ImaGen: A General Framework for Generating Memory- and Power-Efficient Image Processing Accelerators

Nisarg Ujjainkar
nujjaink@ur.rochester.edu
University of Rochester
Rochester, NY, USA

Jingwen Leng
Shanghai Jiaotong University
Shanghai, China
leng-jw@sjtu.edu.cn

Yuhao Zhu
University of Rochester
Rochester, NY, USA
yzhu@rochester.edu

ABSTRACT

Image processing algorithms are prime targets for hardware acceleration as they are commonly used in resource- and power-limited applications. Today's image processing accelerator designs make rigid assumptions about the algorithm structures and/or on-chip memory resources. As a result, they either have narrow applicability or result in inefficient designs.

This paper presents a compiler framework that automatically generates memory- and power-efficient image processing accelerators. We allow programmers to describe generic image processing algorithms (in a domain specific language) and specify on-chip memory structures available. Our framework then formulates a constrained optimization problem that minimizes on-chip memory usage while maintaining theoretical maximum throughput. The key challenge we address is to analytically express the throughput bottleneck, on-chip memory contention, to enable a lightweight compilation. FPGA prototyping and ASIC synthesis show that, compared to existing approaches, accelerators generated by our framework reduce the on-chip memory usage and/or power consumption by double digits. ImaGen code is available at: https://github.com/horizon-research/imagen.

CCS CONCEPTS

• **Computer systems organization → Architectures**; • **Hardware → Power and energy**.

KEYWORDS

Accelerator, Line Buffer, Image Processing, Constrained Optimization, Synthesis, Compiler

ACM Reference Format:
Nisarg Ujjainkar, Jingwen Leng, and Yuhao Zhu. 2023. ImaGen: A General Framework for Generating Memory- and Power-Efficient Image Processing Accelerators. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589076

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '23, June 17–21, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589076

1 INTRODUCTION

Image processing has become ever more important with a plethora of emerging visual computing domains such as Augmented/Virtual Reality, computational photography, and smart cameras. These application domains all present stringent resource and power constraints, leading to many research efforts in building specialized accelerators for image processing [6, 12, 17, 30, 31, 35]. Manually building accelerators, however, is not only time-consuming, error-prone, but also relies heavily on empirical heuristics that do not always deliver optimal designs.

A recent trend is automatically generating accelerators from high-level algorithm descriptions [7, 16, 38]. Prior approaches to generating image processing accelerators either have narrow applicability or yield inefficient designs — for two main reasons (Sec. 3). First, they optimize for simple, single-consumer algorithms where each producer stage has only one consumer. When facing multiple-consumer algorithms such as unsharp filtering [16] and denoising [7], they either have to artificially transform the multiple-consumer algorithm to a single-consumer arrangement, which increases the on-chip memory usage, or increase the total on-chip memory accesses, which increases the power consumption.

Second, there is a large, algorithm-dependent trade-off space between on-chip memory requirement and power consumption that prior work fails to explore. This is because prior work assumes one single memory structure and, critically, use the same memory structure for *all* algorithms and for *all* stages in an algorithm. For instance, FixyNN [38] could generate designs using only single-port SRAMs, and SODA [7] could generate designs using only FIFOs (dual-port SRAMs). The actual design space is much larger: given an algorithm with N stages and M memory structures, there are M^N design points, each providing a unique power-vs-area trade-off.

This paper proposes a compiler framework that generates memory- and power-efficient accelerators (in the form of synthesizable RTL) for image processing (Sec. 4). Instead of artificially restricting algorithm and/or on-chip memory structures, we allow specifying generic algorithms and memory configurations (in terms of size and number of ports). Given the algorithm and hardware specifications, our compiler formulates a constrained optimization problem that, while maintaining theoretically maximum throughput, minimizes the on-chip memory usage and reduces total power consumption.

A key challenge we address is to generate accelerators that consistently deliver theoretically maximum throughput (frame rate) *for every frame*; after all, saving on-chip area and power consumption is of little use when an image processing accelerator has a low frame rate. The central difficulty is to analytically express the throughput bottleneck, i.e., on-chip memory contention, which involves set counting and is incompatible with numerical optimizations. We leverage the data access pattern of stencil operations to transform set counting into equivalent, arithmetic operations that are amenable to numerical optimizations (Sec. 5).

Building on top of the optimization formulation, we propose to judiciously coalesce multiple lines in a line buffer into a single memory block to further reduce on-chip memory consumption. We show that this technique amounts to a static rewriting of the algorithm Directed Acyclic Graph (DAG) and is naturally integrated into our compiler framework (Sec. 6).

We show that our optimization problem is an Integer Linear Programming (ILP), which has efficient solvers. As a result, our compiler is lightweight; it generates synthesizable RTL for common image processing algorithms in milliseconds. We use our framework to generate a wide variety of image processing accelerators, which we evaluate using both an ASIC flow and a Xilinx Spartan-7 FPGA board. Across different input sizes, accelerators generated by our framework reduce on-chip memory usage and power by up to 86.0% and 62.9%, respectively, when compared to designs generated by prior methods, including Darkroom [16], SODA [7], FixyNN [38].

We use our framework to perform a Design Space Exploration (DSE) that explores diverse memory configurations to generate Pareto-optimal designs. We show that the area-vs-power trade-off varies with algorithms, an algorithm-specific design space exploration that our framework uniquely enables.

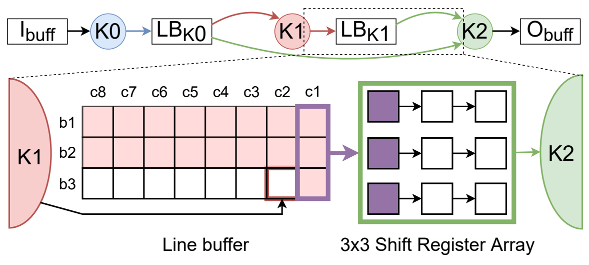
In summary, this paper makes the following contributions:

- We propose a compiler framework that generates memory- and power-efficient image processing accelerators given generic algorithm and on-chip memory specifications. The accelerators guarantee theoretically maximum throughput through constrained optimization.
- We propose a line-coalescing algorithm that coalesces multiple line-buffer lines into one memory block to further reduce on-chip memory usages.
- Accelerators generated by our compiler consume less on-chip memory and power compared to those generated using existing tools. Our compiler is integrated into a DSE process, which reveals algorithmic-dependent area-vs-power trade-offs that prior tools are unable to explore.

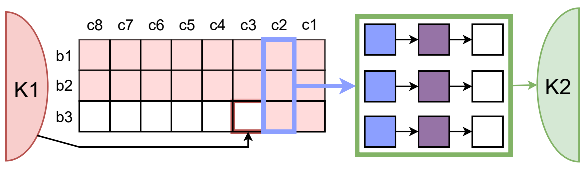
2 BACKGROUND

Image processing pipelines consist of computation stages that operate on regular 2D pixel arrays. Each stage performs a stencil operation, which operates on a window of input pixels to generate an output pixel. The stencil window moves in a raster scan order. An end-to-end algorithm usually cascades multiple stages. Each stage generates an intermediate 2D image read by (potentially multiple) consumer(s). Common image processing algorithms include in-camera image signal processing [16] and High Dynamic Range imaging using burst photography [15].

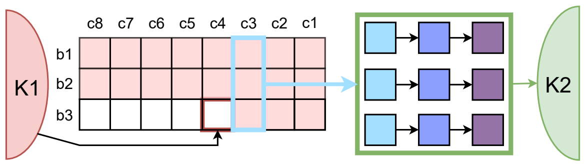
Scope. Our goal is *not* a generic stencil accelerator that runs multiple algorithms. Rather, we focus on accelerators that are specialized for a given algorithm. This is common in both 1) FPGA-based acceleration systems, where the FPGA can be re-programmed for a given algorithm, and 2) low-power ASICs as demonstrated in Image Signal Processors in modern cameras (e.g., Arm Mali [3]



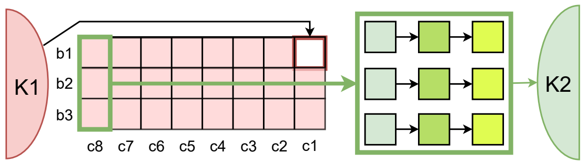
(a) Cycle t.



(b) Cycle t+1.



(c) Cycle t+2.



(d) Cycle t+7.

Fig. 1: A line-buffered accelerator. Every stage has a dedicated line buffer (and the associated shift register array) to write to. We use the line buffer after stage $K1$ to illustrate the line-buffering operations. The line buffer stores three-pixel rows, each stored in a two-port SRAM. As the producer writes the second element in b3, pixels in column c1 are moved to the shift register array. After three cycles the shift register array contains the data from a 3×3 stencil window and $K2$ starts. When $K1$ finishes writing to b3, it writes to b1, since pixels there will no longer be needed by the consumer.

and Qualcomm Spectra [4]), embedded computer vision processors [11, 24, 38, 40], and robotics accelerators [25, 32, 34], where extreme efficiency requires algorithm-specific accelerators.

Image Processing Accelerators. Fig. 1 shows a typical image processing accelerator using a simple pipeline as an example. Each algorithm stage is mapped to a dedicated hardware stage. The input and output stages ($K0$ and $K2$ here) interface with input/output buffers that communicate with off-chip memory. As extensively studied before, those buffers are usually doubled buffered with high access bandwidth, and are *not* the focus of this paper.

Stages communicate intermediate data through another set of on-chip buffers, which make up the majority of the on-chip memory usage. In particular, each producer stage has a dedicated buffer to write its intermediate data to; all its consumers read from that buffer. This is consistent with all image processing accelerator designs [5, 7, 16, 38].

Ideally, all the inter-stage data communication traffic should be fulfilled entire on-chip; otherwise, off-chip memory accesses would stall the pipeline, which requires complicated hardware logic for dynamic scheduling, introduces non-deterministic frame rates, compromises peak throughput, and consumes high power. A naive approach is to buffer all the intermediate data between stages on-chip. This comes with two downsides. First, the intermediate data could be large in size and exceeds a typical on-chip memory capacity. For instance, each 1080p image passed around between stages consume 6 MB of data. Second, it artificially forces the consumer to wait until the producer finishes generating an entire image.

Line Buffer. A common strategy to address these issues is to use a special on-chip buffer structure called "line buffer". The key observation is that each pipeline stage, at any time, operates only on data in a small, local window. Therefore, a consumer stage can start as soon as the data in a stencil window is available, essentially consuming pixels incrementally as they are generated by the producer. A pixel in the buffer can be over-written when it is no longer needed by the consumer (i.e., the stencil window has gone completely past the pixel), reducing the on-chip memory requirement.

Consider the producer-consumer pair $K1$ and $K2$ in Fig. 1, where $K2$ operates on a 3×3 stencil window from the output of $K1$ in every cycle. To support this data communication pattern, the hardware uses a line buffer that stores three rows of pixels generated by $K1$; the line buffer is connected to a 3×3 shift register array, which holds the data in a stencil window and is read by $K2$.

The producer starts writing from the first row, one pixel at a cycle. As soon as the producer finishes producing two rows plus one element (Fig. 1a), the consumer can move the first column of pixels (c1 here) from the line buffer to the shift register array. In the next cycle (Fig. 1b), as the producer writes to the third element of b3, pixels in column c2 are moved to the shift register array. After three cycles (Fig. 1c), the shift register array contains the data from a stencil window, at which point $K2$ can start to produce its output. Once $K1$ finishes writing to b3, it will write to, instead of a new row, the first row in the line buffer (b1), overwriting data in b1, because pixels there are no longer needed by the consumer (Fig. 1d). As a result, the line buffer has to store only three rows of pixels.

Implementation. In the actual hardware implementation of a line buffer, one would use 3 separate SRAMs, each storing one row of pixels. At any give cycle, all three SRAMs are being read from and there is one SRAM that is also being written to. The SRAM being written to will rotate. As a result, all the SRAMs must have at least two ports in this example. Note that when an image row is larger than SRAM block size, the row can be split into multiple SRAM blocks without changing the operating principle of line buffers.

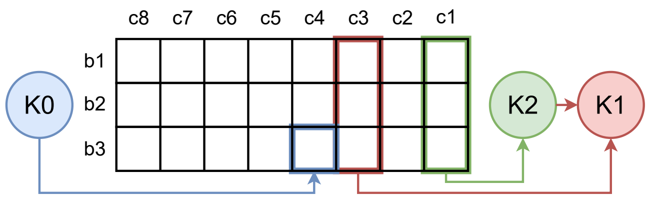


Fig. 2: Illustration of a line buffer with multiple consumers. $K0$ is the producer who is writing to cell c4 in block b3. $K1$ and $K2$ are consumers that are reading columns c3 and c1 respectively. Each line is in an SRAM block. Assuming the common setting where each SRAM has two ports, the pipeline stalls since b3 has to serve three accesses.

Table 1: Prior work makes rigid assumptions about the memory structures, which leads to sub-optimal designs, and/or applies to only specific forms of algorithms.

	Darkroom [16]	**FixyNN [38]**	**SODA [7]**	**Ours**
On-chip memory assumption	Dual port	Single port	Dual port (FIFO)	**Generic**
Algorithm applicability	Single consumer	Single consumer	Generic	**Generic**

3 MOTIVATION

Accelerator design decisions must be made according to the specific algorithm pipeline and the memory resources available. Prior work makes rigid assumptions about the algorithm structures and/or memory resources available. As a result, they either have narrow applicability or result in inefficient designs. We summarize prior work in Tbl. 1 and elaborate next.

3.1 Algorithmic Limitations

The basic design in Sec. 2 assumes dual-port SRAMs and "single-consumer" pipelines, where each producer has only one consumer. Multiple-consumer pipelines such as unsharp mask [16] and denoise2D [7], where multiples consumers read the output from a producer, challenge the simple design. With multiple consumers, one line is accessed by multiple hardware stages. As a result, there could be more accesses to an SRAM block than there are ports, leading to pipeline stalls.

Consider the algorithm in Fig. 1, where $K0$ has two consumers $K1$ and $K2$. The line buffer after $K0$ is simultaneously accessed by three stages (the producer $K0$ and the two consumers). Fig. 2 shows a naive line buffer design, where $K0$ is writing to (b3, c4), and $K1$ and $K2$ are reading columns c3 and c1, respectively. As a result, three different stages are accessing the block b3. If there are only two ports in each SRAM block, b3 will not be able to handle all three accesses. Simply increasing the number of SRAM ports is area-inefficient as SRAM area increases quadratically with the number of ports [37]. Prior work attempts to support multi-consumer pipelines in primarily two ways, each coming with its own downsides, which we explain next.

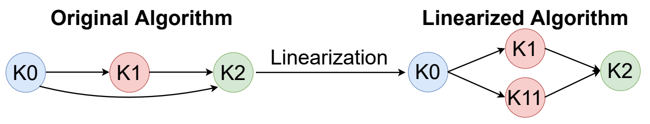
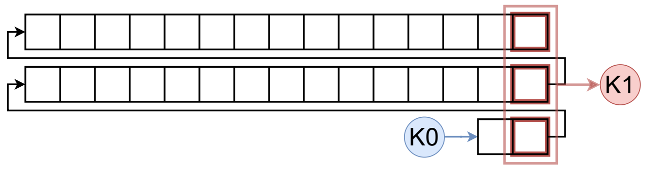
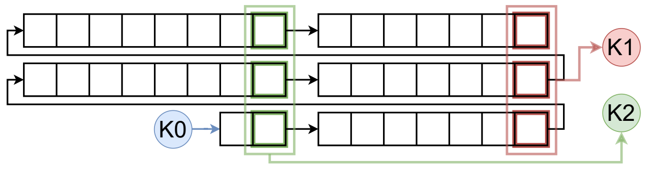


Fig. 3: An example of algorithm linearization. We need an additional line buffer after $K11$. Note that even though $K1$ and $K11$ are both consumers of $K0$, they consume data in exactly the same pattern and act effectively as one consumer.



(a) An FIFO implementation to support a single consumer ($K1$)



(b) An FIFO implementation to support multiple consumers ($K1$ and $K2$).

Fig. 4: Line buffer implementation using FIFO. Figure (a) shows the single consumer case and (b) shows a multiple consumer case, assuming each memory block has two ports. In the multiple consumer case, to accommodate both consumers each FIFO is split into two smaller FIFOs. A FIFO is usually implemented as a dual-port SRAM block. The line being written to by the producer usually holds only a few elements (2 here) and, thus, can be implemented as a shift register using DFFs to save on-chip memory usage.

Algorithm linearization. One can transform an algorithm that has multiple-consumer stages into another functionally identical algorithm with only single-consumer stages, a process dubbed "linearization" by Darkroom [16].

We use Fig. 3 to illustrate linearization. To linearize the pipeline, we add a dummy stage $K11$ between $K0$ and $K2$. Each cycle, $K11$ reads data from $K0$ in *exactly the same pattern* as $K1$ but performs no computation on the data. Essentially, the sole purpose of $K11$ is to simply relay data from $K0$ to $K2$. As a result, $K2$ reads data from $K11$ instead of directly reading from $K0$. Critically, even though $K0$ still has two consumes $K1$ and $K11$, the two consumers read data from $K0$ in exactly the same pattern every cycle, so they effectively act as a single consumer without requiring additional memory port.

The downside of this approach is that the linearized algorithm increases on-chip memory usage than the original algorithm, because each dummy stage requires a dedicated line buffer. In this case, the additional line buffer associated with $K11$ buffers the same data that $K1$ does and is redundant.

Splitting line buffer. Alternatively, one can split a line buffer that has multiple consumers into several smaller line buffers, each serving only one consumer. An implication of this approach is that data from one line buffer must be transferred to another, which requires each line buffer to be realized as a FIFO. This approach is exemplified by SODA [7] and line-buffered designs from Xilinx [5].

Fig. 4a shows FIFO-based line buffer implementation when there is only a single consumer. Usually a FIFO is implemented using a dual-port SRAM/BRAM block. Therefore, every cycle there is one read and one write access to every memory block. When two consumers are trying to access the line buffer, each FIFO must be split into two FIFOs, each in a separate memory block, to accommodate both accesses; this is illustrated in Fig. 4b. Note that if a FIFO becomes very small (e.g., a few elements), which is typically the case for the line that the producer is writing to, it can be implemented as a shift register using DFFs to reduce memory usage.

The downside of FIFOs is high energy consumption, because the nature of FIFO dictates that every cycle there would *always* be two accesses to each SRAM block, whereas in the classic implementation only one of three lines have to serve two accesses; other lines have only one access each cycle. In our FPGA measurement (see Sec. 7 for details), BRAMs with two accesses per cycle consume about 35% more power than BRAMs with only one access per cycle.

3.2 Hardware Limitations

A fundamental limitation of prior approaches is that they use the same form of memory for *all* the algorithms and for *all* the line buffers in an algorithm, e.g., dual-port SRAM in Darkroom [16] or single-port SRAM in FixyNN [38]. The actual design space, however, is much larger. For an algorithm with N stages where a line buffer has M implementation options, there are M^N design points. Navigating a large design space governed by memory resources is especially important in ASIC designs, where, unlike FPGAs where memory resources are fixed, one has the flexibility to customize memories (e.g., size and ports) given area and/or power targets.

Critically, there exists a power-vs-area trade-off when navigating such a large design space. For instance, increasing the number of memory ports increases the per-SRAM area and power consumption but also reduces the number of SRAM blocks needed. As we will show in Sec. 8.5, the exact Pareto-optimal frontier varies across algorithms, a design space exploration that is not possible with existing approaches.

Finally, it is worth noting that prior approaches could not generate any hardware design at all when the available memory resource does not meet their requirement. For instance, the FIFO approach by SODA [7] assumes dual-port memories. It thus does not work when only single-port memories are available, further highlighting the need to consider arbitrary memory configurations, which our framework offers.

4 FRAMEWORK OVERVIEW

Fig. 5 shows the overall workflow of our framework, which takes an image processing pipeline described in a Domain Specific Language (DSL) and the description of available memory resources (i.e., sizes and number of ports) and generates synthesizable Verilog code.

Front End. Literature is rich with DSLs that express image processing pipelines [16, 29], which is *not* the focus of this paper. For simplicity, we use a DSL similar to Darkroom [16]. The code

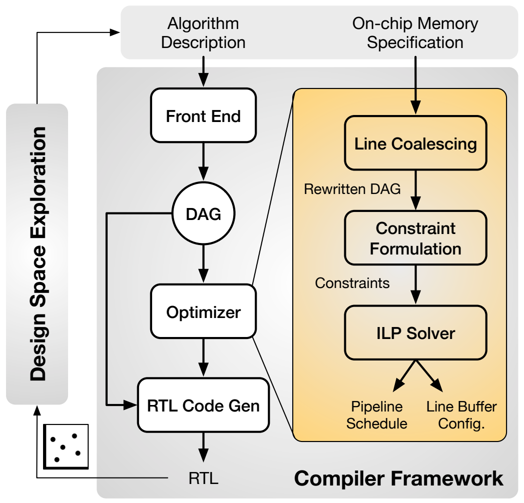


Fig. 5: Compiler framework, which generates synthesizable Verilog code given an image processing algorithm and on-chip memory specifications. The framework can be integrated into a DSE tool to generate Pareto-optimal designs.

below shows the algorithm in Fig. 1. Each stage is defined inside the `im` block. `input` and `output` denote input and output stages of the pipeline, for which off-chip memory accesses are permitted. The front-end parses an algorithm to a DAG as the intermediate representation. Each DAG node is a pipeline stage, and each edge connects a producer-consumer pair. The stencil window sizes are encoded in DAG nodes.

```
input K0;
//K1 reads 3x3 window from K0
K1 = im(x,y)K0(x-1,y-1)+...+K0(x+1,y+1) end
//K2 reads 3x3 window from K1 and 2x2 window from K0
output K2 = im(x,y)K0(x,y)+...+K0(x+1,y+1)+
               K1(x-1,y-1)+...+K1(x+1,y+1) end
```

Optimizer. The core contribution of our work is the optimizer, which takes a DAG and the memory resource specifications to generate accelerator pipeline schedule and on-chip memory configurations, which are then used in generating synthesizable RTL code for by the code generator.

The optimizer forms a constrained optimization problem that minimizes the on-chip memory usage (Sec. 5). Unlike conventional line buffer synthesizers [7, 16, 38], the compiler exploits opportunities to coalesce image rows into the same buffer through a simple DAG re-writing, further reducing on-chip memory usage (Sec. 6).

RTL Code Generation. Given the pipeline schedule and the line buffer configuration, the code generator generates synthesizable RTL. The generated code has compute units that execute the stencil operations, memory blocks that implement line buffers, and control logic to sequence the hardware.

It is worth noting that the code geeneration is largely a *mechanical translation* of arithmetic operations in each pipeline stage to RTL code and, thus, is *not* a contribution of this paper. In fact, any existing HLS tool (e.g., Vivado HLS) can be used for code generation: one can use our optimizer to generate the optimal line buffer configuration, which is then used in the HLS code to annotate the memory sizes. To our best knowledge, today's HLS tools such as Vivado HLS require programmers to explicitly specify line buffer sizes, which our optimizer automatically generates.

Table 2: List of symbols used in our formulation. Subscripts i, j, p, c are used in denoting pipeline stages; p and c, in particular, denote a producer and consumer, respectively. Blackboard-bold symbols $\mathbb{N}, \mathbb{C}, \mathbb{A}$ denote sets.

Symbols	Meaning
N	Total number of stages in the pipeline
LB_i	Size of the line buffer associated with stage i
S_i	Start cycle of a stage i
$\mathbb{N}_i$	Set of stages accessing the line buffer of stage i
SH_i	Stencil height of stage i[1]
$B_{l,t}$	Total number of accesses to a line l at cycle t
P	Number of SRAM ports
W	Width of the input image to each stage[2]
$\mathbb{C}_p$	Set of consumer stages of a producer stage p
$\mathbb{A}_{i,t}$	Set of lines stage i is accessing at cycle t[3]
$L_{i,t}$	The first line that a stage i is accessing at cycle t

5 GENERATING LINE-BUFFERED PIPELINES

We first describe the intuition behind our general idea (Sec. 5.1), followed by a rigorous optimization formulation (Sec. 5.2). We then discuss how on-chip memory contention, the key to our optimization formulation, is modeled (Sec. 5.3), followed by a technique to eliminate redundant hardware constraints (Sec. 5.4). In the end, we show that our formulation amounts of an ILP problem (Sec. 5.5).

5.1 General Idea and Intuition

Objective. Our goal is to minimize the total on-chip memory size while *maintaining the theoretically maximum throughput*, which is quantified by the number of pixels generated per cycle. The theoretically maximum throughput is fundamentally limited by the amount of functional units the hardware can afford to have. Like virtually all prior work [7, 16, 38], we assume that the theoretically maximum throughput is one pixel per cycle. Improving the throughput simply amounts to increasing the compute resources, which is *not* the focus of this paper. Note, however, that the one-pixel-per-cycle assumption is reasonable for real-world applications. Assuming a

[1]A consumer stage can access data from multiple producers and, thus, can have multiple stencil heights. We omit the producer stage from the symbol for simplicity, but the producer stage is evident in each context where h_i appears.
[2]Our current system, as is, can deal with stages without padding, in which case the input image size is different across stages and is trivially calculated given the stencil window size. For the simplicity of the exposition we assume padding and use the same W in the paper.
[3]Similar to SH_i, $\mathbb{A}_{i,t}$ here is tied to a particular producer of i, which could have multiple producers. The producer is omitted in the symbol for simplicity, but should be evident given the context.

100 MHz clock frequency, producing one pixel per cycle is equivalent to providing a 50 frames per second (FPS) frame rate for 1080p images, sufficient for real-time operations.

We emphasize that the accelerator must *consistently* deliver the prescribed throughput across frames. It is unacceptable if some frames are lower that others even if the average frame rate is desirable, because a varying frame rate presents a sluggish user experience. To ensure a consistent throughput, the accelerator pipeline must not stall (once a pipeline starts it never stalls until all the input pixels finish), which translates to meeting three requirements:

$\boxed{R1}$ *Data dependency (causality): any pixel, before can be read by a consumer, must already be generated by the producer and is available in the line buffer*;

$\boxed{R2}$ *No intermediate off-chip memory access: a pixel is evicted from a line buffer only when it is no longer needed by any of its consumers (to avoid DRAM accesses later)*;

$\boxed{R3}$ *No on-chip memory access stall: at any cycle, the number of accesses to any on-chip memory block must be no more than the number of ports.*

Solution Intuition. Intuitively, a generic solution to meeting both memory requirements is to delay the start of certain consumers. For instance, when two consumers of the same producer are allowed to start at the same cycle, each memory block has to serve two simultaneous read accesses. Now if one consumer is delayed by W cycles where W is the width of an image, the two consumers will be reading from two different memory blocks (image rows), avoiding memory-port contention. Delaying the start of a consumer also providing more time for the producer to generate intermediate data.

Delaying consumers, however, increases the line buffer size, because an element is evicted from the line buffer only when it is no longer needed by *any* consumer; delaying a consumer would mean that data will have to stay in the line buffer for longer, increasing the line buffer size. Thus, the central challenge is how to optimally shift: how to schedule different pipeline stages to minimize total line buffer size while meeting the data and hardware constraints.

5.2 Optimization Formulation

Formally, the job of our hardware generator can be described in a constrained optimization formulation:

$$\min_{\phi} LB(\phi) = \sum_{i=0}^{N-1} LB_i(\phi),$$

$$where\ \phi = \{S_i\}, i \in [0, 1, \cdots, N-1] \tag{1a}$$

$$s.t.\ \forall(p,c)\ S_c - S_p \geq (SH_c - 1) \times W + 1, \tag{1b}$$

$$\forall l \forall t\ B_{l,t}(\phi) \leq P. \tag{1c}$$

Optimization Objective. Equ. 1a states the optimization objective. ϕ, the schedule, denotes the collection of the start cycles of all the stages $\{S_i\}$ (i is an integer between 0 and $N-1$, where N is the number of pipeline stages). $LB(\phi)$ denotes the total line buffer size, which is the sum of the N individual line buffer sizes. Recall from Fig. 1 that each stage is associated with a line buffer, so the number of line buffers is the same as the number of pipeline stages, N.

The size of each line buffer is dictated by the start cycles of the producer stage and the consumer stages. For instance, given a producer-consumer pair (p, c) in the pipeline and their start cycles S_p and S_c, there is a $(S_c - S_p)$-cycle delay between the consumer and the producer. According to $\boxed{R2}$ above, the line buffer must have the ability to buffer at least $(S_c - S_p)$ pixels before the consumer starts, because each pixel can be removed from the line buffer only after it has been consumed.

Considering that there could be multiple consumers of the same producer and any element can be removed from the line buffer only after the *last* consumer finishes reading it, the line buffer size of a particular stage p is:

$$LB_p = \left\lceil \frac{\max\{S_c - S_p\}}{W} \right\rceil \times W,\ \forall c \in \mathbb{C}_p, \tag{2}$$

where c is one of p's consumers, denoted by $\mathbb{C}_p$. The ceiling operation is to enforce that a line buffer always stores multiples of a line and, thus, the actual line buffer size must be multiples of W, where W is the image width.

Data Dependency. Equ. 1b states the data dependency requirement $\boxed{R1}$: an element must be in the line buffer before it can be read by its consumer(s). Due to the nature of stencil computations, the data a consumer reads might span multiple image rows. Therefore, a consumer must wait before the line buffer has certain number of pixels. For instance, if we have a consumer whose stencil size is 3×3, the consumer must wait until the line buffer contains two full rows of pixels and one pixel from the third row (see Fig. 1a).

Generally, for any producer-consumer pair (p, c), the consumer start cycle must be delayed $(SH_c - 1) \times W + 1$ cycles after the producer has started, where SH_c is the height of the stencil window read by the consumer.

Equ. 1c states the on-chip memory constraint $\boxed{R3}$, which is far more complicated to express, which we discuss next.

5.3 Modeling On-chip Memory Contention

Equ. 1c states that $B_{l,t}$, the number of accesses to any line l in the line buffer at any given cycle t, must be no more than the number of ports (P) of the SRAM block that contains line l[4]. The key is to mathematically express $B_{l,t}$. To that end, we first express the set of lines that a stage accesses at each cycle.

Consider a pipeline stage i accessing a line buffer at cycle t. The *first* line accessed by stage i at t, denoted by $L_{i,t}$, is:

$$L_{i,t} = \left\lceil \frac{t - S_i}{W} \right\rceil. \tag{3}$$

Thus, the *Access Set* of stage i, i.e., the set of lines that stage i accesses, at cycle t is[5]:

$$\mathbb{A}_{i,t} = \left\{L_{i,t},\ L_{i,t} + 1,\ \cdots,\ L_{i,t} + SH_i - 1\right\}. \tag{4}$$

To satisfy the hardware constraint, no line can belong to the intersection of more than P sets. This is equivalent to saying that

[4]In theory P should be represented as P_l to indicate that each SRAM could have a different number of ports. For simplicity purpose we assume that all the SRAMs in the hardware have the same number of ports. Our formulation, nevertheless, can be easily extended to support different port counts.

[5]A subtlety here is that the stencil height for the stage that writes to the line buffer is always 1.

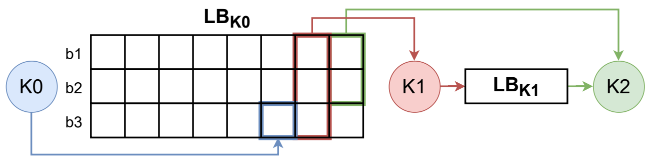


Fig. 6: Example illustrating how to calculate Access Sets. At the current cycle t shown in the figure, $K0$'s Access Set is $A_{0,t}$=(b3), $K1$'s Access Set is $A_{1,t}$=(b1,b2,b3), and $K2$'s Access Set is $A_{2,t}$=(b1,b2). Assuming each line stores in a dual-port SRAM, the hardware constraint would be $A_{0,t} \cap A_{1,t} \cap A_{2,t} = \emptyset$ (Equ. 6), which after constraint pruning is reduced to $A_{0,t} \cap A_{2,t} = \emptyset$ (Sec. 5.4).

the intersection of any $(P+1)$-combination is always an empty set. Hence, the hardware constraint at each stage i can be expressed mathematically as:

$$\forall t \; \forall \mathcal{T} \in \binom{\mathbb{N}_i}{P+1} \bigcap_{i \in \mathcal{T}} \mathbb{A}_{i,t} = \emptyset, \tag{5}$$

where $\binom{\mathbb{N}_i}{P+1}$ denotes the set of all $(P+1)$-combinations of $\mathbb{N}_i$, which itself is the set of all the stages that access the line buffer of stage i.

To concretize Equ. 5, consider the example in Fig. 6, where the line buffer LB_{K0} is accessed by one producer ($K0$) and two consumers ($K1$ and $K2$). Assuming the SRAM has two ports, a common configuration in SRAM/BRAM blocks, the hardware constraint for LB_{K0} is expressed as:

$$\forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{1,t} \cap \mathbb{A}_{2,t} = \emptyset \tag{6}$$

where $\mathbb{A}_{0,t}$, $\mathbb{A}_{1,t}$, and $\mathbb{A}_{2,t}$ are the Access Sets of stages $K0$, $K1$, and $K2$, respectively, at cycle t.

To enforce this constraint, one would enforce *any* of the following three constraints, the intuition being if the intersection of any two sets is empty the intersection of all three sets is necessarily empty:

$$\forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{1,t} = \emptyset, \tag{7a}$$
$$\forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{2,t} = \emptyset, \tag{7b}$$
$$\forall t \; \mathbb{A}_{1,t} \cap \mathbb{A}_{2,t} = \emptyset. \tag{7c}$$

Equ. 7a – Equ. 7c involve calculating set intersections (or, equivalently, counting), not amenable to usual numerical optimizations. We must transform them to equivalent numerical expressions.

Without losing generality, consider the constraint $\forall t \; \mathbb{A}_{i,t} \cap \mathbb{A}_{j,t} = \emptyset$. Enforcing it is equivalent to enforcing that the last line written by s_i (at any cycle t) must be above the first line read by s_j. That is:

$$\forall t \; L_{i,t} + SH_i - 1 < L_{j,t}, \tag{8}$$

which, after applying Equ. 3, becomes:

$$\forall t \; \left\lceil \frac{t - S_i}{W} \right\rceil + SH_i - 1 < \left\lceil \frac{t - S_j}{W} \right\rceil. \tag{9}$$

Equ. 9 depends on t, which does not have an upper bound. Therefore, t must be eliminated for the constraint to be usable. Our strategy is to (somehow) remove the ceiling operator ($\lceil \; \rceil$), which would allow t to be canceled out from both sides of Equ. 9. To that end, observe that:

$$x \le \lceil x \rceil < x + 1, \tag{10}$$

from which we can derive:

$$\left\lceil \frac{t - S_i}{W} \right\rceil < \left(\frac{t - S_i}{W} \right) + 1, \; and$$
$$\left(\frac{t - S_j}{W} \right) \le \left\lceil \frac{t - S_j}{W} \right\rceil. \tag{11}$$

Combining Equ. 11 with Equ. 9, we can transform Equ. 9 into the following constraint:

$$\forall t \; \left(\frac{t - S_i}{W} \right) + 1 + SH_i - 1 \le \left(\frac{t - S_j}{W} \right)$$
$$\equiv S_i - S_j \ge W \times SH_j. \tag{12}$$

Equ. 12 is now independent of t. Note that Equ. 12 is a stricter constraint than Equ. 9[6], which means the solutions obtained with Equ. 12 is a subset of those obtained with Equ. 9, sacrificing the solution optimality. The desirable trade-off, however, is that the constraint is independent of t. Equ. 12 is then applied to re-write Equ. 7a, Equ. 7b, and Equ. 7c.

5.4 Constraint Pruning

One potential issue is that the constraints in Equ. 7a, Equ. 7b, and Equ. 7c are to be "OR-ed"; that is, only one of the three constraints needs to be satisfied. Normally, this would require us to formulate three different sub-optimization problems, each of which considers one of the constraints individually. When a pipeline has multiple stages, each of which has constraints that are to be "OR-ed", the total number of sub-problems grows combinatorially.

To reduce the optimization time, we observe that constraints in Equ. 7a, Equ. 7b, and Equ. 7c are not mutually exclusive, which allows us to prune some of them. We use the example in Fig. 6 to provide the intuition of constraint pruning, and then discuss how it is extended to general cases.

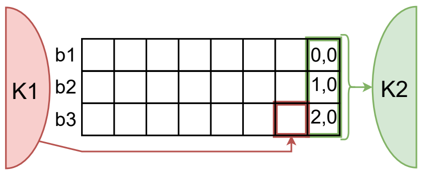
An Example. Observe that the constraint in Equ. 7b is more *relaxed* than that of Equ. 7a and Equ. 7c. That is, if Equ. 7a (or Equ. 7c) holds, Equ. 7b necessarily holds:

$$\forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{1,t} = \emptyset \implies \forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{2,t} = \emptyset, \tag{13a}$$
$$\forall t \; \mathbb{A}_{1,t} \cap \mathbb{A}_{2,t} = \emptyset \implies \forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{2,t} = \emptyset \tag{13b}$$

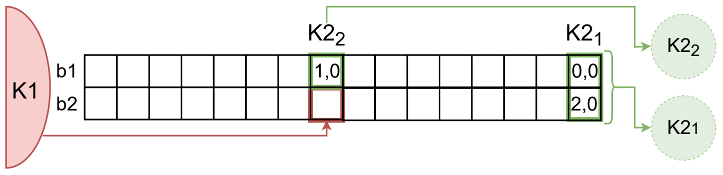
where $A \implies B$ reads "A implies B."

Intuitively, Equ. 13a holds because stage $K2$ data-depends on stage $K1$, which implies that $K2$ must start after $K1$. Therefore, at any time the first line $K2$ writes to must be below the first line $K1$ writes to (Equ. 14a), which in turn must be below the last line $K0$ writes to (Equ. 14b) given $\forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{1,t} = \emptyset$. Therefore, transitivity dictates that the first line $K2$ writes to is below the last line $K0$

[6]The proof is a simple application of the transitivity of the "less than" (<) relation, which we omit here due to space limit.



(a) Without line coalescing.



(b) With line coalescing.

Fig. 7: Illustration of line coalescing optimization. Figure (a) shows the implementation with line combination and (b) shows that with line combination, assuming each memory block has two ports. In line combination, we place two consecutive lines in the same memory block. Effectively, the consumer stage $K2$ in the DAG is replaced with two "virtual" stages, $K2_1$ and $K2_2$, each of which has a stencil height of 2 and 1, respectively.

writes to (Equ. 14c), hence $\forall t\ \mathbb{A}_{0,t} \cap \mathbb{A}_{2,t} = \emptyset$.

$$First(\mathbb{A}_{2,t}) > First(\mathbb{A}_{1,t}), \tag{14a}$$
$$First(\mathbb{A}_{1,t}) > Last(\mathbb{A}_{0,t}) \tag{14b}$$
$$\implies First(\mathbb{A}_{2,t}) > Last(\mathbb{A}_{0,t}) \tag{14c}$$

The validity of Equ. 13b can be similarly reasoned about using the fact that $K1$ data-depends on $K0$.

In general, given two constraints A and B that are to be "ORed", if A is more relaxed than B, it is safe to eliminate B without sacrificing optimality, because any solution that satisfies B must also satisfy A. Therefore, in the example above we could eliminate constraints in Equ. 7a and Equ. 7c.

Generalization. The example above shows that data dependency is key in eliminating redundant constraints. In particular, data dependencies allow us to form partial orders $\preccurlyeq$ between stages. If there is a path in the DAG from stage i to stage j, i.e., stage j (directly or indirectly) depends on stage i, we have a partial order $i \preccurlyeq j$. Reflectivity holds for partial order relation: $i \preccurlyeq i$. We have the following theorem.

THEOREM. Given two generic constraints C_1 and C_2:

$$C_1 : \forall t\ \mathbb{A}_{x,t} \cap \mathbb{A}_{y,t} = \emptyset \tag{15}$$
$$C_2 : \forall t\ \mathbb{A}_{z,t} \cap \mathbb{A}_{w,t} = \emptyset \tag{16}$$

C_1 is more relaxed than C_2 (i.e., C_2 implies C_1) if $x \preccurlyeq z$, $w \preccurlyeq y$, and $SH_x \leq SH_z$.

PROOF. $x \preccurlyeq z$ leads to Equ. 17a, which combined with $SH_x \leq SH_z$ gives Equ. 17b; $w \preccurlyeq y$ gives Equ. 17c; C_2 gives Equ. 17d; transitivity thus yields Equ. 17e, which leads to C_1. Thus, C_2 implies C_1 and can be eliminated given C_1.

$$First(\mathbb{A}_{x,t}) < First(\mathbb{A}_{z,t}), \tag{17a}$$
$$Last(\mathbb{A}_{x,t}) < Last(\mathbb{A}_{z,t}), \tag{17b}$$
$$First(\mathbb{A}_{w,t}) < First(\mathbb{A}_{y,t}), \tag{17c}$$
$$Last(\mathbb{A}_{z,t}) < First(\mathbb{A}_{w,t}), \tag{17d}$$
$$\implies Last(\mathbb{A}_{x,t}) < First(\mathbb{A}_{y,t}) \tag{17e}$$

The theorem is a pruning rule: we examine each pair of constraints and eliminate the stricter one, if it exists, using the pruning rule. Note that given two constraints it is possible that one can *not* make a judgment as to which is more relax/stricter, because there might not always be a partial order between two stages in a DAG.

5.5 Problem Structure and Solver

The optimization problem we formulate is an Integer Linear Programming problem: the optimization variables are the start cycles of each stage—all integers; the objective function (Equ. 1a) and the constraints (Equ. 1b and Equ. 12) are all linear. Note that the ceiling operations in the sub-terms of the objective function (Equ. 2) can be removed without compromising the solution optimality, because minimizing $f(\lceil x \rceil)$ is equivalent to minimizing $f(x)$ given f is monotonically increasingly [7]. The ILP formulation let us use well-established solvers to quickly derive optimal line buffer designs.

6 LINE-COALESCING OPTIMIZATION

So far, we have assumed that each memory (e.g., SRAM/BRAM) block contains one line. It is possible, however, that the capacity of a memory block is large enough to hold multiple lines, in which case combining multiple lines into one single memory block would further reduce the memory requirement. The challenge is how to generate the line-buffered pipeline under line coalescing. We show that our optimization formulation above can be naturally extended to support optimal line coalescing.

Consider the example in Fig. 7, where there are two stages, a producer $K1$ and a consumer $K2$; the consumer operates on a stencil height of 3. Assume for now that each memory block has two ports. Fig. 7a and Fig. 7b show the line buffer implementation without and with line coalescing, respectively. Since each memory block has two ports, we could coalesce up to two lines into one memory block, which is shown in Fig. 7b. The three elements that $K2$ accesses are now spread across two, rather than three, memory blocks, as elements $(0, 0)$ and $(1, 0)$ are in the same memory block.

To express the line-coalesced pipeline to the optimizer, our observation is that line coalescing is equivalent to a transformation of the DAG, where the original stage $K2$ is replaced with two new "virtual" stages $K2_1$ and $K2_2$. In this example, b1 is accessed simultaneously by the two virtual stages, whereas b2 is accessed by only $K2_1$ (along with the producer $K1$). Thus, the virtual stage $K2_1$ now has an effective stencil height of 2, and the virtual stage $K2_2$ has an effective stencil height of 1. Both virtual stages inherit the producer and consumers of the original stage $K2$.

[7]More rigorously, we have: $\operatorname{argmin} f(x) \in \operatorname{argmin} f(\lceil x \rceil)$. That is, the solution that minimizes $f(x)$ is necessarily a solution that minimizes $f(\lceil x \rceil)$ for any monotonically increasing function f. To prove this, let $x_0 = \operatorname{argmin} f(x)$; then $\forall x\ f(x) \geq f(x_0)$, so $\forall x\ x \geq x_0$ (since f is monotonically increasing). Thus, $\forall x\ \lceil x \rceil \geq \lceil x_0 \rceil$, which means $\forall x\ f(\lceil x \rceil) \geq f(\lceil x_0 \rceil)$, i.e., $x_0 \in \operatorname{argmin} f(\lceil x \rceil)$.

We can generalize line coalescing to memory blocks with P ports, where we can coalesce at most P lines in one block and replace the original consumer stage with P virtual stages. This transformation can be done offline, since it depends only on the algorithm DAG and stencil sizes. Algo. 1 describes the general transformation algorithm.

Algorithm 1: Line coalescing algorithm through DAG rewriting. Notation: P is the number of ports, SH_i is the stencil height read by stage i.

```
Data: The original DAG
Result: The transformed DAG
i = input node of the original DAG;
while i is not an empty node do
    if i is not the input node then
        $K = \min(P, SH_i)$;
        split $i$ into $K$ virtual stages;
        for each virtual stage v split from i do
            set $v$'s producer to $i$'s producer;
            set $v$'s consumers to $i$'s consumers;
        end
    end
    $i$ = next node through breath-first search;
end
return the new DAG;
```

From the optimizer's perspective, the transformed DAG is nothing more than another pipeline except all the virtual stages belonging to the same physical stage must share the same start cycle, because logically they must act synchronously. Using the optimization formulation in Equ. 1, the optimizer generates the optimal start cycles for every stage in the new DAG. One special care the code generator takes is that virtual stages that belong to the same physical stage read from a different, but offline-determined offset. For instance in Fig. 7, $K2_2$ will always read from an offset of W (image width) from b1, where $K2_1$ and $K1$ have an offset of 0.

Remarks. We note that the line coalescing optimization is fundamentally incompatible with the FIFO-based approach [7] or designs that assume single-port memories [38]—simply observe the data access behaviors in Fig. 7b.

Line coalescing benefits both an FPGA and an ASIC backend. On FPGAs, BRAM block sizes are fixed on any particular board; forcing each block to hold only one line could result in internal fragmentation of BRAM blocks. ASICs designers could customize the memory for an algorithm; they could properly size the memory blocks to permit line coalescing to reduce the overall area (Sec. 8.5).

7 EXPERIMENTAL METHODOLOGY

Compiler Implementation. We implement our compiler in Python with about 1,500 lines of code. We use Google's optimization library "or-tools" [2] for solving the ILP problem.

Hardware Platform. We evaluate both an ASIC flow and an FPGA flow. We use a Xilinx Spartan-7 FPGA board (xa7s100fgga488-2I) for evaluation. The board has 120 BRAM blocks and each block is of size 36 Kbits. Each block can be configured as either a single-port or a dual-port memory block. We assume SRAM blocks are available at 64 KB for line buffers in the ASIC backend. We evaluate two image resolutions: $320p$ (480×320) and $1080p$ (1920×1080). The SRAM and BRAM block sizes make sure line coalescing applies to $320p$ but not $1080p$, since the block size is not large enough to hold multiple rows in an $1080p$ image.

For the FPGA backend the generated Verilog code goes through the FPGA synthesis and layout flow using Vivado Design Suite 2021.1. We use Vivado's resource monitor to report the BRAM usage. The FPGA communicates with the host through AXI DMA. Through DMA, we first load the input image to the BRAM from the host memory. The frame rate reported (1 pixel per cycle) is the throughput after the accelerator has started, i.e., steady-state throughput. For each design, we perform post-implementation functional simulation to obtain the switching activity, which is then used by Vivado's power analysis tool to obtain power consumption.

For the ASIC backend we build a cycle-level simulator to simulate the line-buffered pipelines. We use the open-source memory compiler OpenRAM[14] with FreePDK45 [1] to estimate the per-access SRAM power, which is then combined with the number of accesses given by our simulator to estimate the total memory power.

Since the goal of this paper is to reduce on-chip memory size and energy, we primarily report results related to the on-chip memory but will also show savings for the entire accelerator. Note that the memory area dominates the accelerator area, so the memory area/power savings are expected to translate to similar total area/power savings. In the ASIC backend, the SRAM area contributes to, on average, 79.8% and 92.7% of the total accelerator area across all algorithms on $320p$ and $1080p$ images, respectively. The reason memory area dominates is that there are very few PEs in line-buffered accelerators: to execute a 3×3 convolution, regardless of the input, we require only 3×3 MAC units (see Fig. 1). The total area, on average, is 0.65 mm^2 and 1.84 mm^2, for the two resolutions, respectively, and the total average power is 72.9 mW and 98.3 mW, for the two resolutions, respectively.

Table 3: Evaluation algorithms. -s or -m indicates if an algorithm has only single-consumer stages or has at least one multiple-consumer stage, respectively. The last column shows the number of multiple-consumer (MC) stages.

Algorithm	Description	# Stages	# of MC Stages
Canny-s	Canny edge detection	9	0
Canny-m		10	1
Harris-s	Harris corner detection	7	0
Harris-m		7	1
Unsharp-m	Unsharp masking	5	1
Xcorr-m	Cross correlation	3	1
Denoise-m	Image denoise	5	2

Algorithms. We evaluate common image processing algorithms listed in Tbl. 3, where each algorithm either ends with an "-s", indicating it has only single-consumer stages or with an "-m", indicating it has at least one multiple-consumer stage. Both Canny and Harris has two versions depending on the implementation details.

Baselines and Variants. We compare with three common line-buffered image processing accelerators.

- FixyNN [38], which is based on the same design described in Sec. 2 but uses only single-port SRAMs.
- SODA [7], which uses FIFO to implement line buffers and splits FIFOs to support multiple-consumer stages (Sec. 3). The FIFOs are implemented using dual-port SRAMs (rather than shift registers).
- Darkroom [16], which linearizes multiple-consumer pipelines (Sec. 3) and uses two-port SRAMs.

We consider two variants of our framework: OURS and OURS+LC. The latter adds line-coalescing to the former.

8 EVALUATION RESULTS

We first show that our compiler maintains the theoretical maximum throughput (Sec. 8.1) and is fast to execute (Sec. 8.2). We then show that the hardware generated by our framework consumes less memory resource (Sec. 8.3) and lower power (Sec. 8.4) compared to existing methods. Finally, we show that our framework can help customize memory modules for individual algorithms (Sec. 8.5).

8.1 Throughput and Latency

Across all algorithms, hardware generated by our compiler maintains a constant throughput of one pixel per cycle, the target laid out and justified in Sec. 5.1. OURS increases the average end-to-end latency by only 0.01% over Darkroom and SODA. Thus, the memory and power savings shown later come with no speed degradation.

8.2 Compilation Speed

On average, our compiler takes 14.5 ms to generate the Verilog code across all the algorithms. For multiple-consumer algorithms, constraint pruning (Sec. 5.4) speeds up the compilation time by 4× on average. This is achieved by pruning redundant constraints that would have led to many sub-optimization problems. Compared to Darkroom's linearization compiler, our compiler, on an average, compiles 37.4% faster. This is because linearization adds adds dummy stages, which adds more constraints to the ILP.

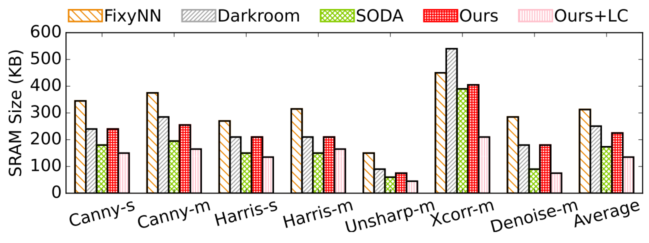
Scalability. We also sweep across different pipelines of varying length from 9 to 60. In each algorithm a third of the stages had multiple consumers. It took 8.7 ms for our compiler to compile 9 stage pipeline and 8.1 s to compile the 60 stage pipeline, showing the scalability of our compiler.

8.3 On-Chip Memory Requirement Reduction

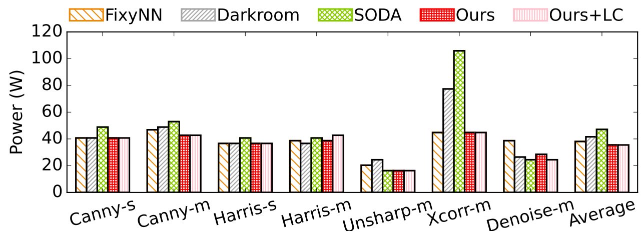
Accelerators generated by our framework reduce the on-chip memory size significantly. Fig. 8a compares the SRAM size of the hardware generated by our framework and the three baselines on 320p images. Averaging over all the algorithms, OURS reduces the SRAM size by 28.0% and 10.2% compared to FixyNN and Darkroom, respectively. After the line-coalescing optimization is applied, the SRAM savings over the baselines increase to 86.0% and 56.8%, respectively.

The SRAM saving on multiple-consumer algorithms is noticeably higher than that on single-consumer algorithms, highlighting the benefits of our framework on the former. On multiple-consumer algorithms, algorithm linearization adds dummy stages and increases line buffer size. FixyNN always has a higher SRAM requirement than OURS, even on single-consumer algorithms, because it uses only single-port memory blocks, where no two stages are allowed to overlap. The SRAM saving is particularly significant on Xcorr-m. This is because when linearizing Xcorr-m, one of the stages that are replicated operates on a tall stencil window (18×1); replicating that stage adds a lot of additional SRAM blocks.

The SRAM requirement of OURS is 31.0% higher than SODA, because SODA, being a FIFO-based approach, is able to implement the last line in the line buffer (the line being written to by the producer) as DFFs (Fig. 4a). With line coalescing, OURS+LC reduces the SRAM requirement by 28.5% compared to SODA.

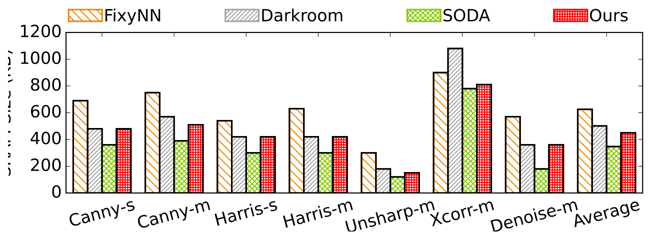


(a) SRAM size.

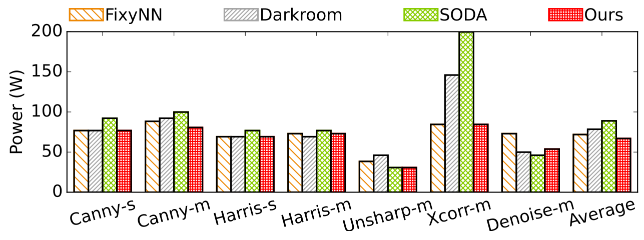


(b) Power consumption.

Fig. 8: SRAM and power comparison on 320p images.



(a) SRAM size.



(b) Power consumption.

Fig. 9: SRAM and power consumption comparison on 1080p images. Note that OURS+LC is not shown, since the SRAM size is not large enough to hold more than one lines.

Fig. 9a shows that the SRAM saving trend on 1080*p* inputs is similar to that on 320*p* inputs, except that the line coalescing optimization could not be applied to 1080*p* images, since the SRAM block size is not large enough to hold more than one line, as discussed in Sec. 7.

Accelerator Results. Memory area dominates the accelerator area, as discussed in Sec. 7. Thus, the memory size saving translates to similar total accelerator area saving. For instance, compared to FixyNN and Darkroom on 320*p* images, Ours+LC saving the total area by 51.2% and 41.9%, respectively. The savings are 27.9% and 12.9% on 1080*p* images.

FPGA Results. Due to the space limit we summarize the main results from the FPGA implementation. On 1080*p* images, Ours reduces the BRAM size by 28.1% and 10.2% compared to FixyNN and Darkroom, respectively, and increase the BRAM usage by 22.8% over SODA, for the same reason described above. On our FPGA, Ours uses 37.5% of the BRAM blocks as opposed to 41.8% by Darkroom.

Multiple Algorithms. Our goal is *not* a generic stencil accelerator that runs multiple algorithms. Rather, we focus on accelerators that are specialized for a given algorithm. Nevertheless, by reducing memory usage our compiler can also help generic stencil accelerators that has one single memory system — by accommodating more algorithms simultaneously. For instance, on our FPGA with 120 BRAM blocks, FixyNN and Darkroom could not simultaneously execute all six algorithms even in the 320*p* resolution because of the BRAM constraint. With Ours+LC, however, the FPGA can accommodate all six algorithms using only 84 BRAM blocks.

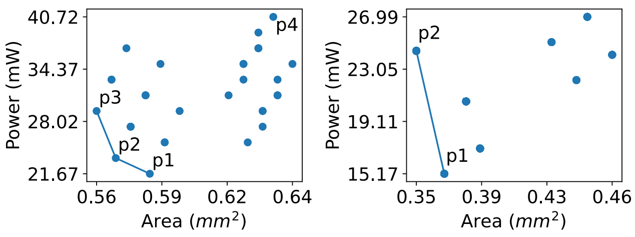
8.4 Power Consumption Reduction

We also generate accelerators that consume lower memory power compared to all baselines. Fig. 8b compares the power consumption on 320*p* images. On average, Ours consumes 7.8%, 13.8%, and 56.0% less power than FixyNN, Darkroom, and SODA respectively. Line coalescing does not change the power by much, since the total memory accesses remain roughly the same. The power savings over Darkroom and FixyNN come from the SRAM size reduction. For instance, while FixyNN, which uses only single-port memories, has lower per-access power, using single-port memories results in more SRAMs, increasing the total power.

It is interesting to observe that Ours has lower power compared to SODA even though Ours require more SRAM arrays than SODA (Fig. 8a). This is because SODA uses FIFOs, which have to serve two accesses *every* cycle. In our design, all but one SRAM array serve only one access per cycle, leading to an overall power reduction. The power saving of Ours+LC over SODA comes from both reducing the SRAM requirement and avoiding power-hungry FIFOs.

Fig. 9b compares the power consumption using 1080*p* images. Ours consumes 7.8%, 13.8%, and 56.0% less power than FixyNN, Darkroom, and SODA, respectively. Again, even though Ours uses more SRAM than SODA, it has lower power consumption because it avoids power-hungry FIFOs.

Accelerator Results. Memory power savings translate to similar accelerator-level savings. On 320*p* images Ours consumes 11.7% and 15.2% less power compared to Darkroom and SODA, respectively; on 1080*p* images the savings are 11.9% and 18.3%.



(a) Canny-m. **(b) Denoise-m.**

Fig. 10: Representative power-vs-area trade-offs under 320*p* for Canny-m and Denoise-m. For each algorithm, we sweep the memory configuration for each stage and generate a corresponding optimal design. Each stage is allowed to use either double-port memory (DP) or DP with line coalescing (DPLC). The Pareto-optimal designs vary with algorithms.

FPGA Results. The power saving trend on the FPGA is similar. On 1080*p* inputs, Ours consumes 19.7%, 5.8%, and 17.7% less power than FixyNN, Darkroom, and SODA, respectively. The FPGA power saving is lower, because FPGAs consume non-trivial static power.

8.5 Design Space Exploration

Evaluation so far assumes that one type of line buffer is used for all the stages in all algorithms, which is the only option in prior work [7, 16, 38]. Since our framework permits specifying arbitrary memory configurations, it can be used (by ASIC designers) to create custom memory modules to explore the power-vs-area trade-off in an *algorithm-specific* manner. Specifically, we allow each line buffer in an algorithm to be implemented as either a double-port memory (DP) or DP with line coalescing (DPLC). For each algorithm, we then sweep all the possible memory configuration combinations and generate the corresponding optimal design for each combination. For example, if there are four stages in an algorithm, we would end up with 16 different designs.

We observe that the Pareto-optimal designs vary with algorithm. Using 320*p* as an example, Fig. 10a shows the power-vs-area comparison for Canny-m, where there are three Pareto-optimal designs. P1 uses the DP configuration for all the stages; in P2, one stage uses the DPLC configuration, and in P3 two stages use the DPLC configuration. In contrast, Fig. 10b shows another trade-off pattern possessed by Denoise-m, where there are only two Pareto optimal configurations. In this case, P1 uses only DP for all the stages and P2 uses only DPLC for all the stages.

We particularly note that for Canny-m the design that uses DPLC for all stages is P4 in Fig. 10a, which is far worse than the three Pareto-optimal designs of Canny-m. DPLC reduces the number of SRAM arrays and the total area, but the per access power also increases. Thus, the total power depends on the total memory accesses, which is necessarily algorithm-specific, an exploration that is uniquely enabled by our tool.

9 RELATED WORK

Agile accelerator design has received considerable attention. A recent theme is languages that close the gap between high-level algorithm semantics and hardware design [20, 21, 26, 33]. These languages allow high-level descriptions of an algorithm and expose the hardware as a set of primitive components. Our work focuses on one particular kind of algorithm domain (image processing), and one particular aspect of hardware components: line-buffered on-chip memory.

Dahlia [26] uses a type system to reject programs that could have unpredictable behaviors in hardware — memory contention being one of them. Our work encodes memory-port contention as a constraint and generates (resource-optimal) hardware that avoids contention. Aetherling [8] is a DSL that compiles high-level image processing algorithms to hardware with the focus of exploring resource-vs-throughput trade-offs. It does not guarantee minimizing memory resource consumption, which we do. HeteroHalide [22] and HeteroCL [21] synthesize hardware accelerators and rely on SODA [7] to generate the on-chip memory system. HalideHLS [28] generates accelerators for image processing algorithms, but rely on the user to optimize the on-chip memory. DSAGEN [36] annotates algorithms using pragmas and automatically searches a large architecture design space for a range of algorithms.

An orthogonal effort is mapping/scheduling an algorithm onto a fixed hardware substrate. Prior work uses constrained optimization methods [11, 19, 27], targeting mostly deep learing workloads. They leverage the fact that the accelerator design space can usually be parameterized and behaviors of algorithms of interest can be mechanically modeled, two traits that our work leverages, too.

10 CONCLUSION AND FUTURE WORK

This paper presents a framework that automatically synthesizes accelerators for image processing. The key is an optimization formulation that permits expressing memory contention as a generic constraint. The explicit memory-constrained optimization allows us to avoid manual heuristics and customize designs in an application-specific way. We show that accelerators generated by our framework reduce on-chip memory usage and power by up to 86.0% and 62.9%, respectively, when compared to state-of-the-art methods.

We demonstrate our framework on image processing because it is central to emerging applications such as autonomous machines. Fundamentally, however, our framework is not limited to image processing; rather, it generalizes to all stencil algorithms, which are central to scientific computing, many of which operate on generic meshes rather than images [18]. Our main technical novelty, i.e., expressing on-chip memory contention in a way that is amenable to numerical optimization, generalizes to any regular algorithm accessing arbitrary on-chip memories, not just line buffers.

Interesting lines of future work include automatically synthesizing sparse image processing accelerators [13, 23] and accelerators that operate on irregular visual data such as meshes and point clouds [9, 10, 39]. These are application domains where accelerators are almost exclusively manually designed.

11 ACKNOWLEDGEMENTS

We thank the anonymous reviewers from HPCA 2023 and ISCA 2023 for their valuable feedback. Jingwen Leng and Yuhao Zhu are the corresponding co-authors. The work was supported, in part, by NSF under grants #2044963 and #2126642.

REFERENCES

[1] [n. d.]. FreePDK45. https://eda.ncsu.edu/freepdk/freepdk45/.
[2] [n. d.]. Google OR-Tools. https://developers.google.com/optimization.
[3] [n. d.]. Mali-C55. https://developer.arm.com/Processors/Mali-C55.
[4] [n. d.]. Snapdragon Makes Significant Leap for Mobile Cameras with Qualcomm Spectra Image Signal Processor and Snapdragon Sight. https://futurumresearch.com/snapdragon-makes-significant-leap-for-mobile-cameras-with-qualcomm-spectra-image-signal-processor-and-snapdragon-sight/.
[5] Daniele Bagni, Pari Kannan, and Stephen Neuendorffer. 2017. Demystifying the Lucas-Kanade optical flow algorithm with Vivado HLS. *Tech. note XAPP1300. Xilinx* (2017).
[6] Nanchini Chandramoorthy, Giuseppe Tagliavini, Kevin Irick, Antonio Pullini, Siddharth Advani, Sulaiman Al Habsi, Matthew Cotter, John Sampson, Vijaykrishnan Narayanan, and Luca Benini. 2015. Exploring Architectural Heterogeneity in Intelligent Vision Systems. In *Proc. of HPCA*.
[7] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.
[8] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 408–422.
[9] Yu Feng, Gunnar Hammonds, Yiming Gan, and Yuhao Zhu. 2022. Crescent: taming memory irregularities for accelerating deep point cloud analytics. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 962–977.
[10] Yu Feng, Boyuan Tian, Tiancheng Xu, Paul Whatmough, and Yuhao Zhu. 2020. Mesorasi: Architecture support for point cloud analytics via delayed-aggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1037–1050.
[11] Yu Feng, Paul Whatmough, and Yuhao Zhu. 2019. Asv: Accelerated stereo vision system. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 643–656.
[12] Yiming Gan, Yu Bo, Boyuan Tian, Leimeng Xu, Wei Hu, Shaoshan Liu, Qiang Liu, Yanjun Zhang, Jie Tang, and Yuhao Zhu. 2021. Eudoxus: Characterizing and accelerating localization in autonomous machines industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 827–840.
[13] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse dnn models without hardware-support via tile-wise sparsity. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.
[14] Matthew R. Guthaus, James E. Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. 2016. OpenRAM: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–6. https://doi.org/10.1145/2966986.2980098
[15] Samuel W Hasinoff, Dillon Sharlet, Ryan Geiss, Andrew Adams, Jonathan T Barron, Florian Kainz, Jiawen Chen, and Marc Levoy. 2016. Burst photography for high dynamic range and low-light imaging on mobile cameras. *ACM Transactions on Graphics (ToG)* 35, 6 (2016), 1–12.
[16] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.* 33, 4 (2014), 144–1.
[17] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible Multi-Rate Image Processing Hardware. In *Proc. of SIGGRAPH*.
[18] Justin Holewinski, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*. 311–320.
[19] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. 2021. Cosa: Scheduling by constrained optimization for spatial accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 554–566.
[20] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 296–311.
[21] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 242–251.

[22] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From image processing DSL to efficient FPGA acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 51–57.

[23] Zhi-Gang Liu, Paul N Whatmough, Yuhao Zhu, and Matthew Mattina. 2022. S2ta: Exploiting structured sparsity for energy-efficient mobile cnn acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 573–586.

[24] Mostafa Mahmoud, Bojian Zheng, Alberto Delmás Lascorz, Felix Heide, Jonathan Assouline, Paul Boucher, Emmanuel Onzon, and Andreas Moshovos. 2017. IDEAL: Image denoising accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 82–95.

[25] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J Sorin. 2016. The microarchitecture of a real-time robot motion planning accelerator. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 45.

[26] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 393–407.

[27] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2013. A general constraint-centric scheduling framework for spatial architectures. *ACM SIGPLAN Notices* 48, 6 (2013), 495–506.

[28] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 3 (2017), 1–25.

[29] 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 Notices* 48, 6 (2013), 519–530.

[30] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, et al. 2021. Warehouse-scale video acceleration: co-design and deployment in the wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 600–615.

[31] Jason Redgrave, Albert Meixner, Nathan Goulding-Hotta, Artem Vasilyev, and Ofer Shacham. 2018. Pixel Visual Core: Google's Fully ProgrammableImage, Vision, and AI Processor For Mobile Devices. In *Proc. IEEE Hot Chips Symp.(HCS)*. 1–18.

[32] Jacob Sacks, Divya Mahajan, Richard C Lawson, and Hadi Esmaeilzadeh. 2018. Robox: an end-to-end solution to accelerate autonomous control in robotics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 479–490.

[33] Robert Stewart, Kirsty Duncan, Greg Michaelson, Paulo Garcia, Deepayan Bhowmik, and Andrew Wallace. 2018. RIPL: A parallel image processing language for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11, 1 (2018), 1–24.

[34] Amr Suleiman, Zhengdong Zhang, Luca Carlone, Sertac Karaman, and Vivienne Sze. 2019. Navion: A 2-mW Fully Integrated Real-Time Visual-Inertial Odometry Accelerator for Autonomous Navigation of Nano Drones. *IEEE Journal of Solid-State Circuits* 54, 4 (2019), 1106–1119.

[35] Artem Vasilyev, Nikhil Bhagdikar, Ardavan Pedram, Stephen Richardson, Shahar Kvatinsky, and Mark Horowitz. 2016. Evaluating Programmable Architectures for Imaging and Vision Applications. In *Proc. of MICRO*.

[36] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 268–281.

[37] Neil HE Weste and David Harris. 2015. *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India.

[38] Paul N Whatmough, Chuteng Zhou, Patrick Hansen, Shreyas Kolala Venkataramanaiah, Jae-sun Seo, and Matthew Mattina. 2019. Fixynn: Efficient hardware for mobile computer vision via transfer learning. *arXiv preprint arXiv:1902.11128* (2019).

[39] Tiancheng Xu, Boyuan Tian, and Yuhao Zhu. 2019. Tigris: Architecture and algorithms for 3d perception in point clouds. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 629–642.

[40] Yuhao Zhu, Anand Samajdar, Matthew Mattina, and Paul Whatmough. 2018. Euphrates: Algorithm-SoC Co-Design for Low-Power Mobile Continuous Vision. In *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture*.