

# Beethoven: A Heterogeneous Multi-Core Accelerator System Composer

Chris Kjellqvist  
Dept. of Computer Science  
Duke University  
Durham, NC, USA  
cmk91@duke.edu

Brendan Peercy  
Dept. of ECE  
Duke University  
Durham, NC, USA  
brendan.peercy@duke.edu

Alvin R. Lebeck  
Dept. of Computer Science  
Duke University  
Durham, NC, USA  
alvy@cs.duke.edu

Lisa Wu Wills  
Dept. of Computer Science  
Duke University  
Durham, NC, USA  
lisa@cs.duke.edu

**Abstract**—Hardware Development is challenging in large part due to the complexity of incorporating realistic designs onto hardware devices (e.g., FPGAs, CGRAs, ASICs). This work proposes a multi-core, hardware-software accelerator design framework called Beethoven. Beethoven provides a flexible and reusable many-core accelerator System-On-Chip integration environment through programming abstractions for Register-Transfer Logic development, generation of software linkage between the host system and the accelerator, and provision of a host software runtime. We thoroughly evaluate Beethoven on microbenchmarks, Mach-Suite, and a compute-limited attention accelerator. We compare Beethoven generated accelerated system performance to High-Level Synthesis generated and hand-written RTL accelerators and show that Beethoven provides on-par or better-performing systems with marginal overheads. Beethoven is open-source and available at <https://github.com/Composer-Team/Beethoven>.

**Index Terms**—Hardware acceleration, Hardware design languages, Accelerator architectures, Computer architecture

## I. INTRODUCTION

Specialized hardware accelerators are a popular approach to providing high-performance computations in the post-Moore’s law era. The design and deployment of accelerators are incredibly challenging, and many industrial and academic efforts have been made to ease the burden of these challenges. However, despite these efforts, the learning curve is still steep, and deploying accelerator designs remains impractical, except in cases of overwhelming need. These burdens require significant time and expertise to overcome.

The challenges of accelerator design can be divided into two general themes. First, the traditional programming paradigm for hardware circuits, called Register-Transfer Logic (RTL) or Hardware Description Language (HDL)<sup>1</sup>, is unintuitive, error-prone, and difficult to debug. There is a large body of work that addresses this challenge, the most widely used of these being High-Level Synthesis (HLS) techniques [4], [5], [11], [16], [17], [43], [49], which compile high-level software, usually C++, into RTL designs. However, HLS requires expert knowledge to achieve optimal results, and the software-to-hardware translation process induces performance, area, and power overheads that are difficult, if not impossible, to diagnose and completely optimize away. Nevertheless, many HLS designs

are very efficient and, in such cases, more productive than a comparable RTL design. Whether one chooses HLS, one of the plethora of HLS-like approaches, or RTL programming is largely a matter of personal preference and tolerance for potential overheads.

The second group of challenges concerns the integration of a hardware circuit onto an actual device. Devices are often complex and vary greatly from one another. There are different classes of devices: Field-Programmable Gate Arrays (FPGAs), Course-Grained Reconfigurable Architectures (CGRAs), and Application-Specific Integrated Circuits (ASICs). Each class (or even devices within a single class) has entirely different resources, structures, and tool flows. Integrating a hardware design onto a device requires mapping the algorithm onto the available hardware resources and interfaces. The complexity of this transformation can depend greatly on the specific algorithm and the specific device and will likely require a significant rewrite to be functional on another device. In practice, this makes most RTL circuit designs non-portable.

In addition, it is typical to instantiate multiple independent copies of the same circuit (or accelerator *cores*) on a device to increase user design throughput and performance. However, adding accelerator cores makes system integration more complex for the increase in throughput. While an additional area on a device may be reserved for an accelerator core, it may be spatially far away from the external interfaces or on another silicon die entirely, making interconnect/System-on-Chip (SoC) design and timing closure more difficult. Constructing an SoC that is scalable with many accelerator cores, each scattered over or throughout different silicon dies, is challenging; doing so in a way that is also flexible to reconfiguration for different accelerators and devices is even more so.

An additional side-effect of device-related challenges is irreproducibility. Many published hardware accelerators are never tested on real devices or otherwise deployed and, instead, are only synthetically analyzed using simulator test benches and synthesis tools or in-house infrastructure that is often never made public, even if the hardware accelerator design is. Although unfortunate, it is an expected consequence because demonstrated deployment, portability, and ease-to-reproduce often add no novelty or measurable value to an academic result

<sup>1</sup>We use the terms RTL and HDL interchangeably in this paper as they refer to the same hardware design methodology.

but only make it more difficult for others to build on or utilize that work.

Although the complexity of these devices presents a significant engineering hurdle and a large learning curve, this complexity is critical for their functionality, flexibility, and performance. However, direct management of this complexity is unnecessary for the underlying accelerator to perform well. For instance, off-chip memory protocols like DDR and HBM are industry-standard and present on most high-performance devices to provide high memory bandwidth using few IO pins. Instead of manually managing these arcane protocols, it is common to use a DDR/HBM controller provided by the device vendor or a third party. This controller manages the off-chip communication while providing the user with a more straightforward interface (e.g., AXI) with negligible performance overhead. The idea of general-purpose protocol IPs has been extended further in recent works to *hardware shells*: an IP block, often supporting multiple devices, that manages all off-chip communication and provides a set of fixed interfaces for the user’s design. While this approach is reusable and improves design portability, the programmer must still map the general-purpose interfaces to the memory and control interfaces for their accelerator core(s), and it can be challenging to modify a third-party shell for a new device.

To alleviate the abovementioned challenges, we propose a multi-core, hardware-software accelerator design framework called Beethoven. Beethoven provides a flexible and reusable accelerator SoC integration environment through “separation of concerns” between accelerator designers and platform developers. Accelerator designers use Beethoven programming interfaces to structure their many-core design and communicate from their accelerator *core* to host memory and other cores. Beethoven’s internal fabric connects to the accelerator core design, constructs device-aware networks, and then presents a device and accelerator-agnostic set of interfaces to the platform developer, who integrates the internal fabric and external interfaces. The connection between Beethoven and the device is reusable across accelerator designs. It allows the accelerator designer to aggressively utilize device resources without expert knowledge of the underlying protocols or device.

We make the following contributions:

- 1) We develop a framework for composing a heterogeneous multi-core accelerator SoC that provides programming abstractions for host-to-accelerator, accelerator-to-memory, and accelerator-to-accelerator communication. Beethoven generates a C++ host-accelerator software interface and provides an FPGA management runtime. Beethoven is tuned to support multiple FPGA target platforms (both data center and edge) as well as ASIC development and technology libraries.
- 2) We thoroughly evaluate Beethoven on microbenchmarks, MachSuite, and a compute-limited attention accelerator. We compare Beethoven-generated accelerated system performance to HLS-generated and hand-written

HDL accelerators and show that Beethoven provides on-par or better-performing systems.

## II. BEETHOVEN

This section details the hardware language abstractions and software systems that Beethoven uses to generate multi-core accelerator architectures and platform-agnostic hardware/software integration. We introduce several structural (Section II-A) and communication (Section II-B) abstractions for designing an accelerator. The developer interacts with their accelerator in software using a cross-platform software runtime and library (Section II-C). Beethoven supports a variety of hardware and host deployments. We enumerate our currently supported platforms in Section II-D.

### A. Beethoven Structure

At the lowest level of the Beethoven hierarchy is a custom functional unit that the developer implements, called a *Core*. A Core can implement arbitrary functionality, ranging from a simple vector add to a matrix multiply pipeline to a systolic array-based neural network inference accelerator. A Core acts independently from other cores and is the part of the user’s design that interacts with external components (e.g., host, memory). The developer instantiates multiple identical Cores into a group called a *System*. The developer may instantiate multiple Beethoven Systems if they desire multiple functions on their accelerator. This hierarchy is shown in Figure 1. Note that the developer only has to supply the accelerator Core logic (shaded light purple boxes in the diagram) while Beethoven generates the rest of the accelerated system.

The host communicates with the Cores through the command subsystem (blue arrows in Figure 1(a)). The commands are communicated using the Rocket Custom Co-processor (RoCC) instruction format – an extension to the RISC-V ISA for accelerators developed by the RocketChip project [2]. Using this format, Beethoven designs can integrate with any RISC-V systems that support the RoCC extensions. Commands are sent from the host to the accelerator over a Memory-Mapped IO (MMIO) interface to the MMIO Command/Response System, which converts the system bus protocol into RoCC instructions. Instructions contain routing information specifying its intended Core and System. To support more complex program flows, Beethoven also allows Cores to communicate with each other.

Beethoven manages the memory subsystem for the developer. Within a Beethoven Core, the user may instantiate any number of access points (e.g., Readers and Writers) into the memory subsystem, described fully in Section II-B. These access points are routed through a TileLink network-on-chip to an external memory controller shown as blue and orange arrows in Figure 1(a).

The Beethoven hierarchy represents the logical design structure of an accelerator, although the developer’s target platform may exhibit certain physical properties that make this structure untenable. In particular, larger FPGAs may be constructed from multiple dies, also called Super Logic Regions (SLRs).

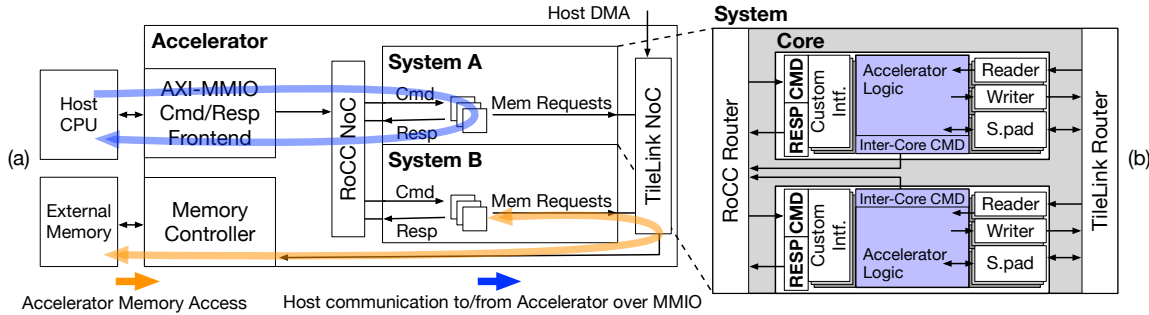


Fig. 1: The structure of a Beethoven-accelerated system. (a) The high-level system diagram. The host sends RoCC commands/responses to the accelerator via the AXI-MMIO command/response system. (b) Accelerator cores interact with the host and other cores via command/response IO and with memory using Readers, Writers, and Scratchpads.

Routing within an SLR is cheap and abundant, whereas routing between SLRs is expensive and limited. When applicable, Beethoven builds SLR-aware command and memory routing networks and maps Cores onto SLRs. This alleviates the user from performing complex floorplanning for large designs and allows accelerators to scale to many Cores easily.

### B. Beethoven Hardware Language Abstractions

We design Beethoven using the Chisel Hardware Description Language [3] and allow developers to tailor the integration between their accelerator and the device in two primary ways: memory stream abstractions and command abstractions.

**Memory Stream Abstractions** HLS tools provide access to external memory by allowing developers to dereference pointers passed into the hardware kernel via function arguments. Although the function pointers reference logically separate allocations in memory, the memory transactions are mediated over a single memory bus. Beethoven takes inspiration from this and provides a set of memory primitives that developers can use to manage logically separate memory streams within their program. While other communication primitives exist (e.g., strided memory access, networking), Beethoven’s implementation does not preclude their addition, and we chose the current offerings as they cover the most common cases. In the appendix, we provide a comprehensive set of the hardware communication interfaces Beethoven currently provides. Under the hood, Beethoven generates a memory subsystem that manages and connects these primitives to external memory. These primitives are called *Readers*, *Writers*, and *Scratchpads*.

Developers can instantiate these primitives however they please: an arbitrary number of them and with data widths, prefetching behavior, and buffer sizes suited to their specific accelerator.

**Readers and Writers** Developers instantiate *Readers/Writers* within their accelerator Core definition to stream data from/to memory. If multiple logical streams are necessary, multiple Readers or Writers can be instantiated. While the platform may have device-specific limitations (e.g., it supports only a subset of a full protocol), the developer does not have to

```
class MyAccelerator(implicit p: Parameters)
  extends AcceleratorCore {
  val io = BeethovenIO(new AccelCommand("my_accel") {
    val addend = UInt(32.W)
    val vec_addr = Address()
    val n_elems = UInt(20.W)
  }, EmptyAccelResponse())
  val addendReg = Reg(UInt(32.W))
  val activeCmd = RegInit(false.B)
  when (io.req.fire) {
    addendReg := io.req.bits.addend
    activeCmd := true.B
  }
  val ReaderModuleChannel(vec_in_request, vec_in_data) =
    getReaderModule("vec_in")
  val WriterModuleChannel(vec_out_request, vec_out_data) =
    getWriterModule("vec_out")

  val write_len_bytes = Cat(io.req.bits.n_elems, 0.U(2.W))
  vec_in_request.valid := io.req.valid
  vec_in_request.bits.addr := io.req.bits.vec_addr
  vec_in_request.bits.len := write_len_bytes
  vec_out_request.valid := io.req.valid
  vec_out_request.bits.addr := io.req.bits.vec_addr
  vec_out_request.bits.len := write_len_bytes
  // for each 32b chunk, add addend and write back
  vec_out_data.data := vec_in_data.data + addendReg

  //manage in/out state machine
  io.req.ready := !active_cmd &&
    vec_in_request.ready && vec_out_request.ready
  io.resp.valid = active_cmd &&
    vec_in_request.ready && vec_out_request.ready
  when (io.resp.fire) {
    activeCmd := false.B
  }
}
```

Fig. 2: Beethoven implementation of a Vector Addition Core using a reader and a writer.

know about these to utilize the DRAM bandwidth correctly and effectively on Beethoven. The developer may customize the data port size to fit their data flow best, and it is the job of Beethoven to perform this memory transaction as efficiently as possible. Beethoven supports advanced functionalities such as read prefetching, which transparently increases performance for the developer.

Readers maximize data throughput by prefetching data and launching parallel read operations to external memory. Read-

```

class MyAcceleratorConfig extends AcceleratorConfig(
  AcceleratorSystemConfig(
    nCores = 1,
    name = "MyAcceleratorSystem",
    moduleConstructor = ModuleBuilder(p => new MyAccelerator()(p)),
    memoryChannelConfig = List(
      ReadChannelConfig("vec_in", dataBytes = 4),
      WriteChannelConfig("vec_out", dataBytes = 4)))
object MyAcceleratorKria extends BeethovenBuild(
  new MyAcceleratorConfig,
  buildMode = BuildMode.Simulation,
  platform = KriaPlatform())

namespace MyAcceleratorSystem {
  response_handle<bool> // no payload for EmptyResponse
  my_accel(int16_t core_idx,
           uint32_t addend,
           const remote_ptr &addr,
           uint32_t n_els);
};

fpga_handle_t handle; // open handle to Beethoven runtime
remote_ptr mem = handle.malloc(1024); // allocate memory for accel.
my_init(mem.getHostAddr()); // initialize memory with data
handle.copy_to_fpga(mem); // copy memory to FPGA-accessible region
auto resp = my_accel(0, 0xCAFE, mem, 1024 / sizeof(uint32_t));
resp.get(); // wait for accel. to complete
handle.copy_from_fpga(mem); // copy to host_accessible region

```

Fig. 3: (a) Beethoven vector addition Core and System configuration is shown on the top. Beethoven accelerator configuration that contains the vector addition System deployed on a Kria KV260 FPGA is shown at the bottom. To build an accelerator for a different platform, the programmer needs only to change the platform to another supported platform (e.g., `AWSF1Platform()`). Developers can create multi-core Systems by simply changing the assigned value of `nCores`. (b) Beethoven generates C++ bindings according to the specifications given in the hardware. (c) The platform-aware Beethoven runtime provides platform-agnostic routines for allocating and managing the accelerator’s memory space.

ers use on-chip memory to store prefetched data internally. This presents a tradeoff between the number of parallel read transactions a single Reader can make and the read length per transaction. Increasing either parameter increases the resources needed to manage the Reader. We tune these parameters for our currently supported platforms to maximize memory throughput with reasonable read length and parallelism.

The vector addition Core implementation in Figure 2 shows how Reader and Writer modules can be used to stream operands from memory. Developers elaborate their code by creating a configuration for it, shown in Figure 3. Configurations allow the developer to declare memory interfaces for a Core, change the number of Cores in a System, or add new Systems to Beethoven without modifying the functional description of their design.

**Multi-Die Designs** Large FPGAs are constructed from multiple (typically between 2 and 4) SLRs, and external interfaces (e.g., host, memory) will have fixed positions in a particular SLR. To achieve good quality of results, designers typically must modify their SoC architecture to account for SLRs by disaggregating global control logic into per-SLR logic, adding appropriate buffering to account for expensive SLR-crossing timing delays, and manually assigning modules to SLRs in

placement constraint files. While FPGA placers may attempt to do this floorplanning automatically, our experience was that the feasible device utilization was massively reduced without explicit floorplanning.

To address the complexities of multi-die FPGAs, Beethoven first places accelerator cores across SLRs. Then, Beethoven generates on-chip networks for memory, host, and intra-accelerator communication that uses this placement information to account for core placement and external interface placement: it constructs a subnetwork for endpoints on the same SLR and then connects these subnetworks with appropriate buffering to account for the high cross-SLR delays. Each subnetwork is itself a tree structure where the internal nodes are buffers. The fanout and buffering parameters that dictate the construction of this network are configurable using the platform development interfaces. Beethoven produces constraint files that enforce the placement of all components onto the intended SLRs. We found that the same RTL without placement constraints consistently yielded poorer quality results and failed timing analysis.

**Scratchpads and On-Chip Memory** Managing on-chip memory in a platform-independent way is challenging. FPGA toolchains do not reliably infer the memory correct cells without using vendor-supplied templates and Verilog annotations. Furthermore, ASIC toolchains require you to instantiate SRAM cells by hand and feature many functionalities not used except in specific circumstances by seasoned engineers (e.g., extra margin adjustment). Nevertheless, even basic functionality requires these features to be tied off correctly.

Beethoven provides a Scratchpad abstraction and an on-chip memory utility to provide simplified, portable, and efficient on-chip memory access. The Scratchpad abstraction is an on-chip memory of the specified size with an initialization routine that uses a Reader to fill the scratchpad with operands from memory. Beethoven interacts with a platform-specific memory backend to implement the memory that the user desires. We currently provide a Xilinx FPGA (UltraScale+ memories) and ASAP7 (SRAM) memory backend.

The use of a memory generation system that is tightly integrated into the RTL generation process is both novel and enables designs *not possible in traditional HDLs (e.g., Verilog) or prior work*. During FPGA synthesis of a single accelerator core in a traditional RTL design, the memory cells will be mapped to device-specific cells (either BRAM or URAM). Duplicating this module for a multi-core design may overutilize those memory cells. In contrast, a mixed use of the memory cells would have succeeded - that is, a design where some cores use URAM, and others use BRAM. Beethoven’s Xilinx backend monitors the per-SLR utilization of memory cells during RTL generation, attempting to map to the most efficient on-chip memory configuration but mapping to other cell types when utilizing more than 80% of the available resources on a given SLR. While an experienced FPGA designer could plan and write their design to achieve this result, it would neither be portable to other platforms nor

robust to changes in the architecture.

**Command Abstractions** Developer designs are accessible from software and other Cores using Beethoven command/response IO (input/output) abstractions. By default, Beethoven uses the RoCC instruction format, but developers can also declare their own command/response formats. Custom commands are transparently mapped onto the RoCC instruction format inside the Core design, allowing Beethoven hardware and software stack to remain oblivious to the commands used by different Cores. Beethoven takes developer-defined custom command format for a core and generates a C++ library with the custom command arguments instead of forcing the developer to perform this mapping themselves, as shown in Figure 3. The generated C++ ensures that the same software testbench can be used across systems where the instrumentation or device details are different. For instance, using different memory address widths will change the bit layout of the payloads delivered to the accelerator, changing the implementation of the `my_accel()` binding without affecting the dataflow of the testbench.

**Platform Development** To add support for a new platform in Beethoven, it is only necessary to provide details for three things. First, whether the platform is an ASIC or FPGA: this allows Beethoven to make educated choices for the latencies of some internal, non-performance critical components. Second, information about the external memory space and protocols, such as protocol parameters (e.g., data bus width, address width, number of AXI ID bits), whether the memory space is shared or discrete from the host, device-specific limitations, and so on. Finally, host-accelerator communication address-space information for the MMIO command/response interface and generated C++ headers. The platform developer provides one function to declare the top-level, external IOs to the communication interface and another function to connect the Beethoven fabric to command and memory endpoints. This structure allows for arbitrary declaration of IOs and any necessary hardware modules for a platform. For instance, an FPGA platform may declare the IOs for communication with an external host. In contrast, a test-chip platform may forego declaring external IOs and instead instantiate the CPU and connect it directly to the Beethoven fabric. While these are the bare minimum to add support for a new platform, there are many more optional additions to help optimize platform-specific performance, such as multi-die information (e.g., how many SLRs, connectivity of SLRs), Reader/Writer internal performance knobs (e.g., registers vs. SRAMs for reader/writer buffers), and network elaboration knobs (e.g., maximum supported degree of crossbars).

### C. Beethoven Software Systems

The developer’s software testbench should be able to access accelerator modules with low latency and without significant resource overheads. Satisfying this objective is challenging for several reasons: 1) the interface between the host and FPGA may differ from system to system, 2) there may be contention

for hardware resources and 3) achieving a high-performance, low-overhead design presents implementation challenges. Our solution comprises two parts: an FPGA management runtime and a user library.

1) *FPGA Management Runtime*: The FPGA management runtime operates as a userspace server responsible for arbitrating fair access to the command-response bus and managing the FPGA memory space. This separation between the FPGA interfaces and user processes helps ensure correctness and simplifies communication over the MMIO interface. The runtime server polls the MMIO interface for command responses when there are in-flight commands. This runtime is implemented as a user module for the current version of Beethoven. It will be expanded to be implemented as a kernel module in the future.

2) *Managing the FPGA Memory Space*: Beethoven provides memory allocator solutions for both embedded and discrete FPGA platforms.

**Embedded Platforms** On embedded platforms, FPGAs share the same memory address space as the host. Beethoven runtime supports physical address allocations for the FPGA by querying the OS mapping tables and allocating memory. Another implication of address space sharing on embedded platforms is that the FPGA and the CPU must maintain a coherent view of memory. Beethoven ensures that DRAM reads/writes from the FPGA are coherent by utilizing AXI-ACE coherence support to mark FPGA reads and writes as coherent. At the moment, Beethoven supports physical addressing inside the FPGA on embedded platforms using Linux “hugepages” and extracting the physical address from the OS page table.

**Discrete Platforms** On discrete platforms, the host and the FPGA have separate address spaces, and the FPGA typically uses physical addresses. The Beethoven runtime provides an allocator for this discrete address space and maintains all states in the host’s address space. This ensures that separate processes can utilize the FPGA kernels and make allocations without memory conflicts.

Despite the differences on these platforms, Beethoven’s abstractions hide platform-specific instrumentation details (e.g., invoking PCIE DMA drivers for discrete platforms vs MMIO for embedded platforms).

3) *Beethoven Software Library*: Beethoven provides a software library for communicating with the runtime. The library provides access to the allocator, DMA routines to FPGA memory, and a command/response interface. If the developer chooses to use a custom command format when designing their accelerator Core, Beethoven creates an additional C++ library with library calls to access their accelerator using the Core’s custom command format, as shown in Figure 3b, c. Sending a command returns a response handle, which the user may use to block while waiting for the command to finish processing. The Beethoven software interfaces are enumerated in the appendix.

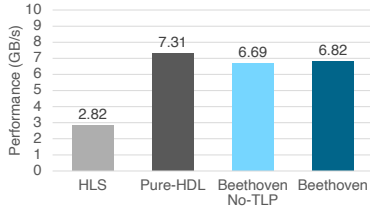


Fig. 4: Performance of Memcpy microbenchmarks on an AWS F1 FPGA platform.

#### D. Platform Support and Instrumentation

We design the Beethoven software libraries so that developers may write platform-agnostic software that utilizes the FPGA.

**Simulation Platform** Beethoven provides a simulation platform for debugging and performance prediction with support for Synopsys VCS and Verilator [46]. We integrate DRAM-Sim3 [26], a cycle-accurate DRAM model, into the simulator to more accurately identify memory bottlenecks and estimate wall-clock execution time.

**FPGA Platforms** Beethoven currently supports the Alveo U200 FPGA available on AWS F1 cloud instances and the Xilinx Zynq FPGA family. The former is a discrete, PCIe-mounted FPGA designed for data center applications. In contrast, the Zynq family are FPGAs embedded on the host die intended for everything from edge to high-performance computing. We chose these targets to demonstrate that Beethoven abstractions are both performant and flexible.

**ASIC Platforms** Beethoven supports ChipKIT [48], a framework for ASIC chip design, as well as multiple ASIC technology libraries: ASAP7 [7], [45], and the Synopsys Academic PDKs [23]. Because ChipKIT relies on the presence of an on-chip CPU, which we cannot freely distribute for licensing reasons, we support ChipKIT by requiring the developer to provide a path to the ARM M0 source, and Beethoven performs the rest of the integration. Special consideration was taken for SRAM macros in particular: Beethoven provides a memory compiler-like utility that cascades and banks the SRAM cells available in the technology library to produce the memory requested by the developer.

Platform support for both FPGAs and ASICs is readily extensible to new platforms. Our emphasis is that our current cross-platform support is indicative that Beethoven primitives allow for seamless cross-platform accelerator integration.

### III. EVALUATION

#### A. Microbenchmarks

In this microbenchmarks evaluation, we present baseline measurements that show the resource, performance, and complexity tradeoffs for Beethoven compared to HLS and hand-written HDL.

We implement a basic memory access kernel, Memory-Copy (MemCpy), in hand-written Chisel HDL, Beethoven,

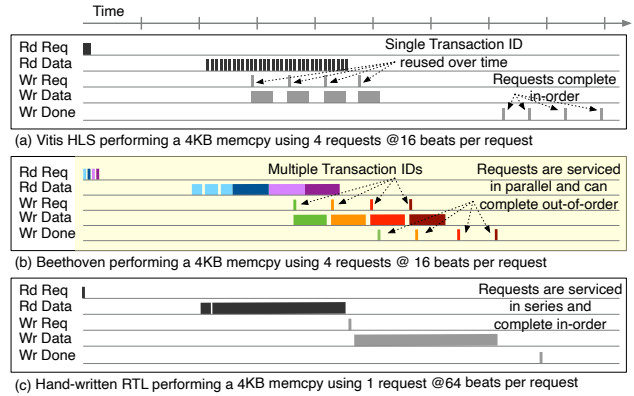


Fig. 5: An annotated reproduction of the timing diagram generated using 4KB memcpy microbenchmarks developed in (a) HLS, (b) Beethoven, and (c) hand-written HDL.

and Vitis HLS for the AWS F1 platform (Xilinx Alveo U200 FPGA). We choose this kernel because it isolates the reader and writer abstractions from externalities (e.g., improper use of HLS loop pragmas). This allows us to draw a clearer baseline comparison for these methodologies’ performance and design complexities.

Our hand-written HDL implementation overlaps read and write transactions but only uses a single AXI ID and emits one transaction per ID concurrently. For Beethoven, we emit long memory transactions as smaller transactions using different AXI IDs. This use of different IDs allows the memory controller to return data out-of-order, potentially increasing achievable bandwidth but significantly increasing design complexity. For all implementations, we use an AXI burst length of 64 as recommended by the Xilinx DDR controller IP specification [1] to maximize throughput.

The HDL implementation of the Memcpy kernel in Chisel HDL (denoted as `Pure-HDL`) took  $\approx 470$  lines of code (LoC). In contrast, the Beethoven implementations took 23 LoC (and 16 LoC for configuration), and the Vitis HLS implementation (denoted as `HLS`) took 4 LoC (and 2 LoC for pragmas). We use the term *Transaction-Level Parallelism* (TLP) to refer to the use of separate AXI IDs for out-of-order execution of a single memory transaction in this evaluation. For the Beethoven implementation, we elaborate a version with and without TLP. The Beethoven and pure-Chisel implementations run at 250MHz and the HLS at 500MHz, although it is performance-limited by the 250MHz DDR controller frequency.

Figure 4 shows the performance of the microbenchmark for our implementations. For the Memcpy benchmark, pure-HDL, Beethoven, and Beethoven No-TLP perform similarly, with pure-HDL implementation performing better by 7%. Although our HLS implementation is annotated to use 64-beat bursts, the compiled output only used 16-beat bursts. To determine if the performance degradation is solely due to a shorter burst length, we compiled a Beethoven memcpy implementation with 16-

beat bursts and found no degradation.

Figure 5 shows the AXI transaction timelines for 4KB memcpy operations on Vitis HLS and Beethoven using 4 requests @ 16-beat bursts. Our hand-written RTL only supports requests with 64-beat bursts and is plotted for comparison. Even for 4 transactions, we notice significant differences between HLS and Beethoven.

While both implementations utilize transaction-level parallelism by emitting many concurrent memory transactions, HLS emits all its transactions on the same AXI ID, whereas Beethoven uses different IDs. Transactions on the same AXI ID must be performed in order, which reduces the memory controller’s achievable parallelism. For this reason, it is common practice to use multiple AXI IDs to achieve better performance [27]. We found that while under heavy load, the latency of memory operations grew tremendously for the HLS memcpy kernel, whereas for the Beethoven implementation, writes finished early, leading to the performance differences we see for the memcpy benchmark.

Our intent is not to advocate for our particular implementation of the AXI protocol — rather, we observe that achieving ideal performance on a real system is complex. Moreover, an ideal solution for one system may not be ideal for another. While HLS compilers could add pragmas to ensure long burst lengths and split transactions over AXI IDs, understanding the memory system sufficiently to resolve these performance problems using HLS pragmas is not trivial. Beethoven addresses these burdens by providing platform abstractions that separate memory usage and the memory system’s implementation into separate development units, allowing accelerator developers to develop platform-agnostic Cores while leaving device-specific tuning and instrumentation to those knowledgeable about the device platform.

Benchmark	Description	Data Size	Parallelism
Gemm - $N^3$	$\mathcal{O}(N^3)$ matrix multiply	$N = 256$	High
NW	$\mathcal{O}(N^2)$ string alignment	$N = 256$	None
Stencil2D	2D stencil pattern	$N = 256$	Medium
Stencil3D	3D stencil pattern	$N = 32$	High
Md - KNN	N-Body problem using $k$ -nearest neighbors approx.	$N = 1024$ $K = 32$	High

TABLE I: MachSuite benchmarks selected for our evaluation.

### B. MachSuite Benchmarks

One objective of Beethoven is to make the HDLs more productive at the lower end of the effort-performance spectrum, providing a performant alternative to HLS and HLS-like methodologies even with low effort. We investigate this by implementing a subset of the MachSuite [41] Benchmark suite in Vitis HLS [49], Spatial [20], and Chisel [3] with Beethoven to show that for similar efforts, we can achieve better performance using Beethoven. Spatial DSL [20] is an HLS-like approach that exposes hardware constructs to the programmer and then compiles the implementation into an RTL design. Like HLS, Spatial designs achieve high performance through loop optimization pragmas.

The MachSuite Benchmark suite features many programs, each with varying amounts of loop parallelism, ranging from “embarrassingly” parallel to completely lacking parallelism. Table I shows the benchmarks we selected and the varying degrees of parallelism (high, medium, low) offered in the algorithm. To the best of our ability, we manually tuned the use of pragmas for Vitis HLS and Spatial implementations. Although Spatial comes with a built-in design-space exploration tool, the reported optimal design points often did not pass FPGA image synthesis, most often due to unresolvable routing congestion.

We explore the effort-performance tradeoff for Beethoven by designing low-effort Chisel implementations for MachSuite for all but one of our designs. These low-effort implementations do not take advantage of loop parallelism in the kernel. One of our authors implemented the chosen low-effort kernels over an afternoon. For the matrix multiply kernel, we design a medium-effort implementation that parallelizes the outer and middle loop bodies by a parameterizable amount, identical to the loop parallelism factors in Vitis HLS or Spatial. After implementing Chisel designs for each kernel, we use Beethoven to create a multi-core accelerator.

For this evaluation, we consider performance to be operations performed per second. Spatial and Beethoven implementations are clocked at the default 125MHz clock rate and deployed on the Xilinx Alveo U200 (VU9P) FPGA on AWS F1 instances. HLS implementations select their clock rate during synthesis.

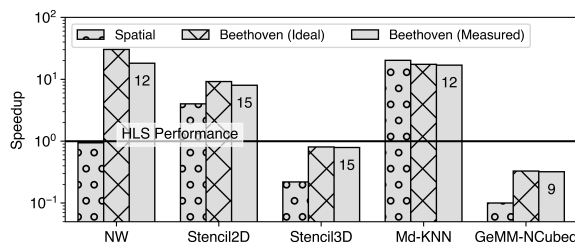


Fig. 6: Speedup of Beethoven and Spatial MachSuite implementations normalized to Vitis HLS performance. The core count for each Beethoven accelerator is shown.

1) *Results:* The results of our MachSuite evaluations are shown in Figure 6. For each Beethoven design, we measured the single-core throughput of the design and computed the ideal throughput as the single-core throughput times the number of Cores in the System (Beethoven (Ideal)). We measured multi-core accelerated system performance using wall clock time for the Beethoven (Measured) results. The numbers of instantiated accelerator Cores are labeled on the Beethoven (Measured) bars. We observe that the difference between ideal and measured throughput is greatest when the kernel’s latency is low. This is because low-latency operations have much higher contention for the runtime server lock, resulting in inefficiencies. This is not a fundamental limitation of Beethoven, and this behavior can be amended in future releases.

We found the greatest speedup in the NW workload. NW has loop-carry dependencies, making the loops unparallelizable using pragmas. Our implementation achieved  $2\times$  higher throughput over the other baselines, even for a single core. In the other benchmarks, the optimization pragmas had varying degrees of effectiveness.

As for utilization in our Beethoven implementations, Stencil2D, Stencil3D, and NW were limited by BRAM overutilization, and GeMM and MdKNN were limited by LUT overutilization. On the other hand, the Vitis HLS and Spatial were never limited by resource overutilization but by design congestion for aggressive unrolling factors.

The success of the low-effort multi-core Beethoven implementations demonstrated that Beethoven is both a productive and performant tool for accelerator design, even at low efforts, consistently outperforming existing accelerator generator toolflows such as Vitis HLS and Spatial.

### C. Attention Accelerator Case Study

In this case study, we showcase how the multi-core capabilities and Beethoven design primitives can be used to accelerate compute-bound applications. For this purpose, we select  $A^3$  [15], an approximate attention mechanism accelerator. Attention is a fundamental building block and critical performance bottleneck for large language models. This has led to great interest in specialized accelerators for this singular operation. The original publication presents the design for a single approximate attention ASIC core (see Figure 7), which they propose as part of a multi-core design that would outperform GPU platforms. We present our FPGA implementation of  $A^3$  using Beethoven primitives, discuss the challenges of producing a multi-core design, and show how Beethoven helps the developer overcome these challenges.

The  $A^3$  design comprises three coarse-grained stages: vector dot product, exponentiation/softmax, and a final output computation. The first stage performs a dot product between a *query vector* and each vector in the *key matrix* to produce a vector of *scores*. The score vector is normalized against the minimum score. This requires a global reduction operation, the first of two in this algorithm. Because the minimum score is not known until the calculations on all of the inputs are completed, the outputs of the dot product module are staged in a FIFO queue as they are calculated. The second stage of the algorithm performs a softmax operation, which requires yet another global reduction. The outputs from the second stage are stored in a second FIFO queue until they can be processed in the final stage, which consists of an element-wise vector multiplication against the *value matrix*.

We parameterize our design to support the BERT model [9]: 64-dimensional embedding vectors and 320 sentences for the key and value matrices. The data type used by  $A^3$  is a 1-byte fixed-point representation, although the width of the intermediates throughout the pipeline varies to maintain accuracy. The key and value matrices are stationary, and the accelerator streams queries from memory, processes them, and writes the results back to memory.

**Results** Our CPU baseline was performed on a 12-core Intel i7-12700K CPU using FP32 operands. Our GPU baseline was performed on an NVIDIA 3090 GPU and used FP16 operands and batch size of  $1024 \times 18$  to form a fair comparison with our FPGA implementation.

We evaluated our accelerator performance using the Xilinx VU9P FPGA (Alveo U200) on an AWS F1 instance. We successfully compiled a 23-core design at 250MHz. Our resource utilization is shown in Table II, and our floorplan is shown in 8. This development of this case study informed the development of many of the features in Beethoven. For instance, the shell consumed significant resources only on SLR0/1, making it beneficial to support core placement with affinity for each SLR individually. Another inspiration from this case study came from the congestion we perceived due to BRAM overutilization; in response, we added resource awareness for each SLR, allowing us to map to alternative on-chip memories when exceeding 80% utilization. This effect is reflected in Table II where some of the Value Scratchpads, for instance, used 15 BRAMs to implement the memory, whereas other Value Scratchpads implemented 16 URAMs. While this would typically indicate a *poor* mapping to memory cells, it alleviated a significant amount of routing congestion in our design and allowed us to successfully route our design with *96% CLB utilization!*

Extreme congestion also motivated us to tune the host and memory interconnect elaboration (this includes the host AXI-MMIO frontend). Although it only represents  $\approx 0.6\%$  resource overhead, the awareness of Cores, Systems, and SLRs yielded interconnects that afforded us significant routability even for the 92 distinct memory interfaces in this design.

Our design achieved  $3.3\times$  higher throughput and  $34\times$  lower energy per attention compared to our GPU baseline. The original authors’s implementation was evaluated as an ASIC (with a tape-out for the smaller parameter sizes, not BERT) at 1GHz and did not outperform their GPU baseline. The authors intended  $A^3$  as part of a multi-core accelerator, though they did not provide evaluation, which speaks to the value of Beethoven; although multi-core is a central argument for the motivation of  $A^3$ , without which their results would’ve been negative, the added implementation challenges of integrating multiple cores and supporting FPGA platforms as we did is secondary to their primary novelty,  $A^3$ ’s approximate pipeline. Beethoven enables the implementation of  $A^3$  as the original authors intended, and our results confirm that  $A^3$  is a scalable architecture and also has value in non-ASIC devices, in this case, FPGAs.

## IV. RELATED WORK

### A. High-Level Synthesis

HLS compilers [4], [5], [8], [10]–[12], [16]–[18], [32], [33], [43], [49], [51] are end-to-end tool flows for transforming developer designs written in high-level software languages into HDL circuit designs (e.g., Verilog) using compiler techniques typically targeting FPGA platforms, but are used in ASIC flows as well. HLS provides a highly productive hardware

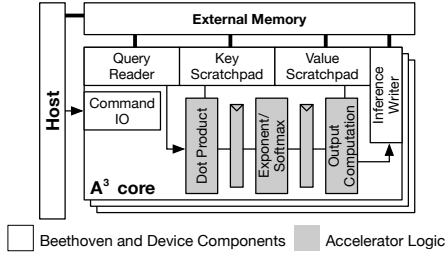


Fig. 7: A<sup>3</sup> Case Study

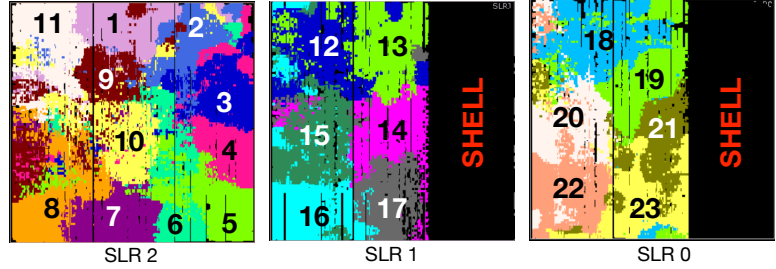


Fig. 8: Floorplan for the 23-core A<sup>3</sup> accelerator.

	CLB	CLB LUT	CLB Reg	BRAM	URAM
<b>Total</b>	139K	887K	541K	658	619
(w/ Shell)	(94.3%)	(75.1%)	(22.9%)	(30.5%)	(64.5%)
Beethoven	108K	737K	335K	518	576
	(96%)	(82%)	(18%)	(30.8%)	(72%)
Interconnect	19K	127K	3.4K	0	0
	(17%)	(14%)	(.1%)		
Core (1)	4K	27K	27K	45/15	0/32
Kernel	2.7-3.3K	16.9K	8.2K	1	0
Value SP	~800	2.6K	2.6K	15 / 0	0 / 16
Reader	600	2.3K	2.6K	7.5 / 0	0 / 8
Memory	0	0	0	7.5 / 0	0 / 8
Keys SP	~800	2.3K	2.8K	15 / 0	0 / 16
Query	602	2.3K	2.8K	7.5	0
Output	304	815	1051	7.5	0

TABLE II: Resource utilization of our 23-core A<sup>3</sup> design.

	Throughput (ops/s)	Energy/Op ( $\mu$ J/op)	Average Power (W)
CPU	$84.8 * 10^3$	885.1	75W
GPU	$5.0 * 10^6$	63.47	320W
<b>Beethoven</b>	$16.59 * 10^6$	1.84	24W
1-Core ASIC	$2.94 * 10^6$	-	-

TABLE III: Performance and Energy statistics for our Beethoven A<sup>3</sup> implementation (a multi-core FPGA implementation @ 250MHz) compared to CPU, GPU baselines, as well as the original 1-Core A<sup>3</sup> published ideal throughput (an ASIC implementation @ 1GHz).

design environment. It takes source codes in high-level languages (e.g., algorithms written in C/C++) as input to the tools and then generates an equivalent RTL hardware design. Although the software abstraction is simple, achieving maximum performance requires expertise in HLS optimization pragmas and knowledge of internal HLS structures (e.g., global memory banks). Even with this expertise, the software-to-hardware transformation is complex, generating outputs that are hard to debug or optimize by hand. For these reasons, HLS serves only as an alternative to traditional, RTL-based hardware design.

### B. Domain-Specific Languages and Compilers

Domain-Specific Languages [13], [21], [25], [29], [31], [34], [35], [38], [42], [44] generate accelerator designs by allowing hardware developers to express their designs using a set of domain-specific primitives. For example, Halide DSL [38]–[40] is designed for high-performance image processing workloads, and CFU Playground [37] is a development

framework for TinyML (Tiny Machine Learning) accelerators. To facilitate DSL development, there is Delite [6], a compiler for generating DSLs. There are works spanning HLS and hardware DSLs, like Spatial DSL [20], which uses hardware-like software structures to build accelerators. Another recent work is OverGen [28], which distills a software codebase into a specialized but still general-purpose architecture hardware accelerator design, offering high reconfigurability and HLS-like performance. In a similar vein, there are also Course-Grained Reconfigurable Architectures (CGRAs) which offer software-like programming semantics through a compiler for an architecture that resembles an FPGA except with higher-level programmable components [14], [21], [24], [30], [31], [36], [47], [50].

### C. Supporting Frameworks for FPGA Multi-tenancy

Hardware Shells are pre-defined HDL modules with fixed inputs and outputs corresponding to the external FPGA interfaces (e.g., PCIE, DDR). One example is the AWS-F1 shell that we use in this work. Coyote [22] and AmorphOS [19] are two academic works that expand on shells, focusing on FPGA multi-tenancy and multiplexing.

Coyote [22] divides the FPGA into several static “virtual FPGAs” (vFPGAs) and provides each partition with access to the typical system interfaces as well as additional utilities like virtual memory. The FPGA images deployed in each vFPGA are individually reconfigurable, allowing the programmer to dynamically switch the set of functions the FPGA provides at run-time. Statically partitioning the FPGA presents problems for larger kernels, which may consume more resources than are available for a single static partition.

AmorphOS [19], on the other hand, provides more flexible multiplexing of FPGA resources at higher compile-time costs. The developer provides AmorphOS with one or several accelerator designs integrated into a shell. Then, it compiles many FPGA images, each with a different permutation of the desired accelerators. At run-time, depending on the application’s demands, an FPGA image with an appropriate permutation of accelerators will be loaded to accelerate the application.

While these works are productive additions to the FPGA stack, their primary contribution is towards multiplexing the FPGA fabric for multi-tenancy purposes. In contrast, Beethoven provides flexible, virtualized channels for host,

memory, and on-chip communication along with multi-core design abstractions to simplify the complexities of concurrent communication in accelerators. As an analogy, prior works operate at the level of the OS (i.e., separate processes), whereas Beethoven operates at the user level (i.e., separate threads) and provides utilities for transparently multiplexing shared communication resources. However, this does not mean that Beethoven is fundamentally more powerful or better. Beethoven is synergistic with these prior works, each providing different, complementary benefits to the accelerator design stacks. We intend Beethoven to be integrated into shells and multi-tenancy frameworks to provide the communication and multi-core design abstractions of Beethoven alongside the capabilities of prior work.

## V. CONCLUSION

While prior work addresses some of the inherent complexity of managing hardware devices, existing approaches still leave much of that complexity to the designer. To address the shortcomings in prior methods, Beethoven simplifies the complexities of concurrency and portability in high-performance, heterogeneous multi-core, and multi-die FPGAs by introducing new programming abstractions for memory and host communication, generating device-topology-aware FPGA floorplans and NoC implementations, and generating host-accelerator software integration. Beethoven allows the developer to focus all their efforts on the algorithm itself instead of complex hardware integration. We demonstrated that Beethoven provides on-par or better accelerated-system performance than HLS-generated or hand-written HDL for the workloads we studied. We believe that Beethoven fits naturally into the accelerator development stack, synergizing with modern HDLs and FPGA management frameworks to provide an end-to-end programming stack for accelerator development.

## VI. ACKNOWLEDGEMENT

This project is supported in part by Meta and in part by a National Science Foundation CAREER award CCF-2045973.

## REFERENCES

- [1] Advanced Micro Device, Inc. Ultrascale architecture-based fpga memory ip v1.4. <https://docs.xilinx.com/v/u/en-US/pg150-ultrascale-memory-ip>.
- [2] 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. 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>, April 2016.
- [3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference, DAC '12*, pages 1216–1225, New York, NY, USA, 2012. Association for Computing Machinery.
- [4] Cadence Design Systems, Inc. Stratus high-level synthesis. [https://www.cadence.com/en\\_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html](https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html), March 2023.
- [5] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: High-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '11*, pages 33–36, New York, NY, USA, 2011. Association for Computing Machinery.
- [6] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPOPP '11*, pages 35–46, New York, NY, USA, 2011. Association for Computing Machinery.
- [7] Lawrence T. Clark and Vinay Vashishtha. Design with sub-10 nm finfet technologies. In *2017 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–87, 2017.
- [8] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April.
- [9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [10] Falcon Computing. Company website: Falcon develops products that simplify the adoption of hardware acceleration. <https://www.falconcomputing.com/about-falcon-computing-solutions/>.
- [11] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1327–1330. IEEE, Dec 2021.
- [12] B. Fort, A. Canis, J. Choi, N. Calagar, R. Lian, S. Hadjis, Y. T. Chen, M. Hall, B. Syrowik, T. Czajkowski, S. Brown, and J. Anderson. Automating the design of processor/accelerator embedded systems with legup high-level synthesis. In *Proceedings of the IEEE International Conference on Embedded and Ubiquitous Computing*, pages 120–129, August 2014.
- [13] C. G. García, D. Meana-Llorián, V. García-Díaz, A. C. Jiménez, and J. P. Anzola. Midgar: Creation of a graphic domain-specific language to generate smart objects for internet of things scenarios using model-driven engineering. *IEEE Access*, July 2020.
- [14] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. Riptide: A programmable, energy-minimal dataflow compiler and architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 546–564, 2022.
- [15] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J. Park, S. Lee, K. Park, J. W. Lee, and D. Jeong. A<sup>3</sup>: Accelerating Attention Mechanisms in Neural Networks with Approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341, February 2020. ISSN: 2378-203X.
- [16] Intel Corporation. Intel fpga sdk for opencl software technology. <https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html>, February 2023.
- [17] Intel Corporation. Intel high level synthesis compiler. <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html>, February 2023.
- [18] Ganwon Jo, Heehoon Kim, Jeesoo Lee, and Jaemin Lee. SOFF: An opencl high-level synthesis framework for FPGAs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 295–308, June 2020.
- [19] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI'18*, pages 107–127, USA, 2018. USENIX Association.
- [20] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszal, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*, pages 296–311, New York, NY, USA, 2018. Association for Computing Machinery.

- [21] T. Kojima, N. A. V. Doan, and H. Amano. Genmap: A genetic algorithmic approach for optimizing spatial mapping of coarse-grained reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–14, July 2020.
- [22] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010. USENIX Association, November 2020.
- [23] Yao-Ming Kuo, Leandro J. Arana, Luis Seva, Cristian Marchese, and Leandro Tozzi. Educational design kit for synopsys tools with a set of characterized standard cell library. In *2018 IEEE 9th Latin American Symposium on Circuits & Systems (LASCAS)*, pages 1–4, 2018.
- [24] Hyoukuun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2018.
- [25] Jiajie Li, Yuze Chi, and Jason Cong. Heterohalide: From image processing dsl to efficient fpga acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '20*, pages 51–57, New York, NY, USA, 2020. Association for Computing Machinery.
- [26] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.
- [27] Arm Limited. Amba axi5 specification. <https://developer.arm.com/documentation/ih10022/latest>, 2023.
- [28] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabzadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. Overgen: Improving fpga usability through domain-specific overlay generation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 35–56, 2022.
- [29] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman. Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development*, 12(3):1165–1187, March 2019.
- [30] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 553–564, February 2017.
- [31] Xingchen Man, Leibo Liu, Jianfeng Zhu, and Shaojun Wei. A general pattern-based dynamic compilation framework for coarse-grained reconfigurable architectures. In *Proceedings of the Design Automation Conference (DAC)*, pages 1–6, June 2019.
- [32] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today’s high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3), September 2012.
- [33] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.
- [34] R. Oliveira, D. Pereira, C. Maia, and P. Santos. A domain specific language for automotive systems integration. In *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, pages 4483–4488, October 2019.
- [35] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
- [36] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A Reconfigurable Architecture For Parallel Patterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 389–402, Toronto ON Canada, June 2017. ACM.
- [37] Shvetank Prakash, Tim Callahan, Joseph Bushagour, Colby Banbury, Alan V. Green, Pete Warden, Tim Ansell, and Vijay Janapa Reddi. Cfu playground: Full-stack open-source framework for tiny machine learning (tinyml) acceleration on fpgas. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 157–167, 2023.
- [38] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Fredo Durand. Halide: decoupling algorithms from schedules for high-performance image processing. *Communications of the ACM*, 61(1), January 2018.
- [39] Jonathan Ragan-Kelley, Andrews Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Fredo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. In *Proceedings of the ACM Special Interest Group on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2012.
- [40] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Fredo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 2013.
- [41] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Raleigh, North Carolina, October 2014.
- [42] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. Musket: A domain-specific language for high-level parallel programming with algorithmic skeletons. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing*, pages 1534–1543, April 2019.
- [43] Siemens. Catapult high-level synthesis and verification. <https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/>, February 2023.
- [44] Lenny Truong, Steven Herbst, Rajsekhar Setaluri, Makai Mann, Ross Daly, Keyi Zhang, Caleb Donovan, Daniel Stanley, Mark Horowitz, Clark Barrett, and Pat Hanrahan. fault: A python embedded domain-specific language for metaprogramming portable hardware verification components. In *Computer Aided Verification*, pages 403–414, July 2020.
- [45] Vinay Vashishtha, Manoj Vangala, Parv Sharma, and Lawrence T. Clark. Robust 7-nm sram design on a predictive pdk. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.
- [46] Veripool. Verilator. <https://www.veripool.org/verilator/>, March 2023.
- [47] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 268–281, June 2020.
- [48] P. Whatmough, M. Donato, G. Ko, S. K. Lee, D. Brooks, and G. Wei. Chipkit: An agile, reusable open-source framework for rapid test chip development. *IEEE Micro*, pages 1–1, 2020.
- [49] Xilinx. Vitis unified software platform. <https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html>, March 2023.
- [50] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Viliam, Muhammad Shahbaz, and Kunle Olukotun. Sara: Scaling a reconfigurable dataflow accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1041–1054, 2021.
- [51] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

APPENDIX

A. Beethoven Hardware Interfaces

Description	Configuration Constructor	Accessor from RTL
Reading contiguous regions from external memory. The width of the data bus is configurable by dataBytes.	<code>ReadChannelConfig(name: String, dataBytes: Int, nChannels: Int = 1)</code>	<code>getReaderModule(name: String, idx: Int = 0) -&gt; (RequestChannel, ReaderDataChannel)</code>
Writing to contiguous regions in external memory. The width of the data bus is configurable by dataBytes.	<code>WriteChannelConfig(name: String, dataBytes: Int, nChannels: Int = 1)</code>	<code>getWriterModule(name: String, idx: Int = 0) -&gt; (RequestChannel, WriteDataChannel)</code>
A Beethoven-managed on-chip memory scratchpad. Allows for variable depth, width, access latency and number of ports. Will attempt to map to an efficient cell mapping for the given platform.	<code>ScratchpadConfig(name: String, dataWidthBits: Int, nDats: Int, nPorts: Int, latency: Int, features: ScratchpadFeatures)</code>	<code>getScratchpad(name: String)</code>
A scratchpad that is writeable from other accelerator cores on chip. commDeg determines if the memories of other cores in this system are identical or independent.	<code>IntraCoreMemoryPortInConfig(name: String, nChannels: Int, portsPerChannel: Int, dataWidthBits: Int, nDats: Int, commDeg: CommunicationDegree, readOnly: Boolean = false, latency: Int = 2)</code>	<code>getIntraCoreMemIns(name: String) -&gt; Seq[Seq[ScratchpadDataPort]]</code>
A scratchpad-like write port that connects to a scratchpad in other systems/cores. The write latency cannot be specified.	<code>IntraCoreMemoryPortOutConfig(name: String, toSystem: String, toMemoryPort: String, nChannels: Int = 1)</code>	<code>getIntraCoreMemOut(name: String) -&gt; Seq[MemReqWritePort]</code>
Declares an on-chip memory that is manually-managed by the programmer. Provides SRAM-like interfaces.	N/A	<code>Memory(latency: Int, dataWidth: Int, nRows: Int, nReadPorts: Int, nWritePorts: Int, nReadWritePorts: Int, withWriteEnable: Boolean = false)</code>

B. Beethoven Software Interfaces

Description	Software Interface
Generating your hardware using Beethoven IOs will produce a C++ stub for calling your function.	<code>response_handle&lt;resp_t&gt; [SystemName]::[IOName] (int core_idx, int arg_a, float arg_b, ...);</code>
Sending commands to an accelerator core returns response handles. The response handle can be used to coordinate asynchronous execution between hardware and software.	<code>resp_t response_handle::get()</code> <code>std::optional&lt;resp_t&gt; response_handle::try_get()</code>
The <code>fpga_handle_t</code> facilitates communication with the Beethoven runtime and is used for allocating memory regions visible from both the accelerator and host domains.	<code>remote_ptr fpga_handle_t::malloc(size_t n_bytes)</code>
Beethoven provides DMA routines between the host and accelerator memory spaces through the <code>fpga_handle_t</code> .	<code>void fpga_handle_t::copy_to_fpga(remote_ptr &amp;t)</code> <code>void fpga_handle_t::copy_from_fpga(remote_ptr &amp;t)</code>