The Celerity Open Source RISC-V Tiered Accelerator Fabric:
Fast Architectures and Design Methodologies for Fast Chips

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

To address these concerns, we propose a chip architecture called Celerity, meaning “swiftness of movement”, that embodies an architectural template that minimizes time to market and allows the chip to maintain high performance and energy efficiency on evolving workloads.

Celerity has three key architectural tiers. First, it has the general purpose tier, a set of OS-capable cores for executing complex codes like networking, control, and decision making. Second it has the specialization tier, highly specialized algorithmic accelerators that have been rapidly developed with high-level synthesis (HLS), and that target specific computations with extreme energy-efficiency and performance requirements. Finally, it has the massively parallel tier, scalable programmable arrays of small tightly-coupled cores that attain high energy-efficiency and flexibility for evolving workloads. We term this architectural trifecta a tiered accelerator fabric.

In response to our target application domain -- quadcopter vision systems -- the 16nm Celerity SoC features 5 Linux-capable RISC-V cores, a 496-core RISC-V manycore array, and a deep neural network accelerator. To bind these components together, we support a heterogeneous remote store programming model that allows any core or accelerator to write to any other core’s 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, including RAM. We designed and implemented Celerity in only 9 months through open-source and other agile hardware techniques.

Celerity is an open-sourced 5x5mm tiered accelerator fabric SoC taped out in TSMC 16nm FFC with 385 million transistors comprising five Linux capable RISC-V cores, a 496-core manycore array, one binarized neural network unit, and an ultra low-power ten core RISC-V manycore array powered by an on-chip DC/DC LDO [1]. The architecture has separate clock domains for I/O (400 MHz), the manycore (1.05 GHz), and the Rocket and BNN complex (625 MHz). Figure 1 shows a block diagram of Celerity highlighting the general purpose tier in green, the massively parallel tier in red, and the specialization tier in blue. Figure 2(a) shows the layout of Celerity. Figure 2(b) is an image from the CAD tools showing the actual SoC floorplan. Finally, Figure 2(c) shows a photomicrograph. The design’s entire sourcebase 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 9 months.
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 cycle time for taping out a complex SoC. Our team designed and taped out Celerity in just 9 months from PDK access, which included:

- Coordinating grad students spread across 4 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 16nm CAD flow; all three teams contributed to physical design with Dreslinski leading.)
- Developing an implementation flow for an advanced 16nm FinFET node
- Satisfying CRAFT program constraints with only $1.3 million USD for non-recurring engineering costs

To meet the aggressive schedule for Celerity, we developed 3 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. For example, we leveraged the Berkeley RISC-V Rocket Core generators to implement the SoC’s general purpose tier, allowing the use 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 Rocket custom coprocessor (RoCC) interface to connect the general purpose tier to the other two tiers. As part of our learning process with RoCC, we created a user guide for others, 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 supporting infrastructure and frameworks for building SoCs, including the Basejump STL Library for SystemVerilog, the Basejump SoC Framework, packaging, and testing motherboards. In Celerity, we built all of our RTL using the Basejump STL and SoC Framework’s pre-validated components which include a unit testing suite. We ported the BaseJump’s I/O socket standard to the CRAFT flip-chip package, and will use the BaseJump Motherboard for the final chip.

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 manycore array from 400 cores to 496 to absorb free die area. By changing just 9 lines of code, we could fully synthesize, place, route, and sign off the new design in a span of 3 days.

Modularization
One key challenge for this project was that our design teams were spread across 4 different physical locations. Because any fine-grain synchronization between teams could cause frequent stalls of work, 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 deadlines. These are designs that we could, in theory, tape out 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 tape out.

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.

Modularization’s key benefit is to greatly reduce team member synchronization and more efficiently utilize their time, which was critical when working with a relatively small design team and tight time constraints.

**Automation**

CRAFT program’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 signoff. 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 testbench 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. We also used a digital design flow to create our PLL and LDO, to speedup component implementation and iteration time as target specs evolved.

**The Celerity Architecture**
When addressing an emerging application domain with a tiered accelerator fabric, a number of key decisions must be made. Clearly, the specialization tier is among the most important, because it is the most integral in determining the chip’s super-capabilities, and 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 ops) but also by the availability and expense of processor IP cores. ARM offers many variants, but low-NRE open source versions of RISC-V are now becoming available, like the Berkeley Rocket processor core used in our design. The massively parallel tier could be comprised of ARM or AMD GPU IPs. Alternatively, our open-source tiled manycore architecture is free and allows for fast and flexible scaling from one to one million cores, at an area cost of 1 mm² per 40 cores in 16nm. We explore each tier in the following three sections.

Figure 1: Celerity block diagram - The green section shows the general purpose tier with a 5-core Rocket core complex, the red section shows the massively parallel tier with a 496-core tiled manycore array, and the blue section shows the specialization tier with binarized neural net accelerator.
The General Purpose Tier

For our SoC to support complex software stacks, exception handling, and memory management, we instantiated 5 Berkeley RISC-V Rocket Cores running the RV64G ISA. The Rocket Core is an open-source [2], 5-stage, in-order, single-issue processor with 64-bit pipelined FPU and size-configurable non-blocking caches that runs Linux. This gives us the flexibility to run SPEC-style applications and network stacks like TCP/IP. Four of the Rocket Cores connect directly to the massively parallel tier using parallel links and one connects directly to the specialization tier. These connections are made using the Berkeley Rocket custom coprocessor (RoCC) interface. L1 Data and Instruction caches are configured at 16 KB each.

Four of the Rocket cores can inject packets directly into the manycore through separate network links. One RoCC instruction sends packets for remote stores to manycore instruction and data memories, and another instruction sends packets for writing to a manycore’s configuration address space, including freeze registers, and arbitration policy. One input to the RoCC instruction is the data to write, and the other input is the address information.

The manycore can also send remote stores directly into the four Rocket cores’ caches, potentially causing cache misses to DRAM. Remote store addresses are translated using a segment address register which maps the 22-bits address space into the Rocket’s 40-bit address space.

The Massively Parallel Tier

To achieve massive amounts of programmable energy-efficient parallel computation, we wanted to use an architecture with a high density of physical threads per area (approximately 40 per mm^2). Therefore, we implemented a 496-core tiled manycore array [3] 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 5-stage, in-order, single-issue processors.
with 4KB instruction and data memories that use the RV32IM ISA. The manycore uses a strict remote store programming model [4] giving us a highly programmable array that allows us 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.

The processor’s store address MSB determines if the store will be local (0) or remote (1). For remote stores, the next set of nine bits will be used to determine the destination tile XY coordinate, and the remaining bits are used for the local address in that tile. This allows both local and remote stores to use the same standard store word, halfword and byte instructions from the ISA. While remote loads, such as found in the Adapteva Parallela architecture [5], could arguably make the manycore array more programmable, they have high round-trip latency costs and lead users astray by offering a high-convenience, low-performance HW mechanism. Remote stores do not incur such a latency penalty because they are pipelined and can therefore be issued once per cycle.

The massively parallel tier has parallel network connections to each Rocket core as well as multiple connections to the BNN in the specialized tier; and all of these components communicate through the manycore’s NOC using remote stores.

**NOC Design.** The manycore’s mesh NoC design targets extreme area efficiency, using only a single physical network for data transfer, no virtual channels, single word / single-flit packets, deterministic XY dimension-ordered routing, and 2-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-bits 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, a core performs a store to a memory address whose XY coordinate is beyond the coordinates of manycore.

When a remote store is performed, a local credit counter will be decremented on a tile. 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 is used to detect whether any outstanding stores exist, allowing a core to pause for memory traffic to finish during a barrier.

**Load-on-broken-reserve.** The manycore features an extension to the load-reserve, store-conditional (LR-SC) atomic instructions called load-reserve, load-on-broken-reserve (LR-LBR). Load-reserve 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. **Load-on-broken-reserve (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 load the target address. Typically user code will check a ready flag with LR, branch away if it is ready, and otherwise fall through to a LBR to wait for it to become ready.
**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 library 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, much like an clock-domain-crossing FIFO.

The producer tile starts by confirming 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. Then it will send the corresponding data via remote stores. After that is done, the producer will update the head pointers via local and remote stores.

The consumer starts by confirming 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 then proceed back to consuming the next set of tokens.

The Specialization Tier

Deciding which workload parts get implemented in the specialization tier takes careful consideration. In Celerity, we chose to implement a binarized neural network (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 [6]. 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 CNNs requires large off-chip memories to store fixed-point weights and activations and carefully hand-crafted digital VLSI architectures. Recent work on binarized neural networks (BNNs) have demonstrated that binarized weights and activations (i.e., +1, -1) can, in certain cases, achieve accuracy comparable to full-precision floating-point CNNs [7]. BNN’s key benefit is that weights can be stored on-chip instead of in DRAM, saving energy.
We employ the specific BNN model shown in Figure 4(a) based on Courbariaux et al. [7]. 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% accuracy on the CIFAR-10 dataset. We target ultra-low latency, requiring a batch size of one image, and a throughput target of 60 classifications/second to enable real-time operation.

Three steps to 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 4(b) shows the number of dynamic instructions, binary weights, and binary activations per layer. Even assuming a very optimistic embedded microarchitecture capable of sustaining one instruction/cycle, executing the entire BNN on the general-purpose tier would be more than 200× slower than the performance target. Although the binarized convolutional layers require over 97% of the dynamic instructions, preliminary analysis suggests all nine layers must be accelerated in order to meet the performance target. The storage requirements for activations are relatively modest, but the storage requirements for weights are still non-trivial and will 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 Rocket custom coprocessor (RoCC) interface. Although the massively parallel tier could be used to implement the BNN at speed, superior energy-efficiency could be attained via specialization. Figure 4(c) shows the BNN accelerator architecture. The BNN accelerator consisted of modules for fixed-point convolution (i.e., 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. 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, but in our initial 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 [6].

**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 use their remote store programming model 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 composed with 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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (ms)</td>
<td>4.024</td>
<td>20.1</td>
<td>3.3</td>
<td>90.0</td>
<td>14.8</td>
<td>0.7</td>
<td>5.9</td>
</tr>
<tr>
<td>Performance (images/sec)</td>
<td>0.3</td>
<td>49.8</td>
<td>303.0</td>
<td>11.1</td>
<td>67.6</td>
<td>1428.6</td>
<td>168.4</td>
</tr>
<tr>
<td>Power (Watts)</td>
<td>0.2–0.5</td>
<td>0.2–0.5</td>
<td>0.5–2.0</td>
<td>3.6</td>
<td>95</td>
<td>235.0</td>
<td>4.7</td>
</tr>
<tr>
<td>Performance / Power</td>
<td>0.5–1.2</td>
<td>99.5–248.8</td>
<td>151.5–606.1</td>
<td>3.0</td>
<td>0.7</td>
<td>6.0</td>
<td>35.8</td>
</tr>
<tr>
<td>Improvement vs. GPT</td>
<td>1.0×</td>
<td>199.0–200.6×</td>
<td>303.0–488.7×</td>
<td>4.9×–196.0×</td>
<td>1.0×</td>
<td>0.2×</td>
<td>2.0×</td>
</tr>
</tbody>
</table>

Table 1: Performance Comparison of Optimized BNN Implementations on Different Platforms – GPT = general-purpose tier; SpT = specialization tier; MPT = massively parallel tier; mGPU = NVIDIA Jetson TK1 embedded GPU board; CPU = Intel Xeon E5-2640; GPU = NVIDIA Tesla K40; FPGA = Xilinx Zynq-7000 SoC. Values denoted with * are ranges: the lower bound of the range estimates the value of a non-clock-gated design, while the higher bound of the range estimates the value of an aggressively clock-gated design.
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 over 10× compared to our prior FPGA implementation [6] and over 100× compared to a mobile GPU.

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 like Rocket and Basejump for key IP blocks, fast motherboard and socket development and FPGA firmware, principled SystemVerilog parameterized component libraries like Basejump STL. Finally, our agile chip development techniques enabled us to quickly a 16nm 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.

References


