packet energy costs 0.056 pJ/bit/mm, making them 1.85× more energy efficient than 2-D mesh.

4.3 Chip Comparison

This section presents both qualitative and quantitative comparisons with previous chip designs.

4.3.1 NoC Comparison

Table 4.1 compares HammerBlade with prior mesh-based manycore designs. HammerBlade operates at significantly higher frequency (1780 MHz at 1.0 V), despite having long-range Ruche links. Its NoC aggregate and bisection bandwidths are also considerably higher, although this advantage is partly due to HB's larger die area and greater number of network routers. The routing density between HB tiles is substantially higher than that of other designs in the same 12 nm technology (e.g. DECADES, FlooNoC). Despite FlooNoC's wide

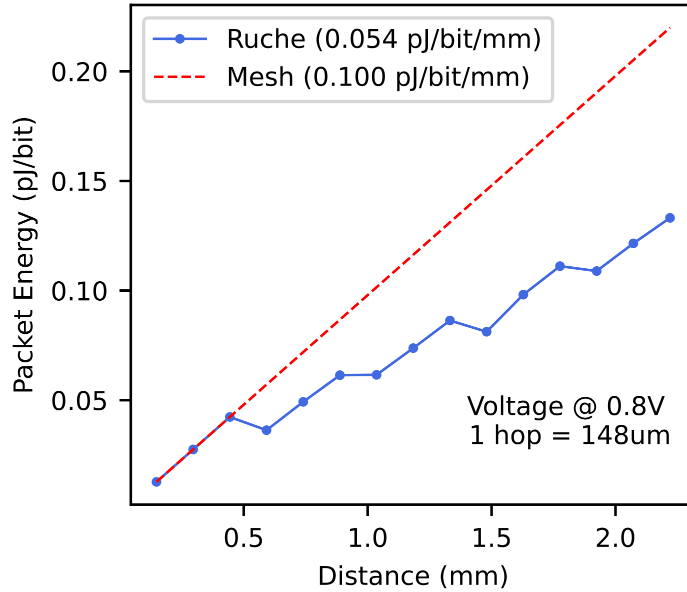


Figure 4.14: Chip measurement: energy per packet.

channel widths (three networks totaling 816 bits), its 2-D mesh topology still struggles to fully utilize interconnect resources. In addition, Ruche Networks reduces packet latency below that of Celerity, which was the previous fastest (x+y).

4.3.2 CoreMark Comparison

Table 4.2 compares the CoreMark score (CM) of HammerBlade with other state-of-the-art parallel architectures that have reported CoreMark scores, as well as with the current single-chip record holder. HB achieves a CM of 8.7M, a 1.43× improvement over the current record holder, AMD's EPYC 9755 at 6.0M – despite the fact that EPYC 9755 used a 4 nm process and 11.4× area. HB also surpasses Celerity, the previous RISC-V record-holder, in both compute density and energy efficiency, with improvements of 1.65× in CM/mm^2 and 1.15× in CM/W, respectively.

Table 4.1: NoC comparison table.

Metric	**KiloCore** [34]	**Piton** [121]	**Celerity** [146]	**DECADES** [60]	**FlooNoC** [56]	**This Work**[a]
Technology	32 nm	32 nm	16 nm	12 nm	12 nm	12 nm
Implementation	Silicon	Silicon	Silicon	Silicon	Post-Layout	Silicon
Voltage (V)	1.1	1.0	0.60 - 0.98	1.2	0.8	0.8-1.0
Frequency (MHz)	1780	500	1400	911	1260	**1480-1780**
NoC Topology	3×Mesh	3×Mesh	2×Mesh	3×Mesh	3×Mesh	**2×Half Ruche**
Network Size	32×31	5×5	16×31	12×9	4×8	**64×40**
Aggregate BW (Tb/s)	388.4	11.3	361	68.2	104	**2572.6**
Bisection BW (Tb/s)	4.23	0.96	4.00	4.20	16.8	**53.2**
Routing Density[b] (bit/mm)	408	354	1002	712	1477	**4289**
Packet Latency[c] (cycle)	2*(x+y)+1	x+y+t+1	x+y	N/A	2*(x+y)	**(x+y)-2*⌊(x-1)/3⌋**

[a] All measurements at 0.8V except for the max freq. [b] Includes both horizontal and vertical sides. [c] Latency for x horizontal, y vertical hops, and t turns.

4.4 Related Work

FlooNoC [56] is specialized for high-bandwidth bulk data transfers required by ML accelerators, featuring very wide (∼512-bit) data links. Similar to Ruche, FlooNoC addresses the challenge of better utilizing on-chip wiring resources. It employs a low-complexity 2-D mesh router with minimal input buffers and shallow pipeline stages. Separate physical channels are used for the three message classes required by AXI4 (req, resp, wide), rather than relying on virtual channels to share a single physical link. However, using a wide mesh is not the most effective way to improve wire utilization, as router area must increase linearly. Moreover, datapaths in the underlying core architecture must be widened to match the NoC channel width (which incurs even more area overhead), otherwise it introduces serialization and deserialization latency.

MemPool [41], implemented in 22 nm, provides a low-latency interconnect between processing elements and distributed L1 memory banks using hierarchical crossbars. It scales to

Table 4.2: CoreMark score (CM) comparison.

Metric	**DECADES [60]**	**CIFER [42]**	**Celerity [146]**	**EPYC 9755**	**This Work**
Technology	12 nm	12 nm	16 nm	4 nm	12 nm
ISA	RISC-V	RISC-V	RISC-V	x86	RISC-V
Area(mm^2)	60.7^d	11.64^d	15.25	1129.6^a	99
# Cores	60	22	496	128	2048
Freq(MHz)	911	1195	1400	2700	1632
Power(W)	N/A	1.792^c	7.47	500^b	69.25
CM	153959	27116	812350	6065430	**8704557**
CM/mm^2	2536	2330	53269	5370	**87925**
CM/W	N/A	15132	108748	12131	**125698**

[a] CPU die area only [3]. [b] Power based on reported TDP [4]. [c] Nominal voltage active power. [d] eFPGA area removed.

256 RISC-V cores and 1024 L1 banks, organized in a three-level hierarchy (Groups, Tiles, Cores). In the absence of network congestion, cores can reach any banks in a maximum of 5 cycles, simplifying programming by allowing data to be freely distributed across remote banks. While this abstraction benefits the programmer, it sacrifices the ability to exploit physical locality – placing data close to where it is processed on chip. Routing the crossbars and interconnect wires presents a critical challenge that limits scalability and efficient utilization of silicon area. Large empty spaces must be reserved between clusters to accommodate crossbar placement and routing. Reducing this space causes routing congestion, while reserving more space decreases area efficiency. TeraPool [195] increases the core count by 4× by adding another level of hierarchy, but it ends up with lower compute density despite being implemented in a more advanced 12 nm technology node.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

HammerBlade achieves unprecedented compute density through its area-optimized RISC-V scalar cores. The combination of Ruche NoCs and a Cellular manycore pattern enable the system to scale efficiently. These scalar processors are generally easier to program for irregular control flow and data access than vector or SIMT processors. HB supports a wide range of parallel algorithms via a familiar C++/SPMD programming interface. Additionally, an extensive set of custom performance debugging and profiling tools has been developed to analyze where and why the processors spend most of their time during kernel execution, and to measure the utilization of DRAM, cache, processors and network routers in order to identify system bottlenecks.

Ruche Networks challenge several long-standing assumptions in NoC research. First, they offer a new perspective on *scaling NoC bandwidth*. On-chip wires have a minimum pitch, which imposes a finite limit on potential bandwidth. Ruche NoCs shift focus to how much of that bandwidth can actually be realized. Previously with 2-D mesh, the only ways to increase utilization were to widen channels or add parallel networks – both of which increase tile area due to linear growth in router area and additional microarchitectural overhead. Ruche NoCs provide architects with a more flexible alternative: adding long-range Ruche links at a fixed cost and adjusting Ruche Factors to control bandwidth utilization.

Furthermore, Ruche NoCs would alter how architects think about *bisection bandwidth*. Traditionally, bisection bandwidth more or less corresponded to the number of links crossing the network bisection. However, our studies suggest that overall network throughput depends not only on bisection link bandwidth but also on bisection *crossbar* bandwidth. Hence, despite doubling the bisection links, folded torus fails to deliver the expected throughput due to bottlenecks imposed by its 2-D mesh crossbars. In torus NoCs, VC routers ex-

pend additional resources (e.g. VC mux) merely to virtualize physical links, whereas Ruche crossbars actually double router bandwidth.

Finally, Ruche NoCs could inspire new designs with *smaller tiles* (Figure 5.1a). Before Ruche NoCs were introduced, tiles featured larger local memories and relied on greater computational intensity to compensate for bandwidth shortage. Although larger tiles potentially have higher inter-tile bandwidth, it is costly to fully utilize with wider channels and additional networks. Furthermore, scaling the Ruche Factor for larger tiles may require pipelining because of increased wire delays. Therefore, a subtle insight from Ruche NoCs is that smaller tiles may represent the optimal design point for maximal wire utilization and on-chip bandwidth.

HammerBlade exemplifies the benefits of very small tiles. With Half Ruche and Ruche Factor of 3, its tiles achieve the highest routing density among tile-based designs. In addition, more tiles and routers provide more aggregate bandwidth for accessing distributed SRAMs. Perhaps the real bottleneck lies not in the network but in the interface between the SRAM banks and the NoC, especially since much of the area overheads (e.g. sense amplifiers) resides within the SRAMs (Figure 5.1b). Therefore, Ruche NoCs paired with smaller tiles (i.e. more distributed SRAMs) may inspire new designs that exploit maximal wire utilization.

5.2 Future Work

This thesis lays down most of the groundwork. Here are some of the future work ideas that could take HammerBlade to the next level.

Adjusting HB Cell Size – Early in the development of HammerBlade, a 2:1 aspect ratio was chosen for the HB Cell, as it allows more cache banks to be placed along the edges. A 16×8 Cell size was primarily selected for practical reasons, since larger Cells would have significantly increased simulation time for benchmark studies. However, the results from this thesis suggest that both the aspect ratio and the Cell size can be increased with Ruche NoCs. Figure 2.18 has shown that it is more advantageous to have larger Cells (32×8) than more of smaller Cells (2×16×8), because tiles can share a larger pool of cache to facilitate greater data reuse. In fact, Table 3.4 shows that a 16×8 Cell with Ruche Factor of 3 over-

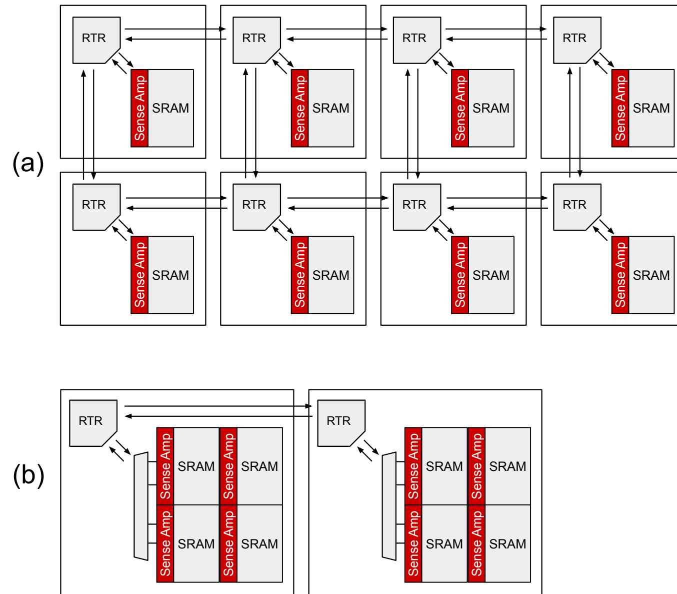


Figure 5.1: Small versus big tiles – illustrating the inefficient utilization of sense amplifiers.

provisions the bisection bandwidth by 2× relative to the cache tile bandwidth. A 32×8 Cell with Ruche Factor of 3 (an aspect ratio of 4:1) could have been a better option, because it matches the bisection and cache tile bandwidth 1:1.

Improving resource utilization – While a single-issue RISC core is area-efficient and has a low cycle time, fundamentally, it struggles to fully utilize functional units (e.g. FPU, ALU) – and more broadly, hardware resources such as scratchpad and NoC bandwidth. Figure 2.3 shows that the 5-stage pipeline naturally splits into integer and floating-point (FP) paths after the instruction issue stage. When the core issues a load/store instruction, the FPU remains idle. Conversely, when the core issues a FP instruction, the integer path remains idle. Figure 2.15 shows that SGEMM kernel, which is supposedly FP-heavy, executes nearly as many integer instructions (e.g. local load/store) as FP instructions (e.g. fused-multiply-add). We have tried to address this underutilization in many different ways.

One approach was to implement a custom *vector load* instruction (e.g. `flw4`) that loads four consecutive words into four standard RISC-V registers. This is achieved by expanding

`flw4` into four separate `flw`, thereby occupying the load/store unit for four cycles with a single instruction fetch. This expansion creates an opportunity to issue FP instructions while `flw4` is expanding.

A major drawback with this approach is the need for additional write ports in the register file, which is more complex than adding read ports. The register file can be *banked* such that each bank has an independent write port. For example, even- and odd-numbered registers can be separated into two banks. With careful compiler scheduling, bank conflicts between concurrent local load and fma instructions can be avoided. However, remote loads with unpredictable return latencies can still cause conflicts.

Another challenge arises with remote vector loads. A vector load must be treated as multiple individual loads by the scoreboard logic responsible for dependency checking, which increases complexity. Another variation is to introduce instructions that access double- or quad-word operands across two or four registers in a single cycle. This approach, however, would require widening the scratchpad SRAM, leading to increased area overhead.

Another proposal was to implement a specialized form of dual-issue logic that can issue one integer and one FP instruction in the same cycle, provided the two instructions are aligned on a two-word boundary. This approach would require modifications to the frontend logic – decode, issue, and PC-increment. It would also necessitate register-file banking and compiler-assisted scheduling, as described above.

It is important to examine how other architectures have tried to address this same challenge. Epiphany-V [127] implements a dual-issue core for this very reason. Snitch [192] implements a configurable loop buffer that can offload the issuance of FP instructions to a side engine; however, the loop buffer has a limited capacity, which limits the maximum kernel size. Similarly, a DMA engine could offload data movement between main memory and the scratchpad, improving energy efficiency by bypassing the register file.

GPU pipelines also contain separate integer and FP execution paths and face similar challenges. Since the NVIDIA V100 [125], GPUs have addressed this issue by provisioning twice as many threads than functional units [125]. For example, each warp consists of 32 threads, but only 16 integer or FP units are available. As a result, each GPU instruction occupies the functional units for two cycles, enabling full utilization of both integer and FP

units while maintaining a single-issue rate.

Expanding the benchmark suite – The parallel benchmark suite used to evaluate HammerBlade covers many of the bases in Berkeley's dwarfs [16], but it is still incomplete. For example, the current suite does not include branch-and-bound, unstructured grid, or finite-state-machine patterns. In addition, the dwarfs themselves lack some fundamental algorithms, such as sorting. Expanding the benchmark suite would help identify more architectural bottlenecks and provide stronger validation for future architectural improvements. Luckily, there is no shortage of benchmarks that can be adapted for HB: Parboil [156], Rodinia [43], PARSEC [33], GAP Benchmark Suite [27], just to name a few.

Comparing performance against GPU simulation is a worthwhile direction to pursue, but it requires more fine tuning. Generally, such comparisons are valuable, because they provide an internal benchmark that motivates continued optimization of both hardware and software. Although we understand the sources of inefficiency in GPUs and demonstrate that HB outperforms the NVIDIA V100 in simulation, we had difficulty quantifying the underlying reasons for this advantage, due to limited effort spent on gaining deeper visibility into GPU simulators and kernel behavior.

Expanding address space – To fully implement the PGAS mapping proposed for inter-Cell communication (Figure 2.7), and more broadly for inter-chip communication, HB needs an address space larger than 32 bits. There has been a proposal to create a 64-bit address by combining two 32-bit registers so that the core datapaths can remain 32-bit to preserve compute density. As with several other future work ideas, this feature would require substantial expertise in compiler engineering.

BIBLIOGRAPHY

[1] RISC-V Bit-manipulation. 2021. `https://github.com/riscv/riscv-bitmanip`.

[2] RISC-V GNU Compiler Toolchain. 2023. `https://github.com/riscv-collab/riscv-gnu-toolchain`.

[3] AMD Granite Ridge and Strix Point Zen 5 Die-sizes and Transistor Counts Confirmed. 2024. TechPowerUp.

[4] AMD EPYC 9005 Series Processors Datasheet. 2025. Advanced Micro Devices, Inc., Santa Clara, CA, USA.

[5] Nilmini Abeyratne, Reetuparna Das, Qingkun Li, Korey Sewell, Bharan Giridhar, Ronald G. Dreslinski, David Blaauw, and Trevor Mudge. Scaling towards kilo-core processors with asymmetric high-radix topologies. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 496–507, 2013.

[6] Dennis Abts, Natalie D. Enright Jerger, John Kim, Dan Gibson, and Mikko H. Lipasti. Achieving predictable performance through better memory controller placement in many-core cmps. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, page 451–461, New York, NY, USA, 2009. Association for Computing Machinery.

[7] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, D. Srikrishna, and M. Taylor. The RAW compiler project. In *Proceedings of the Second SUIF Compiler Workshop*, pages 21–23, 1997.

[8] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, page 248–259, New York, NY, USA, 2000. Association for Computing Machinery.

[9] Nauman Ahmed, Jonathan Lévy, Shanshan Ren, Hamid Mushtaq, Koen Bertels, and Zaid Al-Ars. GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data. *BMC bioinformatics*, 20(1):1–20, 2019.

[10] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, Austin Rovinski, Loai Salem,

Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Xiaoyang Wang, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald G. Dreslinski, Ian Galton, Rajesh K. Gupta, Patrick P. Mercier, Mani Srivastava, Michael Bedford Taylor, and Zhiru Zhang. Celerity: An Open Source RISC-V Tiered Accelerator Fabric. In *HOTCHIPS*, Aug 2017.

[11] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Anuj Rao, Austin Rovinski, Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald Dreslinski, Rajesh Gupta, Michael Taylor, and Zhiru Zhang. Experiences using the RISC-V ecosystem to design an accelerator-centric SoC in TSMC 16nm. In *1st Workshop on Computer Architecture Research with RISC-V (CARRV 2017)*, 2017.

[12] Ashwin M. Aji, Mayank Daga, and Wu Chun Feng. Bounding the Effect of Partition Camping in GPU Kernels. CF '11, New York, NY, USA, 2011. Association for Computing Machinery.

[13] Alric Althoff, Joseph McMahan, Luis Vega, Scott Davidson, Timothy Sherwood, Michael Taylor, and Ryan Kastner. Hiding Intermittant Information Leakage with Architectural Support for Blinking. In *International Symposium on Computer Architecture (ISCA)*, 2018.

[14] Gene M. Amdahl. Computer architecture and amdahl's law. *Computer*, 46(12):38–46, 2013.

[15] Manish Arora, Jack Sampson, Nathan Goulding-Hotta, Jonathan Babb, Ganesh Venkatesh, Michael Bedford Taylor, and Steven Swanson. Reducing the Energy Cost of Irregular Code Bases in Soft Processor Systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2011.

[16] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. 2006.

[17] Saahil Athrij. Vectorizing the Hamerblade Compiler. Master's thesis, University of Washington, 2024.

[18] Z. Azad, G. Yang, R. Agrawal, D. Petrisko, M. Taylor, and A. Joshi. Race: Risc-v soc for en/decryption acceleration on the edge for homomorphic computation. In *ISLPED*, 2022.

[19] Zahra Azad, Guowei Yang, Rashmi Agrawal, Daniel Petrisko, Michael Taylor, and Ajay Joshi. RISE: RISC-V SoC for En/Decryption Acceleration on the Edge for Homomorphic Encryption. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.

[20] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar operand networks: on-chip interconnect for ILP in partitioned architectures. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 341–353, 2003.

[21] Jonathan Babb, Matthew Frank, Victor Lee, Elliot Waingold, Rajeev Barua, Michael Taylor, Jang Kim, Srikrishna Devabhaktuni, and Anant Agarwal. The Raw Benchmark Suite: Computation Structures for General Purpose Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 1997.

[22] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, 2009.

[23] James Balfour and William J. Dally. Design tradeoffs for tiled CMP on-chip networks. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, page 390–401, New York, NY, USA, 2006. Association for Computing Machinery.

[24] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. OpenPiton: An Open Source Manycore Research Framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, page 217–232, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Arnab Banerjee, Pascal T. Wolkotte, Robert D. Mullins, Simon W. Moore, and Gerard J. M. Smit. An energy and performance exploration of network-on-chip architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3):319–329, 2009.

[26] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing Breadth-First Search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2012.

[27] Scott Beamer, Krste Asanović, and David A. Patterson. The GAP Benchmark Suite. *CoRR*, abs/1508.03619, 2015.

[28] Daniel U Becker. *Efficient microarchitecture for network-on-chip routers.* PhD thesis, Stanford University, 2012.

[29] Daniel U. Becker and William J. Dally. Allocator Implementations for Network-on-Chip Routers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. Association for Computing Machinery.

[30] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 88–598, 2008.

[31] B. Beresini, S. Ricketts, and M.B. Taylor. Unifying manycore and fpga processing with the RUSH architecture. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 22 –28, 2011.

[32] Vikram Bhatt, Nathan Goulding-Hotta, Qiaoshi Zheng, Jack Sampson, Steve Swanson, and Michael B. Taylor. Sichrome: Mobile web browsing in Hardware to save Energy . In *Dark Silicon Workshop, ISCA*, 2012.

[33] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.

[34] Brent Bohnenstiehl, Aaron Stillmaker, Jon J. Pimentel, Timothy Andreas, Bin Liu, Anh T. Tran, Emmanuel Adeagbo, and Bevan M. Baas. KiloCore: A 32-nm 1000-Processor Computational Array. *IEEE Journal of Solid-State Circuits*, 52(4):891–902, 2017.

[35] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, page 746–749, New York, NY, USA, 2007. Association for Computing Machinery.

[36] Ajay Brahmakshatriya, Emily Furst, Victor A. Ying, Claire Hsu, Changwan Hong, Max Ruttenberg, Yunming Zhang, Dai Cheol Jung, Dustin Richmond, Michael B. Taylor, Julian Shun, Mark Oskin, Daniel Sanchez, and Saman Amarasinghe. Taming the Zoo: The Unified GraphIt Compiler Framework for Novel Architectures. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 429–442, 2021.

[37] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, 2012.

[38] Martin Burtscher and Keshav Pingali. Chapter 6 - An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm. In Wen mei W. Hwu, editor, *GPU Computing Gems Emerald Edition*, Applications of GPU Computing Series, pages 75–92. Morgan Kaufmann, Boston, 2011.

[39] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *DAC*, 2021.

[40] Sadullah Canakci, Chathura Rajapaksha, Leila Delshadtehrani, Anoop Nataraja, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Processorfuzz: Processor fuzzing with control and status registers guidance. In *HOST*, 2023.

[41] Matheus Cavalcante, Samuel Riedel, Antonio Pullini, and Luca Benini. MemPool: A Shared-L1 Memory Many-Core Cluster with a Low-Latency Interconnect. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 701–706, 2021.

[42] Ting-Jung Chang, Ang Li, Fei Gao, Tuan Ta, Georgios Tziantzioulis, Yanghui Ou, Moyang Wang, Jinzheng Tu, Kaifeng Xu, Paul J. Jackson, August Ning, Grigory Chirkov, Marcelo Orenes-Vera, Shady Agwa, Xiaoyu Yan, Eric Tang, Jonathan Balkind, Christopher Batten, and David Wentzlaff. CIFER: A 12nm, 16mm2, 22-Core SoC with a 1541 LUT6/mm2 1.92 MOPS/LUT, Fully Synthesizable, CacheCoherent, Embedded FPGA. In *2023 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–2, 2023.

[43] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[44] Lin Cheng, Peitian Pan, Zhongyuan Zhao, Krithik Ranjan, Jack Weber, Bandhav Veluri, Seyed Borna Ehsani, Max Ruttenberg, Dai Cheol Jung, Preslav Ivanov, Dustin Richmond, Michael B. Taylor, Zhiru Zhang, and Christopher Batten. A tensor processing framework for cpu-manycore heterogeneous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1620–1635, 2022.

[45] Lin Cheng, Max Ruttenberg, Dai Cheol Jung, Dustin Richmond, Michael Taylor, Mark Oskin, and Christopher Batten. Beyond Static Parallel Loops: Supporting Dynamic Task Parallelism on Manycore Architectures with Software-Managed Scratchpad Memories. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 46–58, New York, NY, USA, 2023. Association for Computing Machinery.

[46] Yuan-Mao Chueh. A Complete Open Source Network Stack For BlackParrot. Master's thesis, University of Washington, 2022.

[47] Dally and Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, 1987.

[48] William J. Dally and Brian Towles. Route packets, not wires: on-chip inteconnection networks. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, page 684–689, New York, NY, USA, 2001. Association for Computing Machinery.

[49] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights. *Proceedings of the IEEE*, 109(10):1706–1752, 2021.

[50] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald Dreslinski, Christopher Batten, and Michael Bedford Taylor. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro*, 38(2):30–41, 2018.

[51] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[52] David R. Ditzel and the Esperanto team. Accelerating ML Recommendation With Over 1,000 RISC-V/Tensor Processors on Esperanto's ET-SoC-1 Chip. *IEEE Micro*, 42(3):31–38, 2022.

[53] Maico Cassel Dos Santos, Tianyu Jia, Joseph Zuckerman, Martin Cochet, Davide Giri, Erik Jens Loscalzo, Karthik Swaminathan, Thierry Tambe, Jeff Jun Zhang, Alper Buyuktosunoglu, Kuan-Lin Chiu, Giuseppe Di Guglielmo, Paolo Mantovani, Luca Piccolboni, Gabriele Tombesi, David Trilla, John-David Wellman, En-Yu Yang, Aporva Amarnath, Ying Jing, Bakshree Mishra, Joshua Park, Vignesh Suresh, Sarita Adve, Pradip Bose, David Brooks, Luca P. Carloni, Kenneth L. Shepard, and Gu-Yeon Wei. 14.5 a 12nm linux-smp-capable risc-v soc with 14 accelerator types, distributed hardware power management and flexible noc-based data orchestration. In *2024 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 67, pages 262–264, 2024.

[54] Anne C. Elster and Tor A. Haugdahl. Nvidia Hopper GPU and Grace CPU Highlights. *Computing in Science & Engineering*, 24(2):95–100, 2022.

[55] Hadi Esmaeilzadeh and Michael Bedford Taylor. Open Source Hardware: Stone Soups and Not Stone Satues, Please. In *SIGARCH Computer Architecture Today*, Dec 2017.

[56] Tim Fischer, Michael Rogenmoser, Thomas Benz, Frank K. Gürkaynak, and Luca Benini. FlooNoC: A 645-Gb/s/link 0.15-pJ/B/hop Open-Source NoC With Wide Physical Links and End-to-End AXI4 Parallel Multistream Support. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 33(4):1094–1107, 2025.

[57] Harry D. Foster. Why the design productivity gap never happened. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 581–584, 2013.

[58] Yaosheng Fu, Tri M. Nguyen, and David Wentzlaff. Coherence Domain Restriction on Large Scale Systems. MICRO-48, page 686–698, New York, NY, USA, 2015. Association for Computing Machinery.

[59] Emily Furst. *Code Generation and Optimization of Graph Programs on a Manycore Architecture*. PhD thesis, University of Washington, 2021.

[60] Fei Gao, Ting-Jung Chang, Ang Li, Marcelo Orenes-Vera, Davide Giri, Paul J. Jackson, August Ning, Georgios Tziantzioulis, Joseph Zuckerman, Jinzheng Tu, Kaifeng Xu, Grigory Chirkov, Gabriele Tombesi, Jonathan Balkind, Margaret Martonosi, Luca Carloni, and David Wentzlaff. DECADES: A 67mm2, 1.46TOPS, 55 Giga Cache-Coherent 64-bit RISC-V Instructions per second, Heterogeneous Manycore SoC with 109 Tiles including Accelerators, Intelligent Storage, and eFPGA in 12nm FinFET. In *2023 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–2, 2023.

[61] Paul Gao, Dai Cheol Jung, Scott Davidson, Daniel Ruelas-Petrisko, Yuan-Mao Chueh, Max Ruttenberg, Kangli Li, Farzam Gilani, Dustin Richmond, Mark Oskin, and Michael Bedford Taylor. HammerBlade in Silicon: A 12-nm 2048-Core RISC-V Manycore SoC With Ruche Networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–5, 2026.

[62] S. Garcia, Donghwan Jeon, C. Louie, and M.B. Taylor. The Kremlin Oracle for Sequential Code Parallelization. *Micro, IEEE*, 32(4):42–53, July-Aug. 2012.

[63] Saturnino Garcia, Donghwan Jeon, Chris Louie, Sravanthi Kota Venkata, and Michael Bedford Taylor. Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning. In *USENIX Workshop on Hot Topics in Parallelism (HOT-PAR)*, 2010.

[64] Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor. Kremlin: Rethinking and Rebooting gprof for the Multicore Age. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[65] Soroush Ghodrati, Sean Kinzer, Hanyang Xu, Rohan Mahapatra, Yoonsung Kim, Byung Hoon Ahn, Dong Kai Wang, Lavanya Karthikeyan, Amir Yazdanbakhsh,

Jongse Park, Nam Sung Kim, and Hadi Esmaeilzadeh. Tandem Processor: Grappling with Emerging Operators in Neural Networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 1165–1182, New York, NY, USA, 2024. Association for Computing Machinery.

[66] Farzam Gilani. *Methodologies for Accelerated Open-Source Hardware Verification and Optimization*. PhD thesis, University of Washington, 2025.

[67] Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael Taylor, and Steven Swanson. GreenDroid: A Mobile Application Processor for a Future of Dark Silicon. In *HOTCHIPS*, 2010.

[68] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *Micro, IEEE*, pages 86–95, March 2011.

[69] Nathan Goulding-Hotta. *Specialization as a Candle in the Dark Silicon Regime*. PhD thesis, University of California, San Diego, 2020.

[70] Nathan Goulding-Hotta, Jack Sampson, Qiaoshi Zheng, Vikram Bhatt, Steven Swanson, and Michael Taylor. GreenDroid: An Architecture for the Dark Silicon Age. In *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012.

[71] Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu. Express Cube Topologies for on-Chip Interconnects. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 163–174, 2009.

[72] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. DR-SNUCA: An energy-scalable dynamically partitioned cache. In *International Conference on Computer Design (ICCD)*, 2013.

[73] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. Time Cube: A Manycore Embedded Processor with Interference-Agnostic Progress Tracking. In *International Conference On Embedded Computer Systems: Architectures, Modeling And Simulation (SAMOS)*, 2013.

[74] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. Qualitytime: A simple online technique for quantifying multicore execution efficiency. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[75] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[76] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 37–47, New York, NY, USA, 2010. Association for Computing Machinery.

[77] John Hauser. Berkeley HardFloat. Available at `https://www.jhauser.us/arithmetic/HardFloat.html`.

[78] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. BlackBox: Lightweight Security Monitoring for COTS Binaries. In *Code Generation and Optimization*, 2016.

[79] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. A Runtime Approach to Security and Privacy. In *European Security and Privacy*, 2016.

[80] R. Ho, K.W. Mai, and M.A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[81] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, page 311–320, New York, NY, USA, 2012. Association for Computing Machinery.

[82] Sara Hooker. The Hardware Lottery. *Commun. ACM*, 64(12):58–65, nov 2021.

[83] Hu, Zhu, Taylor, and Cheng. FPGA Global Routing Architecture Optimization Using a Multicommodity Flow Approach . In *ICCD*, 2007.

[84] Drago Ignjatović, Daniel W. Bailey, and Ljubisa Bajić. The Wormhole AI Training Processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 356–358, 2022.

[85] JEDEC. HIGH BANDWIDTH MEMORY (HBM) DRAM - JESD235D. Jan 2020. Available at `https://www.jedec.org/standards-documents/docs/jesd235a`.

[86] Donghwan Jeon, Saturnino Garcia, Chris Louie, Sravanthi Kota Venkata, and Michael Bedford Taylor. Kremlin: Like gprof, but for Parallelization. In *Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[87] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: Parallel Speedup Estimates for Serial Programs. In *Conference on Object-Oriented Programming, Systems, Language and Applications (OOPSLA)*, 2011.

[88] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Parkour: Parallel Speedup Estimates from Serial Code. In *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*, 2011.

[89] Donghwan Jeon, Saturnino Garcia, and Michael Bedford Taylor. Skadu: Efficient Vector Shadow Memories for Poly-scopic Program Analysis. In *Conference on Code Generation and Optimization (CGO)*, 2013.

[90] Norman P. Jouppi. Cache Write Policies and Performance. *SIGARCH Comput. Archit. News*, 21(2):191–201, may 1993.

[91] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2021.

[92] Dai Cheol Jung. Caches for Complex Open Source System-on-Chip Designs. Master's thesis, University of Washington, 2019.

[93] Dai Cheol Jung, Scott Davidson, Chun Zhao, Dustin Richmond, and Michael Bedford Taylor. Ruche Networks: Wire-Maximal, No-Fuss NoCs. In *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, 2020.

[94] Dai Cheol Jung, Max Ruttenberg, Paul Gao, Scott Davidson, Daniel Petrisko, Kangli Li, Aditya K Kamath, Lin Cheng, Shaolin Xie, Peitian Pan, Zhongyuan Zhao, Zichao Yue, Bandhav Veluri, Sripathi Muralitharan, Adrian Sampson, Andrew Lumsdaine, Zhiru Zhang, Christopher Batten, Mark Oskin, Dustin Richmond, and Michael Bedford Taylor. Scalable, Programmable and Dense: The HammerBlade Open-Source RISC-V Manycore. In *International Symposium on Computer Architecture (ISCA)*, 2024.

[95] Dai Cheol Jung and Michael Taylor. Evaluating Ruche Networks: Physically Scalable, Cost-Effective, Bandwidth-Flexible NoCs. In *International Symposium on Computer Architecture (ISCA)*, 2025.

[96] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. Cutlass: Fast linear algebra in CUDA C++. *NVIDIA Developer Blog*, 2017.

[97] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020.

[98] Moein Khazraee. *Reducing the development cost of customized cloud infrastructure.* PhD thesis, University of California, San Diego, 2020.

[99] Moein Khazraee, Luis Vega, Ikuo Magaki, and Michael Taylor. Specializing a Planet's Computation: ASIC Clouds. *IEEE Micro*, May 2017.

[100] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Taylor. Moonwalk: NRE Optimization in ASIC Clouds or, accelerators will use old silicon. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[101] Jason Kim, Michael B. Taylor, Jason Miller, and David Wentzlaff. Energy Characterization of a Tiled Architecture Processor with On-Chip Networks. In *International Symposium on Low Power Electronics and Design (ISLPED)*, August 2003.

[102] John Kim, James Balfour, and William Dally. Flattened Butterfly Topology for On-Chip Networks. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 172–182, 2007.

[103] Sravanthi Kota Venkata, IkkJin Ahn, Donghwan Jeon, Anshuman Gupta, and Michael Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[104] Tushar Krishna, Amit Kumar, Patrick Chiang, Mattan Erez, and Li-Shiuan Peh. NoC with Near-Ideal Express Virtual Channels Using Global-Line Communication. In *2008 16th IEEE Symposium on High Performance Interconnects*, pages 11–20, 2008.

[105] Tushar Krishna, Amit Kumar, Li-Shiuan Peh, Jacob Postman, Patrick Chiang, and Mattan Erez. Express Virtual Channels with Capacitively Driven Global Links. *IEEE Micro*, 29(4):48–61, 2009.

[106] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A Suite of Parallel Irregular Programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[107] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K. Jha. Express virtual channels: towards the ideal interconnection fabric. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 150–161, New York, NY, USA, 2007. Association for Computing Machinery.

[108] Yann LeCun. Deep Learning Hardware: Past, Present, and Future. In *2019 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 12–19, 2019.

[109] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. Warped-compression: enabling power efficient GPUs through register compression. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 502–514, New York, NY, USA, 2015. Association for Computing Machinery.

[110] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, page 46–57, New York, NY, USA, 1998. Association for Computing Machinery.

[111] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, page 129–140, New York, NY, USA, 2011. Association for Computing Machinery.

[112] Kangli Li. An Open Source Non-Blocking Manycore L2 Cache. Master's thesis, University of Washington, 2024.

[113] Qinjian Li, Chengwen Zhong, Kaiyong Zhao, Xinxin Mei, and Xiaowen Chu. Implementation and Analysis of AES Encryption on GPU. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 843–848, 2012.

[114] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.

[115] Weifeng Liu and Brian Vinter. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 370–381, 2014.

[116] Pejman Lotfi-Kamran, Boris Grot, and Babak Falsafi. NOC-Out: Microarchitecting a Scale-Out Processor. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 177–187, 2012.

[117] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[118] Sheng Ma, Zhiying Wang, Zonglin Liu, and Natalie Enright Jerger. Leaving One Slot Empty: Flit Bubble Flow Control for Torus Cache-Coherent NoCs. *IEEE Transactions on Computers*, 64(3):763–777, 2015.

[119] Ikuo Magaki, Moein Khazraee, Luis Vega, and Michael Taylor. ASIC Clouds: Specializing the Datacenter. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[120] Matt Martineau, Patrick Atkinson, and Simon McIntosh-Smith. Benchmarking the NVIDIA V100 GPU and Tensor Cores. In *Euro-Par 2018: Parallel Processing Workshops*, pages 444–455, Cham, 2019. Springer International Publishing.

[121] Michael McKeown, Alexey Lavrov, Mohammad Shahrad, Paul J Jackson, Yaosheng Fu, Jonathan Balkind, Tri Minh Nguyen, Katie Lim, Yanqi Zhou, and David Wentzlaff. Power and Energy Characterization of an Open Source 25-Core Manycore Processor. In *HPCA*, pages 762–775, 2018.

[122] Robert Mullins, Andrew West, and Simon Moore. The design and implementation of a low-latency on-chip network. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, page 164–169. IEEE Press, 2006.

[123] Sripathi Muralitharan. TinyParrot: An Integration-Optimized Linux-Capable Host Multicore. Master's thesis, University of Washington, 2021.

[124] Tony Nowatzki, Vinay Gangadhan, Karthikeyan Sankaralingam, and Greg Wright. Pushing the limits of accelerator efficiency while retaining programmability. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 27–39, 2016.

[125] NVIDIA. NVIDIA Tesla V100 GPU Architecture, 2017.

[126] NVIDIA. CUDA C++ Progamming Guide, 2023. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[127] Andreas Olofsson. Epiphany-V: A 1024 processor 64-bit RISC System-on-Chip. *arXiv preprint arXiv:1610.01832*, 2016.

[128] Andreas Olofsson, Tomas Nordström, and Zain Ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with Epiphany. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 1719–1726, 2014.

[129] Molly A. O'Neil and Martin Burtscher. Microarchitectural performance characterization of irregular GPU kernels. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 130–139, 2014.

[130] Marcelo Orenes-Vera, Esin Tureci, David Wentzlaff, and Margaret Martonosi. Dalorex: A Data-Local Program Execution and Architecture for Memory-bound Applications. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 718–730, 2023.

[131] Yanghui Ou, Shady Agwa, and Christopher Batten. Implementing Low-Diameter On-Chip Networks for Manycore Processors Using a Tiled Physical Design Methodology. In *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, 2020.

[132] S. Pal, D. Park, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, J. Beaumont, K. Chen, C. Chakrabarti, M. Taylor, T. Mudge, D. Blaauw, H. Kim, and R. Dreslinski. A 7.3 M Output Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm. In *Symposium on VLSI Circuits*, pages C150–C151, 2019.

[133] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. ISCA '97, page 206–218, New York, NY, USA, 1997. Association for Computing Machinery.

[134] D. Park, S. Pal, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, J. Beaumont, K. Chen, C. Chakrabarti, M. B. Taylor, T. Mudge, D. Blaauw, H. Kim, and R. G. Dreslinski. A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix–Matrix Multiplication Accelerator. *IEEE Journal of Solid-State Circuits*, pages 933–944, April 2020.

[135] L.-S. Peh and W.J. Dally. A delay model and speculative architecture for pipelined routers. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 255–266, 2001.

[136] Huwan Peng. *Methodologies and Architectures for AI Inference Hardware: From Foundational Networks to Large Language Models*. PhD thesis, University of Washington, 2025.

[137] Huwan Peng, Scott Davidson, Richard Shi, Shuaiwen Leon Song, and Michael Taylor. Chiplet Cloud: Building AI Supercomputers for Serving Large Generative Language Models. *arXiv:2307.02666 [cs]*, 2024.

[138] Huwan Peng, Scott Davidson, Richard Shi, and Michael Taylor. ReaLLM: A Trace-Driven Framework for Rapid Simulation of Large-Scale LLM Inference. In *ASAP*, 2025.

[139] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro*, 40(4):93–102, 2020.

[140] Daniel Petrisko, Chun Zhao, Scott Davidson, Paul Gao, Dustin Richmond, and Michael Bedford Taylor. NoC Symbiosis. In *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, 2020.

[141] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems)*. Addison-Wesley Professional, 2005.

[142] Robert "Max" Ramstad. Enabling Vector Load and Store instructions on HammerBlade Architecture. Master's thesis, University of Washington, 2024.

[143] Shashank Vijaya Ranga. ParrotPiton and ZynqParrot: FPGA Enablements for the BlackParrot RISC-V Processor. Master's thesis, University of Washington, 2021.

[144] B Ramakrishna Rau. Pseudo-randomly interleaved memory. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 74–83, 1991.

[145] 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 16-nm 695 Giga-RISC-V Instructions/s Manycore Processor With Synthesizable PLL. *IEEE Solid-State Circuits Letters*, 2(12):289–292, 2019.

[146] Austin Rovinski, Chun Zhao, Khalid Al-Hawaj, Paul Gao, Shaolin Xie, Christopher Torng, Scott Davidson, Aporva Amarnath, Luis Vega, Bandhav Veluri, Anuj Rao, Tutu Ajayi, Julian Puscar, Steve Dai, Ritchie Zhao, Dustin Richmond, Zhiru Zhang, Ian Galton, Christopher Batten, Michael B Taylor, and Ronald 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. In *2019 Symposium on VLSI Circuits*, pages C30–C31, 2019.

[147] Daniel Ruelas-Petrisko. *A Qualitative Approach to Agile Hardware Design*. PhD thesis, University of Washington, 2025.

[148] Daniel Ruelas-Petrisko, Farzam Gilani, Anoop Mysore Nataraja, Zoe Taylor, and Michael Taylor. ZynqParrot: A Scale-Down Approach to Cycle-Accurate, FPGA-Accelerated Co-Emulation. *arXiv:2509.20543 [cs]*, 2025.

[149] Jack Sampson, Manish Arora, Nathan Goulding-Hotta, Ganesh Venkatesh, Jonathan Babb, Vikram Bhatt, Michael Bedford Taylor, and Steven Swanson. An Evaluation of Selective Depipelining for FPGA-based Energy-Reducing Irregular Code Coprocessors. In *Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[150] Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor. Efficient Complex Operators for Irregular Codes. In *High Performance Computing Architecture (HPCA)*, 2011.

[151] David Schor. A Look At The ET-SoC-1, Esperanto's Massively Multi-Core RISC-V Approach To AI, 2021. Available at `https://fuse.wikichip.org/news/4911/a-look-at-the-et-soc-1-esperantos-massively-multi-core-risc-v-approach-to-ai/`.

[152] Daeho Seo, Akif Ali, Won-Taek Lim, and N. Rafique. Near-optimal worst-case throughput routing for two-dimensional mesh networks. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 432–443, 2005.

[153] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph Processing on GPUs: A Survey. *ACM Comput. Surv.*, 50(6), jan 2018.

[154] J.P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load Balancing and Data Locality in Adaptive Hierarchical N-Body Methods: Barnes-Hut, Fast Multipole, and Radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, 1995.

[155] Eric Sprangle and Doug Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, page 25–34, USA, 2002. IEEE Computer Society.

[156] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127:27, 2012.

[157] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA, 2017. Association for Computing Machinery.

[158] Steven Swanson and Michael Taylor. GreenDroid: Exploring the next evolution for smartphone application processors. In *IEEE Communications Magazine*, March 2011.

[159] Emil Talpes, Debjit Das Sarma, Doug Williams, Sahil Arora, Thomas Kunjan, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Chandrasekhar Poorna, Vaidehi Samant, John Sicilia, Anantha Kumar Nivarti, Raghuvir Ramachandran, Tim Fischer, Ben Herzberg, Bill McGee, Ganesh Venkataramanan, and Pete Banon. The Microarchitecture of DOJO, Tesla's Exa-Scale Computer. *IEEE Micro*, 43(3):31–39, 2023.

[160] MB Taylor, J Kim, J Miller, F Ghodrat, B Greenwald, P Johnson, W Lee, A Ma, N Shnidman, V Strumpen, et al. The raw processor-a scalable 32-bit fabric for embedded and general purpose computing. In *Proceedings of Hot Chips XIII*, 2001.

[161] M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[162] Michael Taylor. *Tiled Microprocessors*. PhD thesis, Massachusetts Institute of Technology, 2007.

[163] Michael Taylor. A Landscape of the New Dark Silicon Design Regime. *Micro, IEEE*, Sept-Oct. 2013.

[164] Michael Taylor. A Landscape of the New Dark Silicon Design Regime. In *Design Automation and Test in Europe*, April 2014.

[165] Michael Taylor. The Evolution of Bitcoin Hardware. *Computer, IEEE*, Sept-Oct. 2017.

[166] Michael B. Taylor. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Design Automation Conference (DAC)*, 2012.

[167] Michael B. Taylor. Bitcoin and the Age of Bespoke Silicon. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2013.

[168] Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Walter Lee, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Saman Amarasinghe, and Anant Agarwal. A 16-issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network. In *IEEE International Solid-State Circuits Conference (ISSCC)*, February 2003.

[169] Michael B. Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks. In *IEEE Transactions on Parallel and Distributed Systems*, February 2005.

[170] Michael Bedford Taylor. Geocomputers and the Commercial Borg. In *SIGARCH Computer Architecture Today*, Dec 2017.

[171] Michael Bedford Taylor. BaseJump STL: SystemVerilog needs a Standard Template Library for Hardware Design. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[172] Michael Bedford Taylor. Your agile open source HW stinks (because it is not a system). In *ICCAD*, 2020.

[173] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, page 2, USA, 2004.

[174] Michael Bedford Taylor, Luis Vega, Moein Khazraee, Ikuo Magaki, Scott Davidson, and Dustin Richmond. ASIC clouds: Specializing the datacenter for planet-scale applications. *CACM*, pages 103–109, 2020.

[175] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. CortexSuite: A Synthetic Brain Benchmark Suite. In *International Symposium on Workload Characterization (IISWC)*, Oct. 2014.

[176] Dmitrii Tolmachev. VkFFT-A Performant, Cross-Platform and Open-Source GPU FFT Library. *IEEE Access*, 11:12039–12058, 2023.

[177] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[178] Jasmina Vasiljevic and Davor Capalija. Blackhole & TT-Metalium: The Standalone AI Computer and its Programming Model . In *2024 IEEE Hot Chips 36 Symposium (HCS)*, pages 1–30, Los Alamitos, CA, USA, August 2024. IEEE Computer Society.

[179] Luis Vega and Michael Bedford Taylor. RV-IOV: Tethering RISC-V Processors via Scalable I/O Virtualization . In *CARRV*, 2017.

[180] Bandhav Veluri, Collin Pernu, Ali Saffari, Joshua Smith, Michael Taylor, and Shyam Gollakota. Neuricam: Low-power video acquisition using dual-mode iot cameras. In *MobiCom*, 2023.

[181] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[182] Ganesh Venkatesh, John Sampson, Nathan Goulding, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. QsCores: Configurable Co-processors to Trade Dark Silicon for Energy Efficiency in a Scalable Manner. In *International Symposium on Microarchitecture (MICRO)*, 2011.

[183] Marian Verhelst, Luca Benini, and Naveen Verma. How to keep pushing ml accelerator performance? know your rooflines! *IEEE Journal of Solid-State Circuits*, pages 1–18, 2025.

[184] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to Software: Raw Machines. In *IEEE Computer*, September 1997.

[185] David W Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188, 1991.

[186] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. Technical Report UCB/EECS-2014-54, University of California, Berkeley, May 2014. Available at `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html`.

[187] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, 2007.

[188] Chenhao Xie, Xie Li, Yang Hu, Huwan Peng, Michael Taylor, and Shuaiwen Leon Song. Q-VR: System-level design for future mobile collaborative virtual reality. In *ASPLOS*, 2021.

[189] Shaolin Xie, Scott Davidson, Ikuo Magaki, Moein Khazraee, Luis Vega, Lu Zhang, and Michael B. Taylor. Extreme datacenter specialization for planet-scale computing: Asic clouds. In *ACM Sigops Operating System Review*, 2018.

[190] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. YaSpMV: Yet Another SpMV Framework on GPUs. *SIGPLAN Not.*, 49(8):107–118, feb 2014.

[191] Jenq-Shyan Yang and Chung-Ta King. Designing tree-based barrier synchronization on 2D mesh networks. *IEEE Transactions on Parallel and Distributed Systems*, 9(6):526–534, 1998.

[192] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads. *IEEE Transactions on Computers*, 70(11):1845–1860, 2021.

[193] Karen Zee, Viktor Kuncak, Michael Taylor, and Martin C. Rinard. Runtime checking for program verification. In *RV*, 2007.

[194] Xingyao Zhang, Haojun Xia, Donglin Zhuang, Hao Sun, Xin Fu, Michael Taylor, and Shuaiwen Leon Song. η-LSTM: Co-designing highly-efficient large lstm training via exploiting memory-saving and architectural design opportunities. In *ISCA*, 2021.

[195] Yichao Zhang, Marco Bertuletti, Chi Zhang, Samuel Riedel, Diyou Shen, Bowen Wang, Alessandro Vanelli-Coralli, and Luca Benini. TeraPool: A Physical Design Aware, 1024 RISC-V Cores Shared-L1-Memory Scaled-Up Cluster Design With High Bandwidth Main Memory Link. *IEEE Transactions on Computers*, 74(11):3667–3681, 2025.

[196] Ritchie Zhao, Chun Zhao, Shaolin Xie, Bandhav Veluri, Luis Vega, Christopher Torng, Ningxiao Sun, Austin Rovinski, Anuj Rao, Gai Liu, Paul Gao, Scott Davidson, Steve Dai, Aporva Amarnath, KhalidAl-Hawaj, Tutu Ajayi Christopher Batten, Ronald G. Dreslinski, Rajesh K.Gupta, Michael B.Taylor, and Zhiru Zhang. Celerity: An Open Source RISC-V Tiered Accelerator Fabric. In *7th RISC-V Workshop*, 2017.

[197] Qiaoshi Zheng, Nathan Goulding-Hotta, Scott Ricketts, Steven Swanson, Michael Bedford Taylor, and Jack Sampson. Exploring energy scalability in coprocessor-dominated architectures for dark silicon. *Transactions on Embedded Computing Systems (TECS)*, Mar 2014.

[198] Yi Zhu, Yuanfang Hu, Michael Taylor, and Chung-Kuan Cheng. Energy and switch area optimizations for FPGA global routing architectures. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, January 2009.

[199] Yi Zhu, Michael Taylor, Scott B. Baden, and Chung-Kuan Cheng. Advancing supercomputer performance through interconnection topology synthesis. In *International Conference on Computer-Aided Design (ICCAD)*, pages 555–558, 2008.