Lake: An Agile Framework for Designing and Automatically Configuring Physical Unified Buffers

Maxwell Strange
Stanford University
USA
mstrange@stanford.edu

Kavya Sreedhar
Stanford University
USA
skavya@stanford.edu

Mark Horowitz
Stanford University
USA
horowitz@ee.stanford.edu

ABSTRACT

Creating efficient memory subsystems has always been critical for hardware accelerators. While prior works enable flexible memory generation, the task of mapping streaming applications to the hardware is often left to the user. In this paper, we argue that hardware generators must move to generating not only hardware but *also* the software collateral for mapping to that hardware in order to enable rapid design-space explorations. We demonstrate this with Lake, our open-source memory generation system for streaming accelerators, which uses a high-level streaming memory abstraction to cleanly define the hardware/software interface. To demonstrate the viability of this approach, we have used Lake to generate and automatically map to the memory in multiple fabricated coarse-grained reconfigurable arrays. Lake is publicly available at https://github.com/StanfordAHA/lake/.

1 INTRODUCTION

Streaming hardware accelerators [4, 9, 12] generally support a computational model where data is prefetched and then streamed to the functional units in order to maximize the utilization of their large, parallel compute hardware. The streaming applications accelerated by this hardware feature control patterns where the memory accesses can be separated from the compute graph. Accelerator implementations exploit this fact by building machines that implement explicit, decoupled data orchestration [10] where memory access patterns are known *a priori* and can be produced by explicit address generators. This paradigm allows many independent address generator units to work in tandem to keep large numbers of functional units busy. To help find the optimal hardware accelerator for a set of applications, flexible hardware generators are often used to perform rapid design space explorations (DSEs). However, most of these generators only generate RTL and do not provide the necessary software collateral to enable automatic compiler mapping. This approach limits the efficiency of a DSE as there is still manual intervention required to map applications to each design point.

User applications contain graphs of streaming memories that feed data to and process data from compute subgraphs. To enable automatic mapping from these graphs to hardware implementations, the hardware generator and compiler must target a common abstraction. One such abstraction for streaming memories is called the Unified Buffer (UB) [6], which is described in more detail in Section 2. This paper introduces the Lake system, which is built around the UB abstraction in order to enable handling automatic hardware generation and mapping for the *memory* portion of the application graph. Lake is one project within an ecosystem of tools that seek to enable agile hardware design methodologies [5]. The two other tools in this ecosystem, namely PEak [2] and Canal [7], employ different domain-specific languages (DSLs) that target certain abstractions to support building and compiling to processing elements and interconnects, respectively.

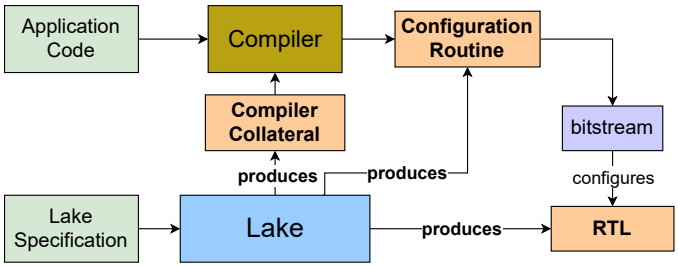

Figure 1: Showing the system-level flow of using Lake to design a compiler-compatible streaming memory accelerator.

As shown in Figure 1, Lake is a hardware generator system that generates all necessary collateral for automatically building and testing streaming memories from a single input specification. In addition to generating physical UB hardware ("RTL" in Figure 1) that implements the UB abstraction, Lake is designed to extract the resource constraints of this hardware and provide them to the compiler ("Compiler Collateral" in Figure 1). With this information, the compiler can generate efficient and implementable schedules for the desired application on the hardware target. In our use case for streaming memories, the original application is written in the image processing DSL Halide [11], although the system is not inherently limited to any specific application language. Then, Clockwork [3], a polyhedral application compiler, analyzes the loop nests of the original application and automatically integrates the "Compiler Collateral" when building the set of constraints that are used by its solver to create a valid schedule for the hardware. Since the hardware implementation is controlled by the user, Lake also creates the mapping routines ("Configuration Routines" in Figure 1) needed to convert these high-level schedules into the low-level operations needed to program the configuration registers in the generated RTL. In the rest of this paper, we briefly describe the Unified Buffer and then describe Lake and show how its connection

LATTE '24, April 27-28, 2024, San Diego, CA, USA

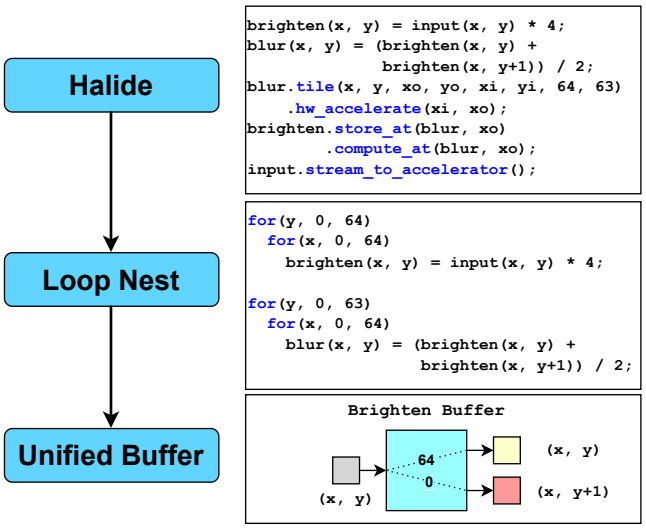

Figure 2: The flow of how an application is written and mapped down to hardware generated by Lake. The compiler extracts and analyses loop nests in the original Halide application and then produces Unified Buffers to implement the streaming memories.

to the UB abstraction makes automatic mapping a feasible option for hardware generators.

2 UNIFIED BUFFER AND COMPILATION

The Unified Buffer is an abstraction for streaming memories that describes operations on streams in both space and time [6]. The UB fundamentally describes streams emitted from a memory as reorderings of streams ingested by the memory. These reorderings are defined over the **IterationDomain** of the original loop nests by a multidimensional set of counters of size *dim*, which are the iteration indices of the loop nest. In addition to the specific reordering of a stream, the UB guarantees data dependencies are maintained by providing specific scheduling information for each buffer access.

For every statement that accesses a specific buffer in the original application, the UB will create a **Port** into a memory representing that buffer and bind to it both an **access map**, which describes the reordering as a sequence of addresses, and a **schedule**, which enforces data dependencies explicitly through a sequence of event timestamps. Each point in this **schedule** corresponds to a read or write at the associated address in the **access map**. These sequences are described as affine maps, using a small set of parameters: *extents*, a vector of size *dim* to describe the bounds of the iteration indices; *strides*, a vector of size *dim* to describe the relation between each iteration index and the sequence value; and a scalar *offset*.

An example of a UB extracted from a brighten followed by a blur kernel written in Halide is shown in Figure 2. First, the loop nests are extracted from the original application and shown in the "Loop Nest" box. These loop nests can be analyzed by Clockwork [3] after which final address and schedule maps are emitted. An example of these maps from the original **IterationDomain** is shown in Figure 3. Since the brighten buffer is accessed by two different statements in the following blur kernel, the UB describing the brighten buffer has two output **Ports**. Note that the cycle offset

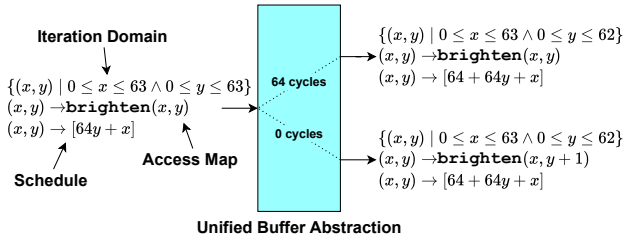

Figure 3: Expanded view of the UB built for the brighten buffer in Figure 2. For each UB Port, a stream is described with an Iteration Domain, Access Map, and a Schedule.

for each output **Port** is 64 since it takes 64 cycles to produce the entire first stencil window (of size 2x1). In addition to emitting naive UBs, Clockwork is capable of exploiting the locality of this sliding window and is able to implement additional ports as time-shifted versions of neighboring ports.

3 SYSTEM OVERVIEW

We define the set of modules used in Lake specifications to directly implement the addressing and scheduling represented by the UB abstraction. Since these modules match the UB abstraction, Lake can both extract the critical resource constraints of these objects to communicate them to the compiler and can provide the mapping routines to create a bitstream for the hardware's configuration registers from the compiler output.

Each Lake design is generated from a specification that describes the topology and resource constraints of the desired memory system. A Lake specification is composed of a set of extensible core components: **Storage**, **MemoryPort**, **Port**, **IterationDomain**, **AddressGenerator**, and **ScheduleGenerator**. The **Storage** and **MemoryPort** components are used to describe parameters of the physical storage device (e.g., SRAM vs. register file, single-port vs. dual-port). The **Port** component corresponds directly to the **Ports** that the UB abstraction binds streams to. The **IterationDomain**, **AddressGenerator**, and **ScheduleGenerator** components are used to provide a **Port** with hardware to implement the data streams that the UB will bind to it (both addressing and scheduling).

We note that Lake supports both statically-scheduled and latency-insensitive (ready/valid) design styles; the user chooses which execution paradigm to use in the input specification. In statically-scheduled designs, the read and write timestamps generated by the compiler are used to program the scheduling control units, while in latency-insensitive designs, coarse scheduling features are extracted during the Clockwork compiler's analysis.

Figure 4 shows an example specification for a coarse-grained reconfigurable array (CGRA) memory tile from one of our chips called Amber [1]. Lake specifications are written in a Python library where components are connected together with a simple API, but the resulting specification is more easily visualized as a block diagram. We used a wide, single-**MemoryPort** SRAM in the Amber memory tile to emulate the bandwidth of a multi-ported memory [5] while getting the higher energy efficiency associated with wider memory busses [8]. In this example, the user specifies that the memory tile will have two input and two output **Ports** with a narrow (16-bit) interface to external components and a wide

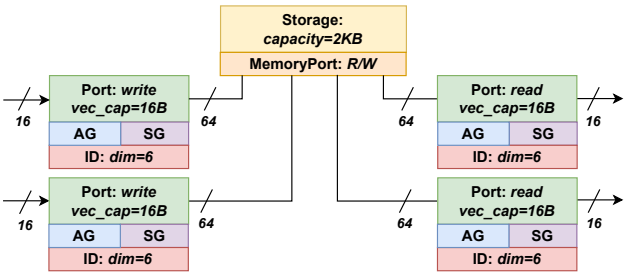

Figure 4: The Lake specification for a memory tile in the Amber CGRA [1]. This design has two input Ports and two output Ports and uses a *vectorization* factor of 4. AG = AddressGenerator, SG = ScheduleGenerator, ID = IterationDomain, *vec_cap* denotes vectorization buffer capacity.

(64-bit) interface to the internal module. For each **Port**, an amount of *vectorization* buffering is specified in its instantiation as well to capture width conversion and potential reuse. For example, for input ports in Amber, we need to pack four 16-bit words to match the 64-bit SRAM interface.

As a hardware generator, Lake generates the RTL that can be used in standard VLSI flows. Lake first stamps out individual components specified by the user and then may infer small *vectorization* buffers and related control logic to enable data width conversion within a **Port**. In cases where the topology implies resource sharing, Lake infers multiplexers and builds the corresponding arbitration logic for the select lines to those multiplexers. Lake then builds hardware for **Storage** components and their associated **MemoryPorts** by either generating custom RTL or picking from a set of provided SRAM macros as dictated by the user. We note that the Lake system is agnostic to circuit-level tricks like double clocking a single-port SRAM to emulate a dual-port memory: this would be simply described as a **Storage** with two **MemoryPorts** in the specification, and the implementation would be handled as a custom generation routine.

We provide optimizations built into Lake to generate efficient hardware for the **IterationDomain**, **AddressGenerator**, and **ScheduleGenerator** components [5], but the user can also substitute alternative implementations and iterate on the hardware generated in a modular fashion. We chose to build Lake on top of a Python-based RTL generation language [13] so that we can create objects for each Lake component using the inheritance of abstract base classes. In this way, Lake enforces that users adhere to designing memory systems composed of parameterized components that match the UB abstraction, which in turn enables Lake to extract information about the capabilities of the generated memories for the compiler.

The collateral that Lake extracts from the design and passes to the compiler is limited to information that impacts the data streams that the hardware can support (e.g., size of **Storage** components, number and characteristics of **Ports**, *dim*ensionality of the sequencing controllers, and *vectorization*). In Amber (Figure 4), the main **Storage** is a single-**MemoryPort** SRAM with a capacity of 2KB and a 1-cycle read and write. There are two input **Ports** and two output **Ports** with 6-dimensional sequencing controllers that all timeshare the single **MemoryPort**. All **Ports** have a *vectorization* factor of 4 with 16B of buffering, which has 1-cycle write and 0-cycle read

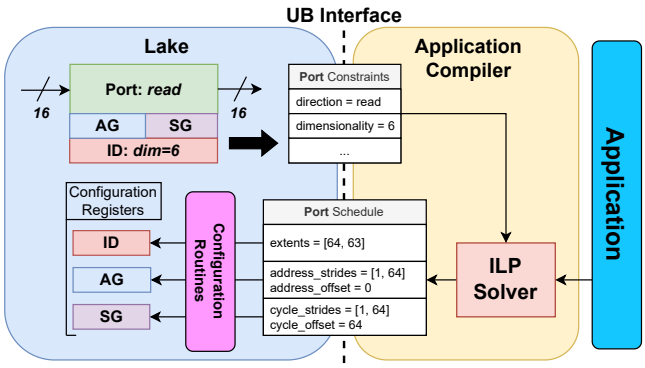

Figure 5: The process of Lake communicating with the compiler for the read Port from the brighten kernel in Figure 3, with a statically-scheduled implementation. Lake passes UB-relevant information up to the compiler that then integrates the constraints when using an Integer Linear Programming (ILP) solver to create an implementable schedule. The resulting schedule is then distributed to the relevant components' configuration routines. AG = AddressGenerator, SG = ScheduleGenerator, ID = IterationDomain.

delays. An example of this information being integrated into the compiler's scheduling system is shown in Figure 5 for a single read **Port** from the brighten buffer in Figure 2.

Finally, after the compiler generates implementable schedules for the hardware design, Lake can automatically map that scheduling information down to the configuration registers ("Configuration Routine" in Figure 1). For each Unified Buffer **Port** in the original application, the compiler generates an address and a schedule pattern as shown in Figure 5. The compiler then distributes the *strides* and *offset* for each **Port** to its associated **AddressGenerator** (and **ScheduleGenerator** in the case of a statically-scheduled design). The **IterationDomain** is configured with the *extents* from these patterns. Depending on the implementation and timing style, different information may be used from each sequence bundle, and in some cases, the hardware optimizations require a transformation on the configuration bits generated by the compiler before the hardware can be programmed. Lake requires the user to specify these transformation functions along with component implementations so that the hardware can continue to be configured automatically.

4 CONCLUSION

We argue that designers should build hardware generators that provide enough information to map to their output designs. To show the feasibility of this approach, we present our streaming memory accelerator generator system, Lake. By using the high-level Unified Buffer abstraction, Lake can produce both hardware and the associated compiler collateral, which enables rapid DSE and automatic configuration of the resulting memory hardware. While adhering to this abstract model limits the design space of the memory systems that can be generated, all hardware generators are limited by the design parameters that their generators support. The Lake system simply takes advantage of these restrictions to guarantee automatic application mapping for all hardware designs.

REFERENCES

[1] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. 2022. Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a Coarse-Grained Reconfigurable Array for Flexible Acceleration of Dense Linear Algebra. *IEEE Symposium on VLSI Technology and Circuits (VLSI)* (2022).

[2] Caleb Donovick, Ross Daly, Jackson Melchert, Lenny Truong, Priyanka Raina, Pat Hanrahan, and Clark Barrett. 2023. PEak: A Single Source of Truth for Hardware Design and Verification. *arXiv preprint arXiv:2308.13106* (2023).

[3] Dillon Huff, Steve Dai, and Pat Hanrahan. 2021. Clockwork: Resource-Efficient Static Scheduling for Multi-Rate Image Processing Applications on FPGAs. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* (2021), 145–146.

[4] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Severn, Chris Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. *International Symposium on Computer Architecture (ISCA)* (2017), 1–12.

[5] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, Keyi Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross Daly, Caleb Donovick, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark Barrett, Mark Horowitz, Christopher Torng, Fredrik Kjolstad, and Priyanka Raina. 2022. AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers. *ACM Transactions on Embedded Computing Systems (TECS)* (2022).

[6] Qiaoyi Liu, Jeff Setter, Dillon Huff, Maxwell Strange, Kathleen Feng, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. 2023. Unified Buffer: Compiling Image Processing and Machine Learning Applications to Push-Memory Accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)* 20, 2 (2023), 1–26.

[7] Jackson Melchert, Keyi Zhang, Yuchen Mei, Mark Horowitz, Christopher Torng, and Priyanka Raina. 2023. Canal: A Flexible Interconnect Generator for Coarse-Grained Reconfigurable Arrays. *IEEE Computer Architecture Letters* 22, 1 (2023), 45–48. https://doi.org/10.1109/LCA.2023.3268126

[8] Vivek Nautiyal, Gaurav Singla, Lalit Gupta, Sagar Dwivedi, and Martin Kinkade. 2017. An ultra high density pseudo dual-port SRAM in 16nm FINFET process for graphics processors. *IEEE International System-on-Chip Conference (SOCC)* (2017), 12–17.

[9] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. *International Symposium on Computer Architecture (ISCA)* (2017), 416–429.

[10] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. 2019. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2019), 137–151.

[11] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2013), 519–530.

[12] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Yanqing Zhang, Brian Zimmer, William J. Dally, Joel Emer, Stephen W. Keckler, and Brucek Khailany. 2019. Magnet: A modular accelerator generator for neural networks. *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2019), 1–8.

[13] Keyi Zhang. [n. d.]. Kratos: Debuggable Hardware Generator. https://github.com/Kuree/kratos