The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric:

Fast Architectures and Design Methodologies for Fast Chips

Scott Davidson and Shaolin Xie
Bespoke Silicon Group

Christopher Torng and Khalid Al-Hawaj
Cornell University

Austin Rovinski and Tutu Ajayi
University of Michigan

Luis Vega and Chun Zhao
Bespoke Silicon Group

Ritchie Zhao and Steve Dai
Cornell University

Aporva Amarnath
University of Michigan

Bandhav Veluri, Paul Gao, and Anuj Rao
Bespoke Silicon Group

Gai Liu
Cornell University

Rajesh K. Gupta
University of California, San Diego

Zhiru Zhang
Cornell University

Ronald G. Dreslinski
University of Michigan

Christopher Batten
Cornell University

Michael Bedford Taylor
Bespoke Silicon Group

Rapidly emerging workloads require rapidly developed chips. The Celerity 16-nm open-source SoC was implemented in nine months using an architectural trifecta to minimize development time: 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 that can be scaled to arbitrary sizes, and a specialization tier that uses high-level synthesis (HLS) to create 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.

Emerging workloads have extremely strict energy-efficiency and performance requirements that are difficult to attain. Increasingly, we see that specialized hardware accelerators are necessary to attain these requirements. But accelerator development is time-intensive, and accelerator behavior cannot be easily modified to adapt to changing workload properties. These factors motivate new architec-

Published by the IEEE Computer Society
0272-1732/18/$33.00 ©2018 IEEE

tures that can be rapidly constructed to address new application domains, while still leveraging specialized hardware and offering high performance and energy efficiency even as applications evolve post-tapeout.

We propose a chip architecture called Celerity, meaning "swiftness of movement," that embodies an architectural design pattern called the tiered accelerator fabric (TAF). TAF minimizes time-to-market and allows the chip to maintain high performance and energy efficiency on evolving workloads.

A TAF has three key architectural tiers:

- The *general-purpose tier* is a set of OS-capable cores for executing complex codes like networking, control, and decision making.
- The *specialization tier* is made of highly specialized algorithmic accelerators to target specific computations with extreme energy-efficiency and performance requirements.
- The *massively parallel tier* is made of scalable programmable arrays of small, tightly coupled cores that attain high energy efficiency and flexibility for evolving workloads.

In response to our target application domain—autonomous vision systems—the Celerity SoC implements the general-purpose, specialization, and massively parallel tiers using five Linux-capable RISC-V cores, a binarized neural network (BNN) accelerator generated with HLS, and a "GPU-killer" 496-core RISC-V manycore array, respectively. Figure 1 shows a block diagram of Celerity highlighting the general-purpose tier in green, the specialization tier in blue, and the massively parallel tier in red. To bind these components together, we support a heterogeneous remote store programming model that allows core 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. This architecture enabled us to design and implement Celerity in only nine months through open-source and agile hardware techniques.

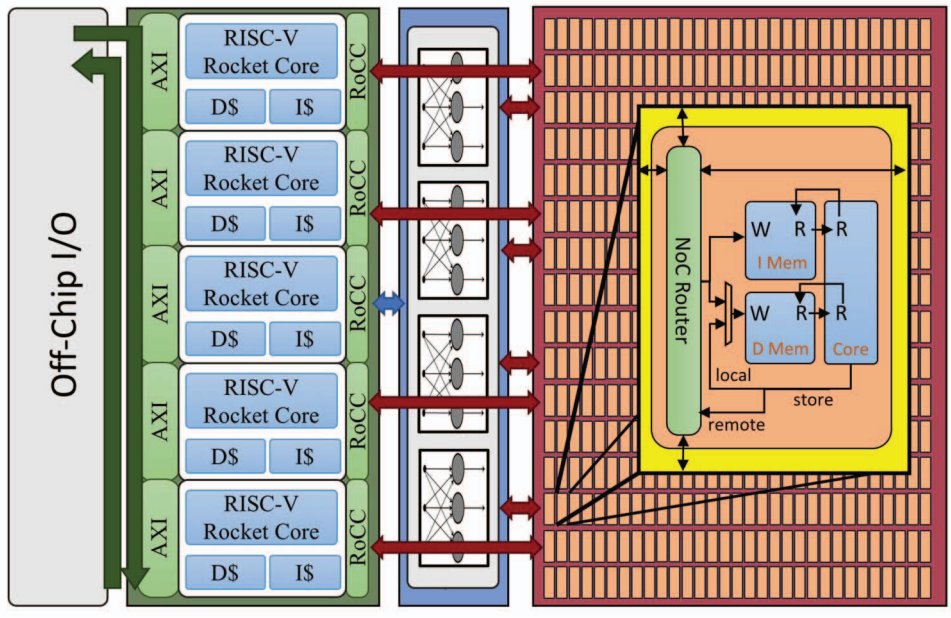

Figure 1. Celerity block diagram. The general-purpose tier (shown in green) has a five-core Rocket core complex, the specialization tier (shown in blue) has a BNN accelerator, and the massively parallel tier (shown in red) has a 496-core tiled manycore array.

Celerity is an open-source 5x5-mm tiered accelerator fabric SoC taped out in Taiwan Semiconductor Manufacturing Company (TSMC) 16-nm Fin field-effect transistor (FinFET) Compact (FFC) with 385 million transistors. In addition to the previously mentioned 501 RISC-V cores, it features an ultra-low-power 10-core RISC-V manycore array powered by an on-chip DC/DC low-dropout (LDO) regulator.[1] The 10-core array shares the same code as the larger 496-core array. The architecture has separate clock domains for I/O (400 MHz), the manycore (1.05 GHz), and the rest of the chip (625 MHz). Figure 2(a) shows the SoC's floorplan image from our CAD tools. Figure 2(b) shows the layout of Celerity. Finally, Figure 2(c) shows a photomicrograph. The design's entire source base is available at http://opencelerity.org. See the sidebar, "Achieving Celerity with Celerity," for the methodologies used to design and tapeout the Celerity chip in less than nine months.

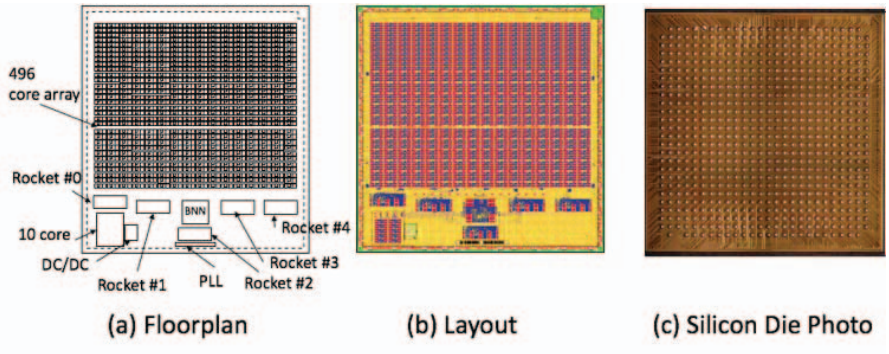

Figure 2. Detailed Celerity images. The floorplan (a) shows the relative sizes and positions of the various blocks in the SoC. The layout (b) shows the physical chip attributes where the red area represents SRAM, the blue area represents logic, and the yellow area represents interconnection between blocks. The silicon die photo (c) shows the real chip taken from a photomicrograph.

THE CELERITY ARCHITECTURE

When addressing emerging application domains with a tiered accelerator fabric, a number of key decisions must be made. The specialization tier is among the most important, because it is the most integral in determining the chip's super-capabilities, and it requires the most effort to design. The choice of general-purpose tier will be determined by feature set (for example, security, debugging features, or raw irregular computation) but also by the availability and expense of processor IP cores. ARM offers many variants, but low-non-recurring-engineering (NRE) open-source versions of RISC-V are becoming available, like the Berkeley Rocket processor core used in our design. The massively parallel tier could be comprised of ARM or Advanced Micro Devices (AMD) GPU IPs. Alternatively, our open-source tiled manycore architecture, BaseJump Manycore, is free and allows for fast and flexible scaling from one to 1 million+ cores, at an area cost of 1 mm^2 per 42 cores in 16 nm. We explore each tier in the following sections, but first discuss how these components are tied together.

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. 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 in Celerity was configured for 512 coordinates (x = 0..15, y = 0..31) and 22-bit addresses. The manycore'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.

While remote loads, such as those found in the Adapteva parallel architecture,[2] 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 manycore or 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.

THE GENERAL-PURPOSE TIER

For our SoC to support complex software stacks, exception handling, and memory management, we instantiated five Berkeley RISC-V Rocket cores running the RV64G ISA. The Rocket core is an open-source,[3] five-stage, in-order, single-issue processor with a 64-bit pipelined FPU and size-configurable non-blocking caches. Each Rocket core can run an independent Linux image. This gives us the flexibility to run SPEC-style applications and network stacks like TCP/IP. Four Rocket cores connect directly to the massively parallel tier using four 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 done 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, do remote stores to other Rocket cores, to any manycore, 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.

THE MASSIVELY PARALLEL TIER

To achieve massive amounts of programmable energy-efficient parallel computation, we wanted an architecture with a high density of physical threads per area. Therefore, we implemented a 496-core tiled manycore array[4] that interconnects low-power RISC-V Vanilla-5 cores using a mesh interconnection network. Each tile contains a simple router and a Vanilla-5 core. Our in-house-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,[5] giving us a highly programmable array to maintain high performance as workloads evolve post-tapeout. A key contribution of our work is to extend the remote store programming model to incorporate heterogeneous processor types and to support fast producer-consumer synchronization.

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 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, running at 1 GHz, 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.

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.

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.

Token Queue

Our design shows that tight producer-consumer synchronization can be layered on top of remote store programming. By using the LR-LBR instruction extension, we implemented the 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 that is done, 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 LR-LBR 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.

Programming Models

The token queue and remote store programming models are particularly well suited for programming with the StreamIt[6] programming model. We are also investigating libraries that will enable CUDA-style applications to be ported more easily, but emphasizing an execution model that is

better able to leverage the inherent locality in parallel computation rather than using double data-rate type five synchronous graphics random-access memory (GDDR5) DRAM as the primary communication mechanism between cores.

THE SPECIALIZATION TIER

Deciding which workload parts get implemented in the specialization tier takes careful consideration. In Celerity, we chose to implement a BNN accelerator. The architecture and reasoning for implementing a BNN in the specialization tier are discussed here.

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 our own prior work on FPGA-based CNN accelerators.[7] Given this context, we chose to use flexible image recognition as a case study for demonstrating the potential of tiered accelerator fabrics in general, and the Celerity SoC specifically.

Most prior work on CNN accelerators uses carefully hand-crafted digital VLSI 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.[8] 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 gain (8-16X) 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.

We employ the specific BNN model shown in Figure 3(a) based on Courbariaux et al.[8] This model includes six convolutional, three max-pooling, and three dense (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-percent accuracy on the CIFAR-10 dataset.

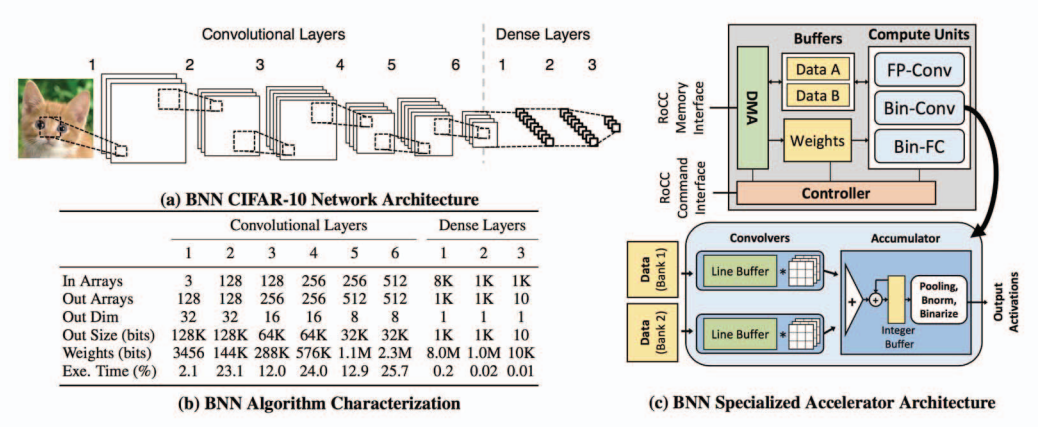

Figure 3. BNN accelerator.

Performance Target

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

Creating and Optimizing the Specialization Tier

We use a three-step process to map applications to tiered accelerator fabrics. First, we implement the algorithm using the general-purpose tier for initial workload characterization and to identify key kernels for acceleration. Second, we can choose to accelerate the algorithm using either the specialization tier or the massively parallel tier. Finally, we can further improve performance and/or efficiency by cooperatively using both the specialization tier and the massively parallel tier.

Establishing the Functionality of the Specialization Tier

In the first step, we implemented the BNN using the general-purpose tier to characterize the computational and storage requirements of each layer. Figure 3(b) 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 200X slower than the performance target. Although the binarized convolutional layers require more than 97 percent 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.

Designing the Specialization Tier

In the second step, we implemented the BNN using 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 3(c) 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.[7]

Combining the Massively Parallel and Specialization Tiers

In the third step, we explored the potential for cooperatively using both the specialization tier and the massively parallel tier. Our early analysis suggested that repeatedly loading the weights from off-chip would significantly impact both performance and energy efficiency. We implemented a novel mechanism that enables cores in the massively parallel tier to send data directly to the BNN. To classify a stream of images, we first load all data memories in the massively parallel tier with the binarized weights. We then repeatedly execute 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.

The Benefits of HLS

We employed HLS to accelerate time-to-market 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.

PERFORMANCE ANALYSIS OF THE SPECIALIZATION TIER

Table 1 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 performance/Watt by more than 10X compared to our prior FPGA implementation[7] and more than 100X compared to a mobile GPU.

Table 1. Performance comparison of optimized BNN implementations on different platforms.

		GPT	SpT	SpT+MPT	mGPU [6]	CPU [6]	GPU [6]	FPGA [6]
Runtime (ms)		4,024.0	20.0	3.2	90.0	14.8	0.7	5.9
Performance (images/sec)		0.3	50.0	312.5	11.1	67.6	1428.6	168.4
Power (Watts)		0.5	0.5	1.9	3.6	95	235.0	4.7
with aggressive clock gating		0.1	0.2	0.4				
Efficiency (images/J)		0.5	100.0	164.5	3.0	0.7	6.0	35.8
with aggressive clock gating		2.5	250.0	625.0				
Relative Efficiency	**vs. GPT**	1×	100×	250×				
	vs. mGPU			208×	1×	0.2×	2×	12×

*GPT = general-purpose tier. SpT = specialization tier with the weights stored in the general-purpose tier's cache. SpT + MPT = specialization tier with the 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.

In the table, runtimes measure processing a single image from the CIFAR-10 dataset. The power of GPT, SpT, and SpT + MPT is estimated using post-place-and-route gate-level simulations with limited clock-gating, as provided in the Celerity SoC (only gating the entire MPT when unused). Aggressive clock-gating assumes an alternate design that can gate unused cores/accelerators in the GPT, SpT, and MPT. Celerity SoC power estimates do not include DRAM power.

NEW DIRECTIONS FOR FAST HARDWARE DESIGN

Our research examines the speedy construction of new classes of chips in response to emerging application domains. Our 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, our 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, our agile chip development techniques enabled us to quickly tape out a 16-nm design with a team of graduate students geographically distributed across the US. Each approach targets the key goal of creating new classes of chips quickly and with low budgets. We hope that the lessons from

our experience will inspire new classes of chips, unlocking the creativity of future students, architects, and chip designers alike.

SIDEBAR: ACHIEVING CELERITY WITH CELERITY: FAST DESIGN METHODOLOGIES FOR FAST CHIPS

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. Our 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 (Batten/Zhang's team designed the specialized tier, Taylor's team designed the general-purpose and massively parallel tiers, and Dreslinski's team implemented the 16-nm CAD flow; all three teams contributed to physical design, with Dreslinski leading.)
- Developing an implementation flow for an advanced 16-nm FinFET node
- Satisfying CRAFT program constraints with only $1.3 million USD for NRE costs

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

Reuse

Reuse for hardware design accelerates both design and implementation time, as well as testing and verification time. For Celerity, we made heavy reuse of open-source designs and infrastructures. We leveraged the Berkeley RISC-V Rocket core generator[3] 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, we 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[9] for SystemVerilog, the BaseJump SoC framework, BaseJump Socket, BaseJump Motherboard, BaseJump FPGA bridge, and BaseJump FMC bridge, as seen in Figure 4. In Celerity, we built all of our 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 test-bench 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.

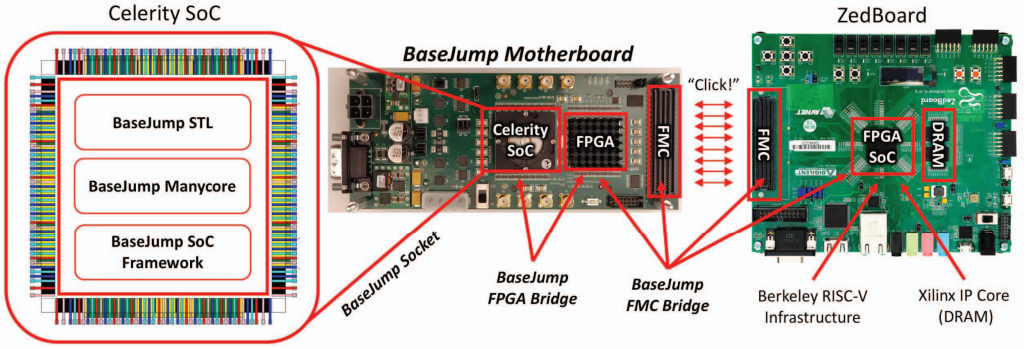

Figure 4. BaseJump open-source hardware components. The SoC framework, including NoC and high-speed off-chip double data-rate (DDR) interface, were implemented using STL, as was the manycore. The fabricated chip conforms to the socket definition and is placed in the motherboard's socket. The motherboard 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.

Modularization

One key challenge for this project was that our design teams were spread across four physical locations. Fine-grained synchronization between teams was not feasible, so we developed techniques to modularize both our design interfaces on chip 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[10] 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.

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 low-level SystemVerilog. In our BNN accelerator development, we used HLS to drastically improve design space exploration and implementation time.

ACKNOWLEDGMENTS

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. We thank Synopsys, Cadence, and Mentor for electronic design automation (EDA) tool donations, ARM for physical IP donations, and Xilinx for both EDA tool and FPGA development board donations. We acknowledge the University of California, Berkeley's contributions to forming the RISC-V ecosystem, for RoCC, and for developing the Rocket chip SoC generator. We thank Ian Galton, Patrick Mercier, Loai Salem, and Julian Puscar for their work on the analog subsystems of the chip. We thank the many contributors to the open-source RISC-V software and hardware ecosystem.

REFERENCES

1. T. Ajayi et al., "Celerity: An Open Source RISC-V Tiered Accelerator Fabric," *Hot Chips: A Symposium on High Performance Chips*, 2017.
2. A. Olofsson, *Epiphany Architecture Reference Manual*, 2008; www.adapteva.com/docs/epiphany_arch_ref.pdf.
3. *Rocket Chip Generator*; https://github.com/freechipsproject/rocket-chip.
4. M.B. Taylor et al., "The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs," *IEEE Micro*, 2002.
5. H. Hoffmann et al., "Remote Store Programming: Mechanisms and Performance," *HiPEAC*, 2010.
6. *StreamIt*; http://groups.csail.mit.edu/cag/streamit/.
7. R. Zhao et al., "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs," *FPGA*, 2017.
8. M. Courbariaux et al., "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1," arXiv preprint, 2016; https://arxiv.org/abs/1602.02830.
9. M.B. Taylor, "BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design," *Design Automation Conference*, 2018.
10. Y. Lee et al., "An agile approach to building risc-v microprocessors," *IEEE Micro*, vol. 36, no. 2, 2016.

ABOUT THE AUTHORS

Scott Davidson is a PhD student in the Bespoke Silicon Group's University of Washington division. Contact him at stdavids@cs.washington.edu.

Shaolin Xie is a research scientist in the Bespoke Silicon Group's University of Washington division. Contact him at shaolin@cs.washington.edu.

Christopher Torng is a PhD student at Cornell University. Contact him at clt67@cornell.edu.

Khalid Al-Hawaj is a PhD student at Cornell University. Contact him at ka429@cornell.edu.

Austin Rovinski is a PhD student at the University of Michigan. Contact him at rovinski@umich.edu.

Tutu Ajayi is a PhD student at the University of Michigan. Contact him at ajayi@umich.edu.

Luis Vega is a PhD student scientist in the Bespoke Silicon Group's University of Washington division. Contact him at vegaluis@cs.washington.edu.

Chun Zhao is a postdoctoral researcher in the Bespoke Silicon Group's University of Washington division. Contact him at chunzhao@uw.edu.

Ritchie Zhao is a PhD student at Cornell University. Contact him at rz252@cornell.edu.

Steve Dai is a PhD student at Cornell University. Contact him at sxdai@sandia.gov.

Aporva Amarnath is a graduate student at the University of Michigan. Contact him at aporvaa@umich.edu.

Bandhav Veluri is a graduate student in the Bespoke Silicon Group's India location. Contact him at bandhav.veluri00@gmail.com.

Paul Gao is a PhD student in the Bespoke Silicon Group's University of Washington division. Contact him at gaozihou1@gmail.com.

Anuj Rao is a master's student in the Bespoke Silicon Group's University of California, San Diego division. Contact him at anr044@eng.ucsd.edu.

Gai Liu is a graduate student at Cornell University. Contact him at gl387@cornell.edu.

Rajesh K. Gupta is a professor of computer science and engineering at the University of California, San Diego. Contact him at rgupta@ucsd.edu.

Zhiru Zhang is a professor of electrical and computer engineering at Cornell University. Contact him at zhiruz@cornell.edu.

Ronald G. Dreslinski is a professor at the University of Michigan. Contact him at rdreslin@umich.edu.

Christopher Batten is a professor of electrical and computer engineering at Cornell University. Contact him at cbatten@cornell.edu.

Michael Bedford Taylor is the leader of the Bespoke Silicon Group and a jointly appointed professor in the Departments of Computer Science and Engineering and Electrical Engineering at the University of Washington. Contact him at profmbt@uw.edu.