Towards Free, Open, and Ubiquitous Hardware Design

by

Austin John Rovinski

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2022

Doctoral Committee:

Associate Professor Ronald G. Dreslinski, Chair
Professor Scott Mahlke
Professor Trevor Mudge
Professor Dennis Sylvester

Austin John Rovinski
rovinski@umich.edu
ORCID iD: 0000-0003-1761-1898

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Ron Dreslinski, for guiding me. I have known Ron since I began collaborating with him as a fledging undergrad student eager to try academic research. Ron took me under his wing and immersed me in the field of computer chip design, the field which has ensnared my passion and become a key part of my identity. From many lab festivities to many late-night tapeout crunches, Ron brought our lab together through thick and thin. Although it's not in his nature to push people, I would like to thank him for pressuring me to keep working on this dissertation during the doldrums of the COVID-19 pandemic and allowing me to move on to the next phase of my life. I would especially like to thank Ron for allowing me dedicated time for self-discovery – I approached him in the middle of my PhD and asked if I could take a semester for myself. I was facing severe burnout and needed time for personal reflection. He agreed without hesitation and ensured that I had funding. I credit that semester for allowing me to rebalance my life and obtain a new perspective, and I credit Ron for being the only person I know who would take this risk. Lastly, I thank Ron for being an overall genuine and extremely kind person. Few look after the well-being of their students as well as Ron does.

I would also like to thank the numerous student colleagues and collaborators whom I've worked with during undergrad and grad school. Johann Hauswald, Chang-Hong Hsu, Yiping Kang, Arjun Khurana, Mike Laurenzano, Kris Smith, Yunqi Zhang, and Steve Zekany in particular guided my early research and cemented my decision to pursue a PhD. Maxime Lawton, Saurabh Deo, Tim Kenny, and Dinker Ambe are all excellent friends and project teammates. I thank all my lab mates who have helped me, collaborated with me,

entertained me, and commiserated with me. Tutu Ajayi, Aporva Amarnath, Subhankar Pal, and I were Ron's first students, and we share a close relationship. Tutu Ajayi and Jielun Tan have always been friends I can reach out to, and I sincerely appreciate it. Many students outside of Michigan have also been great friends and collaborators, but I'd like to thank Khalid Al-Hawaj, Scott Davidson, and Chris Torng in particular.

My committee members and other faculty have been invaluable resources during my PhD. Mike Taylor and Chris Batten were my first collaborators outside of Michigan, and they've been extremely helpful in offering new perspectives and advice. Jason Mars and Lingjia Tang introduced me to research and encouraged me to pursue a PhD. Peter Chen inspired my interest in computer engineering during my freshman year of undergrad and has been a model for lecturing and mentoring students. Andrew Kahng, Tom Spyrou, and Matt Liberty were invaluable resources for learning about electronic design automation tools and developing the OpenROAD Project.

I would like to thank all my friends who have made grad school more bearable, usually through the lens of Dungeons & Dragons and video games: Dan Braunstein, Evan Brisita, Branden Carlson, Tony D'Agostini, Alex Dishaw, Sara Eskandari, Tahmid Hasan, Rob Swor, and Emily Wisniewski. Evan Daykin and Charlie Moshier also provided me with only the freshest and dankest memes. Rob Swor has been a best friend, an ear to listen, a source of advice, and a creator of bad puns since our days in middle school.

Lastly, I'd like to thank my parents, who have supported me throughout my entire academic career. By encouraging scholarship from a young age, they prepared me for the demands of Michigan's curriculum and academia. They have shown me unwavering love and support during my journey, and I am glad they are able to witness its culmination.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In 1965, Gordon Moore posited that the number of transistors on an integrated circuit would double every 18 months. With some adjustments, the prediction largely remained true for decades and revolutionized technology as we know it. A lesser-known contributor to Moore's Law was the "Mead and Conway Revolution" in VLSI, initiated by Carver Mead and Lynn Conway. Prior to this revolution, integrated circuit design was mostly done manually and required deep expertise from top-level integration all the way down to fabrication effects. Mead and Conway transformed this manufacturing knowledge into a set of design rules which were fit for use in automation. This innovation enabled integrated circuit design to scale with computers rather than humans, and effectively kick-started the electronic design automation (EDA) industry.

Moore's Law is reaching a slow but inevitable end as transistor counts take longer and longer to double. Because chips can no longer rely on foundry improvements to improve performance, architectural innovations need to pick up the slack. Certain domains such as machine learning, genomics, graph processing, drug discovery, financial trading, and others have turned to hardware acceleration. EDA software has not seen such focus and is at risk of stagnating chip development. In this dissertation, I discuss key issues limiting the pace of innovation in hardware design, including complexity of design integration, inaccessibility of EDA tools, and lagging EDA tool performance. Then, I present three of my works which address these issues: Celerity, OpenROAD, and SpeEDAr. These works represent a multi-faceted approach to speed up design innovation by improving design methodologies, providing open-source EDA tooling, and improving end-to-end EDA tool performance.

Celerity addresses the issue of long hardware design schedules and integration of complex systems-on-chip (SoCs). Methodologies are presented which sped up chip development to 9 months: approximately half of a normal design schedule. At the time of publication, the resulting chip design produced a single-chip record 695 Giga RISC-V instructions/s, a record CoreMark benchmark score of 825,320 and a record CoreMark score/MHz of 580.25. Celerity outperformed prior manycore works in energy efficiency by 4.2× and normalized area efficiency by 1.8×.

OpenROAD tackles the long-standing issue of closed-source EDA software. OpenROAD is the first and only open-source EDA software capable of producing design-rule-clean chips in advanced sub-20nm nodes. OpenROAD also includes the OpenROAD Design Suite, a diverse EDA benchmarking suite composed of real-world designs. To date, OpenROAD has enabled over 100 designs to be taped out, including by many designers with no prior chip design experience.

Lastly, I present a characterization of OpenROAD. This is the first known full EDA flow characterization to date. This characterization reveals that the implementation flow runtime is dominated by detailed routing (40%) and synthesis (30%). I then present SpeEDAr, an accelerator for detailed routing which accounts for the detailed router's unique costing while prior work does not. SpeEDAr achieves a mean 67× speedup on the detailed router's graph search, which translates to a mean 1.2× end-to-end flow speedup.

CHAPTER 1

Introduction

1.1 The Techno-Social Scripted-Process

In 1965, Gordon Moore posited that the number of transistors on an integrated circuit (IC) would double every 18 months. Transistors are the fundamental building blocks of ICs, and transistor counts serve as a rough measure of the IC's complexity and capability. With some adjustments, Moore's prediction largely remained true for decades and became immortalized as **Moore's Law**. Moore's Law is now synonymous with the exponential rise in computing power, as well as computing's pervasiveness in society.

A lessor known contributor to Moore's Law was the "Mead and Conway Revolution", initiated by Carver Mead[1] and Lynn Conway. Prior to this revolution, integrated circuit design was done mostly manually and required full-stack domain expertise – circuit designers needed to be knowledgeable of top-level integration all the way down to the minutiae of fabrication effects on design geometry. Mead and Conway observed that the complexities of the manufacturing process could be abstracted out into simpler sets of **design rules**. Not only did these rules make IC design more accessible, but the rules could be expressed *programmatically* and enable many manual aspects to be automated. While designs were previously limited by the ability and expertise of a human, new designs could use computers to automate and scale up tasks. This innovation launched the electronic design industry into a process which Conway calls the "techno-social scripted-process" (Algorithm 1).

Algorithm 1: Conway's Techno-Social Scripted-Process

Step i

> Use design tools on current computers to design chip-sets for more powerful computers;
> Print the more powerful chip-sets using foundries' next-denser fabrication processes;
> Use some of those chip-sets to update current computer-design computers and design tools;

Repeat as Step $(i + 1)$

[1] Coincidentally, Carver Mead is credited with coining the term "Moore's Law"

Coupled with Moore's Law, Conway's Process enabled steady, generational improvement in the development of integrated circuits, with complex software and processors being used to develop even more complex software and processors.

In Moore's own words, however, "no exponential is forever", and experts generally agree that Moore's Law is slowly coming to an end as transistor count growth slows. A slowing of Moore's Law entails a slowing of Conway's Process and vice versa. This precarious situation forms a negative feedback loop that risks stagnating new hardware development, and much technological innovation along with it. This problem is not unique to Conway's Process, however. Because of the slowing of Moore's Law, many other computational domains are experiencing diminished performance, and they have begun developing their own solutions.

1.2 Adapting to a Post-Moore Era

In this "Post-Moore" era, several phenomena prevent the further scaling of fabrication processes, such as the end of Dennard Scaling [21], Dark Silicon [22], transistor reliability [15], and thermal limitations. To remedy this, others have proposed architectural techniques for continued performance scaling [51]. In the near term, it appears that *specialization* is winning out as the predominant solution, as evidenced by the rapid proliferation of on-chip or on-package accelerators in the past decade [27].

Conceptually, one can imagine a computer chip as a piece of silicon which is divided into a *fixed* number of transistors: a "transistor budget". Computer architects are responsible for using the budget to provide power and performance improvements to the computer, while Moore's Law enables raising the budget on a regular basis. The end of Moore's Law means that the regular budget increases are becoming less and less regular, yet demands for improvements remain high. Specialization may be the most enticing path forward because specialized circuits improve power and performance *while also using fewer transistors* than their general counterparts. Most importantly, these improvements are independent of the technology the chip is developed on, meaning architects can provide improvements without relying on Moore's Law.

One might ask that if specialization can offer technology-independent benefits, why are such architectures only being pursued now? Specialization has in fact always been a technique to improve performance of certain operations, such as math co-processors and graphics processing units (GPUs) of the '80s and '90s. The main difference is that in the post-Moore era, the power, performance, and area of silicon (**PPA**) gains from technological improvements are dwindling, so a much greater focus has been placed on architectures

to pick up the slack. Enhancements to an existing architecture are much less complicated in terms of design and verification than developing an entirely new architecture with a specialized accelerator. In the prime of Moore's Law, this often meant that time spent developing an accelerator was less efficient than simply creating fast, incremental improvements to existing architectures to reduce the processor's time to market. Today, the balance has shifted, and accelerator development is significantly more favorable than in the past.

As such, specialized hardware has been rapidly developed for certain domains, including machine learning, genomics, graph processing, drug discovery, financial trading, and others. All of these computationally intensive workloads have shown substantial benefit from specialization. But what about electronic design automation (EDA) tools, the software referenced in Conway's Process? EDA software faces several barriers which cause a slowdown in Conway's Process, and in the development of new integrated circuits.

1.3 The Slowing of Conway's Process

Despite hardware acceleration proliferating among other workloads, EDA software has not seen the same degree of advancement. While many EDA tools have gradually shifted to multithreaded or distributed processing, no commercial state-of-the-art tool employs hardware more specialized than CPUs to date. Even for academic endeavors, very few works explore the use of hardware acceleration.[2] To understand the slowdown in Conway's Process, we must understand the barriers facing modern EDA software.

1.3.1 Design Time

Figure 1.1 illustrates the general methodology for designing a digital integrated circuit. The design cycle is highly iterative, where each iteration consists of:

1. Perform any alterations to the design specification.
2. Implement the new design specification.
3. Verify the implementation satisfies the specification and manufacturing constraints.

This process is iterated until the design satisfies the specification and the finalized design files are sent to the manufacturer for fabrication. This handoff of design files marks the end of the design process and is frequently referred to as **tapeout**. In terms of the real time / engineering hours consumed by this process, the time for an iteration is referred to

[2] One notable exception is DREAMPlace [39], which offers GPU-accelerated global placement.

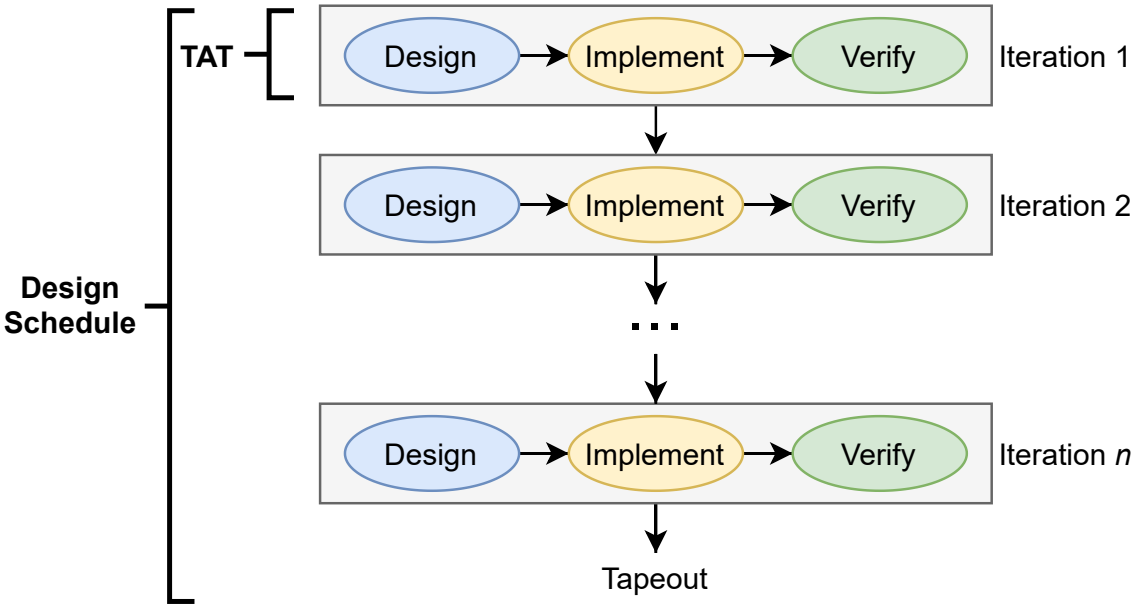

Figure 1.1: Illustration of an example design schedule

the **turnaround time** (TAT), and the total time from design inception to tapeout is referred to as the **design schedule**.

One significant constraint in IC design is the turnaround time, due to the amount of time it takes for EDA tools to implement a design from source code. Because changes in TAT can greatly affect the design schedule, an industry rule of thumb is often to target no greater than a 24-hour TAT.

A key challenge in creating a more complex design is managing the impact on design schedule, because it represents the time to market for a new IC product. In Conway's Process, the new generation of chips generally enables faster computation and mitigates some of the increasing design complexity. In the Post-Moore era, however, complexity can easily outpace computational performance improvements. Without improvements to EDA tools and design strategies, TAT increases lead to slower design schedules and stagnation in technological advancement.

1.3.2 Closed Source

Perhaps the largest challenge facing EDA tools is their closed-source nature. While many other domains of software have free and open-source substrates to build on, EDA tools are notoriously closed-source and proprietary. EDA tools are dominated by commercial providers who are incentivized to keep their competitive advantages shrouded in secrecy. Because of this, very few open-source EDA tools exist in the modern IC design era, and

none of them offer anywhere near the same feature set nor quality of results (QoR) that commercial tools provide. In an industry where every percent of PPA improvement matters, commercial tools are often the only acceptable option. Even in less competitive segments of the IC market, the lack of necessary features dissuades designers from using free or open-source tools.

Without any open infrastructures to build from, development of state-of-the-art EDA tools outside a commercial environment becomes extremely difficult and costly [29]. In other software communities, perhaps Linux being the most prominent example, open-source and closed-source developers alike have a state-of-the-art substrate to build innovation on top of. In EDA, such an analogue doesn't exist. Therefore, creating an incremental contribution to the state-of-the-art in EDA requires significant costs in re-implementing common substrates. With increasing barriers to entry in state-of-the-art EDA, many potential innovations are lost. In the Post-Moore Era where technology-independent innovations are key to performance improvements, new innovations are key to keeping the improvement from Conway's Process alive.

1.3.3 Acceleration Difficulties

Because TAT is a key concern from designers, ensuring that EDA tools run with high efficiency is also a key concern. During the height of Moore's Law, steady improvements to CPU performance helped keep the increasing complexity of designs from drastically impacting TAT. As improvements to single-thread performance waned, many EDA algorithms pivoted to partitioning and parallel algorithms to make use of multicore architectures and distributed systems. However, EDA tools are a mixed bag of algorithms which are each important for implementing a chip. Some algorithms are facing severe diminishing returns on parallelization (e.g., Amdahl's Law), and others are not amenable to parallelization at all. This leaves the performance trajectory of EDA tools in a tenuous position, as increasing the capability of the tools as well as providing them with more complex designs can lead to a severe slowdown in TAT.

1.4 A Path Forward

The challenges for EDA software listed in Section 1.3 pose a significant risk to innovation in hardware design. Interestingly, the problem space of EDA slowdown maps very closely to one of my co-authored works, Sirius[3] [25].

[3] Since renamed to Lucida

In 2014, intelligent personal assistants (IPAs) were becoming workloads of significant interest. Assistants such as Apple's Siri, Google's Google Now, and Microsoft's Cortana offered a natural method of interfacing with mobile devices and search engines. However, search engines return a list of results from a text query; intelligent personal assistants process speech inputs from the user, perform a search, *and* attempt to identify a single answer. Because of these additional steps, we hypothesized that IPA queries consumed a much greater amount of computation than the underlying search query alone. Studying IPAs proved difficult because all major IPAs were closed-source. Thus, Sirius' first major contribution was to *create an open-source platform for benchmarking*. An open-source platform was pivotal in enabling IPA research and performance measuring. Sirius also created *a representative benchmark suite* of IPA queries so that performance could be benchmarked on the platform.

With an open-source platform and benchmark suite, Sirius found that IPA queries consume over 100× the computation of traditional search engine queries; datacenters wouldn't be able to scale efficiently if IPA queries supplanted traditional search queries. To address this computationally intensive workload, Sirius proposed first *characterizing the platform for computationally expensive kernels* then *surveying hardware accelerators to improve these kernels' performance*. Sirius identified 7 kernels which comprise 92% of the CPU time. Then, Sirius surveyed commercially available hardware accelerators including multicore CPUs, manycore CPUs, GPUs, and field-programmable gate arrays (FPGAs). From this survey, Sirius found that GPUs and FPGAs can reduce the end-to-end query latency by 10× and 16×, respectively.

In summary, Sirius defined a roadmap for improving performance of intelligent personal assistants:

1. Create an open-source platform for benchmarking.
2. Create a a representative benchmark suite.
3. Characterize the open-source platform to identify key computational kernels.
4. Survey hardware accelerators to determine speedup for the end-to-end workload.

Returning to the challenges with EDA software, several similarities with IPA software can be observed: EDA software is closed-source, computationally expensive, and composed by a multi-step flow. Therefore, the roadmap presented by Sirius also maps very well to the problem space presented by EDA software. One additional problem posed by Sirius, however, is that *accelerator implementation and integration is difficult*. Changing

the computational platform from CPU almost always requires a change in programming language and/or model, leading to a high implementation overhead.

1.5 Reinvigorating Conway's Process

In this dissertation, I explore the challenges presented in Section 1.3 and propose solutions to push Conway's Process forward in the Post-Moore era. Using the roadmap created by Sirius (Section 1.4), I present 3 works which address the issues of design time, closed source, and difficulty in acceleration, respectively.

1.5.1 Celerity

A major challenge encountered with Sirius was the difficulty in implementing specialized accelerators. Implementing a design in hardware takes orders of magnitude more effort than implementing it in software. Aside from architecting, optimizing, and verifying the accelerator, integrating the accelerator into a full system requires further time and effort.

The difficulties experienced in creating Sirius led to the inception of Celerity. Celerity is a system-on-chip (SoC) with a 496-core manycore array, 10-core low-power array, 5 host cores, digital LDO, and digital PLL. Celerity is one of the most complex single-chip academic projects to date, but the main innovations of the project were to create design techniques which drastically reduced the amount of design time to create a complex SoC. My team, consisting only of grad students, created the chip with in only 9 months in an advanced FinFET node, whereas the typical SoC design time is on the order of 12-36 months for a team of engineers. This work is presented in Chapter 2.

1.5.2 OpenROAD

Celerity elucidated design techniques which can drastically reduce design time; however, it also revealed the challenges of working with proprietary electronic design automation tools and process development kits (PDKs). Restrictive licensing agreements forbid sharing design data and implementation scripts. Tools are inscrutable when they don't work as expected. Along with Sirius' roadmap, it was clear that EDA needed an open-source platform.

OpenROAD [8, 9, 45] is a fully open-source EDA tool for digital SoCs. In collaboration with my co-authors, I created OpenROAD-flow, an open-source implementation flow with the long-term goal of becoming fully autonomous. OpenROAD enables sharing

of implementation scripts and flows, community collaboration, and full implementation of chips without licensing fees. With the open-sourcing of the SkyWater 130nm PDK, it is now possible to create fully open-source chips from start to finish, as demonstrated by the Google/Efabless OpenMPW shuttle runs [7] which have enabled over 100 open-source designs to be taped out to date. This work is presented in Chapter 3.

1.5.3 SpeEDAr

Even with the availability of OpenROAD in place of commercial EDA tools, a major hurdle stands in the way. The algorithms which power EDA tools are extremely computationally expensive, and it is common for the turnaround time to consume more than 24 hours on the best available CPU platforms. OpenROAD presented a huge opportunity, as open-source code can be profiled and optimized with hardware acceleration (as Sirius was). In addition, it can be designed with software interoperability in mind – a distinguishing factor from closed-source software. This leads my work on SpeEDAr, which identifies the key kernels of computation in OpenROAD and accelerates them with a simulated ASIC implementation. SpeEDAr targets the search kernel of OpenROAD's detailed router and achieves a mean kernel speedup of 74×, which translates to a mean flow speedup of 1.20×. This work is presented in Chapter 4.

CHAPTER 2

Celerity

Sirius highlighted a major challenge with incorporating hardware accelerators into systems (Section 1.4). Implementation of a hardware design requires significantly more effort than software, and further effort is required to integrate the accelerator into a full system. To address this challenge, the Celerity project was formed with two primary goals:

1. Architect a system-on-chip (SoC) representative of new, innovative chip designs.
2. Shrink the SoC's design schedule to half of the typical 12-36 month schedule.

To meet the first goal, Celerity addresses the broad issue of a **usability gap**. When a new, computationally expensive software algorithm is introduced, chips typically take years to employ accelerators for the algorithm. The time gap between when an algorithm is introduced and when it becomes *usable* due to hardware acceleration is what forms the usability gap. As an example, AlexNet [34] from 2012 is one of the seminal works in machine learning which catapulted the field into everyday use. However, the first dedicated accelerators for convolutional neural networks such as AlexNet did not appear in consumer devices until around 2017 with Apple's Neural Engine and Google's Pixel Visual Core.

Even when new accelerator architectures are introduced, the accelerator behavior cannot be easily modified to adapt to changing workload properties. These factors motivate new architectures which can be rapidly constructed to address new application domains while still leveraging specialized hardware. In addition, the architectures need to offer high performance and energy efficiency even as applications evolve post-tapeout.

To meet Celerity's second goal, the project examined the steps in the design process which consume the most engineering time. The project targets many different techniques in order to reduce the total design schedule, which are broken down into 3 categories: reuse, modularization, and automation. These techniques are used on all stages of the design cycle, including the design, implementation, and verification.

2.1 Introduction

To address the usability gap, Celerity proposes a **tiered accelerator fabric** chip architecture. This architecture uses different tiers of computation where each tier represents a tradeoff between energy efficiency and workload flexibility. The tiered accelerator fabric design pattern minimizes time-to-market and allows the chip to maintain high performance and energy efficiency on evolving workloads. The Celerity SoC is implemented in 16nm technology with 3 tiers in the fabric: a general-purpose tier comprised of open-source Linux-capable RISC-V cores, a massively parallel tier comprised of a RISC-V tiled manycore array, and a specialization tier that implements an algorithmic neural-network accelerator. These tiers are tied together with an efficient heterogeneous remote store programming model on top of a flexible partial global address space memory system. In this chapter, I first discuss Celerity's architecture and how it implements a tiered accelerator fabric (Section 2.2). The following section, 2.3, discusses the implementation details of Celerity's chip. Section 2.4 reports Celerity's measurements and compares them to prior work. Lastly, Section 2.5 discusses the design methodologies used throughout the entire design process in order to meet the project goal of reducing the design schedule by half.

The key contributions of this work are as follows:

- The tiered accelerator fabric is presented as a methodology for rapid, high-efficiency SoC design. In addition, the design's entire source base is available at `http://opencelerity.org`.
- Celerity presents a novel network-on-chip (NoC) that provides high-bandwidth, low-latency communication with 61% less area and up to 67% less data overhead than prior work.
- Celerity's architecture provides best-in-class performance for energy efficiency and normalized area efficiency, outperforming prior work by 2.0× and 1.3×, respectively.
- I describe several design methodologies which enabled my team to reduce the SoC design schedule down to just 9 months.

This work was a large collaboration between many students and professors. My role was the student lead for the more than 20 students that participated in the project. My contributions included arranging meetings and organizing the design schedule among all groups, leading the SoC's physical design and full-chip integration (including implementation flow), innovating techniques to reduce design time, and collaborating with the architecture designers to create an efficient implementation.

2.2 Celerity Architecture

Celerity's tiered accelerator fabric has three architectural tiers:

- The general-purpose tier is a set of cores capable of executing operating systems, networking, control, and other decision-making code.
- The massively parallel tier is composed of scalable and programmable arrays of small, tightly coupled cores that attain high energy efficiency and flexibility for evolving workloads.
- The specialization tier is composed of algorithmic accelerators which target computational kernels with extreme energy-efficiency and performance requirements.

Celerity uses *autonomous vision systems* as a target application domain due to their rising prominence as a workload. Thus, the Celerity architecture implements the general-purpose, massively parallel, and specialization tiers using 5 Linux-capable RISC-V cores, a 496-core RISC-V manycore array, and a binarized neural network (BNN) accelerator, respectively. Figure 2.1 shows a block diagram of Celerity with each tier highlighted. To bind these components together, Celerity supports a heterogeneous remote store programming model that allows cores and accelerators to write to each other's memories through a partitioned global address space. Layered upon this model are two novel synchronization mechanisms: load-reserved, load-on-broken-reservation (**LR-LBR**), which extends load-reserved store conditional for efficient producer-consumer synchronization; and the **token queue**, which uses LR-LBR to achieve efficient producer-consumer transfer of resource ownership.

In the remainder of this section, I first discuss the partitioned global address memory system and how it connects the 3 tiers. Then, the architecture of each tier is discussed.

2.2.1 Partitioned Global Address Space

Communication among accelerators and cores in the three tiers is accomplished through a partitioned global address space over a unified mesh network-on-chip (NoC). When a remote store is performed, a wide single-word packet is injected into the NoC, which contains x,y coordinates of the destination core, the local word address at that core, 32 bits of data to store, and a byte mask. The addressing scheme is illustrated in Section 2.2.3. When the message arrives at the destination, the address is translated and the store is performed. Ordering of messages sent from one node to another is maintained. The parameterized NoC

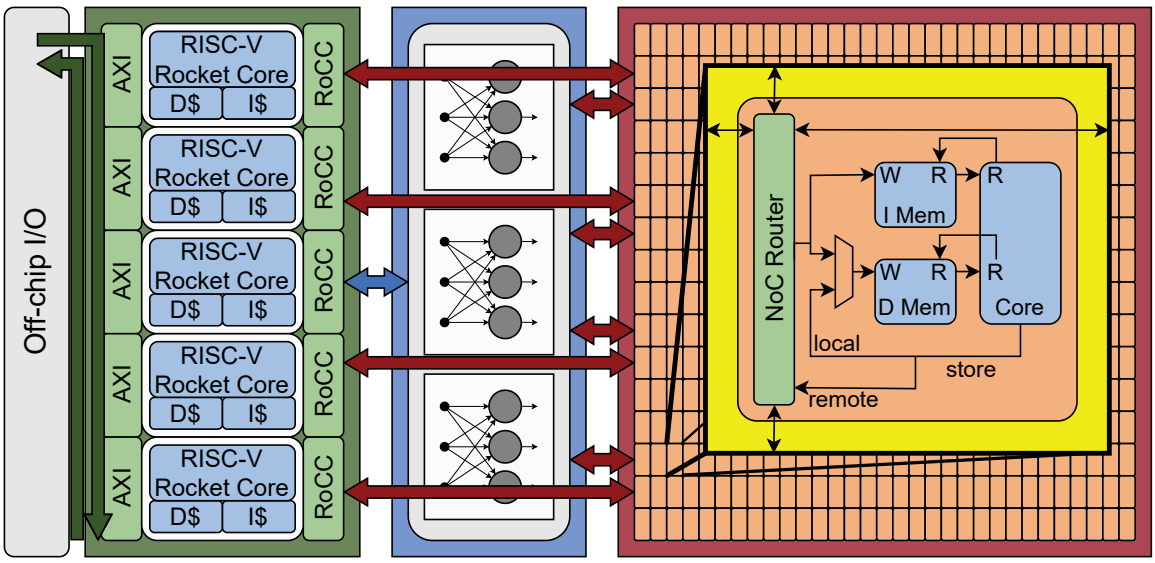

Figure 2.1: Celerity block diagram. The general-purpose tier (green) has a five-core Rocket core complex, the specialization tier (blue) has a BNN accelerator, and the massively parallel tier (red) has a 496-core tiled manycore array. [18]

in Celerity was configured for 512 coordinates (x = 0..15, y = 0..31) and 22-bit addresses. The mesh's cores map one-to-one to all of these addresses except y = 31, which demarcates the south edge of the manycore. The remaining 16 positions on the south edge are used for four parallel connections to the BNN and four connections to the Linux-capable Rocket cores. The remaining connections are unused in this implementation.

While remote loads, such as those found in the Adapteva parallel architecture [42], are easy to add and could arguably make the system more programmable, they have high round-trip latency costs and lead users astray by offering a high-convenience, low-performance mechanism. Remote stores do not incur such a latency penalty because they are pipelined and can therefore be issued once per cycle.

When a remote store is performed, a local credit counter will be decremented at the sender. When the store is successful at the remote node, a store credit is placed on the store network that is routed back to the original tile on a separate 9-bit physical network, incrementing the counter. A RISC-V fence instruction on either a manycore tile or a Rocket core is used to detect whether any outstanding remote stores exist, allowing a core to pause for memory traffic to finish during a barrier.

2.2.2 The General-Purpose Tier

For the SoC to support complex software stacks, exception handling, and memory management, Celerity instantiates five Berkeley RISC-V Rocket cores running the RV64G ISA. The Rocket core is an open-source [10], five-stage, in-order, single-issue processor with a 64-bit pipelined floating-point unit and size-configurable, non-blocking caches. Each Rocket core can run an independent Linux image. This provides the flexibility to run SPEC-style applications and network stacks like TCP/IP. Four Rocket cores connect directly to the massively parallel tier using parallel remote store links on the global mesh NoC. One Rocket core connects directly to the specialization tier through a dedicated Rocket custom coprocessor (RoCC) interface. These connections are made using the Berkeley RoCC interface. L1 data and instruction caches are configured at 16 KBs each.

When remote stores are performed to the Rocket cores, they go directly into the four Rocket cores' caches, potentially causing cache misses to DRAM. Remote store addresses are translated using a segment address register that maps the 22-bit address space into the Rocket's 40-bit address space. Rocket cores issue remote stores through a single RoCC instruction and can, for example, perform remote stores to other Rocket cores, to any manycore tile, or to any of the BNN input links. Remote stores to manycore tiles are used to write instruction and data memories, as well as to set configuration registers, such as freeze registers and arbitration policies for the local data memory.

2.2.3 The Massively Parallel Tier

To achieve massive amounts of programmable energy-efficient parallel computation, Celerity implements a 496-core tiled manycore array [53] that interconnects low-power RISC-V Vanilla-5 cores using a mesh interconnection network (Figure 2.2). Each tile contains a simple router and a Vanilla-5 core. The inhouse-developed Vanilla-5 cores are five-stage, in-order, single-issue processors with 4-KB instruction and data memories that use the RV32IM ISA. The manycore uses a strict remote store programming model [26], providing a highly programmable array to maintain high performance as workloads evolve post-tapeout. A key contribution of this work is to extend the remote store programming model to incorporate heterogeneous processor types and to support fast producer-consumer synchronization.

2.2.3.1 NoC Design

The manycore's mesh NoC design, which facilitates the remote store fabric that ties the chip together, targets extreme area efficiency using only a single physical network for data

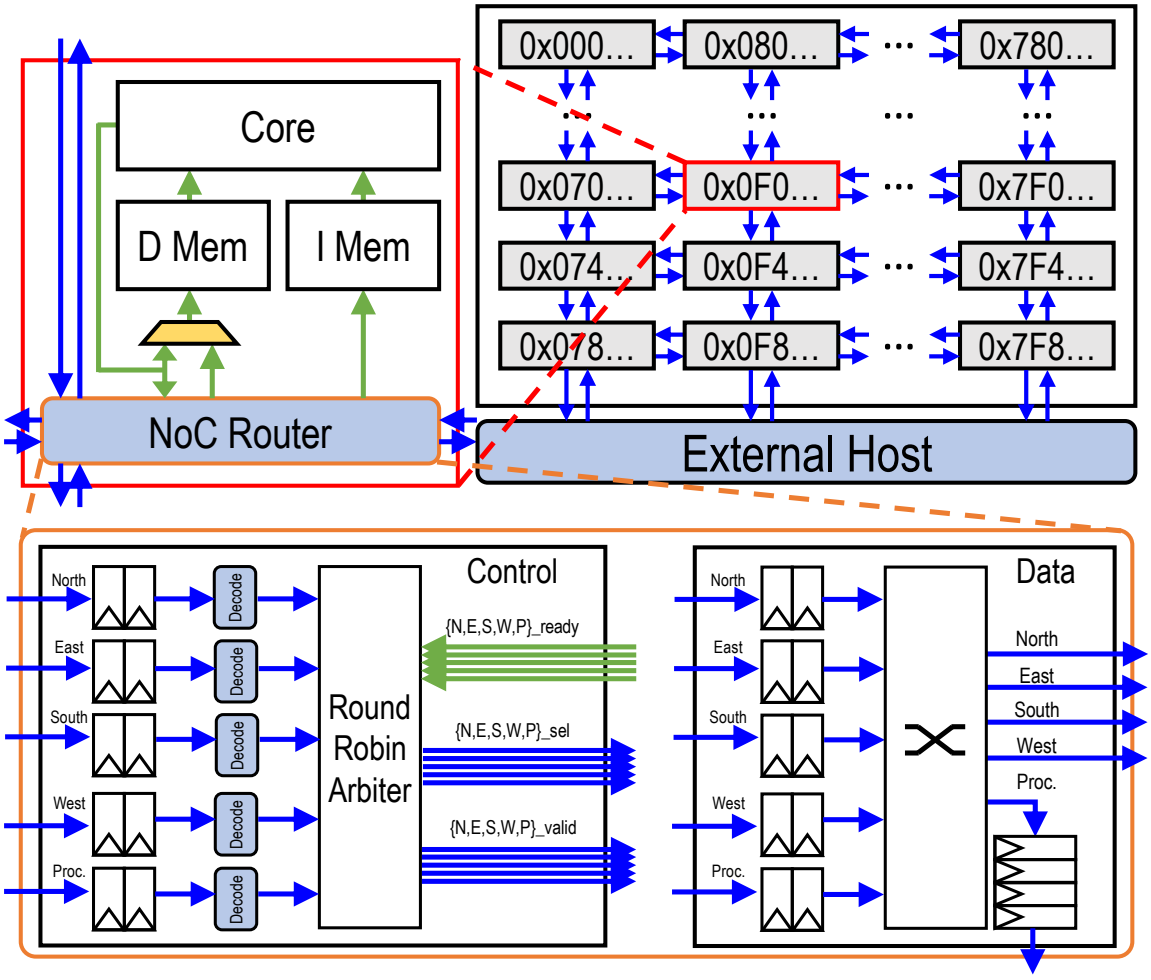

Figure 2.2: Manycore mesh architecture with callouts to an individual tile and router [46]

transfer, no virtual channels, single-word/single-flit packets, deterministic x,y dimension-ordered routing, and two-element router input buffers. Head-of-line blocking and deadlock are eliminated because remote stores can always be written to a core's local memory, removing the word from the network. Connections between neighboring tiles are 80-bit wide full duplex, allowing address, command, and data information to be routed in a single wide word, and each hop takes one cycle. To generate packets that go off the array's south side to the specialized and general-purpose tiers, a NoC client performs a store to a memory address whose x,y coordinate is beyond the coordinates of manycore. Both local and remote stores use the same standard store word, half-word, and byte instructions from the ISA.

2.2.3.2 Remote Stores

Each time a store is about to be performed, the high bit of the address determines if the store address is local (0) or remote (1). The local address space uses the remaining 31 bits to determine the memory address. The remote address space uses the next 9 bits as a destination coordinate (x = 0..15, y = 0..31) of the target core on the NoC. The remaining 22 bits are translated at the destination into a local address, and the store is performed.

2.2.3.3 LBR

The manycore features an extension to the LR store-conditional (LR-SC) atomic instructions called LR-LBR. LR operates much like in LR-SC by performing a load and then adding the target address to a reservation register, which is then cleared if an external core writes to that address. LBR is a new instruction that places the core's pipeline in a low-power mode until another core remote stores to that address and breaks the reservation, at which point the core will wake up and perform a load on the target address. Typically, user code will load a memory location's value with LR, branch away if it is satisfied with the value (a ready flag is set, or a FIFO pointer has sufficiently advanced), and otherwise fall through to a LBR to wait for it to change, so it can be rechecked.

2.2.3.4 Token Queue

Celerity's design shows that tight producer-consumer synchronization can be layered on top of remote store programming. With the LR-LBR instruction extension, Celerity implements a token queue, a software construct used to asynchronously transfer control of buffer address between producer and consumer tiles. The consumer will allocate a circular buffer to which tokens can be enqueued and dequeued. A token can be a simple data value, a pointer to a memory buffer, or identifiers for more abstract resources. Producer

and consumer can consume different quantities of tokens at each step. By enqueuing a set of tokens, the producer is transferring read/write ownership of those resources to the consumer. By dequeuing a set of tokens, the consumer is transferring write ownership of the resource back to the producer. The producer and consumer each have local copies of head and tail pointers to the circular buffer, but only the producer will modify the head pointers, and only the consumer will modify the tail pointers. The remote versions of the pointers will be updated after the local versions, similar to a clock-domain-crossing FIFO.

The producer tile confirms there is enough space in the token queue to enqueue a particular group of tokens, using LR-LBR to wait in low-power mode for remote updates to the local tail pointers if there is not enough space in the queue. Then, it will send the corresponding data through remote stores. After completion, the producer will update the head pointers through local and remote stores.

The consumer confirms that it has enough tokens in the token queue to proceed, using the LRLBR instructions to wait in low-power mode until the head pointer is updated by the producer, and checking if enough tokens have been enqueued. When there is enough, the consumer will wake up and start accessing the data represented by the new tokens in the buffer. When done, the consumer will dequeue the tokens by updating the tail pointers and proceeding back to consuming the next set of tokens.

2.2.3.5 Programming Models

Software programs are compiled using a different workflow from shared memory systems. Because each tile is a RISC-V core, C/C++ programs can be compiled using the standard RISC-V toolchains. Celerity uses a custom GCC linker script which maps data and instructions to separate 4KB segments such that instructions and data may be placed into the respective instruction and data memories. When compiling, the program must target a single tile and fit within a tile's instruction/data memory. For Single-Program, Multiple-Data (SPMD) class programs, the same program can simply be replicated across each tile in the mesh. Larger programs can be constructed by partitioning instructions across tiles and explicitly passing data between them. For example, a large program can be split into multiple program segments. Each segment is stored in a different tile's instruction memory, and data is passed between tiles. In the case of streaming applications, this works particularly well for separating consumer and producer functions across tiles and streaming data between them. Infrastructures have been developed to simplify compiling such workloads, such as StreamIt [54], an infrastructure to automatically partition programs using annotations, and `bsg_manycore_lib`, my team's library for sending, receiving, and synchronizing data across tiles. These infrastructures enable compilers/libraries to orchestrate the data transfer.

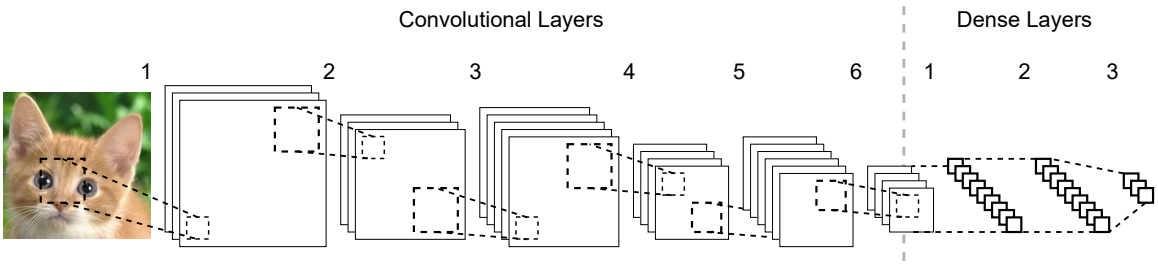

Figure 2.3: BNN CIFAR-10 Network Architecture [18]

2.2.4 The Specialization Tier

Careful consideration is required when deciding which workload kernels to implement in the specialization tier. Celerity implements a BNN accelerator. This section discusses the architecture and reasoning for selecting a BNN as the specialization tier.

2.2.4.1 Choosing the Neural Network

Deep convolutional neural networks (CNNs) are now the state of the art for image classification, detection, and localization tasks. However, using CNN software implementations for real-time inference in embedded platforms can be challenging due to strict power and memory constraints. This has sparked significant interest in hardware acceleration for CNN inference, including my team's own prior work on FPGA-based CNN accelerators [60]. Given the context of autonomous vision systems, Celerity uses flexible image recognition as a case study for demonstrating the potential of tiered accelerator fabrics in general, and this SoC specifically.

Most prior work on CNN accelerators uses carefully hand-crafted digital architectures and represent the weights and activations in 8- to 16-bit fixed-point precision. Recent work on BNNs has demonstrated that binarized weights and activations (+1, -1) can, in certain cases, achieve accuracy comparable to full-precision floating-point CNNs [17]. BNNs' key benefit is that the computation in convolutional and dense layers can be realized with simple exclusive-negated-OR (XNOR) and pop-count operations. This removes the need for more expensive multipliers and adder trees, saving area and energy. BNNs can also achieve a substantial reduction (8-16×) in the memory size of weights compared to a fixed-point CNN using the same network structure, making the model easier to fit on-chip. Additionally, there is an active body of research on BNNs attempting to further improve classification performance and reduce training time.

Celerity employs the specific BNN model shown in Figure 2.3 based on Courbariaux et al. [17]. This model includes six convolutional, three max-pooling, and three dense

	Convolutional layers						Dense Layers		
	1	2	3	4	5	6	1	2	3
In Arrays	3	128	128	256	256	512	8K	1K	1K
Out Arrays	128	128	256	256	512	512	1K	1K	10
Out Dim	32	32	16	16	8	8	1	1	1
Out Size (bits)	128K	128K	64K	64K	32K	32K	1K	1K	10
Weights (bits)	3456	144K	288K	576K	1.1M	2.3M	8.0M	1.0M	10K
Exe. Time (%)	2.1	23.1	12.0	24.0	12.9	25.7	0.2	0.02	0.01

Table 2.1: BNN Algorithm Characterization [18]

(fully connected) layers. The input image is quantized to 20-bit fixed-point, and the first convolutional layer takes this representation as input. All remaining layers use binarized weights and activations. BNN-specific optimizations include eliminating the bias, reducing the batch norm calculation's complexity, and carefully managing convolutional edge padding. This network achieves 89.8% accuracy on the CIFAR-10 dataset.

2.2.4.2 Performance Target

The BNN targets ultra-low latency, requiring a batch size of one image and a throughput target of 60 classifications per second to enable real-time operation.

2.2.4.3 Creating and Optimizing the Specialization Tier

The BNN was implemented using a three-step process to map the application to the tiered accelerator fabric. First, the algorithm is implemented using the general-purpose tier for initial workload characterization and to identify key kernels for acceleration. Second, the algorithm can be accelerated using either the specialization tier or the massively parallel tier. Finally, performance and/or efficiency can be further improved by cooperatively using both the specialization tier and the massively parallel tier.

2.2.4.4 Establishing the Functionality of the Specialization Tier

In the first step, the BNN is implemented using the general-purpose tier to characterize the computational and storage requirements of each layer. Table 2.1 shows the number of binary weights and binary activations per layer in addition to the execution time breakdown, assuming a very optimistic embedded microarchitecture capable of sustaining one instruction per cycle. The total estimated execution time for the BNN software model (estimated to be around 2 billion instructions) on the general-purpose tier would be approximately

200× slower than the performance target. Although the binarized convolutional layers require more than 97% of the dynamic instructions, preliminary analysis suggests that all nine layers must be accelerated to meet the performance target. The storage requirements for activations are relatively modest, but the storage requirements for weights are non-trivial and require careful consideration.

2.2.4.5 Designing the Specialization Tier

In the second step, the BNN is implemented as a configurable application-specific accelerator in the specialization tier. This accelerator was designed to integrate with a Rocket core in the general-purpose tier through the RoCC interface. Although the massively parallel tier could be used to implement the BNN at speed, superior energy efficiency could be attained through specialization. Figure 2.4 shows the BNN accelerator architecture. The BNN accelerator consisted of modules for fixed-point convolution (first layer), binarized convolution, dense layer processing, weight and activation buffers, and a DMA engine to move data in and out of the buffers. The BNN accelerator processes one image layer at a time and can perform 128 binary multiplications (XNORs) per cycle using two convolvers. Any non-binarized computation is performed completely within each module to limit the amount of non-binarized intermediate data stored in the accelerator buffers and/or memory system. The activation buffers are large enough to hold all activations; however, in this design, the sizeable binarized weights necessitated off-chip storage using the general-purpose RoCC memory interface. The binarized convolution unit includes two convolvers implemented with a flexible line buffer based on Zhao et al. [60].

2.2.4.6 Combining the Massively Parallel and Specialization Tiers

In the third step, the potential for cooperatively using both the specialization tier and the massively parallel tier is explored. Early analysis suggested that repeatedly loading the weights from off-chip would significantly impact both performance and energy efficiency. Celerity implements a novel mechanism that enables cores in the massively parallel tier to send data directly to the BNN. To classify a stream of images, Celerity first loads all data memories in the massively parallel tier with the binarized weights. It then repeatedly executes a small remote-store program on the massively parallel tier; each core takes turns sending its portion of the binarized weights to the BNN in just the right order. The BNN can be configured to read its weights from queues connected to the massively parallel tier instead of from the general-purpose tier.

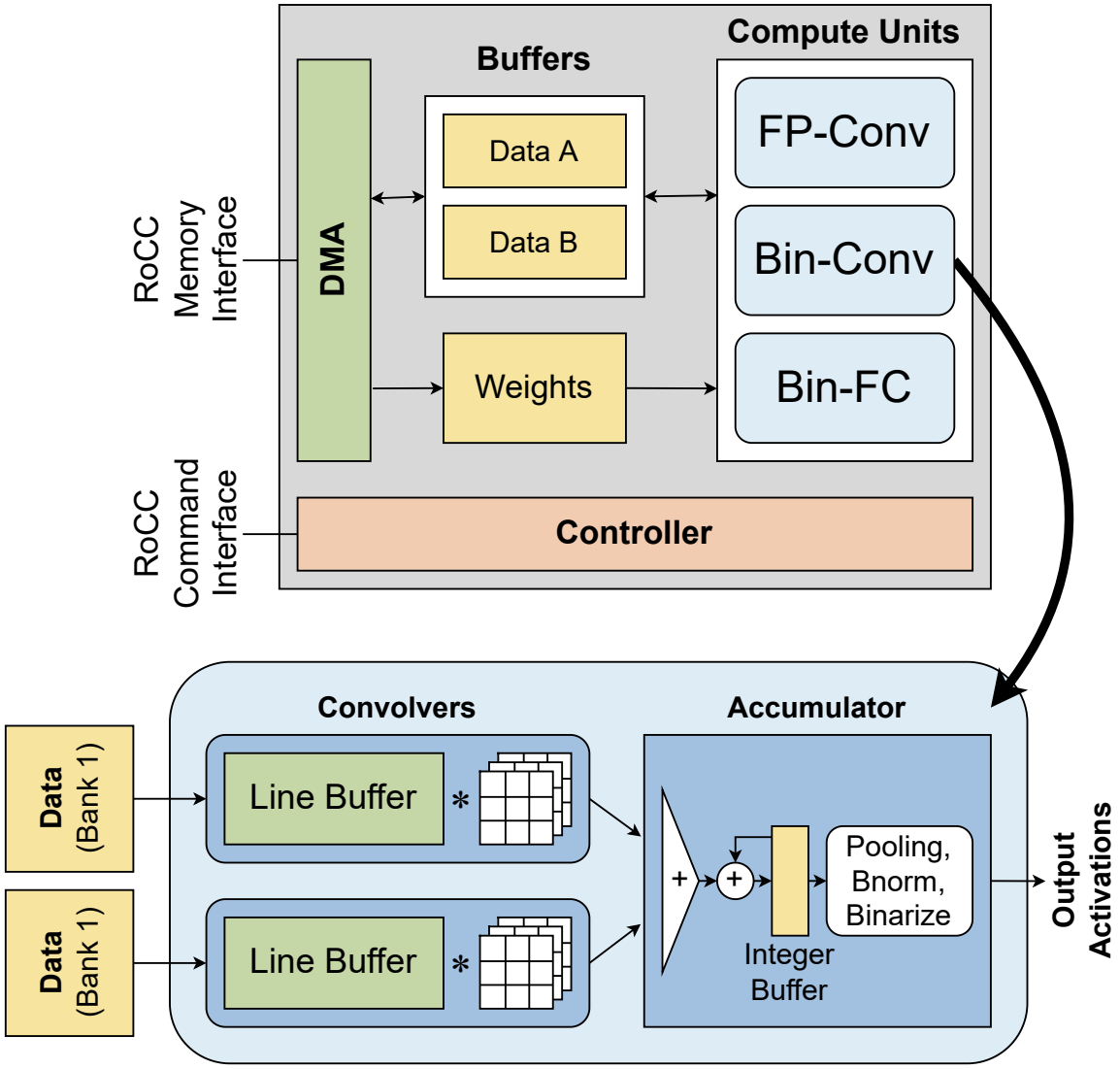

Figure 2.4: BNN specialized accelerator architecture [18]

	Runtime (ms)	Performance (images/s)	Power (W)	Efficiency (images/J)	Relative Efficiency vs. GPT	Relative Efficiency vs. mGPU
GPT	4024.0	0.3	0.1	2.5	1.0×	
SpT	20.0	50.0	0.2	250.0	100.0×	
SpT+MPT	3.2	312.5	0.4	625.0	250.0×	208.3×
mGPU [60]	90.0	11.1	3.6	3.0		1.0×
CPU [60]	14.8	67.6	95.0	0.7		0.2×
GPU [60]	0.7	1428.6	235.0	6.0		2.0×
FPGA [60]	5.9	168.4	4.7	35.8		11.9×

Table 2.2: Performance comparison of optimized BNN implementations on different platforms. GPT = general-purpose tier. SpT = specialization tier with weights stored in GPT cache. SpT+MPT = specialization tier with weights stored in the massively parallel tier. mGPU = Nvidia Jetson TK1 embedded GPU board. CPU = Intel Xeon E5-2640. GPU = Nvidia Tesla K40. FPGA = Xilinx Zynq-7000 SoC [18]

2.2.4.7 The Benefits of HLS

Celerity employed HLS to accelerate the design time and to enable significant design-space exploration for the BNN algorithm. The BNN model was first implemented in C++ for rapid algorithmic development, before adding HLS-specific pragmas and cycle-accurate SystemC interface specifications. Cadence Stratus HLS transformed the SystemC code into cycle-accurate RTL. Very similar C++ test benches were used to verify the BNN algorithm, the SystemC BNN accelerator, the generated BNN RTL, and the Rocket core running the BNN accelerator. This HLS-based design methodology enabled three graduate students with near-zero neural-network experience to rapidly design, implement, and verify a complex application-specific accelerator.

2.2.4.8 Performance Analysis of the Specialization Tier

Table 2.2 shows the performance and power of optimized BNN implementations on the Celerity SoC and other platforms. Although each platform uses a different implementation methodology, technology, and memory system, these results can still provide a rough high-level comparison. These results suggest that the Celerity SoC can potentially improve energy efficiency by more than 10× compared to my team's prior FPGA implementation [60] and more than 100× compared to a mobile GPU.

In the table, runtimes measure processing a single image from the CIFAR-10 dataset. The power of GPT, SpT, and SpT + MPT are estimated using post-place-and-route gate-level simulations. DRAM power is excluded from the power estimates.

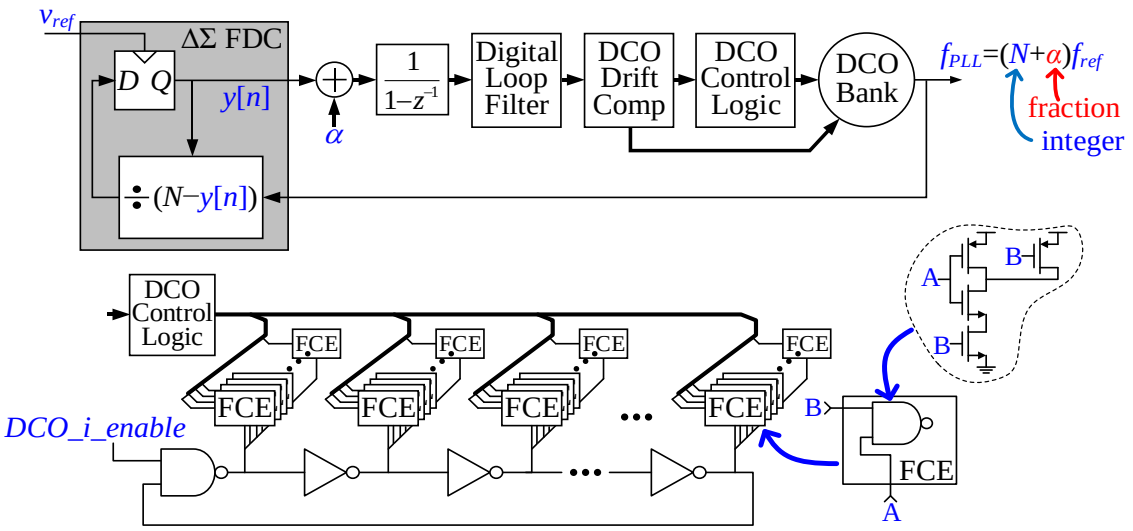

Figure 2.5: Synthesizable PLL architecture

2.2.5 Digital PLL

Three of Celerity's clock domains are provided by three instances of a custom, fully synthesized, and automatically placed and routed clock generator. The generator operates from an isolated 0.8 V supply and occupies 5,898 μm^2. With a reference frequency of f_{ref} = 26 MHz, its output frequency is tunable from 10 MHz to 3.3 GHz with minimum increments of no more than 2%, and consumes 1.5–3.5 mW at the min and max frequencies, respectively. The PLL achieves a (simulated worst-case) period jitter of 2.5 ps. Jitter was obtained using a bit-exact, event-driven simulation which accounts for phase noise. The simulation forgoes supply noise, as the design was done in parallel to the SoC before supply characteristics were known. However, the synthesizable architecture was created to be tolerant of supply noise. The PLL locks both frequency and phase with a simulated worst-case lock time of 230 μs.

The clock generator's PLL core (Figure 2.5) consists of a first-order ΔΣ frequency-to-digital converter [12], an α adder, a frequency-to-phase accumulator, a digital low pass loop filter, DCO drift compensation logic, DCO control logic, and a bank of 16 DCOs. The 16 DCOs together cover a frequency range of 1.3–3.3 GHz, and only one DCO is enabled for each output frequency setting. Each DCO (Figure 2.5) is a ring oscillator wherein each inverting delay element is loaded with a bank of NAND gate frequency control elements (FCEs) [16]. The PLL targets a 50% frequency range overlap above and below for each DCO in order to margin against process, voltage, and temperature variation (Figure 2.6). The DCO drift compensator dynamically controls 37 of the FCEs to compensate for drift of

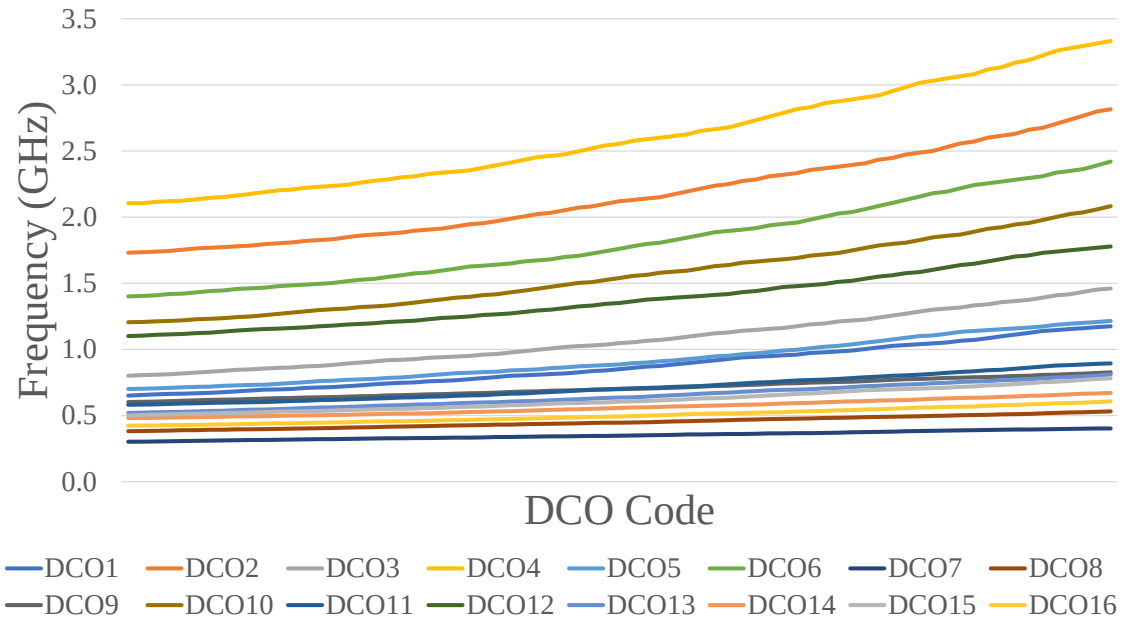

Figure 2.6: PLL DCO code vs. simulated output frequency

the DCO's center frequency over temperature and supply. The DCO control logic partitions its input into integer and fractional parts. The integer part drives all but 8 of the remaining FCEs with an update rate of f_{ref}. The fractional part is oversampled by a second-order $\Delta\Sigma$ modulator followed by a dynamic element matching encoder, the output of which drives the final 8 FCEs.

Ur Rahman et al. [55] propose a similar architecture to this work, however a key distinction is that this work uses NAND gates as loading elements to vary node capacitance, whereas ur Rahman et al. use inverters in parallel to vary drive current. NAND gate loading is compatible with synthesis tools, whereas parallel driving cells are usually not, due to a lack of tristate devices in most digital cell libraries.

2.3 Celerity Implementation

Celerity was implemented on a 5×5mm chip in TSMC's 16nm FFC process using 385 million transistors. The chip has 4 separate clock domains: the source-synchronous I/O, the manycore, the LDO (from an external clock), and all other components (Rocket, BNN, low-power array, core logic). Figure 2.7 shows a floorplan diagram of Celerity, Figure 2.8 shows a photomicrograph of the die, and Table 2.3 lists the area consumption and maximum frequency of each component.

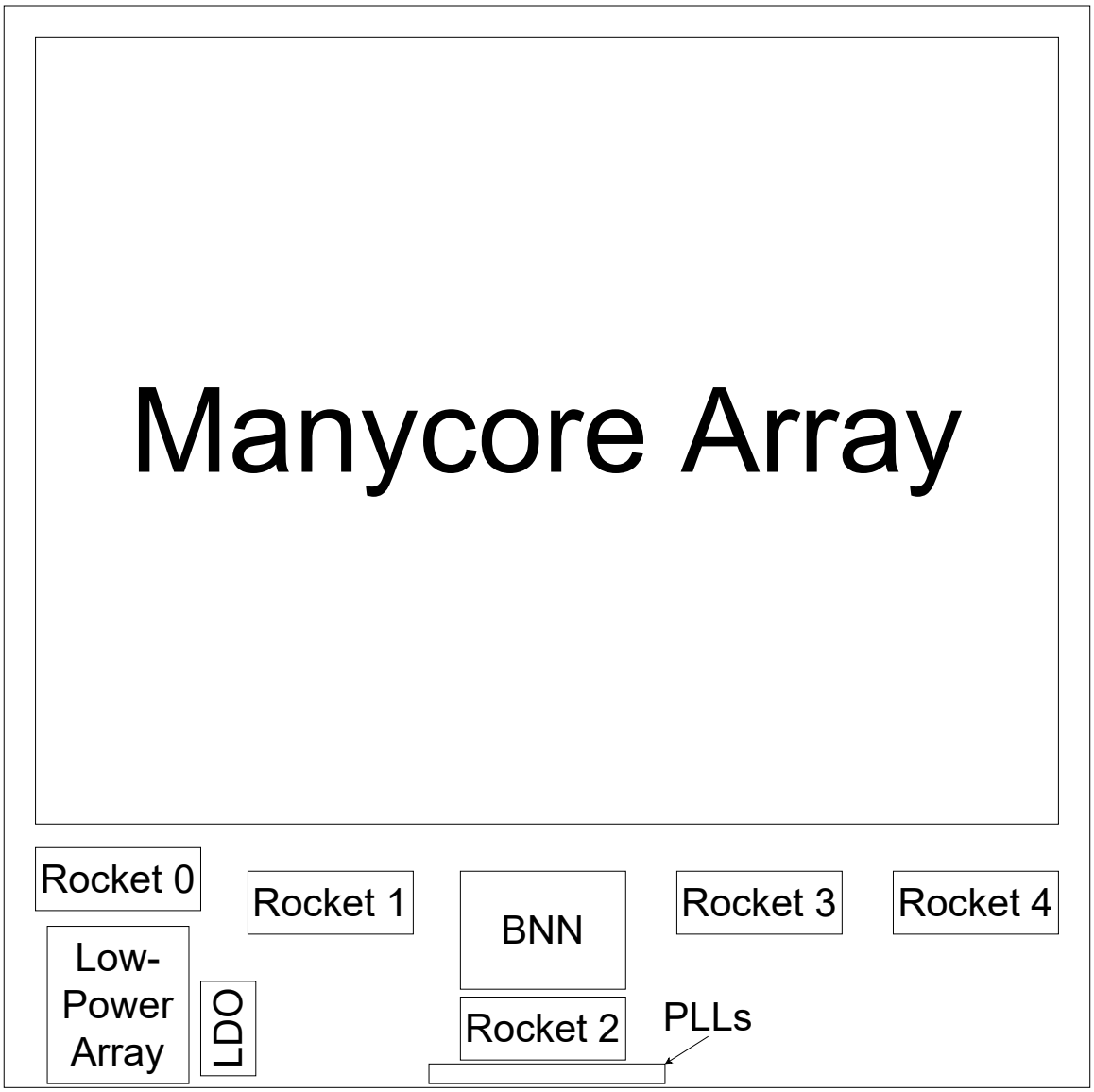

Figure 2.7: Celerity implementation floorplan

Component	**Die area** (mm^2)	**Max frequency** (MHz)
Rocket core	0.202	800
Manycore tile	0.024	1400
Low-power tile	0.024	800
BNN	0.356	800
PLL	0.006	–
LDO (controller)	0.002	–
LDO (decap)	0.074	–

Table 2.3: Celerity component area breakdown

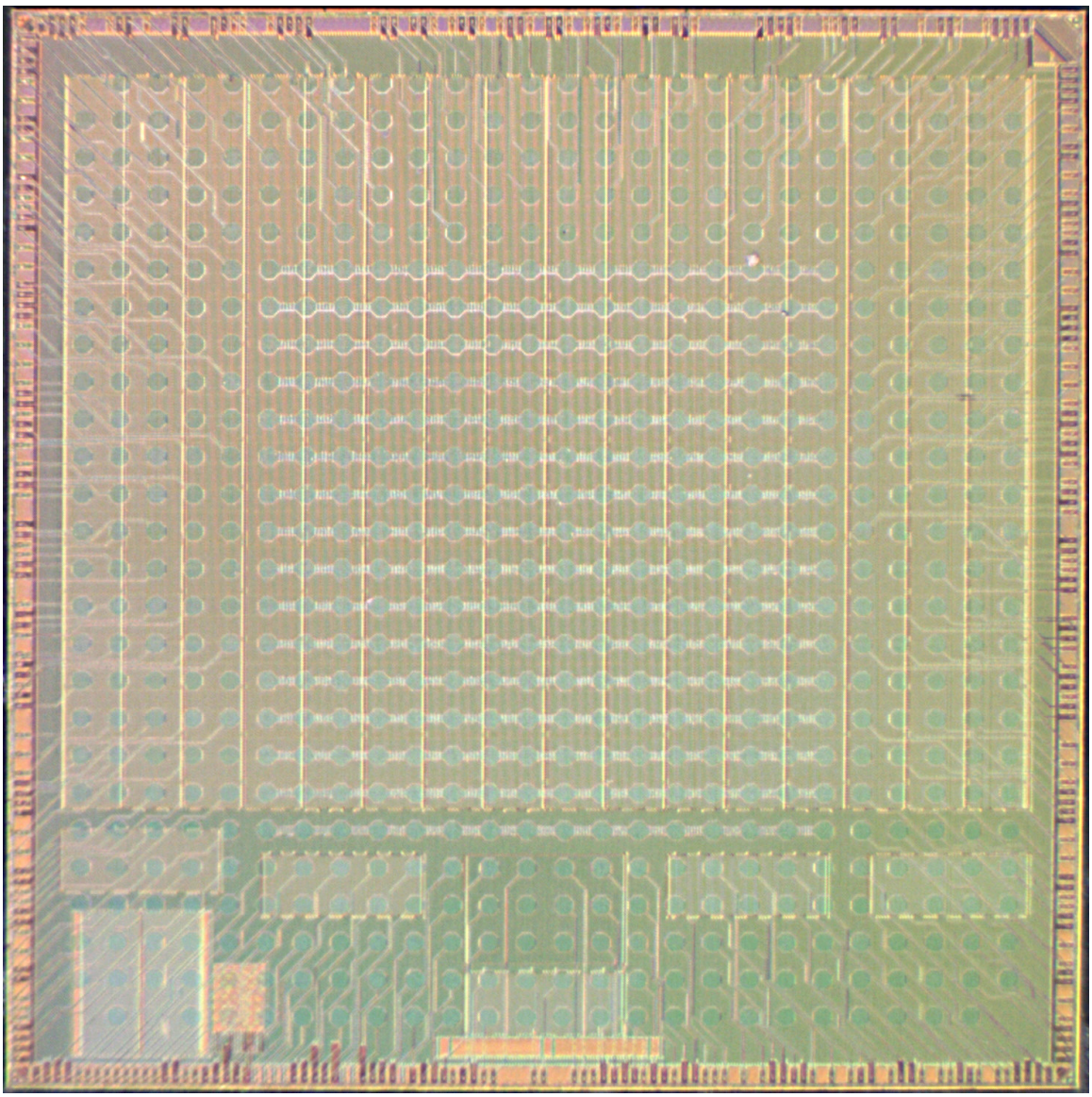

Figure 2.8: Celerity die photomicrograph after removal of bumps and top metal layer

2.3.1 Manycore Implementation

Figure 2.9 shows the layout of a single tile, which contains a "Vanilla-5" core and the routing logic for that node. A core implements the 32-bit RISC-V base instruction set and the multiply/divide extension (RV32IM) in a 5-stage pipeline. Each tile contains 2× 4KB SRAMs for instruction/data memories (IMEM/DMEM), and a 32-entry, 32b register file implemented using two 1r1w latch-based memories. The router is a single-stage design, allowing it to arbitrate, route, and send flits in a single cycle. In addition to providing low latency, the area of the router is reduced over a multi-stage design. Because there are no pipeline registers between nodes, flits take only 1 cycle per hop. Two-element FIFOs are used at the input for each direction to hold packets in case of congestion. To implement both rate limiting and memory fences, the router uses a source-controlled credit counter. The credit counter is decremented on each packet injected into the network from a remote store, and incremented when a remote store completes. Credits are returned over a separate 9-bit NoC with the same architecture as in Figure 2.2. The per-module physical area breakdown is listed in Table 2.9, with the NoC occupying only 1881 μm^2 (7.8%) of the tile. The router supports 80b transfers per cycle, which packages data, address, and commands into a single flit. The router and core run on the same clock domain up to 1.4 GHz, allowing each tile to both transfer 750 Gb/s and process 1.4 giga-RISC-V instructions per second (GRVIS). Several gaps were created between rows of tiles to allow for electrostatic discharge (ESD) cells and in-cell overlays (ICOVL) as required for fabrication. The total die area of the manycore is 15.25 mm^2 as fabricated with ESD and ICOVL (or 12.03 mm^2 without). This yields an area efficiency of 45.57 GRVIS/mm^2 (57.77 GRVIS/mm^2).

2.4 Measurements and Comparison to Prior Work

2.4.1 Partitioned Global Address Space

Celerity's use of a partitioned global address space (Section 2.2.1) enables the manycore to achieve a high compute density. Figure 2.10 shows the area overhead of a traditional memory system (directory-based coherent cache) vs. Celerity's PGAS system, demonstrating that PGAS offers over a 20× reduction in area overhead. The comparison system breakdown was extracted from Celerity's RV64G control cores, with directory area conservatively estimated from Sanchez and Kozyrakis [48]. The cost of removing these structures is mainly the ease of programming that comes from shared memory. However, streaming and highly parallel workloads often have well-defined dataflow patterns, which can enable compilers to manage data movement and mitigate this cost (see Section 2.2.3.5).

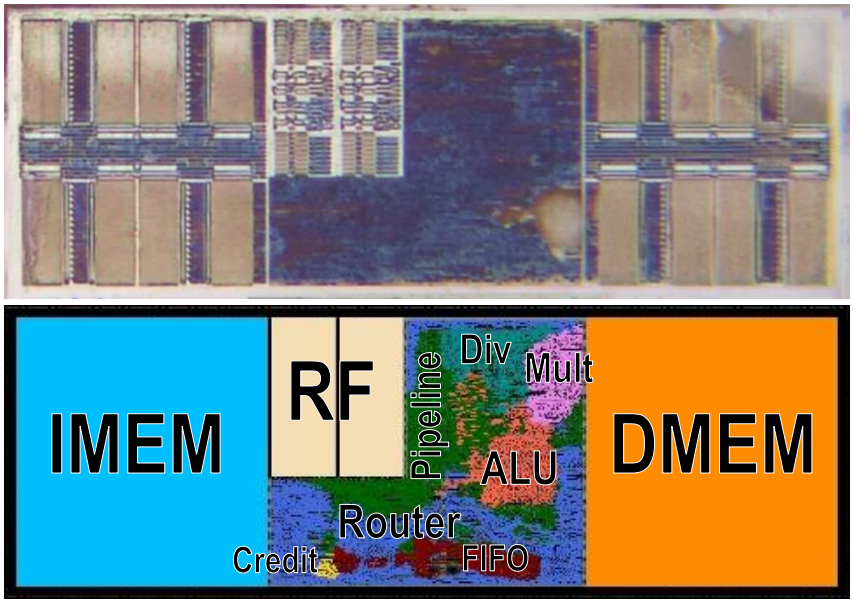

Figure 2.9: A manycore tile die photograph (top) and corresponding floorplan (bottom). The die photograph has most metal layers removed.

	Cell Type	Area (μm^2)	%
	IMEM	6,691	27.59
	DMEM	6,691	27.59
	RF	2,008	8.28
	Core logic	2,473	10.20
	ALU	485	2.00
	Div	412	1.70
	Mult	301	1.24
	Pipeline/other	1,275	5.26
	NoC	1,881	7.76
	Endpoint FIFO	303	1.25
	Credit counter	23	0.09
	Router	1,555	6.41
	Endcap/welltap	281	1.16
	Filler	1,635	6.74
	Unutilized	2,591	10.68
	Total	24,251	100.00

Table 2.4: Physical area breakdown of each manycore tile

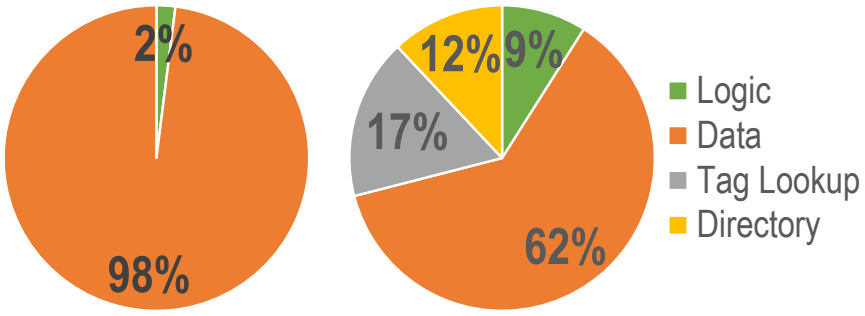

Figure 2.10: Area breakdown for Celerity's PGAS memory system (left) vs. a comparable directory-based coherent cache (right)

2.4.2 Remote Store Programming

The remote store programming model (Section 2.2.3.2) disallows loads from remote memory to remove a low-performance programming paradigm. A node can freely load or store to its local memory, but can only perform stores to remote memory. The use of remote store programming in Celerity enabled a 10% reduction in router area over a router with remote loads. In addition, pipeline stalls associated with long-latency remote loads are prevented.

2.4.3 Single-Flit Packets

Celerity implements a different flow control scheme compared to prior work. While wormhole routing is common due to its relative efficiency, it still has inefficiencies related to packet ingestion in the network. Most wormhole schemes require head and/or tail flits to reserve routes and communicate metadata. This results in network overhead, as sending a single data flit results in 2-3 flits being injected into the network. In addition, wormhole routing can cause head-of-line blocking when packet route reservations conflict.

Celerity instead implements a single-flit packet protocol, where the command, address, and data of a packet is contained in a single flit. This flow control scheme offers several benefits over wormhole routing:

- No head or tail flits – no overhead flits in a packet
- No head-of-line blocking because routes are not reserved (congestion can still occur)
- Small core-to-core latency, especially for adjacent cores
- An in-order pipeline can execute one store per cycle, because a store injects only one flit into the network

Work	Routing model	Arbitrary destination	Head-of-line blocking	Packet throughput	Min. latency (cycles)	Overhead flit fraction
TILE64[13] Piton[40]	Wormhole	Yes	Yes	0.33 / cycle	h + n + t + 1	2 / (n + 1)
KiloCore [14]	Wormhole	Yes	Yes	0.33 / cycle	2h + n + 1*	2 / (n + 2)
	Circuit switched	No	N/A^{+}	Not Reported	Not Reported	Not Reported
Celerity	Single-flit packet	Yes	No	1 / cycle	h + n - 1	0

h hops, n data flits, t turns in the network path. Core↔router is 1 hop.
* KiloCore's network is GALS and requires synchronization for each hop.
$^{+}$ KiloCore's circuit switched NoC can only be reprogrammed during the processor configuration phase.

Table 2.5: Comparison of flow control models

Table 2.5 provides a comparison of the single-flit flow control model versus the related work, and Figure 2.11 provides an example of each flow control model sending one flit of data to another node. The model allows the network to outperform prior works in both packet throughput and latency for small data transfers. The difference in latency between the models diminishes towards larger data transfers, however data streaming workloads favor smaller transfer sizes with smaller latency in order to allow processing at the next node sooner. Critical paths in the design lie in both the core and NoC, although experiments show that the NoC tends to be the limitation on frequency. In terms of impact on improvement over related work, the NoC and core architecture both contribute significantly.

2.4.4 Performance

The primary workload used to benchmark the manycore is CoreMark, a computationally intensive benchmark that stresses pipeline performance. CoreMark is ported to the manycore platform by starting with the "barebones" implementation provided by EEMBC. With this implementation, a simple linker script is created to identify which functions to distribute to the manycore tiles versus the functions to run on the host control cores. CoreMark's parallelization interface is then used to load the program binaries to all manycore tiles and run the program. The CoreMark benchmark enumerates the criteria to submit a valid CoreMark score, which Celerity adheres to. Unlike previously reported [46], the scores reported in Section 2.4.4.1 do not use modified core benchmark code. A change in compiler version and compiler flags allowed my team to fit the benchmark within a single tile's IMEM, as well as modestly improve the score.

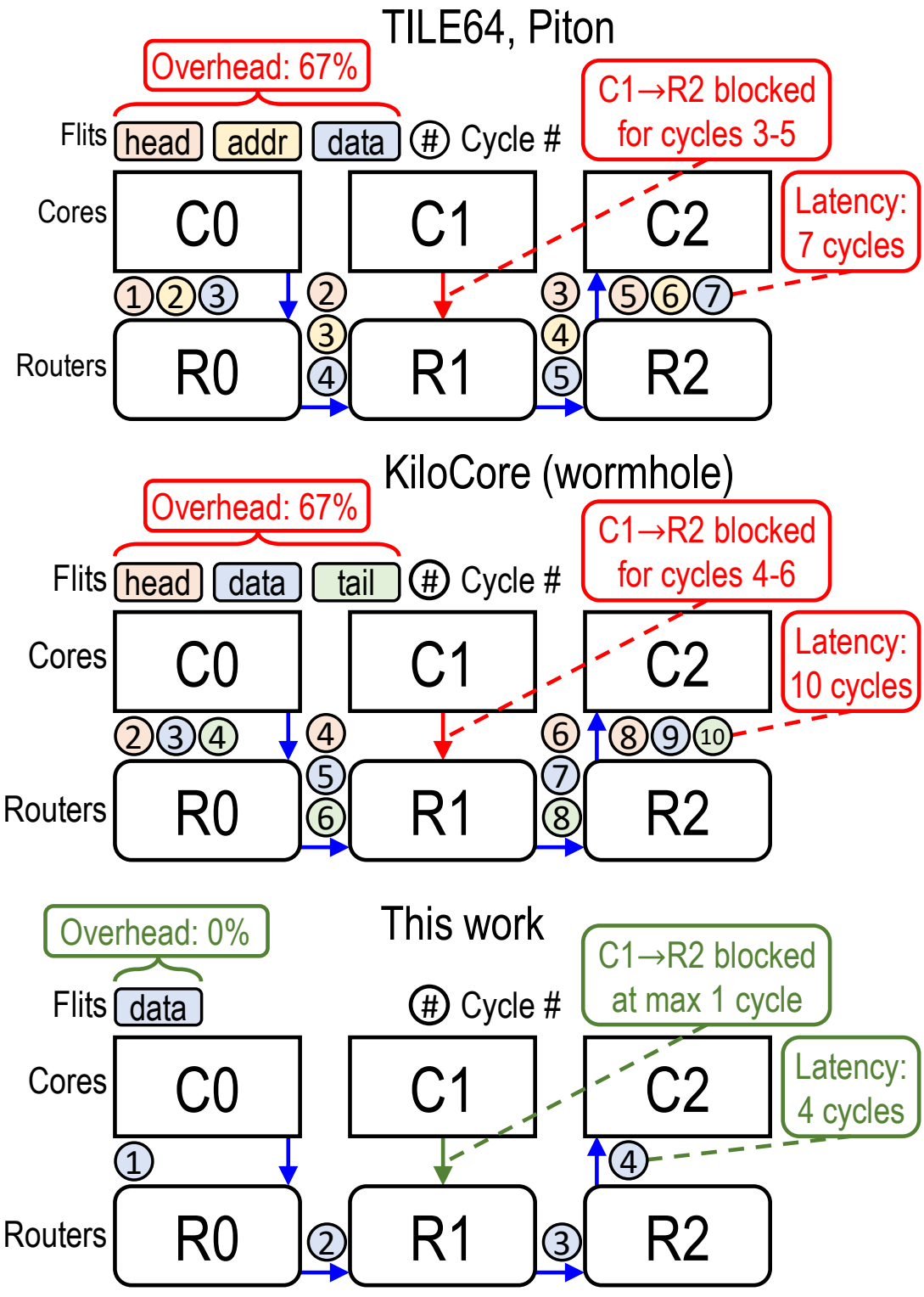

Figure 2.11: An example of sending a packet with one data flit from Core 0 to Core 2 for each flow control model

2.4.4.1 Experimental Results

To validate the manycore processor, Celerity runs CoreMark distributed across all cores simultaneously. CoreMark is structured as self-validating benchmark: each iteration depends on the previous iteration and a hash of the final state is used to check correctness. Figure 2.12 identifies the operating configurations where CoreMark reports a correct result for all tiles. The processor achieves a max throughput of 695 GRVIS at 1.4GHz and 0.98V – the highest single-chip RISC-V throughput to date – and a max energy efficiency of 314.89 GRVIS/W at 500MHz and 0.60V. It achieves a record CoreMark score of 825,320, outperforming the next best score by more than 2×, as well as my team's previously reported score [46] by a small margin. The evaluation uses GRVIS as a measure of performance because it signifies compliance with the RISC-V ISA. A custom ISA can increase efficiency by tailoring instructions, but this extricates the architecture from the benefits of open-source software and toolchains. In the comparison, non-RISC-V performance is quantified with giga-operations per second (GOPS). For direct comparisons, GRVIS are GOPS, but GOPS are not GRVIS. Table 2.6 compares Celerity against prior manycore works. In most metrics, this work compares very favorably against related works. Celerity exceeds all compared works for normalized NoC area (2.5×-44×), area efficiency (1.8×-160×), and energy efficiency (4.2×-37×). Throughput measurements are normalized to 32-bit operations, under the optimistic assumption that two 16-bit operations are equivalent to one 32-bit operation.

KiloCore [14] modestly exceeds this work in network aggregate and bisection bandwidth, although a majority of its bandwidth comes from the statically-routed, circuit-switched network. KiloCore also uses only 1.1KB memory per tile, whereas Celerity uses 8KB per tile. In terms of RISC-V performance, Lee et al. [36] report state-of-the-art in single-chip GRVIS throughput, which Celerity outperforms by 267×.

2.5 Design Methodologies

Celerity was designed under the DARPA Circuit Realization at Faster Timescales (CRAFT) program, whose goal was to reduce the design time for taping out complex SoCs. My team designed and taped out Celerity in just nine months from process design kit (PDK) access, which included:

- Coordinating graduate students spread across four universities
- Developing an implementation flow for an advanced 16nm FinFET node

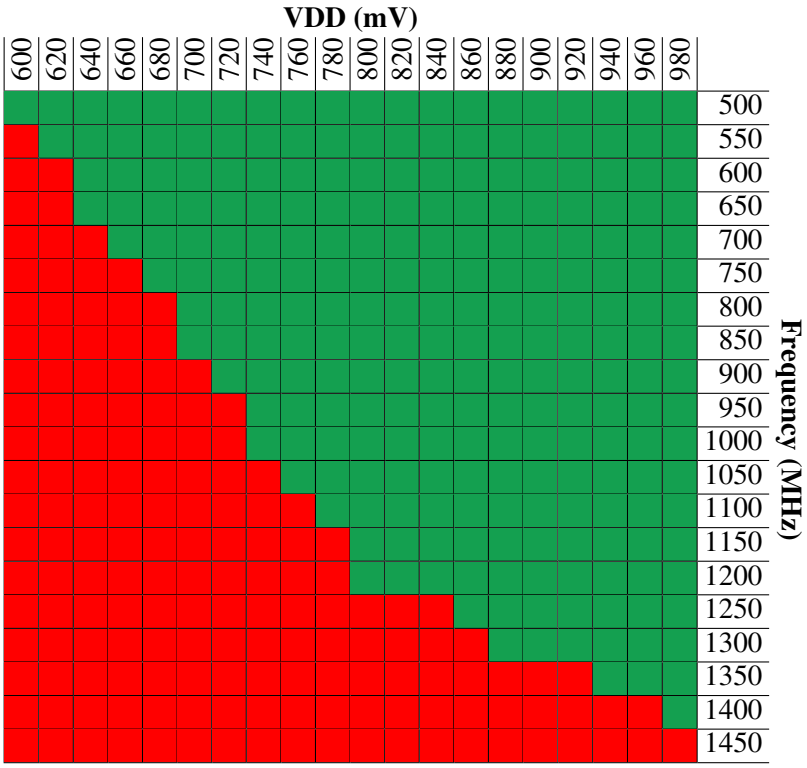

Figure 2.12: Shmoo plot of operation points

	ISSCC '08[13]	HPCA '18[40]	JSSC '17[14]	ESSCIRC '14[36]	Celerity
ISA	VLIW	SPARC V9	RISC	RISC-V	RISC-V
Datapath Width	32-bit	64-bit	16-bit	64-bit	32-bit
Technology	90nm Planar	32nm SOI	32nm SOI	45nm SOI	16nm FinFET
Voltage	0.90 - 1.30 V	0.80 - 1.20 V	0.67 - 1.10 V	0.65 - 1.20 V	0.60 – 0.98 V
Area[a]	232.16 mm^2	29.37 mm^2	57.41 mm^2	3.08 mm^2	15.25 (12.03[c]) mm^2
Normalized Area[ab]	32.65 mm^2	14.08 mm^2	27.52 mm^2	0.69 mm^2	15.25 (12.03[c]) mm^2
Normalized NoC Router Area[ab]	~82894 μm^2 (5x32 bit)	16214 μm^2 (3x64 bit)	4784 μm^2 (16 + 2x16 bit)	–	**1881 μm^2** (80 + 9 bit)
Cores (Threads)	64 (64)	25 (50)	1000 (1000)[d]	2 (2)	496 (496)
Frequency	750 MHz	500 MHz	1770 MHz	200 - 1300 MHz	10 - 1400 MHz
Power	10.8 W	2 W	39.6 W[d]	0.96 W	7.47 W
Norm. Throughput[e]	144 GOPS	5 GOPS	885 GOPS	5.2 GRVIS	695 GRVIS
Network Aggregate Bandwidth[f]	33.79 Tb/s	11.33 Tb/s	53.4 Tb/s (wormhole) 335 Tb/s (circuit)	–	361 Tb/s
Network Bisection Bandwidth[g]	1.92 Tb/s	0.96 Tb/s	0.58 Tb/s (wormhole) 3.65 Tb/s (circuit)	–	4.00 Tb/s
Routing Model	Wormhole	Wormhole	Wormhole+circuit	–	Single-flit packet
Energy Efficiency[e]	13.33 GOPS/W	2.50 GOPS/W	22.35 GOPS/W[d]	5.42 GRVIS/W	**93.04 GRVIS/W**
Normalized Area Efficiency[abe]	4.41 GOPS/mm^2	0.36 GOPS/mm^2	32.16 GOPS/mm^2	7.54 GRVIS/mm^2	**45.57 (57.77[c])** GRVIS/mm^2

[a] Area only includes die area allocated to tiles
[b] Area normalized to 16nm based on Contacted Poly Pitch (CPP) scaling
[c] Excluding ESD and ICOVL area
[d] KiloCore can only power 160 cores from its package. Power extrapolated to 1000 cores
[e] Throughput normalized to 32-bit GOPS/GRVIS
[f] Network Aggregate Bandwidth = (# usable links) * (link bandwidth)
[g] Network Bisection Bandwidth = (min. # links cut to bisect network) * (link bandwidth)

Table 2.6: Comparison to related works

- Satisfying the CRAFT program constraints with only $1.3 million USD for non-recurring engineering costs

To meet the aggressive schedule for Celerity, my team developed three classes of techniques to decrease design time and cost: reuse, modularization, and automation.

2.5.1 Reuse

Reuse for hardware design accelerates both design and implementation time, as well as testing and verification time. For Celerity, my team made heavy reuse of open-source designs and infrastructures. We leveraged the Berkeley RISC-V Rocket core generator [10] to implement the SoC's general-purpose tier, allowing the reuse of Rocket's testing infrastructure and the RISC-V toolchain. The same infrastructure was used for the manycore array's Vanilla-5 core. Because validation is usually more work than design, inheriting a robust test infrastructure greatly reduced overall design time. We leveraged the RoCC interface for all connections to the general-purpose tier. As part of our learning process with RoCC, my team created the "RoCC Doc," located at `http://opencelerity.org`.

Beyond the RISC-V ecosystem, we leveraged the BaseJump open-source hardware components, which can be found at `http://bjump.org`. BaseJump provides open-source infrastructure and frameworks for designing and building SoCs, including the BaseJump STL [52] for SystemVerilog, the BaseJump SoC framework, BaseJump Socket, BaseJump Motherboard, BaseJump FPGA bridge, and BaseJump FMC bridge, as seen in Figure 2.13. In Celerity, we built all of the RTL using the Basejump STL and SoC framework's pre-validated components and unit testing suite. We ported the BaseJump Socket to the CRAFT flip-chip package and will use the BaseJump Motherboard for the final chip.

By leveraging the unit testing suite from BaseJump and RISC-V testing infrastructure, we could focus our verification efforts primarily on integration testing. Using an FPGA in place of the SoC, the BaseJump infrastructure allows for designs to be simulated in the same two board environment they will be running in post-tapeout. All firmware and testbench code written during simulation will be reused during bring-up once the chip returns from fabrication, giving us a robust verification and validation suite. Reuse is also enabled by extensibility and parameterization. Due to the scalable nature of tiled architectures, BaseJump STL's parameterization, and the flexibility of our backend flow methodology, we were able to extend the BaseJump manycore array from 400 cores to 496 to absorb free die area. By changing just nine lines of code, we could fully synthesize, place, route, and sign off on the new design in a span of three days.

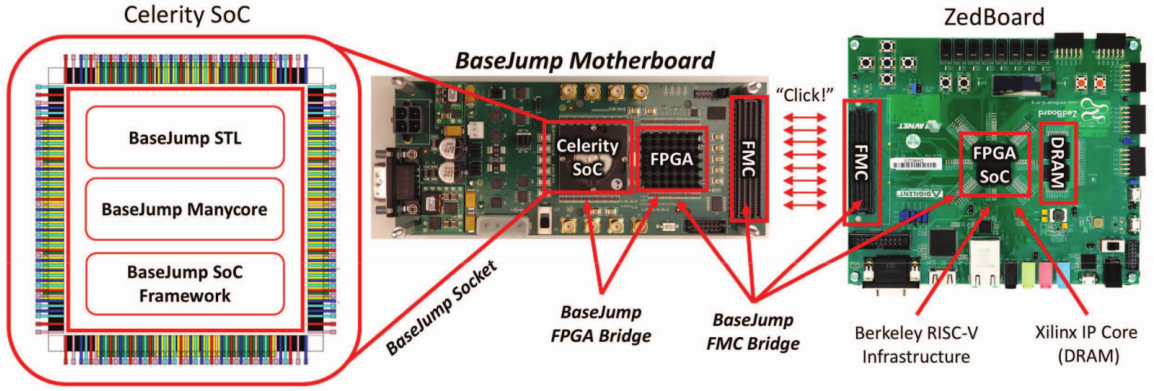

Figure 2.13: BaseJump open-source hardware components. The NoC, manycore, and high-speed off-chip interface were implemented using STL. The fabricated chip conforms to the socket definition and is placed in the motherboard's socket. The motherboard connects through an FMC connector to a ZedBoard hosting RISC-V testing infrastructure. Communication between the motherboard and ZedBoard is handled with the open-source FMC bridge code.

2.5.2 Modularization

One key challenge for this project was managing design teams spread across four physical locations. Fine-grained synchronization between teams was not feasible, so we developed techniques to modularize both our chip design interfaces and our interfaces between teams.

Many techniques we used can be compared to an agile design methodology as it applies to hardware. We used a bottom-up design flow to build, iterate, and integrate smaller components into a larger design. We also used a SCRUM-like task management system, where we clearly identified and prioritized various tasks and issues, minimized synchronization issues, and distributed tasks across team members without assigning rigid specialized roles.

We also defined tape-in [37] deadlines. These are simpler designs that were tapeout ready before the deadline. This allowed us to stress-test our physical design flow early in the design cycle, in addition to identifying big-picture problems early on, which we found particularly useful when dealing with an advanced technology node. Each successive tape-in incorporated an additional IP block, building up to what we see in Celerity. We performed daily chip builds to ensure no changes broke the overall design and that we always had a working design to tapeout.

To help modularize the RTL, chip component interfaces were established early. We selected RoCC early on for on-chip communication and BaseJump for off-chip communication. Because we used BaseJump STL's pervasive latency-insensitive interfaces, our architecture-specific dependencies between components were minimized.

2.5.3 Automation

CRAFT's tight time constraints required that we employ higher degrees of automation to accelerate the design cycle. We developed an abstracted implementation flow to minimize the changes necessary for different designs to go from synthesis through sign-off. We combined vendor reference scripts with an integration layer to coalesce implementation parameters and separate scripts into design-specific and process-specific groups. We could then quickly identify which scripts needed to be modified between designs.

We also took advantage of emerging tools and methodologies. We used the PyMTL framework for rapid test-bench development using high-level languages and abstractions rather than lowlevel SystemVerilog. In our BNN accelerator development, we used HLS to drastically improve design space exploration and implementation time.

2.6 Conclusions

This research examined the speedy construction of new classes of chips in response to emerging application domains. The approach was successful due to a heterogeneous architecture that offers fast construction, scalability, and heterogeneous interoperability through the remote store programming model and advanced producer-consumer synchronization methods like LR-LBR and token queues. At the same time, the design methodology combines HLS for specialized tier accelerator development, open-source technology like Rocket and BaseJump for key IP blocks, fast motherboard and socket development and FPGA firmware, and principled SystemVerilog parameterized component libraries like BaseJump Standard Template Library (STL). Finally, agile chip development techniques enabled my team of geographically distributed grad students to quickly tape out a 16nm design. Each approach targets the key goal of creating new classes of chips quickly and with low budgets.

2.7 Acknowledgements

I would like to personally thank all of my collaborators who made this ambitious project possible. I would like to especially thank Tutu Ajayi and Aporva Amarnath who shared the physical design tasks with me.

This research was supported in part by DARPA CRAFT Award HR0011-16-C-0037 and NSF CRI Award #1059333, NSF CRI Award #1512937, NSF SaTC Award #1563767, NSF SaTC Award #1565446, and NSF XPS Award #1337240. I thank Synopsys, Cadence,

and Mentor Graphics (now Siemens EDA) for EDA tool donations, ARM for physical IP donations, and Xilinx for both EDA tool and FPGA development board donations. I acknowledge the University of California, Berkeley's contributions to forming the RISC-V ecosystem, for RoCC, and for developing the Rocket chip SoC generator. The bringup effort was partly funded by the DARPA/SRC JUMP ADA center.

CHAPTER 3

OpenROAD

Several academic EDA tools have been released; however, few are used in real tapeouts, even by other academics. Robust open-source tools require feedback and direction from users. To this end, OpenROAD employs end-users as internal design advisors who bring with them the experience of multiple tapeouts and EDA tool flow development. This chapter discusses the OpenROAD my work as a design advisor to bring OpenROAD from a collection of tools to an end-to-end autonomous design flow. I discuss the design advisors work to fill in the gaps for a full RTL-to-GDS design flow, assemble a full-flow test suite reflective of real tapeouts, debug flow-level issues between tools, and bridge the gap between OpenROAD developers and others in the open-source community. Lastly, I discuss OpenROAD's long-term goal to become fully autonomous, and what that means from a user's perspective.

3.1 Introduction

Access to high-quality electronic design automation (EDA) tools, required engineering expertise, and lengthy project schedules have long been a barrier to hardware startups and hobbyists. Restrictive licensing has also been a massive impediment to the academic community due to virtually all licenses prohibiting sharing of scripts or results. With this motivation in mind, The OpenROAD Project aims to create a fully autonomous, open-source tool chain for full "RTL-to-GDS" digital layout generation. With such a tool, numerous issues can be addressed, including engineering resources, licensing, collaboration, and reproducibility [8, 9, 29].

A critical aspect of creating usable open-source software is to receive feedback and direction from users. As such, OpenROAD employs a group of experienced digital SoC designers as *internal design advisors*. Our job is to act as the first users of the OpenROAD tools and form a quick feedback loop with developers in order to iterate software quickly.

The original intent of the internal design advisor was to use real-world designs for tool testing and feedback, as well as provide human intelligence to guide tool development (similar to application or product engineers). Over the course of the project, however, it became apparent that key responsibilities for realization of OpenROAD goals [8, 9] did not fit cleanly into the project's organization structure. As such, the design advisor role has evolved to encompass several additional tasks which can be categorized under two responsibilities:

Flow Development. Even with well-defined OpenROAD tool interfaces, orchestrating a full RTL-to-GDS flow is a non-trivial task. Since creating the previous iterations of OpenROAD-flow [8, 9], the advisors have substantially overhauled the flow to transition from a tool chain of stand-alone binaries to an integrated app. In addition, we are responsible for interjecting flow-level solutions which increase autonomy and reduce burden on developers. Such solutions act as initial scaffolding to improve autonomy from a user's standpoint and allow developers to focus on critical tool features.

Test Infrastructure. The integration of OpenROAD tools into a single app highlighted the project's need for continuous integration (CI) infrastructure (as noted by Kahng [29]). The OpenROAD Project operates in a delicate situation of testing with proprietary commercial data but developing with public infrastructure (e.g. GitHub). In addition, CI infrastructure requires maintaining good unit and integration tests for code coverage and metric tracking. We have set up a Jenkins CI infrastructure to balance these demands and create a secure but productive CI flow for the tool developers.

In this work, I discuss the background of The OpenROAD Project (Section 3.2), and the main responsibilities of the internal design advisors (Sections 3.3 & 3.4) in order to bring OpenROAD from a collection of tools to a full RTL-to-GDS flow. Next, I discuss OpenROAD's long-term goals of autonomy and quality, and the roadmap I want from a user's perspective (Sections 3.5 & 3.6). I conclude with our key lessons learned and goals for OpenROAD (Section 3.7).

3.2 Background

The OpenROAD Project was launched in June 2018 within the DARPA IDEA program to create an open-source, fully autonomous RTL-to-GDS flow. The RTL-to-GDS process implements a register-transfer-level (RTL) description of a circuit into the Graphic Design System (GDS) format, representing a mask layout which fabrication facilities use to manufacture chips. This implementation process is broken down into several sequential steps, generally referred to as a design implementation flow. Flows differ among designers for

a variety of reasons, but the most basic flows include synthesis, floorplanning, placement, clock tree synthesis (CTS), optimization, and routing. Verification is also important to verify that the design is manufacturable and free from critical bugs.

"OpenROAD" or "the OpenROAD app" is a collection of physical design tools which can take a Verilog netlist and perform placement and routing (PnR) to output a physical design in the Design Exchange Format (DEF). The OpenROAD app only covers the PnR portion of the flow (floorplanning to routing[1]). The other important steps – synthesis, DEF to GDS conversion, and verification – are performed using third-party open-source tools (Yosys [58] and KLayout [33]). To fulfill OpenROAD's RTL-to-GDS commitment, "OpenROAD-flow" acts as a wrapper around these tools and forms a full design implementation flow. These tools are all available from github.com/The-OpenROAD-Project.

This past year, The OpenROAD Project has spent significant effort on support for a commercial 14nm platform. With this milestone coming to a close, OpenROAD is entering the second phase of the project and ready to focus on additional directions.

3.3 Flow Development

3.3.1 Original Intent

The initial task of the internal design advisors was to test the OpenROAD tools with previously taped-out designs and provide rapid feedback throughout the development process. It quickly became apparent that a flow would be required to achieve this task; however, flow development was not clearly defined at the launch of The OpenROAD Project.

OpenROAD started as a collection of separate tool binaries accomplishing various steps of a design flow. The tools varied in maturity and some tools/steps were not initially available. In addition, the interfaces and expectations between tools were not concretely defined. To address this, we needed a flow able to pass designs through all available steps provided by the development team. This included the ability to skip or work around flow steps as necessary. Therefore, the earliest iterations of our flow leveraged commercial tools to generate "clean" artifacts (DEFs, floorplans, guides, etc.) needed to exercise each tool/step. The initial iteration of the flow chained steps together using GNU Make, and this flow has continued to evolve with the maturing tools.

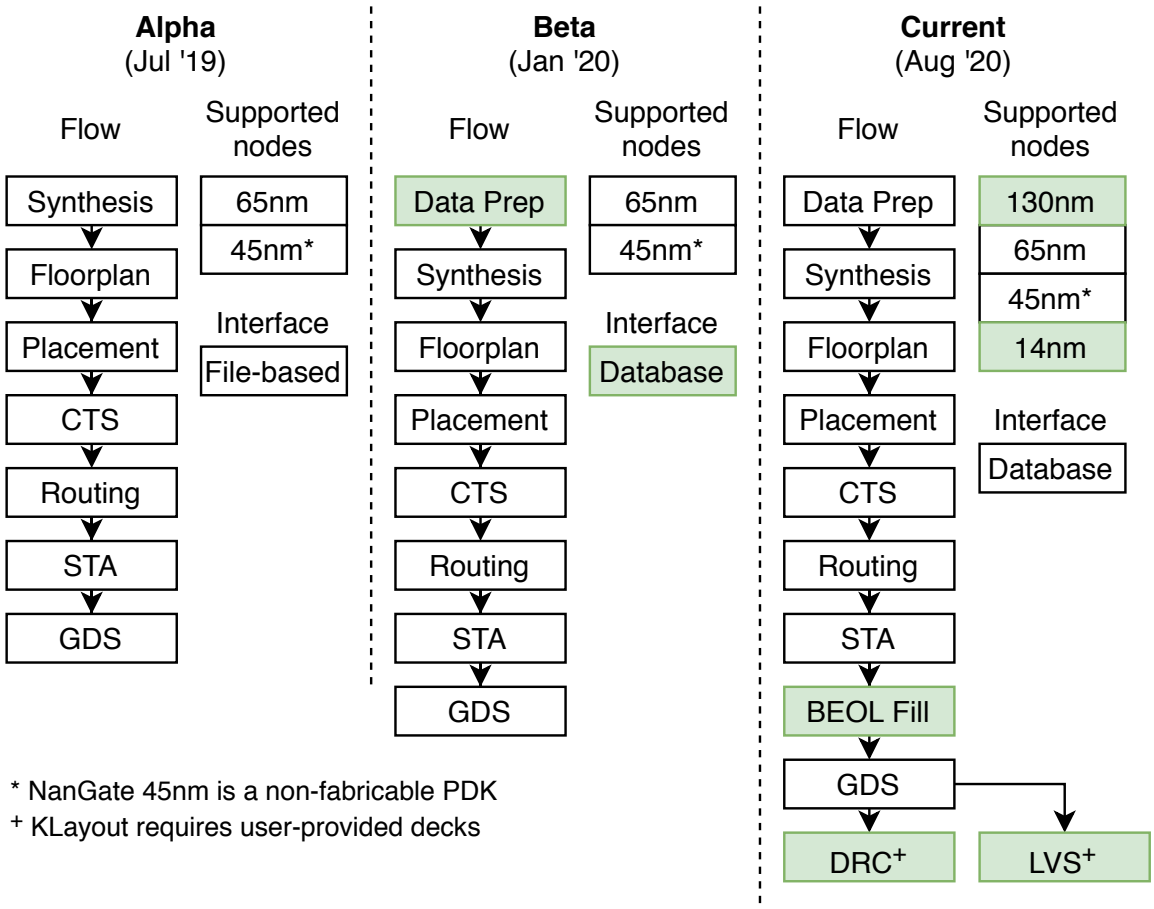

Figure 3.1: Evolution of OpenROAD-flow

3.3.2 Responsibilities

Maintenance of the OpenROAD-flow repository is now the largest and foremost responsibility of the design advisors. While the OpenROAD app offers a collection of PnR tools, there remains a huge gap between the OpenROAD app and a full RTL-to-GDS flow. OpenROAD-flow provides wrappers around Yosys, the OpenROAD app, and KLayout to fulfill this role. Yosys provides synthesis from Verilog RTL to netlist, OpenROAD provides PnR from netlist to DEF, and KLayout provides DEF to GDS conversion as well as design rule checking (DRC) and layout-versus-schematic (LVS) checking.

Figure 3.1 shows the evolution of OpenROAD-flow from its initial implementation [8, 9] to the current iteration. The largest improvement to OpenROAD-flow was the shift from a file-based interface to a unified database interface. The alpha release of OpenROAD used separate binaries for each tool and relied on a patchwork of configuration files and command-line arguments to run each tool. Once the tools were integrated into a unified app[1], all interfaces were also unified into a single binary with a Tcl interface. OpenROAD-flow has expanded its support to include 14nm FinFET and the newly open-sourced SkyWater 130nm platform. In addition, open-source DRC and LVS are available through KLayout; however, very few process development kits (PDKs) have KLayout rule decks available. Community-sourced decks are available for NanGate45 and are expected for SkyWater130, but the outlook for other commercial PDKs remains dim.

In addition to adding more flow stages as shown in Figure 3.1, another key responsibility is simplifying the user's flow interactions. To this end, the important data preparation step aims to minimize the number of user adjustments needed to set up a commercial PDK, such that it is usable by OpenROAD-flow. Optimization, while not explicitly shown, is embedded in the placement and CTS stages.

3.3.3 Challenges

Developing and maintaining OpenROAD-flow has led to several challenges along the way to meeting OpenROAD's development schedule goals. In this section, I discuss some of the most difficult aspects we faced.

3.3.3.1 Tool Synchronization

One of the main challenges for flow maintenance has been synchronization. OpenROAD-flow acts as a wrapper on top of OpenROAD and other tools. Therefore, whenever the

[1] TritonRoute is a separate binary at time of writing, but integration is expected soon.

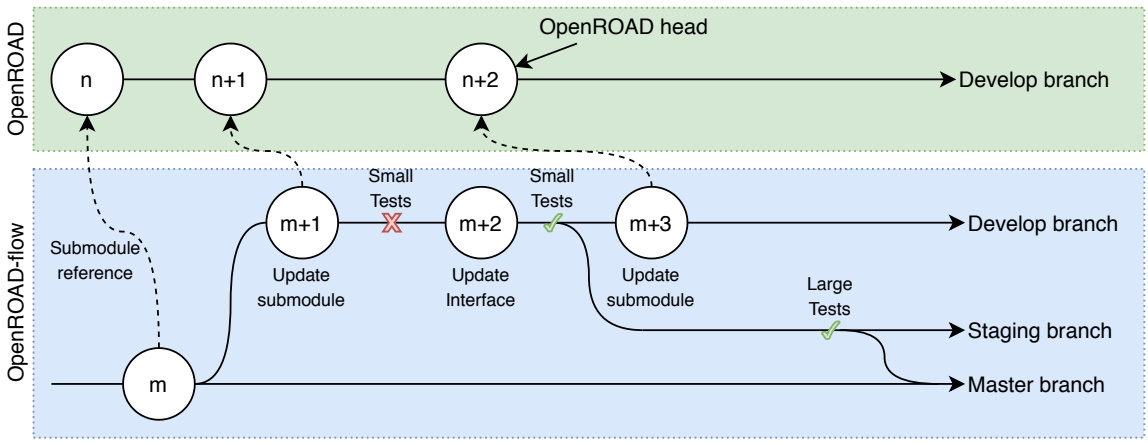

Figure 3.2: OpenROAD-flow submodule branching methodology

underlying tools change, OpenROAD-flow must change as well. This problem bears resemblance to software libraries and packaging. However, OpenROAD developers frequently update the application programming interface (API) or introduce new features, and they look for feedback within on the order of hours or days. Therefore, synchronization occurs at a scale that is too fine-grain to rely on software packaging.

Our approach to this problem relies on asynchronous, regular updates to OpenROAD-flow. The current methodology is shown in Figure 3.2. OpenROAD-flow maintains a reference to a specific OpenROAD commit (and other tools) via git submodules, which allows OpenROAD-flow to maintain API synchronization with OpenROAD. When updates are committed to OpenROAD, we asynchronously update OpenROAD-flow's submodule reference, perform API updates, and merge the changes into the master branch. This model differs from the one described by Kahng [29] in a few regards. In particular, OpenROAD-flow separates tests into small pipe-cleaning tests and large quality of result (QoR) tests across separate Jenkins pipelines triggered by separate branches. The main reason for this strategy is to reduce build server load: if a large QoR test is triggered from every commit, the build server can quickly become overloaded. We instead use automatically triggered small tests for every commit to the development branch, and then large tests only on merges to the "staging" branch. Merges to the staging branch occur at a regular interval (i.e., nightly), and merges to the master branch are triggered by a successful QoR test.

3.3.3.2 Tool Workarounds

In cases where the root issue resided between tools, the design advisors would often be able to perform external processing as a workaround for lack of tool support. This allowed developers to focus on more critical issues. Some previous examples include fixing incorrect

or inconsistent DEF outputs between tools, fixing technology-dependent tool issues, and adding partial support for yet-to-be-supported file formats (e.g., interconnect parasitics).

Adding workarounds remain a balancing act, however, as they often add technical debt which burdens future development. Several of these workarounds have enabled the development team to hit our schedule for 14nm node support, but now underlying issues must be thoroughly investigated to reduce technical debt.

3.3.3.3 Parameter Tuning

Virtually all EDA tools require human input to identify design intent and constraints. While OpenROAD's long-term goal is to automate much of this process, OpenROAD developers today need reasonably selected inputs in order to debug algorithms and identify smaller-scale problems. The design advisors have often provided the human intelligence to select and tune design parameters, such as design area and utilization, cell placement padding, and global routing settings. Parameter tuning is a normal part of designing with EDA tools and is quite straightforward for experienced designers. The more challenging aspect of parameter tuning is trying to identify tool pitfalls to the developers. For example, if decreasing global placement density and reducing global routing per-layer resource allocations do not result in reduced routing violations, then the issue may be that the detailed placer is causing pin access issues for the detailed router. As design advisors, we have to be familiar with the entire flow and process rules in order to narrow down issues appropriately.

In addition, the challenge becomes more difficult when applying this tuning across platforms. The design advisors are expected to maintain good configurations for each design across all supported nodes - from 130nm down to 14nm. The diverse routing rules, cell libraries, metal stacks, etc. across supported nodes dictate separate parameters across platforms, in addition to tuning designs.

3.3.3.4 Generic Node Enablement

Detailed routing for FinFET technology nodes is a significant challenge for academic research. A routing tool must understand all of the hundreds of complex design rules in order to properly route designs without design rule violations (DRVs). To the best of our knowledge, no academic detailed router other than OpenROAD's TritonRoute [6] supports FinFET (sub-20nm) nodes. Even among commercial tools, few commercial routers successfully route FinFET-node designs without DRVs.

As internal design advisors, helping TritonRoute to achieve zero DRVs with limited time and resources was a significant challenge. Following the "generic node enablement"

methodology [29], we made and enforced assumptions in routing rules to minimize the required design rule support:

- **Unidirectional and on-track routing.** Only on-track routing in the preferred direction is allowed. Bidirectional routing triggers complex design rules, such as spacing to convex or concave corners and color-aware design rules for multiple patterning technology layers.
- **Minimum-width routing.** Non-default routing (NDR) is not allowed. Wide metals trigger width-aware spacing rules, via enclosure rules and minimum numbers of vias.
- **On-track pins for macro cells.** Macro cells, such as SRAMs and other IPs, often have non-uniform pin widths and shapes. By creating "wrapped LEF" views with on-track, minimum-width pins, we can provide a view with simplified pin access without violating the two previous assumptions.
- **Routing-friendly P/G distribution.** Minimum-width routing is generally not an option for the power delivery network (PDN). In order to avoid forcing the router to consider NDRs, OpenROAD's PDN generator adds routing blockages and uses on-track, minimum-width stacked vias between power stripes. This allows the router to perform clean routing without any additional rule support.

3.4 Test Infrastructure

3.4.1 Design Suite

3.4.1.1 Original Intent

A critical aspect of making OpenROAD a usable tool is ensuring it is tested on real-world design data. The original intent of the design advisors was to curate a suite of test cases based on previous tapeouts of real designs. We quickly ran into several issues that prevented us from creating meaningful test cases for the developers.

Candidate designs that we identified for test cases proved too complex, and they overwhelmed the early OpenROAD tool capabilities. Our initial test cases had many advanced features not yet supported by OpenROAD, such as dense floorplans, multiple power domains, multiple clocks, etc. Additionally, some designs relied on circuit-level techniques which were incompatible with OpenROAD's all-digital design flow, or contained proprietary IP which prevented transmission to the developers.

To address this, we stripped down the test cases to accommodate the tool capabilities, project milestones and specific features we needed to evaluate. As the tools matured, we progressively tightened design constraints and added more test cases from both the design advisors and the open-source community.

Design	Description	Source	Instance count	Macro count	RTL taped out?
gcd	Greatest common denominator	Authors	250	0	Yes
dynamic_node	2D mesh router	[11]	8,090	0	Yes
vanilla5	Vanilla-5 CPU core	[18, 46]	12,300	4	Yes
ibex	Ibex CPU core	[4]	14,600	0	Yes
aes	AES encryption	[2]	15,000	0	Yes
bp_fe_top	Black Parrot CPU front-end	[43]	24,400	11	Yes
tinyRocket	Rocket generator CPU	[10]	25,100	2	No*
bp_be_top	Black Parrot CPU back-end	[43]	39,400	10	Yes
jpeg	JPEG encoder	[1]	52,600	0	Unsure
swerv	SweRV EH1 CPU (core only)	[3]	82,300	0	Yes
swerv_wrapper	SweRV EH1 CPU (with macros)	[3]	83,900	28	Yes
black_parrot	Black Parrot CPU core	[43]	121,000	24	Yes
ariane	Ariane CPU core	[59]	150,000	37	Yes
coyote	Coyote CPU core	[18, 46]	222,000	15	Yes
bp_multi_top	Black Parrot quad-core CPU	[43]	852,000	196	Yes

* several chips have been taped out using the Rocket generator, but tinyRocket RTL has not

Table 3.1: OpenROAD-flow design suite. Instance counts are collected post-synthesis from Yosys with a commercial cell library.

3.4.1.2 Responsibilities

OpenROAD-flow now maintains a suite of designs containing source RTL and multi-platform constraints. The suite is a combination of open-source designs provided by both the design advisors and the community (summarized in Table 3.1). This suite provides several properties critical to maintaining an open-source flow:

- **Diverse** - The designs range from a few hundred instances to over 400k instances. Small designs allow developers to pipe-clean and debug tools quickly. In addition, users can run small designs quickly to validate their tool and flow setup. Large designs provide more complex developer test cases as well as benchmarks for QoR.
- **SoC-level** - OpenROAD-flow provides full-chip designs with I/O rings, enabling developers to test tools at the SoC level rather than only at the block level.[2]

[2] A lack of open-source I/O cells limits what can be distributed publicly. The SkyWater130 platform plans on releasing I/O cells, and we expect to incorporate them shortly thereafter.

- **Compact** - OpenROAD-flow provides a variety of test cases, but not so many that full regression testing becomes infeasible. The included designs are curated to be representative of a spectrum of real-world designs.
- **Real** - Almost all of the included designs have been taped out and are representative of real-world designs.
- **Cross-platform** - All designs are platform-independent and can be ported across processes. Macros must be regenerated for each platform, but the interfaces are designed to require no source changes.

In addition to design sources, we maintain flow configuration parameters for each design, including SDC and OpenROAD parameters. As the tools change, changing design parameters may be warranted. For example, placement density may increase over time as OpenROAD becomes more capable and can realize the benefits from shorter inter-cell distances without incurring DRVs.

3.4.1.3 Challenges

The main challenge to maintaining the design suite is likely tool compatibility. Many of our designs' sources are in SystemVerilog, which is only partially supported by Yosys. Our solution was to automatically convert the source to Verilog and maintain the generated source in the repository. This solution unfortunately loses the semantics and readability of the original designs, but it satisfied our use case of obtaining a test design. Many new open-source efforts in SystemVerilog parsing and conversion have arisen since the start of the OpenROAD Project, and our existing approach may be revisited as these efforts mature.

3.4.2 Continuous Integration

3.4.2.1 Original Intent

Continuous integration was another area which was not clearly enumerated in the original project task structure. Writing tests is often considered a responsibility of the programmers who write the code, and it was believed that regression testing could be handled by individual developers. Such a structure was sensible at the beginning of the project due to all of the tools being self-contained programs. However, several issues limited the scalability of this approach:

- **Issue reporting** - Even with developers being responsible for unit tests and design advisors for integration tests, problems are reported for the tool which fails and not

necessarily the tool which creates the problem. This behavior can create a "hot-potato" issue where developers have to pass the issue around to figure out the root cause.

- **Delayed feedback** - Lag time between introducing a change and receiving design advisor feedback inherently limits the rate at which developers can commit stable updates.
- **Portability** - Some code changes may only be stable in the developer's environment, where the code was tested, and unstable elsewhere.

With these problems, it became clear that investment in a continuous integration solution would be required to move forward. With CI for both unit tests and integration tests, developers can immediately pin down which code change broke the full-flow tests.

3.4.2.2 Responsibilities

The design advisors are partially responsible for maintaining the Jenkins CI infrastructure for the whole project, and wholly responsible for maintaining regression tests for the OpenROAD-flow repository. As mentioned in Section 3.3.3.1, we update the submodule reference to OpenROAD regularly and perform full-flow regression tests on the tools. Updates which pass all regression tests (which is currently every design in OpenROAD-flow's design suite) will be pushed to the master branch and be ready for use by the community.

3.4.2.3 Challenges

OpenROAD's CI has faced two main challenges.

Test scale. EDA tools are renowned for consuming significant computation time, and OpenROAD is no different. The largest designs in the OpenROAD-flow design suite can take more than 12 hours to run, meaning that running the full test suite on every push is not feasible. Instead, we adopt the approach mentioned in Section 3.3.3.1. A subset of test cases are triggered on every commit to the development branch, and the full test suite is only run nightly. The small tests are curated to run in a small amount of time, fail quickly if a change completely breaks the flow, and include designs both with and without macros. Our small test currently completes in under 30 minutes and includes gcd, aes, and tinyRocket. Our large test includes the full design suite and takes approximately 12 hours when fully parallelized.

Private tests. Using real, commercial platforms is critical to testing tool correctness. However, nearly all commercial platforms are regarded as trade secret and require significant security to ensure their privacy. Therefore, neither typical open-source CI practices nor closed-source CI practices fit our testing requirements. To maintain security but still incorporate open development, the private CI server can only test protected branches and only reports details back to trusted developers.

3.5 Towards Full Autonomy

While The OpenROAD Project has made long strides toward more automation, many complex challenges remain. Our main goals are to (1) focus on improving the user experience in the short term, and (2) focus on improving autonomy in the long term.

3.5.1 Improving User Experience

Achieving full autonomy is no small feat and we expect it to take a significant amount of time. In the short term, we aim to improve the user experience so that providing human input is more intuitive and tool issue resolution is less cumbersome. The following subsections detail key milestones we want to see from The OpenROAD Project as designers.

3.5.1.1 Improved Documentation

Significant developer effort was invested in achieving a tapeout-ready design in a 14nm FinFET node. After completing this goal, we propose providing several resources that users have come to expect from commercial tools:

- Documentation on all required OpenROAD-flow input files/ parameters, and instructions on how to generate/select them.
- Tutorials for setting up new designs and new platforms.
- Documentation, uniformity, and adjustable filtering for all tool messages (info, warning, error, and critical).
- Generated documentation for OpenROAD code and APIs.

Common feedback from community members indicates that OpenROAD-flow's biggest hurdle is determining the source of errors – whether from user parameters, user designs, or tool bugs. We believe the points above, particularly the improvement of tool messages, will enhance users' abilities to resolve issues themselves.

3.5.1.2 Improved Access

Open software installation needs to be easy, fast, and widely available. Currently, OpenROAD's only officially supported OS is CentOS 7. We advocate adding official support to more operating systems, including CentOS 8 and Ubuntu 18, by testing builds in our CI system. All operating systems which support OpenROAD, KLayout, and Yosys dependencies should be able to build and run OpenROAD-flow, even without official support.

We also advocate making software packages available in the long term. The main difficulty with packaging is that OpenROAD-flow directly depends on OpenROAD's frequently changing API, as mentioned in Section 3.3.3.1. The current solution of git submodules only works well when building from source. Versioning also becomes an issue, as matching a version of OpenROAD-flow with the corresponding version of OpenROAD would be cumbersome and unintuitive. This is currently an open problem, but we believe packaging is important to simplifying access to OpenROAD.

3.5.2 Reducing Manual Effort

The goal of full autonomy is reducing the number of required user inputs. OpenROAD-flow is currently in-line with commercial workflows in terms of manually specifying parameters and fine-tuning a design with human guidance. To reach full autonomy, these human inputs will need to be replaced with machine intelligence.

OpenROAD-flow currently requires about 50 configuration parameters to be set per-process, not including the PDN, I/O, or design-specific configurations. For an inexperienced user, setting these parameters correctly can be difficult. We propose aggressively reducing the number of required manual parameters by replacing them with ones automatically extracted from platform data. This will allow OpenROAD-flow to reduce the level of experience needed to set up new platforms, while still allowing expert users to manually tune parameters if desired.

One such example parameter is the target design utilization (cell density). Setting this parameter can be a difficult task: a utilization target that is too high will blow up tool runtime and potentially provide an unroutable design; on the other hand, a utilization that is too low can turn competitive power-performance-area (PPA) results into non-competitive. The maximum utilization often correlates most strongly to the process, but can also be influenced by the design. To automate, OpenROAD will need heuristics to select a utilization which balances runtime with PPA based on process and design characteristics.

3.6 Towards Improved Results

In contrast to many commercial tools, The OpenROAD Project sees autonomy as a primary constraint and QoR as secondary. This means that OpenROAD will focus on providing a clean and manufacturable design, without human intervention, over improvements to PPA. However, improving QoR is not always orthogonal to autonomy, as improved algorithms can lead to reductions in design rule violations. Improving QoR is also important for increasing the user base, as low QoR can lead users towards closed-source proprietary tools.

3.6.1 Inter-tool Feedback

OpenROAD's shift from standalone binaries to an integrated app was a major advancement due to the common database substrate. This common substrate allows tools to interact with each other much more easily, but OpenROAD is only beginning to take advantage of this. We advocate focusing on mechanisms which allow inter-tool feedback to enhance QoR.

For example, the global and detailed placers focus mainly on half-perimeter wire length and cell displacement as their main optimization metrics. However, pin access is a primary constraint which can determine whether the router will be able to perform clean routing. Because of this limitation, users may try to reduce placement density, increase cell padding, and/or disable use of certain standard cells to avoid DRVs. Yet, macro placements in the floorplan, or the setup of global routing layer resources, could ultimately turn out to be "the culprit". We would like to see OpenROAD incorporate mechanisms to make upstream tools more aware of downstream problems so that users can save iteration time and achieve higher QoR. Simple versions of this might see the router's pin access analysis invoked by the detailed placer, or the global router being run under the hood of the placer. The common database substrate could also enable greater empowerments of OpenROAD's tools – e.g., the router curing a pin access-induced DRV by modifying the detailed placement.

3.6.2 Automatic Clock Gating

Automatic clock gating (ACG) is one of the most significant features that OpenROAD-flow does not yet support. Traditionally, ACG is implemented as part of synthesis, but Yosys does not currently support ACG. Due to Yosys being a third-party tool with separate infrastructure, addition of ACG may be difficult in that respect. We believe that the benefits to power and area are worth pursuing. We advocate that OpenROAD investigate a path towards implementing ACG, whether by upstreaming changes to Yosys, or by working within OpenROAD on a post-synthesis netlist.

3.6.3 Parasitic Extraction

Parasitic extraction is a key step which has been missing from OpenROAD-flow. We have worked around the issue by (1) extracting per-unit parasitics, (2) multiplying by wire length, then (3) derating to correlate with extracted parasitics. Parasitic correlation can be a cumbersome process, and more importantly, still relies on an external golden tool for accurate data. The project has invested significant effort toward bringing up a parasitic extraction tool, OpenRCX [5], which is expected to be introduced to OpenROAD soon.

OpenRCX will significantly aid in parasitic calibration as it reduces the trial and error from per-unit parasitics, but it still faces the issue of calibration from a golden model, which often uses data in an encrypted, unreadable format [29]. We advocate that OpenROAD explore two different paths for parasitic extraction:

- **Automatic golden correlation** - Proprietary golden model use is unavoidable for most commercial platforms. Commensurate with OpenROAD's vision, working with these golden models should be as automated as possible. For example, OpenRCX creates DEFs to provide to a 3D solver, and the user need only provide the corresponding parasitics (SPEFs) back to OpenRCX.
- **Full-stack solution** - The release of the open-source SkyWater130 platform provides a tremendous opportunity for OpenROAD. We advocate that OpenROAD develop a fully open-source parasitic extraction stack using the PDK.

3.7 Conclusions

The OpenROAD internal design advisors have greatly helped OpenROAD to get off the ground. Our expectations changed over time as we better understood the challenges facing open-source EDA flows, and we have learned important lessons along the way:

- **EDA is not typical open-source software** - Open-source projects typically have access to open data, whereas EDA must work with proprietary data to be practical. Proprietary data necessitates additional infrastructure and maintenance.
- **Test early and test often** - An RTL-to-GDS tool has an extraordinarily high minimum viable product of "placed and routed chip". Breaking down test cases into appropriate scope and complexity for early testing is incredibly important.
- **Expect the unexpected** - Even with significant EDA experience across my team, several unforeseen tasks arose. Project members stepping up to handle tasks outside

their expertise significantly helped to keep the project moving forward.

We also have an optimistic outlook for OpenROAD's future, and we have identified some features that would offer the most benefit from a designer's perspective:

- **Improved user experience** - Intuitive, accessible, and documented software is important for reducing user effort. While full automation is a long-term strategy to improving the user experience, improving documentation and accessibility can provide quick returns on investment.
- **Improved quality** - The easier it is to get high-quality results from OpenROAD, the easier it will be to get community investment in it. Features such as inter-tool feedback, automatic clock gating, and accurate parasitic extraction can significantly improve QoR.

3.8 Acknowledgements

I would like to thank the rest of the OpenROAD team, and especially thank OpenROAD's early adopters and contributors. I hope that OpenROAD can continue to grow into a tool that reshapes EDA.

This work was funded under the DARPA IDEA program (grant HR0011-18-2-0032).

CHAPTER 4

SpeEDAr

An overarching theme from both Celerity and OpenROAD was that shifting work to automated tools is frequently more efficient than manual work. This is a fairly evident statement, but it also means automated tools are at times *not* more efficient than manual work. Designing hardware is about striking the balance between manual effort and automated effort to minimize the design schedule. Figure 4.1 illustrates design time and effort with and without the slowing of Moore's Law. In this illustration, **effort** represents the work output of the engineer or automated tool, the combined effort required represents the **design complexity**, **time** represents the total wall time used to achieve that effort, and the **design schedule** is determined by the combined engineering and automation time.

In the first scenario, Moore's Law enables designers to maintain a fixed design schedule each generation due to offloading the increased complexity to the EDA tools. The exponential increase in complexity is paid with increasing effort from the EDA tools, but with fixed tool time. In the second scenario, Moore's Law is slowing. Automation alone cannot keep up with design complexity without dramatically increasing the design schedule. Therefore, a new balance is struck between engineering effort, automation effort, and total complexity. In this instance, some design complexity is sacrificed, some engineering effort is added, and the time constraint for automated tools is relaxed. The net effect is that with Moore's Law slowing, designs take longer with more engineering effort for a relatively less complex design.

Due to EDA tools' issue of closed source (See Section 1.3.2), Celerity treated the automation time and effort as a hard constraint and developed methodologies to minimize the design schedule. As an example, careful partitioning of the design hierarchy and reusable scripting enabled our team to extend Celerity's manycore array from 400 cores to 496 cores in just 3 days, including verification and signoff.

With the stable release of the OpenROAD app, a platform now exists for full EDA flow introspection, the likes of which haven't been available in decades, possibly ever (See

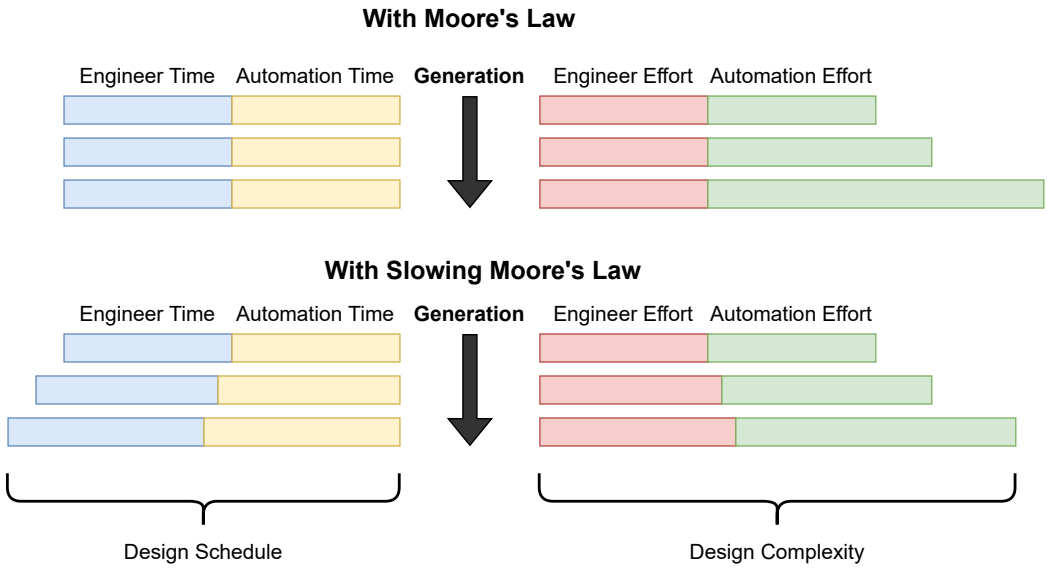

Figure 4.1: Tool effort

Section 1.3.2). OpenROAD enables examining the automation side of the design schedule for potential improvements. As demonstrated in other fields such as machine learning, hardware accelerators can be applied to computationally intensive workloads to dramatically reduce computation time over CPU substrates. Thus, OpenROAD can be used to investigate hardware accelerators for EDA in order to improve automation time/effort.

4.1 Introduction

In Section 1.4, Sirius presented a roadmap for accelerating computationally intensive workloads. Chapter 3 presented both OpenROAD, an open-source EDA tool, and OpenROAD Design Suite, a benchmarking suite for EDA tools. Following the direction of Sirius, OpenROAD can be used to characterize the EDA implementation flow. This characterization is presented in Section 4.2. Using this characterization, I identify the key stages of computation in the end-to-end flow and analyze their feasibility for acceleration. From this characterization, I select detailed routing (performed by TritonRoute) to examine for hardware acceleration, and I explore the TritonRoute algorithm in more detail in Section 4.2.1. I then examine prior work in the area of detailed routing hardware acceleration and identify the insufficiency of prior work to meet the needs of TritonRoute (Section 4.3). Section 4.4 presents SpeEDAr, a hardware accelerator for detailed routing, and Section 4.5 presents the simulated performance impacts of SpeEDAr on the entire OpenROAD flow.

The key contributions of this work are as follows:

CPU	Intel Xeon W-2245
Base Frequency	3.9 GHz
Max Frequency	4.7 GHz
Cores (threads)	8 (16)
Cache	16.5 MB
Memory	256 GB
Disk	2 TB NVMe Flash

Table 4.1: OpenROAD benchmark machine specifications

- I characterize OpenROAD-flow, an open-source EDA end-to-end implementation flow. To the best of my knowledge, this is the first published characterization of a full state-of-the-art EDA flow.
- The key computational steps of OpenROAD-flow are identified, including the detailed router TritonRoute.
- I examine previous accelerators for detailed routing and identify the disparity between TritonRoute's requirements and prior work.
- I present SpeEDAr, an accelerator for TritonRoute's detailed algorithm. SpeEDAr's performance is examined both in the context of detailed routing and in the full flow, yielding on average 74× and 1.20× speedups, respectively.

4.2 OpenROAD Software Characterization

Section 3.4.1 presented OpenROAD-flow's built-in design suite, which enables convenient access for OpenROAD software characterization on real workloads. These designs run out-of-the-box with OpenROAD-flow with the included open-source PDKs. OpenROAD-flow also runs with commercial PDKs, provided that the PDK is linked to the flow. For this software characterization, I use all available benchmark designs for the OpenROAD public platforms (`nangate45`, and `asap7`) and private platforms (`tsmc65lp` and `gf12`) at the time of writing. Using the CPU platform described in Table 4.1, OpenROAD is benchmarked with the OpenROAD Design Suite (via OpenROAD-flow's Makefile). All 16 threads are used for any multithreaded flow steps. The runtimes and breakdowns for each step are shown in Figure 4.2 and the normalized breakdowns are shown in Figure 4.3.

From these measurements, it is shown that detailed routing and synthesis consume 41% and 30% on average, while each other workload consumes less than 7% on average. Mean

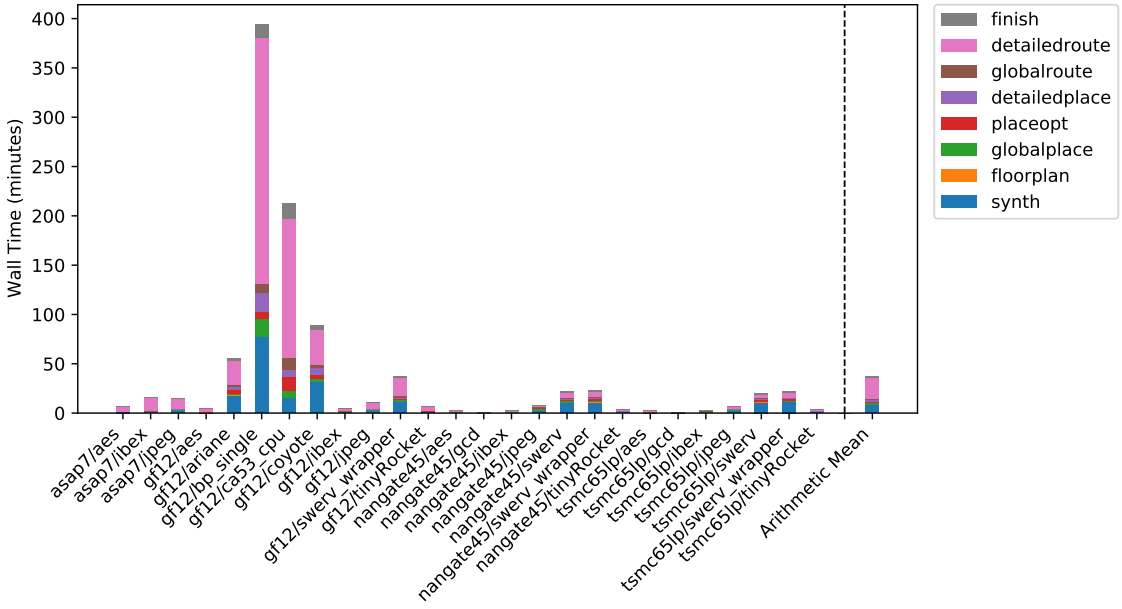

Figure 4.2: OpenROAD wall time breakdown for each workload.

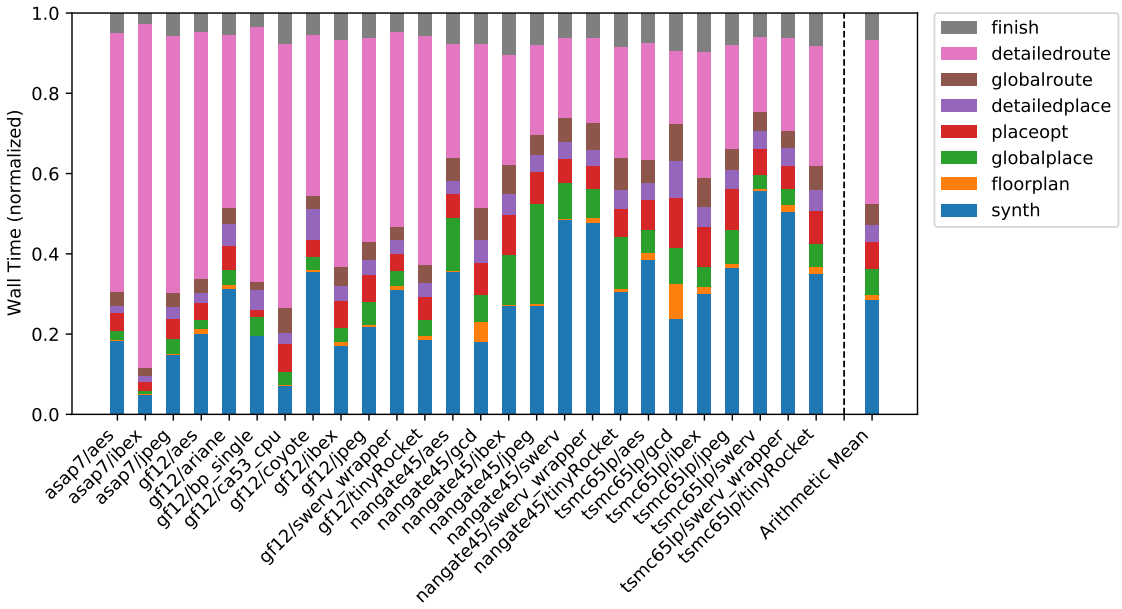

Figure 4.3: OpenROAD normalized wall time breakdown for each workload.

values were calculated as the arithmetic mean of the normalized components. From this characterization, several insights can be realized about the workloads:

- Detailed routing and synthesis are the two most time-consuming workloads
- For more advanced nodes (`asap7` and `gf12`), detailed routing dominates the execution time. This stands to reason because the routing rules for advanced nodes are significantly more complicated than older technologies
- For planar nodes (`nangate45` and `tsmc65lp`), synthesis overtakes routing for the most time consuming. This is mostly due to detailed routing consuming less time rather than synthesis consuming more time.
- Detailed routing is the only (significant) stage which is multi-threaded, and was characterized with 16 threads in this work. OpenROAD's synthesis engine, Yosys, is single-threaded. Parallel logic synthesis algorithms are used by commercial tools, however Yosys currently does not implement any.

For the remainder of this work, I focus on detailed routing as the primary candidate for acceleration. Not only is detailed routing the largest time consumption on average, but the time consumption is significantly greater among designs in advanced nodes.

4.2.1 Detailed Routing Algorithm

OpenROAD uses TritonRoute [30, 32] for its detailed router. TritonRoute is one of the only steps in OpenROAD which is multi-threaded, yet it still consumes a significant portion of wall time. TritonRoute consumes on average 40% of the total flow time using 16 threads on the benchmark machine. In this section, I explore the detailed routing algorithm in more detail.

TritonRoute is composed of 5 main phases: route guide pre-processing, design rule lookup table generation, pin access, track assignment, and detailed routing. To avoid confusion with the name of the flow step, TritonRoute's detailed routing phase will be referred to as the "search" phase.

4.2.1.1 Route Guide Pre-Processing

Prior to running TritonRoute, the global route step will create a list of **guide segments** as input to TritonRoute. A guide segment is a rectangular segment that identifies the suggested region and layer on which to route a net. Groups of adjacent guide segments form

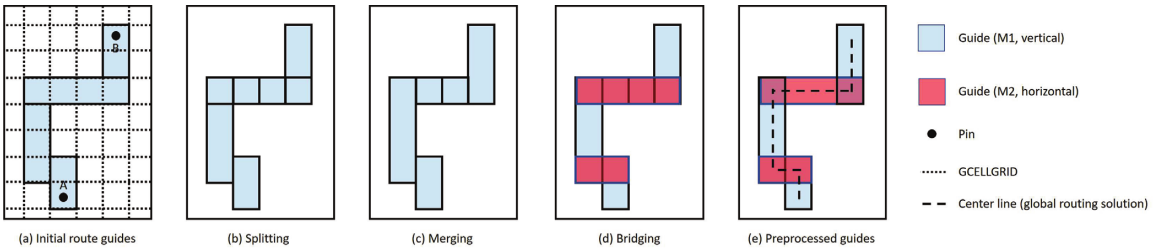

Figure 4.4: TritonRoute's guide pre-processing [32]

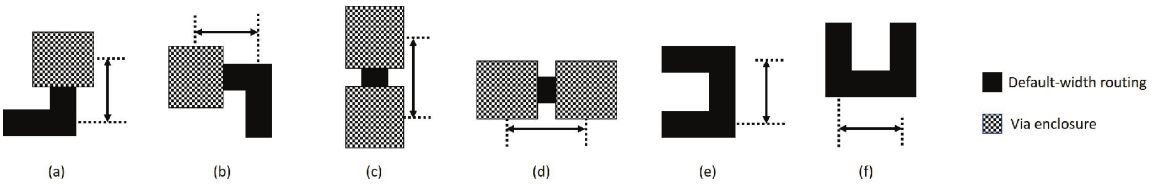

Figure 4.5: TritonRoute's LUT entry types: (a) vertical via to jog, (b) horizontal via to jog, (c) vertical via to via, (d) horizontal via to via, (e) vertical jog to jog, (f) horizontal jog to jog [32]

a **route guide** as guidance on how to route a net from source to destination. The minimum granularity of a route segment is typically 10-15 metal tracks (OpenROAD uses 10 metal3 tracks), referred to as a **GCell**. The purpose of this step is to simplify the routing problem by breaking it into separate steps. Global routing seeks to allocate routes such that routing tracks are not over-congested, and detailed routing performs the actual routing of wires with the focus of avoiding design rule violations.

As the first phase, TritonRoute pre-processes the guides into a standardized format. Figure 4.4 from Kahng et al. [32] illustrates this process. The route guides are first split into segments based on the preferred routing direction of the layer (Figure 4.4b). These segments are then merged such that adjacent guides along the same preferred direction form a single route guide (Figure 4.4c). Guide segments in the non-preferred direction are then moved to an adjacent layer so that the guide is in the preferred direction (Figure 4.4d). The final guides appear as shown in Figure 4.4e.

4.2.1.2 Design Rule Lookup Table Generation

The second phase of TritonRoute is lookup table (LUT) generation. TritonRoute reads the routing rules provided by the technology kit and simplifies them into a lookup table. Figure 4.5 from Kahng et al. [32] shows the list of entries in the LUT, which identify minimum distances among vias (changes in layer) and jogs (turns in the same layer). TritonRoute uses the LUT during the search phase to avoid routing violations.

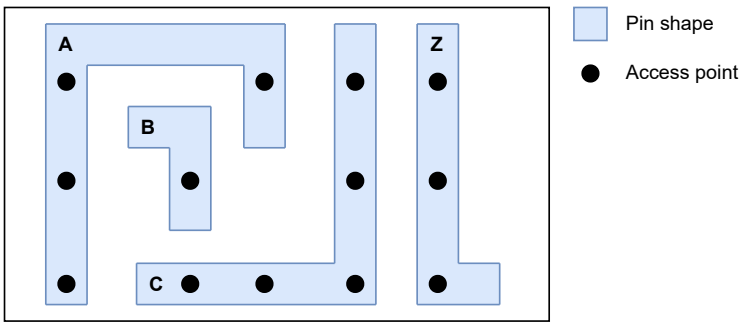

Figure 4.6: An example of a TritonRoute pin access strategy [31]

4.2.1.3 Pin Access

The third phase is pin access, whereby TritonRoute identifies **access points** on each pin. An access point is a potential point where a route can connect to a pin. A group of access points together form an **access strategy**. Figure 4.6 from Kahng et al. [31] illustrates one example of an access strategy. Certain groupings of access points are preferred because they reduce the likelihood of causing congestion and design rule violations near the pin. Pin access strategies are formed for each pairing of standard cells to identify candidates when standard cells abut each other. These pin access points ultimately will form the source and destination nodes when performing the path search.

4.2.1.4 Track Assignment

TritonRoute performs track assignment during the next phase of the algorithm. Because the input guides represent multiple tracks that TritonRoute can use to route, it must match nets to tracks; TritonRoute uses a greedy algorithm to do so. The matching is done to try to minimize the amount of congestion, that is, the degree to which wires overlap each other in their track assignment. An example of this process is shown in Figure 4.6.

4.2.1.5 Search (Detailed Routing)

The final phase of TritonRoute is the search phase, illustrated in Figure 4.8. First, TritonRoute breaks down the chip into groups of GCells, called a **route box** (Figure 4.8a). A route box is adjustable in size, but it must be an odd square of GCells (e.g. 3×3, 5×5, 7×7). Larger route boxes can offer better solutions because they offer the search algorithm more space to find valid routes. On the other hand, larger route boxes also increase runtime due to the larger search space. TritonRoute uses a 7×7 route box by default as a balance between search quality and runtime.

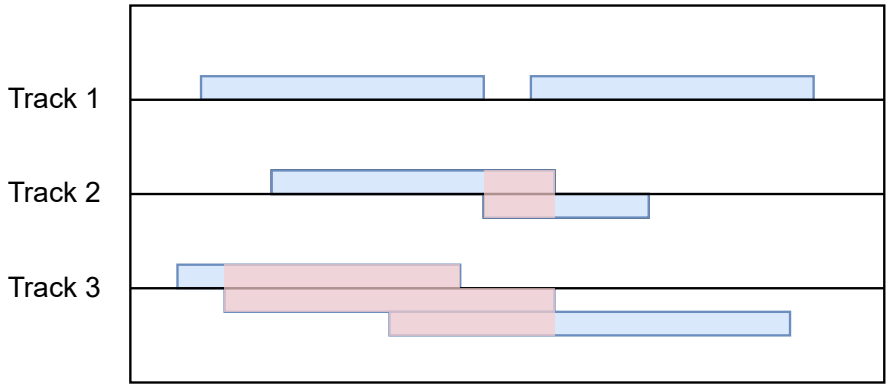

Figure 4.7: An example of track assignment for 7 routes to 3 tracks. Red areas indicate over-congestion.

Once the route boxes are formed, each one is pushed into a **routing queue** (Figure 4.8b). The routing queue is a serial data structure that functions as a work queue for parallel processing. TritonRoute creates a number of worker threads specified by the user, and each worker thread atomically pops the routing queue until empty (Figure 4.8c).

Within a worker thread, TritonRoute runs an A* search to search for a path from one of the source nodes to one of the destination nodes (Figure 4.8d). A key detail about TritonRoute's search is that multiple source nodes and destination nodes can exist for a single path. Each source is assigned an initial cost based on the pin access mentioned in Section 4.2.1.1, and the search terminates once any of the destination nodes are reached. TritonRoute notably differs from a true A* search because the design rule costing function causes it to be *inconsistent*, therefore a node has the possibility of being revisited. TritonRoute does not always find the lowest-cost route because it limits the number of times a given node can be expanded; by default, the limit is two expansions. The authors state that the reasoning for this limitation is to prevent excessive runtime and searching. After completing the search, TritonRoute backtraces the path and updates the associated costs for each node in the route box (Figure 4.8e). This process is repeated for each net within the route box.

The next step in the algorithm is for TritonRoute to wait and synchronize the worker threads to ensure all nets have been routed. Then, it performs **geometry checking** (design rule checking) on each route box (Figure 4.8f). The route box is bloated slightly to ensure that shapes outside the route box are accounted for when checking design rules. The graph is then updated with the DRC costs (Figure 4.8g).

Steps a-f combined form one iteration. TritonRoute continues iterating with updated costs until the design either reaches 0 violations or the maximum number of iterations.

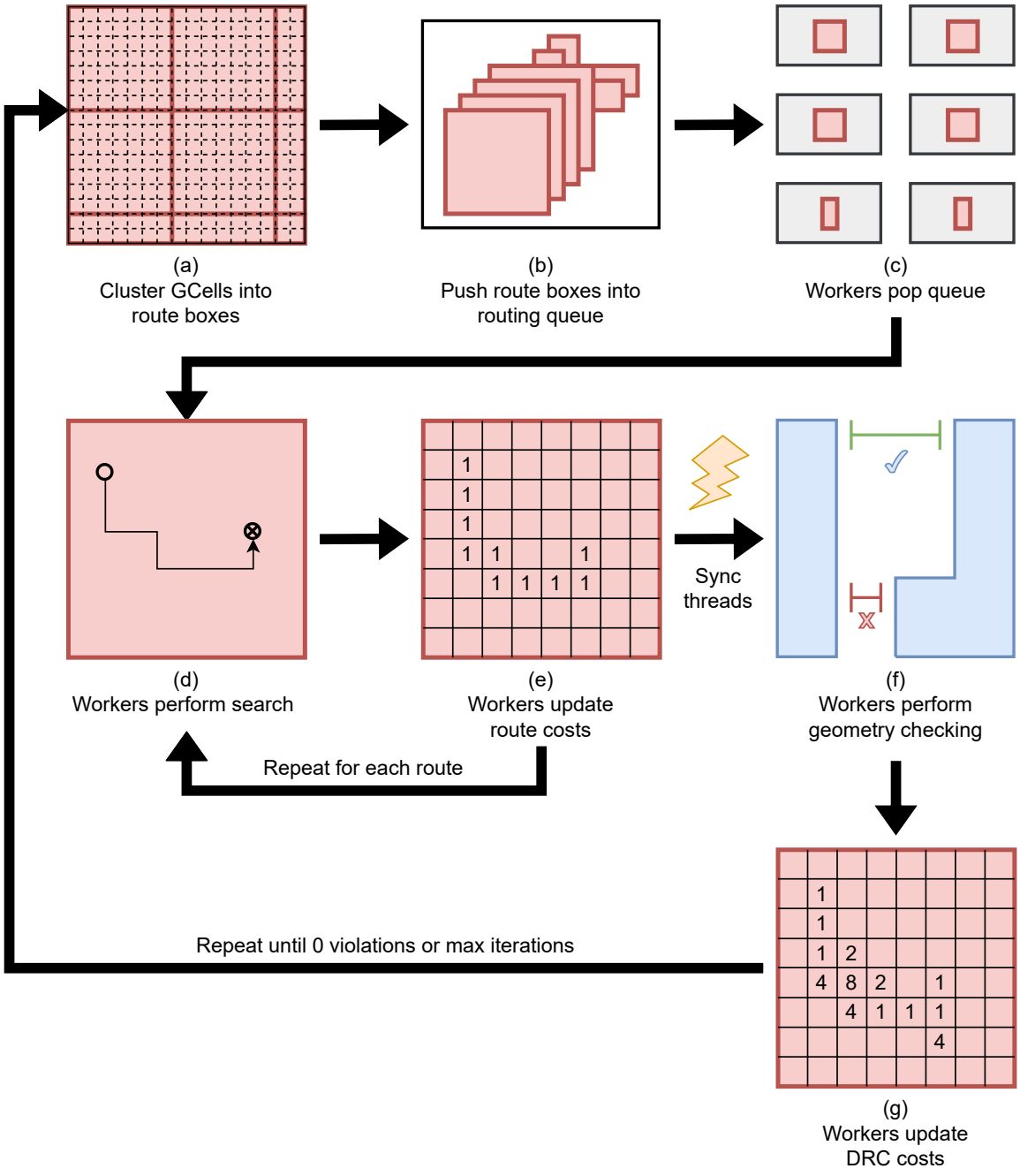

Figure 4.8: Flow diagram of TritonRoute's routing (search) phase.

4.2.2 Detailed Routing Characterization

With TritonRoute as the primary acceleration candidate, I further examine the computational breakdown of TritonRoute on the benchmark platform. Profiling with Intel VTune reveals that `FlexGridGraph::search()` consumes a significant portion of the runtime: on average 41%. This function represents the A* search phase of TritonRoute mentioned in the previous section. The pre-processing components do not compose a significant portion of the workload, which follows from being executed only once at the start of TritonRoute. Additionally, geometry checking composes a significant, albeit smaller, portion of the execution time at 11.5%.

This characterization demonstrates that the A* search is the key computational kernel underpinning TritonRoute, and it forms a prime target for acceleration. By combining the flow-level characterization with the TritonRoute characterization, this single function consumes on average 22% of the total flow runtime. In the next section, I examine TritonRoute's particular implementation of A* search, as well as prior work in accelerating this class of workload.

4.3 Related Work

Although this work is the first (to the best of my knowledge) to characterize an end-to-end implementation flow and identify the key computational kernels, other works have previously identified detailed routing as an important workload. However, the most recent work in this area by Nestor and Lavine [41] dates to 2007, when the most advanced technology node commercially available was a 45nm planar process. Routing requirements for modern finFET nodes (<20 nm) have dramatically increased since then, including a more than doubling of number of routing rules, double patterning, and more. TritonRoute was designed to address these modern technology nodes, and as such, differs substantially from the algorithms targeted by previous hardware accelerators for detailed routing. This section will discuss previous work and how the requirements for TritonRoute differ.

4.3.1 Lee's Algorithm Accelerators

The key work that underpins most detailed routing algorithms is Lee's Algorithm [35]. Lee's Algorithm describes a breadth-first-search approach to finding a minimum path from a source node to a destination node on a grid. The greatest strength of the algorithm is that it finds a shortest path, if one exists. The drawback to this algorithm is that it is computationally expensive to perform a breadth-first search. As such, derivatives of Lee's

Algorithm have been developed to reduce the computational cost of the search, such as the A* algorithm. The key difference is that the A* algorithm uses a heuristic to estimate cost to the destination node, which allows the algorithm to prioritize searching in the lowest-cost direction.

Prior hardware approaches to detailed routing have proposed using multiple processing elements (PEs) to compute the search expansion in parallel, thus reducing the search time. Previous *full-grid* approaches [28, 47] use a PE to represent each node in the graph, whereas *virtual-grid* approaches [49, 57, 56] reduce hardware cost by mapping multiple nodes onto the same PE. In particular, Nestor and Lavine proposed the L4 architecture [41], which is a virtual-grid approach to accelerating Lee's algorithm. These architectures rely on several assumptions that do not apply to TritonRoute:

- **TritonRoute uses non-uniform edge costs between nodes**. A key assumption of Lee's algorithm and prior approaches is that each edge has uniform cost, thus the cost is equal to the number of edges traversed. Furthermore, this assumption meant that the shortest path is also the lowest-cost path, which is not the case with TritonRoute.
- **TritonRoute does not have hard blockages**. Prior approaches assume that blocked nodes cannot be routed through and therefore do not need to be expanded. Additionally, these approaches can fail to find a path if the destination nodes are completely blocked. TritonRoute *does* allow routing through blocked nodes, albeit with higher cost, and will always return a solution if the input is well-formed.
- **TritonRoute has cost history**. TritonRoute uses costs from previous iterations to represent routes, blockages, and design rule violations in order to iterate on the previous routing solution. Prior solutions do not implement any cost history other than blockages.

Because of these differences, TritonRoute requires a new solution which is compatible with its search algorithm.

4.3.2 Potential Platforms

4.3.2.1 Multicore CPUs

TritonRoute is already multithreaded and shows good speedup. On the benchmark platform, TritonRoute shows on average a 6.9× speedup for 16 threads vs. a single-thread baseline. The key limitation for parallelism in TritonRoute is the synchronization of threads after every routing iteration. Figure 4.9 shows the cumulative distribution of routes versus

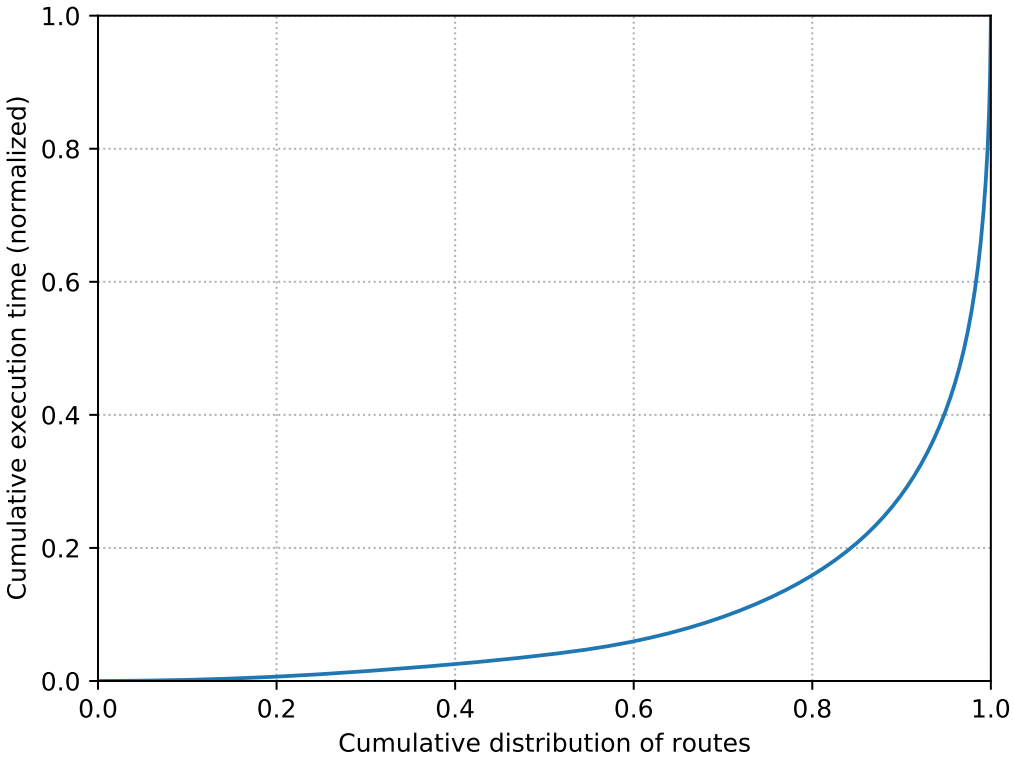

Figure 4.9: Cumulative distribution of all routes from the OpenROAD Design Suite vs. their cumulative execution time on the benchmark platform (normalized)

their normalized runtime. The figure shows that 80% of routes consume less than 16% of the total runtime, whereas the top 5% of routes consume 51% of the runtime. While TritonRoute can parallelize each route box, the algorithm cannot reduce the latency of an individual route box because each net must be routed sequentially. Parallel algorithms for A* search exist, however many of them perform extra work in order to properly terminate with the lowest-cost path or do not achieve a lowest-cost solution [23]. A solution might be designed to balance threads between workers and within workers, however this solution is not considered and left for future exploration.

4.3.2.2 GPU

Prior work has shown that GPUs map very poorly to detailed routing algorithms due to a) lack of parallelism for each net, b) divergence of computation patterns between nets, and c) a huge number of random memory accesses [38]. Tangjittaweechai et al. attempted to accelerate detailed routing using a GPU [50], however their solution only achieved a 25%

average speedup over a single-thread CPU, which is slower than a multithreaded solution.

4.3.2.3 FPGA/ASIC

The results of prior work show substantial promise. The L4 architecture shows a 29-93× speedup over the classic Lee Algorithm and 5-19× speedup over the A* algorithm using a CPU baseline [41]. However, there are many limitations to this study that limit its usefulness. Firstly, the work is from 2007 and does not consider any advanced-node technologies. Design constraints have increased substantially, requiring entirely new classes of rules which do not map well to the L4 architecture. Secondly, the architecture is based off a custom C implementation of the A* algorithm which is not derived from real detailed routing software. Thirdly, the implementation was only tested against small micro-benchmarks which each complete in under 1 second on a CPU. Despite this, the work lays a valid foundation and demonstrates that FPGA/ASIC is a promising platform for hardware acceleration for this problem.

4.4 SpeEDAr Architecture

To address the deficiencies of prior work, I propose SpeEDAr, an architecture designed to accelerate TritonRoute's detailed routing algorithm. Figure 4.10 shows the block diagram for SpeEDAr's mesh architecture. Similar to prior work, SpeEDAr uses a virtual-grid mesh of PEs to represent each node in TritonRoute's grid graph. The mesh is an $n \times m$ grid representing one layer of the grid graph, and each physical node stores the data for all z layers at its (x, y) location.

4.4.1 Mesh Architecture

Within a PE, SpeEDAr differs greatly from prior work. L4's PE is a simple finite state machine which uses an expansion bit from the neighbors to determine when to expand. SpeEDAr instead receives cost inputs from each direction and computes the new cost for the node. The cost is calculated the same as TritonRoute by adding the path cost, distance cost, turn cost, and via spacing cost. These costs are detailed in Table 4.2. The path cost is the input cost to this node, the distance cost is the physical length between the two nodes, the turn cost is a penalty added if there is a turn in the path to get to this node, and the via spacing cost is an additional cost added if the last via (up or down turn) is too close to this node and is a via.

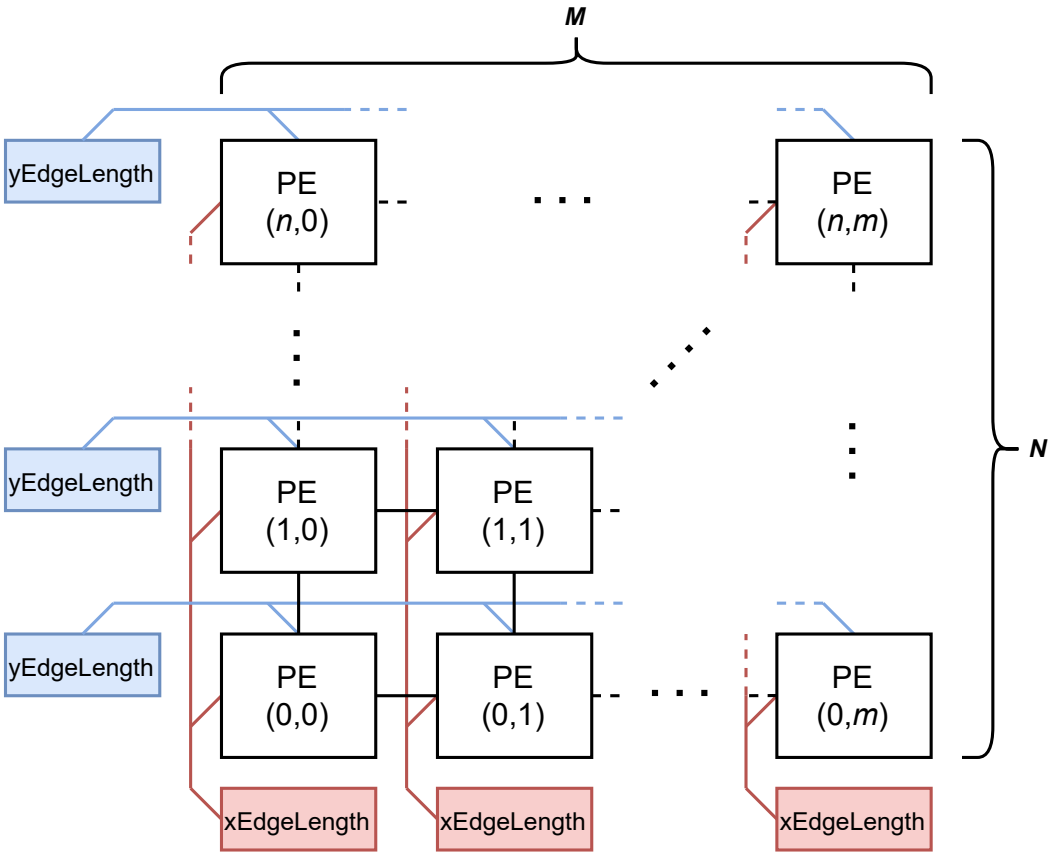

Figure 4.10: SpeEDAr mesh block diagram for N rows and M columns

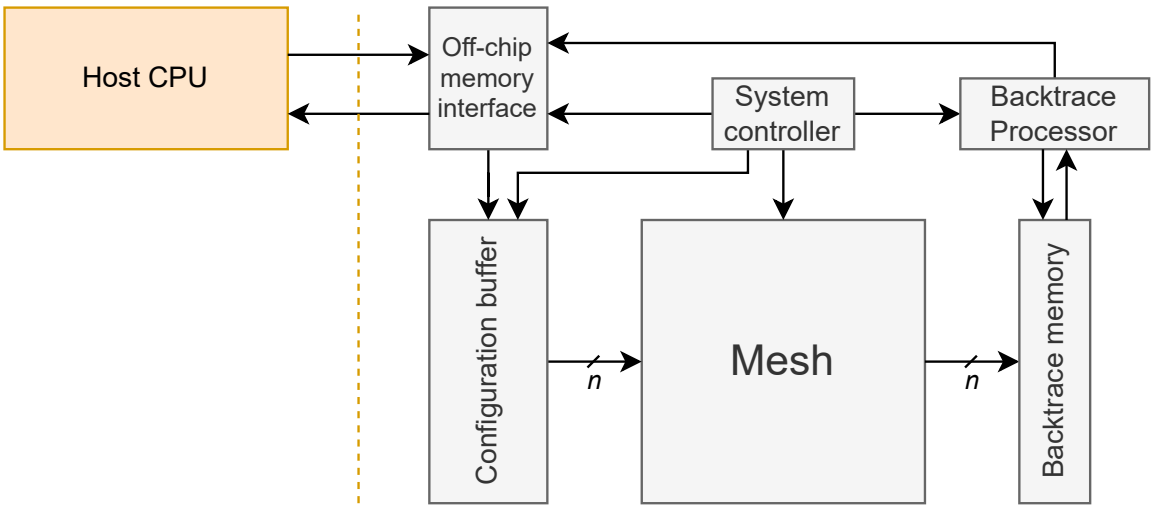

Figure 4.11: SpeEDAr system block diagram. The buses between the configuration buffer, mesh, and backtrace memory are each n lanes wide where n is the number of mesh rows.

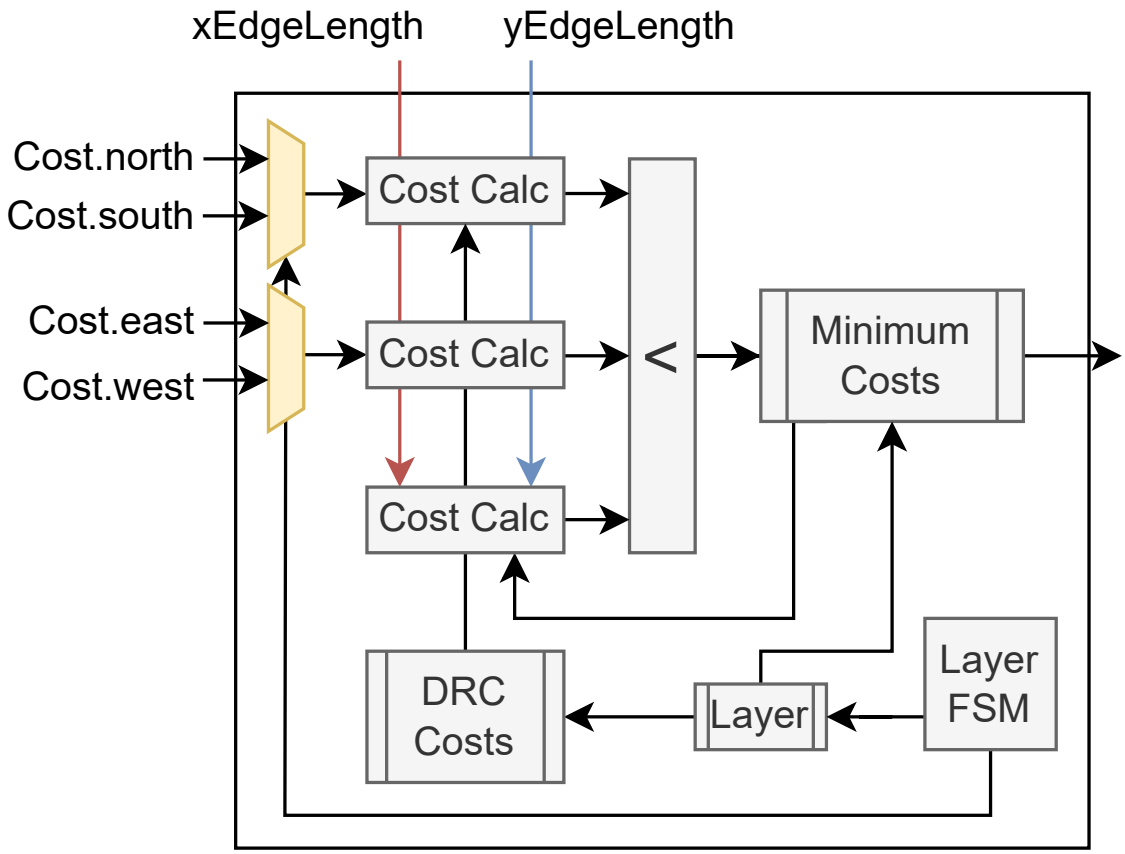

Figure 4.12: SpeEDAr processing element block diagram. Cost input signals include path cost, via distance, and previous direction. The layer finite state machine determines whether the {north, east} inputs or {south, west} inputs are selected. The {up, down} costs are passed from the cost of the previous layer. Minimum costs and DRC Costs are indexed by the layer.

Cost	**Formula**
Path	Accumulated cost to previous node
Distance	Edge length × `GRID_COST`
Turn	Direction != previous direction
Via spacing	(Direction == Up || Direction == Down) && (distance < min via distance)
Marker	Marker_cost[current node]
Route	Route_cost[current node]

Table 4.2: List of SpeEDAr cost calculations as derived from TritonRoute. `GRID_COST` is a fixed value. Marker and route costs are stored in PE lookup tables.

Once each of these costs are calculated, the PE selects the minimum cost and direction and stores them in a register file indexed by the current layer. Up and down inputs differ from planar inputs because the path cost comes from within the PE. Once the cost for the current layer is stored, it can be used as the path cost input to the next layer.

As noted in Figure 4.12, there are only 3 cost calculators for the 6 input directions. SpeEDAr time shares the cost calculations in order to reduce hardware costs and to complement the cost propagation scheme. SpEDAr has a similar wavefront pattern to L4: a wavefront is started from a point and expands in each direction. SpEDAr starts expansion from a corner node while L4 starts expansion from the source nodes. While L4 is slightly faster by expanding from the source nodes first, it requires expansion in 5 directions simultaneously (North, South, East, West, and Up) and therefore the associated hardware to do so. L4 also assumes that the source node is always on the bottom layer, which is not the case for TritonRoute. L4's algorithm breaks down if a source is not on the bottom layer, because it requires expanding up and down simultaneously. By starting expansion from the corner, SpeEDAr always expands nodes in 3 directions (North, East, and Up). This provides important hardware savings over the L4 architecture.

SpeEDAr continues cost propagation through each node until it reaches the opposite corner of the route box (e.g. starting at $(0, 0, 0)$ and ending at $(x_{\text{max}}, y_{\text{max}}, z_{\text{max}})$). Propagating the cost from one corner to the opposite corner is considered *one propagation*. SpeEDAr then reverses the direction and starts propagating costs backwards (e.g. starting at $(x_{\text{max}}, y_{\text{max}}, z_{\text{max}})$ and ending at $(0, 0, 0)$). When propagating in the reverse direction, SpeEDAr propagates costs along the West, South, and Down directions. This approach differs substantially from prior work because prior work stops as soon as a destination node is reached. However, In TritonRoute, the shortest path is not necessarily the lowest-cost path, so SpeEDAr continues propagating.

With each propagation, SpeEDAr finds a path with the same or lower cost than the previous propagation. This approach differs from TritonRoute's algorithm. Because TritonRoute uses a DRC costing function, the path search breaks the *consistency* quality of the A* algorithm. An ideal algorithm which always finds the minimum-cost path requires $O(n^2)$ complexity, which is why TritonRoute instead allows each node two re-expansions as a balance between runtime and solution quality. This approach means that SpeEDAr and TritonRoute both provide *approximate* solutions. TritonRoute's solution quality converges to optimal as more re-expansions are allowed, and SpeEDAr converges to optimal as more propagations are allowed. Due to difference in implementations, it is not possible to replicate TritonRoute's approximate solution with SpeEDAr, however it is possible to converge to an equivalent or lower-cost solution with more propagations.

If we define a **direction domain change** as a change in direction from {north, east, up} to {south, west, down}, SpeEDAr's algorithm guarantees that the optimal path with n direction domain changes is found within $n + 1$ propagations. For example, any path which consists of a north, east, and south segment has 1 direction domain change (east to south), and therefore will be found within 2 propagations. A path which contains a north, east, south, east, up, east, and down segment has 3 direction domain changes (east to south, south to east, east to down) and will be found within 4 propagations.

4.4.2 Windowing Optimization

One drawback to SpeEDAr's architecture is that the search time scales with the graph size ($O(n + m)$), regardless of the route length. However, because the optimal path usually lies within the bounding box of the source nodes and destination nodes, a *windowing* optimization can be applied where the search is limited to this bounding box. Thus the search time is reduced to $O(n_w + m_w)$, where n_w and m_w are the window size. The tradeoff for this optimization is that optimal routes outside of the bounding box are not found. Other works which use the windowing optimization rely on window expansion, which bloats the window if a solution is not found. Because SpeEDAr always finds a solution, my approach instead relies on the TritonRoute iteration number to determine the window bloat. The iteration number is a direct representation of the difficulty in finding a clean solution for a given route box. Therefore, earlier iterations can use a window bloat of 0 while later iterations use an increased bloat size.

Figure 4.13 displays the cumulative distribution of bloating required for SpeEDAr to produce to produce the same solution as TritonRoute. This metric for required bloating provides a conservative estimate for the amount of required bloating for SpeEDAr, because SpeEDAr is able to find routes that TritonRoute does not within a given window. The figure shows that a vast majority of routes, 81%, require no bloating to find a clean route. The next 11% require a bloat of $<$10, and 7% require a bloat of $<$100. Only the last 0.05% of routes require a bloat of $>$100. Therefore SpeEDAr exponentially increases the window bloat to converge to a window size for a clean route without increasing runtime dramatically. SpeEDAr uses a bloat size of $\{0, 10, 50, 100\}$ for detailed routing iterations 0-3. After iteration 3, the bloat size doubles until the mesh size is reached. From testing on the OpenROAD Design Suite, this strategy enables SpeEDAr to converge to the required window size within 2 iterations of TritonRoute. These additional iterations contribute a maximum of 1% computation overhead versus a solution with 0 bloating.

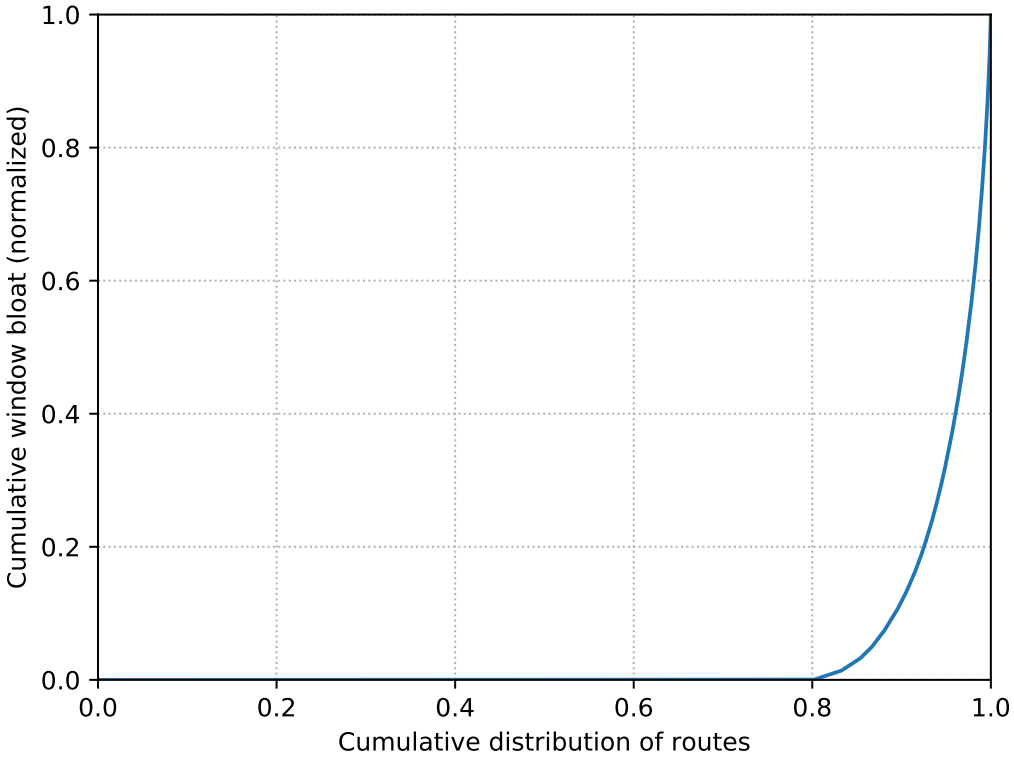

Figure 4.13: Cumulative distribution of all routes from the OpenROAD Design Suite vs. the cumulative window bloating required to find the same solution as TritonRoute (normalized)

Data	Size	Frequency
x & y edge lengths	$8 \times (x + y)$ bits	Once per route box
Grid graph	$x_w \times y_w \times l_w$ bits	Once per route box
Cost update	Varies	Each route
Path result	$48 \times \#$turns bits	Each route

Table 4.3: SpeEDAr off-chip communication costs. Subscript w represents that the input has been windowed.

4.4.3 Backtracing

Because SpeEDAr always finds a solution within 2 propagations, the architecture does not have any signal to indicate when a destination node is reached. Instead, it terminates after a predefined number of propagations have completed. The number of propagations is a direct trade-off between solution quality and runtime. Once the search phase terminates, the previous directions are shifted 1 column per cycle into an off-mesh backtrace memory. While prior solutions perform backtracing in-mesh, this approach presents substantial hardware cost added to each node as well as global signaling to support it. SpeEDAr instead opts for a dedicated backtrace processor which starts at the destination node and iterates through the memory until a source node is found. This path is compressed into a list of coordinates representing each segment in the path, which is then returned as the result from SpeEDAr.

4.4.4 Communication

SpeEDAr simulates a high-speed interconnect to a host machine. In this work, I model a PCIe 3.0 interface with 8 lanes for communication to the host PC. The required communication to and from SpeEDAr is summarized in Table 4.3. SpeEDAr uses a buffering scheme to allow incoming data transfers while the current route box is being searched and backtraced.

4.5 Measurements

SpeEDAr is implemented in SystemVerilog and verified using Synopsys VCS against test examples from TritonRoute. SpeEDAr matches TritonRoute solutions on trivial routes but differs on more difficult routes. However, SpeEDAr finds an equivalent or superior solution to TritonRoute in all cases tested by comparing path costs of the backtraced path. TritonRoute and SpeEDAr both converge to 0 design rule violations on all cases tested. SpeEDAr was synthesized to the Global Foundries 12nm PDK to measure the hardware

Mesh Size	200×200	250×250	300×300	350×350	400×400
Area (mm^2)	20.1	31.6	45.8	62.6	82.1

Table 4.4: SpeEDAr mesh logic area for various sizes. Results obtained from synthesis in Global Foundries' 12nm node

cost of an ASIC implementation. Table 4.4 shows the area cost of SpeEDAr at various mesh sizes.

Figure 4.14 shows the speedup obtained by SpeEDAr on the OpenROAD Design Suite for the grid graph search function over the 16-thread CPU baseline. Figure 4.15 then simulates the impact of SpeEDAr on the end-to-end flow for the benchmark suite. SpeEDAr achieves a mean speedup of 74× on the grid graph portion of TritonRoute, and a mean flow speedup of 1.20×. The results show that the speedup is higher for more advanced nodes (7/12nm) and lesser for older nodes (45/65nm). This is due to the routing rules and grid graph sizes being smaller on these nodes. Because more advanced nodes require larger graph sizes, searches to traverse them take longer. In addition, more complicated design rules translate to more costing and searching required to find a DRC clean solution.

The overall flow speedup is higher for advances nodes (1.19-1.51×) and lower for older nodes (1.07-1.20×). This stands to reason because TritonRoute composes a smaller portion of flow runtime and does not achieve as high speedup on older nodes. However, SpeEDAr still achieves a worst case flow speedup of 1.08×.

4.6 Conclusions

This chapter characterized and investigated the computational characteristics of OpenROAD, an end-to-end open-source EDA flow. My characterization revealed that detailed routing consumes on average 41% of the flow wall time, and the grid graph search consumes a majority of the detailed routing runtime. Prior work in this area does not map well to TritonRoute due to its different path and design rule costing. I presented SpeEDAr, a hardware architecture to accelerate TritonRoute. Simulated ASIC results show that SpeEDAr improves search performance on average 74×, which translates to a 1.20× average speed up for the entire OpenROAD implementation flow.

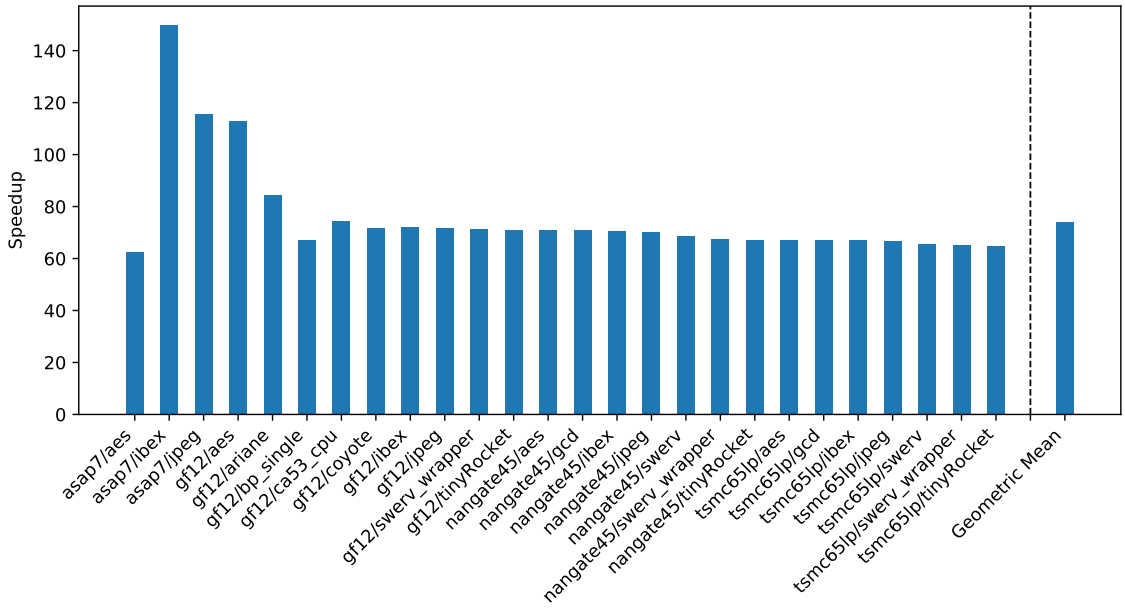

Figure 4.14: SpeEDAr speedup of TritonRoute grid graph search function over the 16-thread CPU baseline

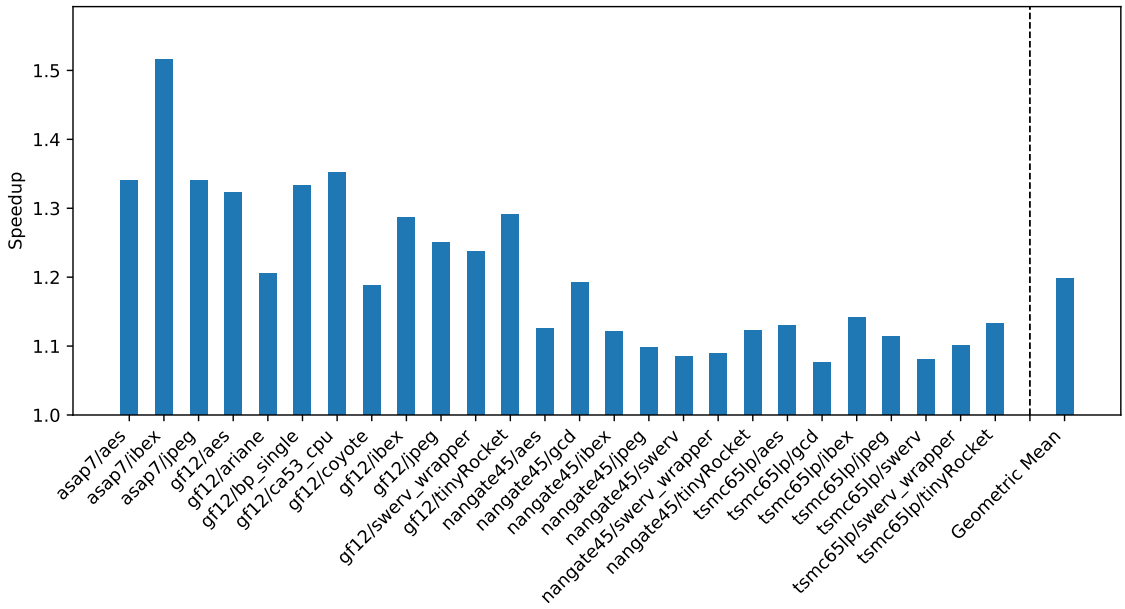

Figure 4.15: SpeEDAr speedup of OpenROAD-flow over 16-thread CPU baseline

4.7 Future Work

While SpeEDAr presented an accelerator to significantly reduce the automation time used by EDA tools, there remain many avenues to be more fully explored.

4.7.1 Implementation

While SpeEDAr's speedup results are very promising, furtherance of this idea requires implementation of the accelerator and integration with a host CPU platform. Development of such system will enable more accurate measurement of end-to-end speedup. The main goals of such a system would be to examine the data transfer overheads and to develop schemes to minimize the amount of communication required between the accelerator and host platform.

Another area of focus for implementation would be to optimize the die area of the architecture. Currently, a significant portion of each PE is dedicated to storing cost values. Optimizations schemes such as **delta compression** may be able to greatly reduce the required storage in a PE without greatly affecting performance. In theory, there is a maximum cost difference between nodes on layer n and layer $n + 1$. If this cost difference is small enough, it may be possible to store data in a base + offset format where the base is stored using i bits and the offsets are stored using j bits, where $i < j$. The overhead cost is the adders required to decompress the data from base + offset format.

4.7.2 Further Acceleration

As noted by Amdahl's Law, the maximum speedup for accelerating a program is proportional to the fraction of the program that is sped up. While SpeEDAr provides on average 74× speedup for the grid graph search, this portion of the program constitutes only 55% of TritonRoute's runtime, and therefore a limitation of 2.2× speedup for the whole program. To further accelerate TritonRoute, it is worth exploring extensions of this architecture to accelerate adjacent portions of TritonRoute. Key functions of interest include grid graph cost updating and geometry checking. Adding cost updating seems to be a promising extension of SpeEDAr, because cost updates constitute a majority of the transfers between SpeEDAr and the host CPU. By adding this functionality on-chip, the required communication between the host platform and SpeEDAr can be reduced dramatically.

Geometry checking is another function of interest because it composes the other portion of a routing iteration. Routing iterations dominate the total runtime of TritonRoute, and hardware which accelerates the entire routing iteration bodes promise for increased

speedup. Incorporating both algorithms on an acceleration platform would require much less communication to the host platform – data would only need to be synchronized for the whole route box rather than per route.

4.7.3 Alternate Architectures

SpeEDAr achieves very high speedup over the CPU baseline at the cost of relatively high area consumption. While SpeEDAr can be synthesized to an FPGA target, current limitations on FPGA sizes mean that only small meshes ($<$100×100) can fit on the FPGA fabric. For more accessible acceleration, exploring less area-intensive architectures is warranted. Prior work mainly explores mesh-based architectures wherein Lee's algorithm and other breadth-first searches are accelerated. However, TritonRoute uses an A*-based search which generally has much smaller memory requirements than a full breadth-first search. Some recent work on A* hardware acceleration shows promising results with a 37-75× improvement over a CPU baseline [61]. However, this work faces similar pitfalls to prior Lee's Algorithm accelerators in terms of their applicability to TritonRoute. Additionally, significant focus is placed on maintaining an accurate OPEN list of nodes, whereas TritonRoute allows node revisiting. An A*-based accelerator for TritonRoute could offer significant benefits over a full- or virtual-grid accelerator, including accelerator area reduction as well as solutions more consistent with the TritonRoute software implementation.

4.7.4 Other Workloads

Section 4.2 presented a characterization of the full OpenROAD flow. Detailed routing with TritonRoute was identified as the most computationally intensive step, however other steps such as synthesis pose significant time consumption as well. A straightforward step to reducing total flow time would be to investigate and incorporate hardware acceleration options into these other steps. Detailed Routing is the only major flow step which incorporates multithreaded processing; other flow stages run using a single thread. Some workloads such as global placement have readily-available hardware solutions available and should be incorporated into OpenROAD [39, 24]. Other workloads such as synthesis have an abundance of literature on parallel implementations of algorithms [20, 19, 44]; these implementations should be explored to improve CPU utilization over the current single-thread implementation.

CHAPTER 5

Conclusions

With the waning of Moore's Law, new foundry processes cannot be relied upon for performance gains. Many fields such as machine learning, genomics, graph processing, drug discovery, financial trading, and others have shifted towards designing specialized hardware to continue improving performance. One notable field that has not seen such a shift is electronic design automation (EDA), the software and algorithms underpinning computer chip design. The slowing of Moore's Law is at risk of creating a negative feedback loop: slowing computer performance leads to slowing chip design which leads to further slowing of computers.

Section 1.4 presented Sirius, a case study for hardware acceleration of intelligent personal assistants (IPAs). This case study mapped extremely well to the current EDA landscape, because state-of-the-art software is proprietary and closed source. Sirius also posed a challenge that integrating hardware accelerators into a system remains challenging.

Chapter 2 presented Celerity, which addressed the design and integration of complex systems-on-chip (SoCs). Celerity presented a complex, tiered accelerator fabric to offer both highly programmable general computation and highly efficient specialized computation. Celerity's manycore array broke several performance records at time of publication, including 1) single-chip peak RISC-V instruction throughput (695 GRVIS), 2) CoreMark benchmark raw score (825,320), and 3) CoreMark score/MHz (580.25). Celerity also outperformed prior manycore works in energy efficiency and normalized area efficiency by 4.2× and 1.8×, respectively.

Even with Celerity's complexity, the chip itself was designed and taped out in 9 months: about half the time of a normal chip development cycle. Celerity proposed 3 key design strategies for reducing the amount of time for chip development: 1) reuse designs and intellectual property (IP), 2) modularize designs and interfaces to enable hierarchical design, and 3) automate high-cost manual tasks. Reuse may be one of the most powerful strategies available, because it reduces design time, verification, and integration. These strategies address and simplify the design integration issues faced in Sirius.

Chapter 3 presented OpenROAD, a fully open-source RTL-to-GDS flow. Similar to how Sirius created an open-source platform for studying IPAs, OpenROAD created a platform for studying EDA. I also presented the OpenROAD Design Suite, which hosts a collection of real-world, representative designs for benchmarking and validating OpenROAD. OpenROAD has had a tremendous impact on the chip design community – 3 multi-project wafer (MPW) submissions through EFabless' Open MPW Shuttle program as well as 2 MPW submissions from Efabless' ChipIgnite program were all enabled by OpenROAD. In total, hundreds of designs have been taped out using OpenROAD, including a reported 60% from first-time chip designers [7]. OpenROAD succeeded not only in providing a platform for EDA study, but it has also spurred innovation in hardware design by lowering the barrier for access.

In Chapter 4, I presented SpeEDAr, an accelerator for detailed routing. With the release of OpenROAD and the OpenROAD Design Suite, I was now able to characterize an EDA flow and examine the computationally intensive kernels. Characterization revealed that a single function in the entire flow, TritonRoute's grid graph search, composed approximately 22% of the total flow wall time. Prior accelerators for detailed routing lack compatibility with TritonRoute's detailed routing algorithm; therefore, I presented SpeEDAr, an accelerator architecture for detailed routing. SpeEDAr achieves a mean speedup of 74× over the 16-thread CPU baseline for grid graph search, and a mean 1.20× end-to-end flow speedup.

In a broad sense, this dissertation examined modern hardware design from multiple different perspectives: design, integration, automation, and tooling. Through each of these lenses, I presented solutions to speed up hardware design and enable increased complexity. The key techniques behind these works involved making hardware design cheaper, faster, and more accessible through fast design methodologies, open-source designs, and open-source software; thus, we move closer toward a vision of free, open, and ubiquitous hardware design.

BIBLIOGRAPHY

[1] 2008. Video compression systems. https://opencores.org/projects/video_systems.

[2] 2018. AES (Rijndael) IP Core. https://opencores.org/projects/aes_core.

[3] 2019. SweRV RISC-V Core 1.1 from Western Digital. https://github.com/westerndigitalcorporation/swerv_eh1.

[4] 2020. Ibex RISC-V Core. https://github.com/lowRISC/ibex.

[5] 2020. OpenRCX. https://github.com/The-OpenROAD-Project/OpenRCX.

[6] 2020. TritonRoute. https://github.com/The-OpenROAD-Project/TritonRoute.

[7] 2021. First Google-Sponsored MPW Shuttle Launched at SkyWater with 40 Open Source Community Submitted Designs. *Skywater Technologies* (April 2021). https://www.skywatertechnology.com/press-releases/first-google-sponsored-mpw-shuttle-launched-at-skywater-with-40-open-source-community-submitted-designs/

[8] Tutu Ajayi, David Blaauw, Tuck-Boon Chan, Chung-Kuan Cheng, Vidya A. Chhabria, David K. Choo, Matteo Coltella, Sorin Dobre, Ronald G. Dreslinski, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jiajia Li, Zhaoxin Liang, Uday Mallappa, Paul Penzes, Geraldo Pradipta, Sherief Reda, Austin Rovinski, Kambiz Samadi, Sachin S. Sapatnekar, Lawrence Saul, Carl Sechen, Vaishnav Srinivas, William Swartz, Dennis Sylvester, David Urquhart, Lutong Wang, Mingyu Woo, and Bangqi Xu. 2019. OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain. In *Proceedings of Government Microcircuit Applications and Critical Technology Conference (GOMACTech '19)*. 6.

[9] Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, Lutong Wang, Zhehong Wang, Mingyu

Woo, and Bangqi Xu. 2019. Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 76, 4 pages. https://doi.org/10.1145/3316781.3326334

[10] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[11] 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. 2016. OpenPiton: An Open Source Manycore Research Framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 217–232. https://doi.org/10.1145/2872362.2872414

[12] R. Douglas Beards and Miles A. Copeland. 1994. An Oversampling Delta-Sigma Frequency Discriminator. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 41, 1 (1994), 26–32. https://doi.org/10.1109/82.275664

[13] 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. 2008. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers (ISSCC '08)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 88–598. https://doi.org/10.1109/ISSCC.2008.4523070

[14] Brent Bohnenstiehl, Aaron Stillmaker, Jon J. Pimentel, Timothy Andreas, Bin Liu, Anh T. Tran, Emmanuel Adeagbo, and Bevan M Baas. 2017. KiloCore: A 32-nm 1000-Processor Computational Array. *IEEE Journal of Solid-State Circuits* 52, 4 (2017), 891–902. https://doi.org/10.1109/JSSC.2016.2638459

[15] Shekhar Borkar. 2005. Designing Reliable Systems from Unreliable Components: the Challenges of Transistor Variability and Degradation. *IEEE Micro* 25, 6 (2005), 10–16. https://doi.org/10.1109/MM.2005.110

[16] Pao-Lung Chen, Ching-Che Chung, and Chen-Yi Lee. 2005. A Portable Digitally Controlled Oscillator Using Novel Varactors. *IEEE Transactions on Circuits and Systems II: Express Briefs* 52, 5 (2005), 233–237. https://doi.org/10.1109/TCSII.2005.846307

[17] Matthieu Courbariaux and Yoshua Bengio. 2016. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR* abs/1602.02830 (2016). arXiv:1602.02830 http://arxiv.org/abs/1602.02830

[18] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawaj, 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 G. Dreslinski, Christopher Batten, and Michael B. Taylor. 2018. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro* 38, 2 (2018), 30–41.

[19] Kaushik De and Prithviraj Banerjee. 1994. Parallel Logic Synthesis Using Partitioning. In *International Conference on Parallel Processing*, Vol. 3. Institute of Electrical and Electronics Engineers, New York, NY, USA, 135–142. https://doi.org/10.1109/ICPP.1994.150

[20] Kaushik De, Balkrishna Ramkumar, and Prithviraj Banerjee. 1992. ProperSYN: A Portable Parallel Algorithm for Logic Synthesis. In *IEEE International Conference on Computer-Aided (ICCAD '92)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 412–416.

[21] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, Rideout V. Leo, Ernest Bassous, and Andre R. LeBlanc. 1974. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (1974), 256–268. https://doi.org/10.1109/JSSC.1974.1050511

[22] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 365–376. https://doi.org/10.1145/2000064.2000108

[23] Alex Fukunaga, Adi Botea, Yuu Jinnai, and Akihiro Kishimoto. 2017. A Survey of Parallel A*. *CoRR* abs/1708.05296 (2017). arXiv:1708.05296 http://arxiv.org/abs/1708.05296

[24] Frédéric Gessler, Philip Brisk, and Mirjana Stojilovič. 2020. A Shared-Memory Parallel Implementation of the RePlAce Global Cell Placer. In *33rd International Conference on VLSI Design and 19th International Conference on Embedded Systems (VLSID '20)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 78–83. https://doi.org/10.1109/VLSID49098.2020.00031

[25] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. 2015. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 223–238.

[26] Henry Hoffmann, David Wentzlaff, and Anant Agarwal. 2010. Remote Store Programming. In *International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC '10)*. Springer Berlin Heidelberg, Berlin, Germany, 3–17.

[27] Wen-mei Hwu and Sanjay Patel. 2018. Accelerator Architectures — A Ten-Year Retrospective. *IEEE Micro* 38, 6 (2018), 56–62. https://doi.org/10.1109/MM.2018.2877839

[28] Alexander Iosupovici. 1986. A Class of Array Architectures for Hardware Grid Routers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 5, 2 (1986), 245–255. https://doi.org/10.1109/TCAD.1986.1270193

[29] Andrew B. Kahng. 2019. Looking Into the Mirror of Open Source: Invited Paper. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '19)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942131

[30] Andrew B. Kahng, Lutong Wang, and Bangqi Xu. 2018. Tritonroute: An initial detailed router for advanced vlsi technologies. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '18)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 1–8.

[31] Andrew B. Kahng, Lutong Wang, and Bangqi Xu. 2020. The Tao of PAO: Anatomy of a Pin Access Oracle for Detailed Routing. In *57th ACM/IEEE Design Automation Conference*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 1–6. https://doi.org/10.1109/DAC18072.2020.9218532

[32] Andrew B. Kahng, Lutong Wang, and Bangqi Xu. 2020. Tritonroute: The open-source detailed router. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 3 (2020), 547–559.

[33] Matthias Köfferlein. 2018. KLayout.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 25. Curran Associates,

Inc., 9. https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[35] Chin Yang Lee. 1961. An Algorithm for Path Connections and Its Applications. *IRE Transactions on Electronic Computers* EC-10, 3 (September 1961), 346–365. https://doi.org/10.1109/TEC.1961.5219222

[36] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanović, and Krste Asanović. 2014. A 45nm 1.3 GHz 16.7 Double-Precision GFLOPS/W RISC-V Processor with Vector Accelerators. In *40th European Solid State Circuits Conference (ESSCIRC '14)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 199–202. https://doi.org/10.1109/ESSCIRC.2014.6942056

[37] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ružica Jevtić, Stevo Bailey, Milovan Blagojević, Pi-Feng Chiu, Rimas Avižienis, Brian Richards, Jonathan Bachrach, David Patterson, Elad Alon, Borivoje Nikolić, and Krste Asanović. 2016. An Agile Approach to Building RISC-V Microprocessors. *IEEE Micro* 36, 2 (2016), 8–20. https://doi.org/10.1109/MM.2016.11

[38] Yibo Lin. 2020. GPU Acceleration in VLSI Back-end Design: Overview and Case Studies. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD '20)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 1–4.

[39] Yibo Lin, Shounak Dhar, Wuxi Li, Haoxing Ren, Brucek Khailany, and David Z. Pan. 2019. DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 117, 6 pages. https://doi.org/10.1145/3316781.3317803

[40] Michael McKeown, Alexey Lavrov, Mohammad Shahrad, Paul J. Jackson, Yaosheng Fu, Jonathan Balkind, Tri Minh Nguyen, Katie Lim, Yanqi Zhou, and David Wentzlaff. 2018. Power and Energy Characterization of an Open Source 25-Core Manycore Processor. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA '18)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 762–775. https://doi.org/10.1109/HPCA.2018.00070

[41] John A. Nestor and Jeremy Lavine. 2007. L4: An FPGA-based Accelerator for Detailed Maze Routing. In *2007 International Conference on Field Programmable Logic and Applications*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 357–362.

[42] Andreas Olofsson. 2014. *Epiphany Architecture Reference* (14.03.11 ed.). Adapteva, Lexington, MA, USA. https://www.adapteva.com/docs/epiphany_arch_ref.pdf.

[43] Daniel Petrisko, Farzam Gilani, Mark Wyse, Tommy Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael B. Taylor. 2020. BlackParrot: An Agile Open Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro* 40, 4 (2020), 93–102. https://doi.org/10.1109/MM.2020.2996145

[44] Vinícius Neves Possani. 2019. *Parallel Algorithms for Scalable Logic Synthesis & Verification*. Ph.D. Dissertation. Federal University of Rio Grande do Sul.

[45] Austin Rovinski, Tutu Ajayi, Minsoo Kim, Guanru Wang, and Mehdi Saligane. 2020. Bridging Academic Open-Source EDA to Real-World Usability. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD '20)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 1–7.

[46] 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. 2019. 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 (VLSIC '19)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, C30–C31. https://doi.org/10.23919/VLSIC.2019.8778031

[47] Thomas Ryan and Edwin Rogers. 1987. An ISMA Lee Router Accelerator. *IEEE Design & Test of Computers* 4, 5 (1987), 38–45. https://doi.org/10.1109/MDT.1987.295231

[48] Daniel Sanchez and Christos Kozyrakis. 2012. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *IEEE International Symposium on High-Performance Comp Architecture (HPCA '12)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 1–12. https://doi.org/10.1109/HPCA.2012.6168950

[49] Kei Suzuki, Yusuke Matsunaga, Masayoshi Tachibana, and Tatsuo Ohtsuki. 1986. A Hardware Maze Router with Application to Interactive Rip-Up and Reroute. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 5, 4 (1986), 466–476. https://doi.org/10.1109/TCAD.1986.1270218

[50] Lalinthip Tangjittaweechai, Mongkol Ekpanyapong, Kanchana Kanchanasut, Adriano Tavares, Sung Kyu Lim, and Prabhas Chongstitvatana. 2012. Parallel VLSI Detailed Routing Using General-Purpose Computing on Graphics Processing Unit. In *9th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 1–4. https://doi.org/10.1109/ECTICon.2012.6254140

[51] Michael B. Taylor. 2012. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1131–1136. https://doi.org/10.1145/2228360.2228567

[52] Michael B. Taylor. 2018. INVITED: BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC '18)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 1–6. https://doi.org/10.1109/DAC.2018.8465909

[53] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. 2002. The RAW Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro* 22, 2 (2002), 25–35. https://doi.org/10.1109/MM.2002.997877

[54] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction (CC '02)*. Springer Berlin Heidelberg, Berlin, Germany, 179–196.

[55] Fahim ur Rahman, Greg Taylor, and Visvesh Sathe. 2019. A 1–2 GHz Computational-Locking ADPLL With Sub-20-Cycle Locktime Across PVT Variation. *IEEE Journal of Solid-State Circuits* 54, 9 (2019), 2487–2500. https://doi.org/10.1109/JSSC.2019.2926191

[56] Raja Venkateswaran and Pinaki Maxumder. 1990. A Hexagonal Array Machine for Multilayer Wire Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 9, 10 (1990), 1096–1112. https://doi.org/10.1109/43.62734

[57] Takumi Watanabe, Hitoshi Kitazawa, and Yoshi Sugiyama. 1987. A Parallel Adaptable Routing Algorithm and its Implementation on a Two-Dimensional Array Processor. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6, 2 (1987), 241–250. https://doi.org/10.1109/TCAD.1987.1270268

[58] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a Free Verilog Synthesis Suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip 21)*.

[59] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (Nov 2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114

[60] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 15–24. https://doi.org/10.1145/3020078.3021741

[61] Yuzhi Zhou, Xi Jin, and Tianqi Wang. 2020. FPGA Implementation of A* Algorithm for Real-Time Path Planning. *International Journal of Reconfigurable Computing* 2020 (August 2020). https://doi.org/10.1155/2020/8896386