Scalable Window Generation for the Intel Broadwell+Arria 10 and High-Bandwidth FPGA Systems

Greg Stitt, Abhay Gupta, Madison N. Emas, David Wilson, Austin Baylis
University of Florida, Department of Electrical and Computer Engineering
gstitt@ece.ufl.edu,abhayg271@gmail.com,madisel@ufl.edu,d.wilson@ufl.edu,abaylis@ufl.edu

ABSTRACT

Emerging FPGA systems are providing higher external memory bandwidth to compete with GPU performance. However, because FPGAs often achieve parallelism through deep pipelines, traditional FPGA design strategies do not necessarily scale well to large amounts of replicated pipelines that can take advantage of higher bandwidth. We show that sliding-window applications—an important subset of digital signal processing—demonstrate this scalability problem. We introduce a window generator architecture that enables replication to over 330 GB/s, which is an 8.7× improvement over previous work. We evaluate the window generator on the Intel Broadwell+Arria10 system for 2D convolution and show that for traditional convolution (one filter per image), our approach outperforms a 12-core Xeon Broadwell E5 by 81× and a high-end Nvidia P6000 GPU by an order of magnitude for most input sizes, while improving energy by 15.7×. For convolutional neural nets (CNNs), we show that although the GPU and Xeon typically outperform existing FPGA systems, projected performances of the window generator running on FPGAs with sufficient bandwidth can outperform high-end GPUs for many common CNN parameters.

CCS CONCEPTS

• **Hardware → Hardware accelerators**;

KEYWORDS

FPGA, convolution, neural networks

ACM Reference format:
Greg Stitt, Abhay Gupta, Madison N. Emas, David Wilson, Austin Baylis. 2018. Scalable Window Generation for the Intel Broadwell+Arria 10 and High-Bandwidth FPGA Systems. In *Proceedings of 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 25–27, 2018 (FPGA '18),* 10 pages.
https://doi.org/10.1145/3174243.3174262

1 INTRODUCTION

Digital signal processing applications commonly use field-programmable gate arrays (FPGAs) or graphics-processing units (GPUs), but GPUs are generally the more popular alternative for most high-performance use cases. Although FPGAs often provide more performance per unit of memory bandwidth (e.g., [7]), GPUs tend to provide external memory bandwidth that is at least an order-of-magnitude higher than FPGA systems. For example, the Nvidia P100 provides 732 GB/s [19], whereas most FPGA boards provide on the order of tens of GB/s (e.g., [8, 15, 21]).

Traditionally, high-performance FPGA designs have dealt with limited memory bandwidth using deep pipelines (e.g., [6, 13, 27]). Pipeline replication and loop unrolling are also common FPGA optimizations, but with limited bandwidth such replication provides limited improvements. For FPGAs to compete with GPU performance, emerging and future FPGAs will need to significantly increase memory bandwidth [16].

Although higher-bandwidth FPGA systems enable increased replication, not all FPGA design patterns scale well to large amounts of pipeline replication. We show that sliding-window generation has limited replication scaling, which is a significant problem for FPGAs given the prevalence of sliding windows in convolutional neural nets (CNNs) and other image-processing applications [6, 26].

Previous approaches provide window generators that are either limited to one window per cycle [4, 9], lack scalability to numerous windows [23], or support a specific window and/or image size [20]. For data-center usage, an FPGA circuit must support a wide range of window and image sizes due to prohibitively long reconfiguration times. Although previous approaches can support different window and image sizes by padding the input, such padding is often an expensive overhead, which in our tests exceeded the FPGA execution times of the presented case studies.

In this paper, we introduce a window generator architecture that greatly improves scalability, enabling generation of numerous windows in parallel to support anticipated bandwidth increases, while addressing flexibility problems by supporting runtime-configurable window and image sizes with no padding overhead. Our results demonstrate pipeline replication to more than 330 GB/s of memory bandwidth, which to our knowledge is more than any existing FPGA system and an 8.7× improvement over previous work [23].

Although the window generator can be used with any sliding-window application, we evaluate 2D convolution running on an Intel Broadwell+Arria 10 (BDW+A10) [10]. For traditional convolution (one filter per image), our approach outperforms optimized implementations from DeepBench [22] and the Intel Math Kernel Library (MKL) [24] running on a 12-core Xeon Broadwell E5 and a high-end Nvidia P6000 GPU with 3,840 CUDA cores. Despite the FPGA having a theoretical peak performance that is comparable to the Xeon and significantly less than the P6000, our approach achieves an average speedup of 81× over the Xeon and 12.6× over the P6000. We obtain these improvements by exploiting different types of parallelism that provide near-peak FPGA performance in

FPGA '18, February 25–27, 2018, Monterey, CA, USA

ACM ISBN 978-1-4503-5614-5/18/02…$15.00
https://doi.org/10.1145/3174243.3174262

situations where existing Xeon and GPU implementations leave many resources underutilized. Energy improvements are even more significant, with improvements of 96× and 15.7× for the Xeon and GPU, respectively. Even for the ideal situation where a GPU can completely avoid or amortize PCIe transfers, our approach achieves an average speedup of 1.2× and energy improvement of 1.5×.

We also evaluate our approach using convolution parameters common to CNNs for theoretical bandwidth improvements. Although the P6000 outperforms existing FPGA systems for CNNs, we show that our approach running on a shared-memory Stratix 10 system with sufficient bandwidth is projected to outperform the P6000 for common CNN use cases.

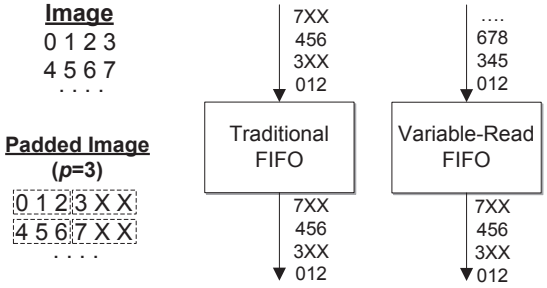


Figure 1: Unlike traditional FIFOs, the variable-read FIFO obviates padding for parallel windows (e.g., $p = 3$), which reduces pre-processing and PCIe overhead.

2 RELATED WORK

The most closely related study is the window generator from [23], which had similar goals of generating multiple windows per cycle, while also supporting runtime-configurable window and image sizes. We show that the previous approach does not scale past 256 parallel windows on the BDW+A10, and has clock speeds that are more than 2× slower than the presented approach for 64 or more parallel windows. Overall, the presented approach is able to support memory bandwidth that is 8.7× higher than this previous work.

Although there are many previous window-generation studies [4, 9, 20, 23], none of those studies address scalability, high clock frequencies, and runtime-configurable inputs with no padding overhead. A recent approach [20] investigated minimizing register usage during loop coarsening with high-level synthesis. Our approach has an alternative goal of maximizing scalability, which sacrifices register usage for improved clock speeds and scalability up to 1024 replicated pipelines, whereas the previous study reports up to 64. Although a direct comparison is not feasible due to the previous approach only providing high-level synthesis estimates, that approach complements our work with new border-handling techniques.

Tradeoff analyses between FPGAs, GPUs, and microprocessors for sliding-window applications are a well-studied topic [1, 3, 6, 26], and have established Pareto-optimal implementations for different use cases. This paper complements those studies with a window generator that improves FPGA performance, especially for emerging high-bandwidth systems.

FPGA performance evaluations for CNNs have received significant attention recently. Zhang [27] evaluated an FFT-based FPGA implementation for CNNs on a shared-memory system. Our work is complementary, focusing on sliding windows needed by time-domain implementations. The two approaches could be combined to efficiently support CNNs over many use cases, with time-domain implementations for smaller windows and frequency-domain implementations for larger windows [2]. Nurvitadhi et al. [18] compared Arria 10 performance with a CPU, GPU, and ASIC for binarized neural networks. In more recent work, Nurvitadhi et al. [17] presented Stratix 10 performance and energy projections compared to a Titan X GPU for CNNs, which showed projections similar to our presented work. Our work is complementary by presenting a window generator capable of realizing such projections for high-bandwidth FPGA systems. In general, the focus of our paper is on efficient sliding-window generation, which we evaluate using different convolution use cases, whereas earlier studies focus primarily on an architectural tradeoff analysis for CNNs.

3 WINDOW GENERATOR

Sliding-window applications are a domain of digital signal processing that perform application-specific computation on "windows" (sub-images) of an image. Applications generally slide these windows across an image from left to right at the top of the image, then move down one row and repeat until all windows have been processed. Although there are a variety of different sliding behaviors, we focus on single strides where each window slides by one column, and fully immersed windows where the window does not slide past the image borders. Our approach can be easily adapted to other variations, albeit with potentially less reuse for larger strides.

In this section, we present an architecture for generating all necessary windows independently from the application-specific computation. For consistency, we adopt the same terminology as the previous approach in [23], where the inputs are an image i with i_r rows and i_c columns, and a window w with w_r rows and w_c columns. We define the top-left pixel of an image to be $i[0, 0]$. We use p to represent the number of pixels provided each cycle, the number of windows generated each cycle, and the amount of pipeline replication, which are all equivalent in our architecture.

The window generator consists of a variable-read FIFO, a window buffer, and a window coalescer. Initially, the user passes a stream of pixels (p each cycle) to the variable-read FIFO (Section 3.1), which provides a variable number of pixels from that stream to the window buffer to obviate input padding. The window buffer assembles the pixels into columns of window data that the coalescer combines into approximately p complete windows each cycle (Section 3.2).

3.1 Variable-Read FIFO

The variable-read FIFO (VRF) streams pixels into the window buffer in a way that enables the window buffer to generate p windows in parallel, while eliminating the need for input padding. Although padding the image to have columns that are a multiple of p can eliminate the need for the VRF, such padding has a software preprocessing and PCIe overhead. In our experiments, software padding times often exceeded the FPGA execution time, which significantly reduced or eliminated speedup. Although such padding is conceptually easy to implement on the FPGA, creating a circuit that adds

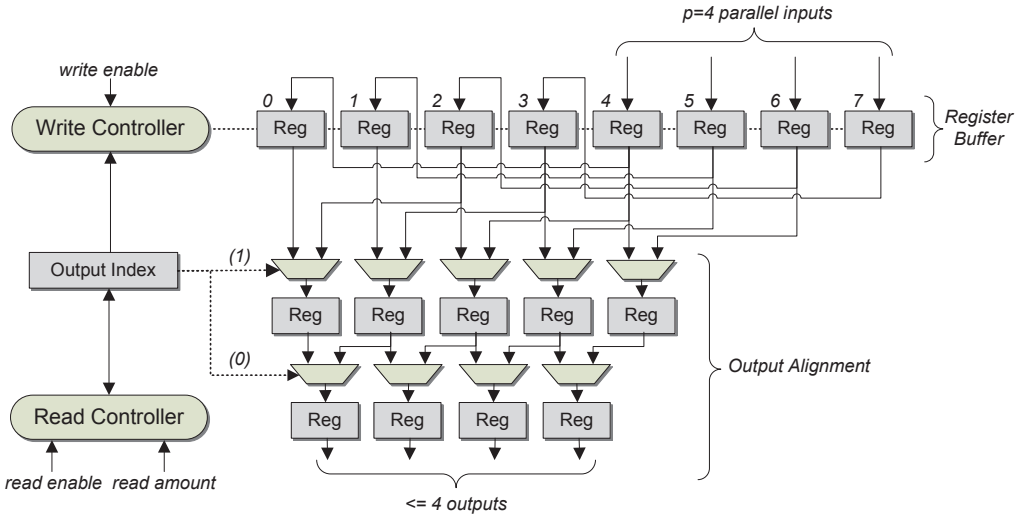


Figure 2: Variable-read FIFO architecture for four parallel inputs and outputs ($p = 4$).

variable padding based on runtime parameters with no performance overhead is not trivial. The VRF provides this functionality.

Figure 1 compares the difference between input streams when using a traditional FIFO with padding and the VRF. For an image with four columns and a system that can provide three pixels ($p = 3$) each cycle, the first read from the FIFO provides elements 0-2. To complete the processing of the first row, the application only needs element 3. However, if the FIFO provides three elements instead of one, the application will either be corrupted with invalid data for the first row, or must buffer the extra data somewhere internally. Since the FIFO already provides buffering, it makes more sense to read a variable amount of data from the FIFO instead of adding additional buffering elsewhere, which would also need to support variable amounts. With the VRF, the application reads three pixels to get pixels 0-2, then one pixel (3) to complete the first row, then pixels 4-6, then pixel 7, etc. This approach eliminates up to $p - 1$ padded pixels from the right edge of each image row, which is critically important for large p values where the padding overhead could be prohibitive.

Figure 2 illustrates the VRF architecture. The structure is conceptually similar to the FIFO in [23], but has been modified to improve timing scalability for larger p. Both approaches write p pixels into a fixed set of registers within a buffer of $2p$ registers. In this buffer, the output index varies depending on previous reads. Initially, the output index starts at 0 and then increases after each read by the number of elements read from the FIFO. When the output index exceeds $p - 1$, the write controller shifts the register buffer left by p positions to ensure the index is always between 0 and $p - 1$.

Because the output index changes, the read controller must align the outputs with the appropriate p registers. The previous approach implemented output alignment using p separate p:1 muxes to select the appropriate register for each output, which is a significant timing-closure bottleneck. Although those muxes could potentially be pipelined, muxes in general are an expensive FPGA resource. To avoid this problem, and to ensure better scalability, our new approach ensures that the critical-path propagation delay (ignoring routing delays) is independent of p.

In our approach, the VRF aligns outputs using a pipelined barrel shifter that shifts by the amount in the output index. Although this approach creates a several-cycle output delay, the user can still read every cycle. With this strategy, there is never more than a 2:1 mux in between registers for any value of p, which potentially enables the architecture to scale indefinitely up to any resource constraint without experiencing a timing-closure bottleneck.

In addition to the logic in the figure, the VRF outputs a count of the words in the FIFO, in addition to bits that specify the validity of each output. For example, when the user requests one output, the VRF will provide p outputs, but will mark $p - 1$ outputs as invalid. Although the propagation delay of the count logic increases with larger p, that increase is logarithmic. For any realistic value of p, the count logic is is not a timing-closure bottleneck. Even for $p = 1024$, the count logic only requires a 10-bit adder and subtractor.

3.2 Window Buffer and Coalescer

The window buffer is responsible for buffering the input stream of pixels into separate rows, and then passing p columns from each row into the window coalescer each cycle.

Figure 3(a) provides an overview of the window-buffer architecture. The basic structure is a chained sequence of w_r FIFOs, which each buffer an entire row of the image. The window buffer reads p pixels at a time from the VRF into the bottom row FIFO, where each word consists of p pixels. When there are fewer than p pixels left in a row of the image, the window buffer requests the remaining number of pixels from the VRF (e.g, pixel 3 in Figure 1). In this case, the window buffer marks any extra pixels as invalid to avoid including them in windows.

A previous approach [23] similarly used w_r FIFOs, but after outputting an entire row of windows, that approach would erase the top FIFO. The previous approach would then reuse the top FIFO for the next row of the image. As a result, the first row of each window gradually moves into different FIFOs. Similar to the VRF,

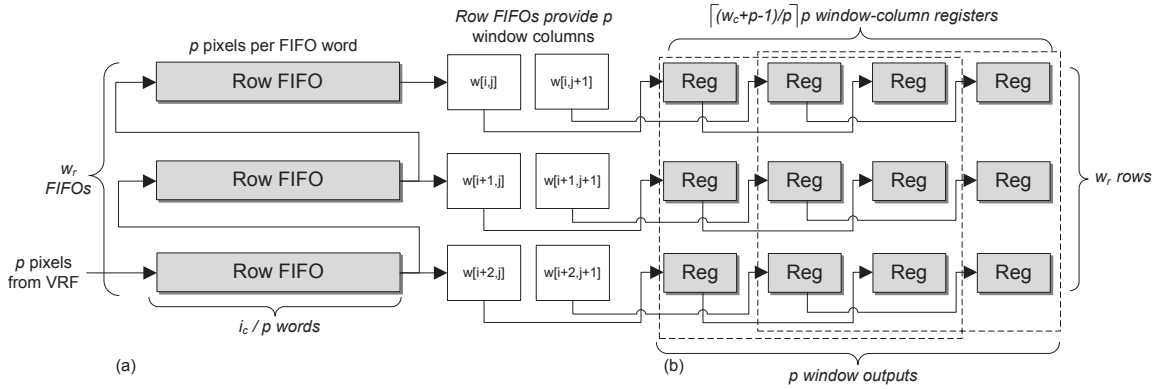


Figure 3: Overview of the (a) window buffer and (b) coalescer architecture for two parallel ($p = 2$) 3×3 windows.

the previous approach maintained an index that specified which FIFO stored the top window row. To align the outputs, that approach used w_r separate w_r:1 muxes, which created another significant scalability bottleneck that has similar problems to the previous VRF.

Our approach uses an alternative strategy that completely eliminates the need for alignment by always using the top FIFO as the top row of a window, the bottom FIFO as the bottom row, etc. To enable this functionality, the window buffer initially writes all incoming pixels into the bottom FIFO, storing p pixels per word. When the bottom FIFO contains $\lceil i_c/p \rceil$ words (an entire image row), every new write into the bottom FIFO also triggers a read from that FIFO. For each FIFO read, the buffer writes the read data into the next higher FIFO, which enables the pixels to gradually shift to the top FIFO. This process repeats until all w_r FIFOs contain $\lceil i_c/p \rceil$ words.

At this point, a controller (not shown) starts reading from all FIFOs, which passes p columns each cycle into the coalescer (Figure 3(b)). The coalescer assembles p windows by dividing each window into w_c register columns that get shifted by p positions for every new set of columns from the window buffer. After $\lceil (w_c + p - 1)/p \rceil$ shifts, the coalescer contains p complete windows, where the index of the first window starts at the first column, the second window at the second column, etc. Overall, the coalescer consists of $\lceil (w_c + p - 1)/p \rceil \cdot p$ columns, where each column has w_r registers.

The controller continues streaming columns from the buffer for $\lceil i_c/p \rceil$ cycles, and then returns to buffering incoming data. The controller continues to buffer new pixels while outputting columns. If pixels arrive every cycle, the controller will immediately begin outputting columns again after completing the current row. If the window buffer outputs all the current columns before another i_c pixels arrive, the controller delays the next set of columns until all the FIFOs have sufficient pixels.

In addition to removing all muxes, one critically important timing optimization was the removal of register enable logic. The previous approach stalled the coalescer by clearing an enable until all FIFOs were not empty. Because that enable had to control every register in the coalescer, the previous approach had a timing-closure bottleneck resulting from a prohibitive enable fanout of $\lceil (w_c + p - 1)/p \rceil \cdot p w_r d$, where d is the bit width of each pixel.

We address this problem by removing all enable logic from the coalescer. To remove the logic, the controller delays reads from the window buffer until the row FIFOs contain all required pixels for the next row of windows. When the controller starts reading, the p column inputs to the coalescer will always be valid until the end of the row, which eliminates the need for an enable. Although this approach delays the first columns, the throughput is identical. Most importantly, this optimization eliminates the fanout, which results in propagation delays that are independent of p.

To support any window size, the controller determines how far to slide the maximum-sized window across the image. For example, if the FPGA provides a 10×10 window, but the user requests a 3×3 window, the controller would slide the 10×10 window seven pixels past the right edge and bottom edge of the image. The controller pads all unused window elements with 0, which is done automatically on reset. With this strategy, although many window elements are unused, generation times for 3×3 windows are similar regardless of the maximum window size, with the only overhead being the initial time to fill up the extra row FIFOs. To support arbitrary image sizes, the architecture sets the FIFO depth to the maximum image width divided by p, and then simply starts reading from the FIFOs when the requested image columns i_c are buffered in each FIFO ($\lceil i_c/p \rceil$ words).

4 2D CONVOLUTION ON BDW+A10

In this section, we describe our custom RTL implementation of 2D convolution on the Broadwell+Arria 10 (BDW+A10). Because convolution implementations have been widely studied [6], we focus on BDW+A10 specific issues due to space constraints.

The BDW+A10 shares the Xeon's main memory with the FPGA, which the FPGA accesses over both PCIe and/or QPI. To access memory, the BDW+A10 provides a cache coherent interface (CCI) that provides basic memory-access mechanisms. In VHDL, we extended the provided mechanisms with our own DMA memory interface that provided memory access reordering, virtual-to-physical address translation, in addition to width conversion to enable the application to request any data width from memory. We also went to great effort with timing optimizations to ensure the DMA interface can run at 400 MHz to maximize memory bandwidth. Intel provides a Memory Properties Factory core that enables much of this functionality, which we have not yet evaluated, but could potentially improve reported memory bandwidth.

In our BDW+A10 implementation, software initially loads the bitfile and initializes page tables inside the FPGA using memory-mapped I/O (MMIO) to enable sharing of memory between software and the FPGA. For data-center usage, such functionality would normally be done while booting. Next, the software transfers the convolution kernel into FPGA registers, specifies the convolution parameters, and starts the FPGA execution, all using MMIO. The FPGA circuit then starts a DMA access to read the image from memory, which the circuit streams into the window generator from Section 3. After receiving enough pixels, the window generator generates a stream of parallel windows that are pushed into replicated 2D convolution pipelines. The FPGA also initiates another DMA access to simultaneously write results to memory.

Each 2D convolution pipeline uses a row of $w_r w_c$ multipliers, one for each element of the convolution kernel, followed by a balanced adder tree consisting of $w_r w_c - 1$ adders. The pipelines include registers between each operation, which eventually provides an output from each pipeline every cycle.

For the floating-point implementation, we use a different strategy to maximize DSP utilization and clock frequency. Arria 10 DSPs can perform both a single-precision multiply and addition. The DSPs also contain chained routing that connects each DSP to an adjacent DSP. Our implementation utilizes this chaining, which has the side-effect of converting the balanced adder tree with a depth that grows logarithmically with the kernel size into a sequence of adds whose length grows linearly. This chaining improves clock frequencies at the cost of each adder input requiring a longer alignment delay than in the balanced tree. To handle these delays, we use registers when the delay is below a certain threshold and switch to block RAM for larger delays. Such an approach will not scale to large kernel sizes due to limited block RAM, in which case a combination of the balanced approach and chaining approach can be used.

To improve clock frequency, we performed a number of timing optimizations. Chaining the DSPs made a significant impact on timing by avoiding routing delays from numerous 32-bit signals. We also performed an optimization similar to Section 3.2 where we eliminated stall functionality for each individual pipeline stage to eliminate the enable signal's fanout. We replaced the fine-grained stall strategy with a strategy that uses a large FIFO to absorb the entire state of the pipeline when writes to output memory have to stall. When this FIFO is almost full, it triggers the window generator to stop producing windows. An additional advantage of this strategy is the enabling of Stratix 10 HyperFlex interconnect registers, which should further improve clock frequencies in future work.

One critical timing optimization for the 2D convolution pipelines was register duplication. Because the window coalescer (Section 3.2) shares registers for window elements that overlap in consecutive windows, many of these registers will fanout to numerous pipelines, which can significantly restrict maximum clock frequency for large values of p. Although we could not find an exact description of register-duplication restrictions in Quartus, we removed the fine-grained stalling of each individual register, removed combinational logic before the first pipeline register, and added registers before the multipliers, at which point Quartus started replicating the registers with high fanout. We plan to manually evaluate area/clock tradeoffs for manually specified replication thresholds, but lengthy compilation times prohibited such analysis for this study.

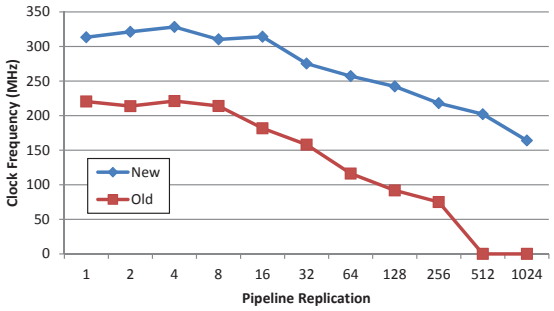


Figure 4: A comparison of Arria 10 clock frequencies for the presented (*new*) window generator and previous work (*old*) for different amounts of pipeline replication.

5 EXPERIMENTS

In this section, we first evaluate the scalability of the window generator (Section 5.1). We then provide performance and energy comparisons between the BDW+A10, Broadwell Xeon E5, and Nvidia P6000 for traditional 2D convolution (Section 5.2), in addition to performance projections for convolutional neural nets (Section 5.3).

5.1 Scalability

Figure 4 compares the maximum clock frequency of the presented approach with previous work [23], which we evaluated using the open-source release at https://github.com/ARC-Lab-UF/window_gen. We use Quartus 16 Prime Pro to determine the maximum clock frequency after synthesis, placement, and routing on an Arria 10 GX1150 FPGA. Results use 3×3 windows and image sizes of 2048×2048. Frequencies for other window and image sizes were similar.

The figure demonstrates the potential scalability problem in window generation with the previous approach decreasing to under 100 MHz for 128 pipelines, whereas the new approach runs at 242 MHz. More importantly, the old approach does not scale past 256 pipelines, whereas we were able to evaluate the new approach for 1024 pipelines. Although the frequency of both approaches decreases with more replication, the new approach decreases at a slower rate, and provides frequencies over 200 MHz even for 512 pipelines. Overall, the maximum bandwidth that the new approach can leverage is 336 GB/s, compared to 38.4 GB/s in the old approach—an improvement of 8.7×. These results suggest that the presented approach will enable FPGAs to fully utilize increased memory bandwidth for the foreseeable future.

Table 1(a) compares lookup table (LUT), flip flop (FF), and block RAM (RAM) utilizations between the presented approach and previous work for different window sizes and replication amounts. For almost all examples, the new approach used fewer LUTs, with an average reduction of 40% from eliminating muxes as described in Section 3.2. Register usage (FFs) increased by an average of 20%, which is likely an attractive trade off considering the 8.7× bandwidth improvement. RAM usage increased on average by 40%. Note that the previous approach does not synthesize past 256 pipelines.

Table 1(b) shows resource counts for the new approach. The Arria 10 GX1150 has over one million LUTs and FFs, which likely

Table 1: (a) Resource utilization relative to previous work for window sizes from 3×3 to 9×9. (b) Absolute resource numbers for the presented approach.

	3x3			5x5			7x7			9x9		
Replication	*LUTs*	*FFs*	*RAM*	*LUTs*	*FFs*	*RAM*	*LUTs*	*FFs*	*RAM*	*LUTs*	*FFs*	*RAM*
1	0.7x	1.1x	1.3x	0.6x	1.0x	1.2x	0.6x	1.0x	1.1x	0.6x	1.0x	1.1x
2	0.7x	1.1x	1.7x	0.6x	1.0x	1.4x	0.6x	1.0x	1.3x	0.6x	1.0x	1.2x
4	0.7x	1.2x	1.7x	0.6x	1.1x	1.4x	0.6x	1.1x	1.3x	0.5x	1.0x	1.2x
8	0.9x	1.3x	1.7x	0.7x	1.2x	1.4x	0.7x	1.1x	1.3x	0.5x	1.1x	1.2x
16	0.9x	1.4x	1.7x	0.7x	1.3x	1.4x	0.7x	1.2x	1.3x	0.6x	1.1x	1.2x
32	0.9x	1.5x	1.7x	0.7x	1.3x	1.4x	0.6x	1.2x	1.3x	0.5x	1.2x	1.2x
64	1.0x	1.6x	1.7x	0.4x	1.4x	1.4x	0.4x	1.3x	1.3x	0.5x	1.2x	1.2x
128	0.3x	1.6x	1.7x	0.3x	1.4x	1.4x	0.4x	1.3x	1.3x	0.5x	1.2x	1.2x
256	0.2x	1.7x	1.7x	0.3x	1.5x	1.4x	0.4x	1.3x	1.3x	0.5x	1.3x	1.2x
512	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Avg	0.7x	1.4x	1.6x	0.6x	1.2x	1.4x	0.6x	1.2x	1.3x	0.5x	1.1x	1.2x

(a)

	3x3			5x5			7x7			9x9		
Replication	*LUTs*	*FFs*	*RAM*	*LUTs*	*FFs*	*RAM*	*LUTs*	*FFs*	*RAM*	*LUTs*	*FFs*	*RAM*
1	425	631	4	602	917	6	805	1278	8	1040	1695	10
2	461	704	5	638	1032	7	859	1428	9	1107	1891	11
4	560	990	5	763	1366	7	996	1952	9	1264	2377	11
8	823	1632	10	1158	2252	14	1528	2856	18	1911	3484	22
16	1379	3017	20	1925	4121	28	2489	5231	36	3059	6354	44
32	2444	5919	35	3216	8070	49	3997	10176	63	4742	12284	77
64	4826	11967	65	6265	16119	91	7711	20290	117	9218	24465	143
128	10001	24661	130	12747	32879	182	15494	41194	234	18306	49486	286
256	20704	51117	260	26262	67636	364	31603	83955	468	37062	100310	572
512	43360	106385	515	54097	1E+05	721	64928	2E+05	927	75862	204114	1133

(b)

makes these amounts acceptable for most use cases. All results use a maximum image size of 2048×2048, which makes the RAM usage pessimistic for common use cases with smaller images.

5.2 2D Convolution on BDW+A10

This section evaluates the window generator using 2D convolution on the Intel BDW+A10, while comparing performance and energy to a GPU and parallelized software.

5.2.1 Experimental Setup. All experiments compare the BDW+A10 with a 12-core Broadwell Xeon E5 and an Nvidia Quadro P6000 GPU. The P6000 is a high-end GPU using the latest Pascal architecture, which has 3840 CUDA cores, 24 GB of GDDR5X RAM, and costs approximately $5000. The BDW+A10 does not have a publicly announced price, but uses an Arria 10 GX1150 FPGA, which costs several thousand dollars [5].

To evaluate software, we used the optimized convolution from DeepBench [22], which provides two algorithms that leverage the Intel Math Kernel Library (MKL) 2017 Update 3 [24]. We also created our own MKL-based implementation to optimize for large images. All software implementations used AVX2 instructions on 12 cores. For the GPU, we used DeepBench GPU code, which selects from eight different algorithms for a given input. We also used Nvidia *convolutionFFT2D* code from the CUDA-8.0 SDK to include an optimized frequency-domain implementation. FPGA details are given in Section 4. For synthesis, we used Quartus 16 Prime Pro, which is required for the BDW+A10. All examples run the convolution pipelines at 271 MHz and the DMA interface at 400 MHz.

To measure performance, we used *gettimeofday()* around relevant regions of code. For all devices, measurements exclude initialization that is common to all devices. We also exclude times for device initialization on all devices, which would be amortized over many executions when used in a data center. On the FPGA, we excluded time for bitstream configuration and memory allocation, which requires several seconds. For the GPU, we exclude the time of the first execution, which adds 0.3 seconds. For all devices, we exclude the time to initialize the convolution kernel, which generally changes infrequently. All FPGA results include PCIe and QPI transfer times for accessing memory.

We measured Xeon and memory power using the RAPL (Running Average Power Limit) component of Performance API (PAPI) 5.5.1.0 [25]. RAPL uses the Model Specific Registers (MSR) kernel module to read registers that capture the energy and time between two points in the code. To measure power, we performed convolution in a loop, capturing the PAPI readings before and after the loop to give an average value.

For Arria 10 power measurements, we used *tempPowMon* from system release 5.0.3, which reads power and temperature measurements from the FPGA. To get total system power for the FPGA, we added FPGA power to the measured Xeon and memory power during FPGA execution.

To measure GPU power, we used *nvmlDeviceGetPowerUsage* from the Nvidia Management Library, which provided power of the entire GPU board. For the GPU, we measured power and time in separate executions because the power measurements significantly increased time measurements. For total system power with the GPU, we added the GPU power to the idle Xeon power and idle memory power. Ideally, we would measure Xeon power during GPU execution, but since we could not put the GPU in the server with the Broadwell processor, such power would not be a fair comparison. Therefore, total system power for the GPU is likely optimistic.

For all devices, we measured time and power by putting the relevant code in a loop and averaging numerous measurements, with the exact amount depending on the variation for each device.

Because 2D convolution can be used for a variety of purposes, a complete analysis is outside the scope of this paper. This section focuses on traditional use cases of one filter per image, using kernel sizes of 3×3, 5×5, 7×7, and 9×9, along with images ranging from 256×256 to 2048×2048. All examples use inseparable kernels to get worst-case performances. Color channels are 8 bits.

5.2.2 Performance Evaluation. To ensure good FPGA performance, we replicated the 2D convolution pipelines using the presented window generator. Figure 5 demonstrates the improvements in FPGA execution time for different amounts of pipeline replication for a 3×3 kernel on a 2048×2048 image using 8-bit color channels. Trends were similar for other window and image sizes.

The results show near-perfect performance improvements initially, with each replication achieving a 1.99× speedup over the previous amount of replication. However, for 32 replications, that improvement fell to 1.18× due to memory bandwidth being exhausted. For 64 replications, there was no improvement. For nearly all the presented results, this trend was the main performance bottleneck. Average DSP utilization was only 52% of the available 1280 DSPs, with additional DSP usage being prevented primarily by insufficient bandwidth. This utilization suggests improved memory bandwidth can provide significantly improved performance.

Table 2 presents BDW+A10 speedup over the Xeon for both a 16-bit fixed-point kernel and a 32-bit floating-point kernel. The software baseline only uses floating point due to MKL not including

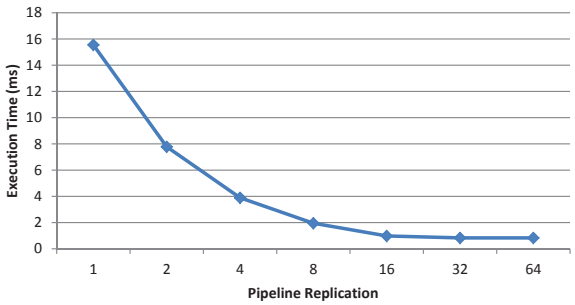


Figure 5: FPGA 2D convolution execution times with varying amounts pipeline replication (unrolling) for a 3x3 kernel, 2048x2048 image, and 8-bit color channels.

Table 2: BDW+Arria10 2D convolution speedup compared to a 12-core Xeon Broadwell E5.

		Image Size				
Precision	*Kernel Size*	*256x256*	*512x512*	*1024x1024*	*2048x2048*	*Avg*
16-bit Fixed	3x3	29×	38×	52×	55×	44×
	5x5	57×	96×	135×	146×	108×
	7x7	50×	115×	126×	145×	109×
	9x9	38×	80×	110×	123×	88×
	Avg	44×	82×	106×	117×	
32-bit Float	3x3	27×	38×	52×	55×	43×
	5x5	57×	96×	136×	146×	109×
	7x7	48×	97×	109×	123×	94×
	9x9	27×	48×	58×	62×	49×
	Avg	40×	70×	89×	97×	

fixed-point implementations. The BDW+A10 shows clear improvements over the Xeon, with speedups ranging from 27× to 146×, and an average of 81× across all examples. Speedup from fixed-point implementations tended to be larger than floating point, primarily due to a larger amount of pipeline replication at larger kernel sizes.

The FPGA speedup is achieved from several contributing factors. Most significantly, the FPGA exploited a massive amount of parallelism every cycle. For example, the fixed-point 5×5 kernel is capable of 1,600 multiplies and 1,536 adds every cycle at 271 MHz, which is approximately 850 GOPS. Floating-point results are similar, with the 5×5 kernel performing 800 single-precision multiplies and 768 adds each cycle. Although memory bandwidth prevented those resources from being fully realized, the parallelism still far exceeded that achieved by the Xeon. For fixed-point kernels, the pipeline replication was 64 for 3×3 windows, 64 for 5×5, 32 for 7×7, and 16 for 9×9. These pipelines used 576, 1600, 1568, and 1296 multipliers, respectively, and a similar number of adders. For floating-point kernels, the replication was 64 for 3×3 windows, 32 for 5×5, 16 for 7×7, and 8 for 9×9, which used 576, 800, 784, and 648 DSP resources, respectively, with each performing a multiply and add.

For traditional convolution, the Xeon efficiency was surprisingly low considering its peak potential throughput of 700 GFLOPS when using AVX2 across 12 cores. We have observed that DeepBench and MKL appear to exploit parallelism across larger kernel sizes and larger numbers of kernels, as opposed to computing multiple outputs from the same kernel in parallel. As as a result, much of the potential parallelism of the Xeon is left underutilized. It may be possible to optimize the Xeon code to exploit such parallelism, but for current convolution software implementations, our presented approach is able to exploit parallelism that is not leveraged by the Xeon.

Another significant contributor to FPGA performance compared to previous studies is that the BDW+A10 has negligible overhead for initiating FPGA execution. For systems using FPGAs on PCIe boards, the application generally has to copy all relevant inputs to the FPGA board, and then read back all results. On the BDW+A10, even though the FPGA accesses memory over PCIe and/or QPI, the FPGA shares the Xeon's memory, which provides a significant performance improvement by eliminating such copying. For the BDW+A10, execution time is roughly equivalent to the time to read inputs and write outputs to memory.

Figure 6 compares BDW+A10 execution times with the GPU for different kernel and image sizes. GPUs can be used in a variety of usage scenarios, where in some cases inputs and outputs must be transferred over PCIe every execution, and in others results are reused from GPU memory for a large number of execution. In these results, we evaluate the maximum possible GPU performance by excluding all PCIe transfers from the GPU execution times. Note that all FPGA results still include all PCIe and QPI transfer times.

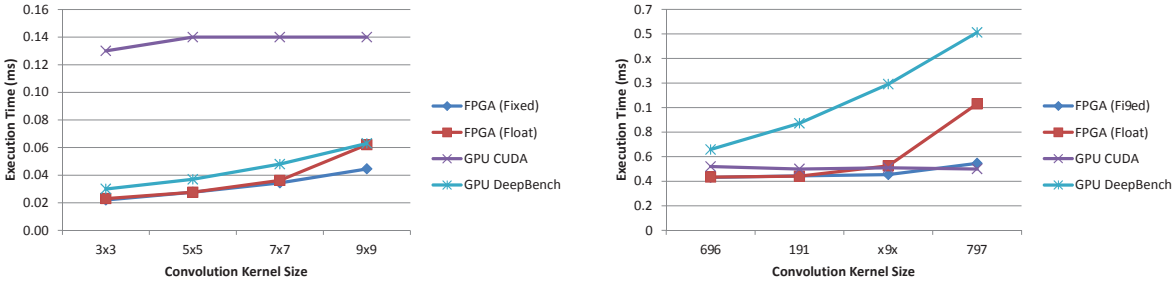
For 256×256 images, the fixed-point FPGA implementation always provided the best performance, with the floating-point FPGA version achieving nearly identical results, except for the 9×9 kernel size. The GPU DeepBench implementation was slightly slower, with FPGA speedup ranging from 1.0× to 1.4×. The GPU CUDA-SDK implementation was significantly slower at this image size due to the added initial overhead of performing the FFT. For 512×512 images, trends are similar, with the FPGA speedup range increasing from 1.4× to 2.3×.

At 1024×1024, performances of the GPU CUDA-SDK and FPGA fixed-point version were comparable, with the GPU slightly overtaking the FPGA at the 9×9 kernel size. The FPGA floating-point version experienced a 2× slowdown for the 9×9 kernel due to lower parallelism than the fixed-point version. Trends were similar for 2048×2048 images, with the GPU CUDA-SDK slightly increasing its advantage.

Table 3 shows BDW+A10 speedup across all inputs compared to the fastest GPU implementation. The left side of the table summarizes the results from Figure 6, which excluded PCIe transfers. The right side shows the BDW+A10 speedup with GPU PCIe transfers. The results show that GPU transfers are an expensive overhead, resulting in FPGA speedup of more than an order of magnitude in most cases. Overall, the average FPGA speedup increased from 1.2× with no GPU PCIe transfers to 12.6× with PCIe transfers.

Like the Xeon, the P6000 performed far below its peak performance of 12 TFLOPS. The decreasing speedup for larger window sizes suggests that current GPU implementations parallelize across large windows, and as shown later, across multiple filters per image. Because traditional convolution uses a single kernel per image, the P6000 was significantly underutilized, whereas the FPGA was able to exploit parallelism across multiple outputs of the same kernel.

5.2.3 Energy Comparison. In this section, we repeat the experiments from the previous section for energy consumption using power measurements described in Section 5.2.1.



(a) 256×256 Image **(b) 1024×1024 Image**

Figure 6: 2D convolution execution times for different kernel and image sizes. FPGA results include 16-bit fixed point and 32-bit floating point. GPU results include CUDA SDK and DeepBench implementations and *exclude* all PCIe transfer times.

Table 3: BDW+Arria10 speedup over the P6000 GPU when excluding (left) and including (right) GPU PCIe transfer times.

			Image Size				
GPU PCIe	*Precision*	*Kernel Size*	*256x256*	*512x512*	*1024x1024*	*2048x2048*	*Avg*
Excluded	16-bit Fixed	3x3	1.4×	1.4×	1.2×	0.9×	1.2×
		5x5	1.3×	1.7×	1.1×	0.9×	1.3×
		7x7	1.4×	2.2×	1.1×	0.9×	1.4×
		9x9	1.4×	2.3×	0.9×	0.8×	1.3×
		Avg	1.4×	1.9×	1.1×	0.9×	
	32-bit Float	3x3	1.3×	1.4×	1.2×	0.9×	1.2×
		5x5	1.3×	1.7×	1.1×	0.9×	1.3×
		7x7	1.3×	1.9×	1.0×	0.8×	1.2×
		9x9	1.0×	1.3×	0.5×	0.4×	0.8×
		Avg	1.2×	1.6×	0.9×	0.8×	

			Image Size				
GPU PCIe	*Precision*	*Kernel Size*	*256x256*	*512x512*	*1024x1024*	*2048x2048*	*Avg*
Included	16-bit Fixed	3x3	13.1×	16.4×	14.5×	14.1×	14.5×
		5x5	11.1×	15.5×	14.6×	14.5×	13.9×
		7x7	9.3×	15.7×	14.9×	15.1×	13.7×
		9x9	7.0×	12.6×	13.2×	13.4×	11.5×
		Avg	10.1×	15.0×	14.3×	14.3×	
	32-bit Float	3x3	12.5×	16.4×	14.5×	14.0×	14.3×
		5x5	11.1×	15.7×	14.7×	14.5×	14.0×
		7x7	8.8×	13.3×	12.9×	12.8×	12.0×
		9x9	5.0×	7.5×	7.0×	6.8×	6.6×
		Avg	9.4×	13.2×	12.3×	12.0×	

Table 4: BDW+Arria10 2D convolution energy improvements over a Xeon Broadwell E5.

		Image Size				
Precision	*Kernel Size*	*256x256*	*512x512*	*1024x1024*	*2048x2048*	*Avg*
16-bit Fixed	3x3	329	×89	689	659	2×9
	2x2	659	1169	1819	1849	1339
	8x8	259	1329	1239	16×9	1759
	4x4	××9	459	1329	1×59	1069
	Avg	219	449	1319	1×09	
37-bit Float	3x3	339	×69	629	669	239
	2x2	6×9	1119	1679	1649	1789
	8x8	229	1139	1319	1349	1049
	4x4	379	259	819	829	249
	Avg	×69	579	1059	1179	

Table 4 compares BDW+A10 energy with the Xeon. BDW+A10 energy improvements were more significant than performance improvements, with the FPGA providing an average 96× improvement in energy. FPGA device power ranged from 8.9 W to 15.4 W. Memory power during FPGA execution added another 33 W, and the Xeon power added 42 W. For software execution on the Xeon, Xeon power ranged from 53 W to 77 W, with memory power ranging from 25 W to 49 W. Overall, the average system power across all FPGA tests was 87 W, compared to 105 W when running software.

Table 5 compares BDW+A10 energy to the most energy-efficient GPU implementation for each input. When excluding GPU PCIe transfers, the BDW+A10 achieved an average energy improvement of 1.5×, and was more efficient than the GPU for all but two examples. When including CPU PCIe transfers, the BDW+A10 shows significant improvements, achieving an average energy improvement of 15.7×. GPU device power ranged from 61 W to 190 W. The FPGA device power ranged from 8.9 W to 15.4 W. Total system power with the GPU ranged from 98 W to 227 W, whereas the total system power for the FPGA was from 84 W to 90 W.

One potential power optimization for the FPGA is to use interrupts instead of polling to check for completion. Although FPGA-generated interrupts are not documented yet for the BDW+A10, we imitated this optimization by putting the processor to sleep during FPGA execution. For these tests, the Xeon power and memory during FPGA execution decreased to 27 W and 24 W, respectively, reducing the average total system power to 62 W.

5.3 CNN Performance Projections

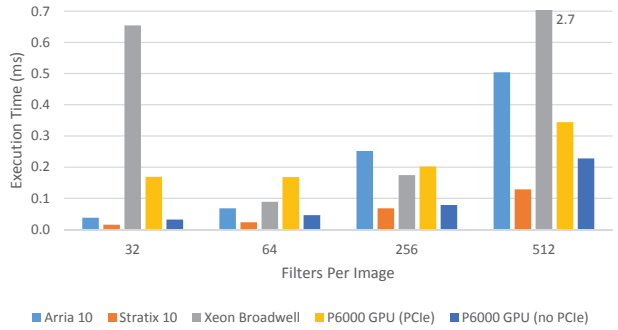
In this section, we evaluate the window generator for convolution parameters common to CNNs. Specifically, we use an image size of 256×256, filter sizes of 3×3 and 5×5, filters per image ranging from 32 to 512, which are common to DeepBench and AlexNet [14].

Unlike traditional convolution, the Xeon and P6000 outperform existing FPGA systems for most CNN use cases due to efficient parallelization across multiple filters for an image. Because this paper focuses on the benefits of scalable window generation, in these experiments we evaluate projected performance of shared-memory FPGA systems with theoretical amounts of memory bandwidth that would achieve full utilization of the presented 2D convolution pipelines up to the resource limits of an Arria 10 GX1150 and a Stratix 10 GX2800. For the FPGA projections, we manually determined a parallelization strategy for each example that both replicated pipelines and performed multiple filters per pipeline without exceeding resource constraints. We then simulated the

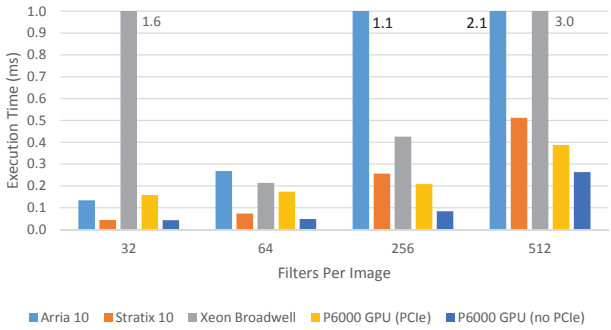
Table 5: BDW+Arria10 2D convolution energy improvements over the P6000 GPU when excluding (left) and including (right) GPU PCIe transfer times.

GPU PCIe	Precision	Kernel Size	Image Size 256x256	512x512	1024x1024	2048x2048	Avg
Excluded	16-bit Fixed	3x3	1.64	1.×4	1.24	1.04	1.24
		2x2	1.24	8.14	1.04	1.34	1.64
		×××	1.64	8.54	1.34	1.34	1.×4
		nxn	1.×4	3.34	1.14	1.14	1.54
		Avg	1.64	8.24	1.34	1.34	
	38-bit Float	3x3	1.24	1.×4	1.24	1.34	1.24
		2x2	1.24	8.74	1.34	1.34	1.24
		×××	1.24	8.34	1.14	1.14	1.24
		nxn	1.84	8.74	7.64	7.24	1.14
		Avg	1.04	8.74	1.14	1.74	

GPU PCIe	Precision	Kernel Size	Image Size 256x256	512x512	1024x1024	2048x2048	Avg
9 cluded	16-bit Fixed	3x3	12.24	1n.54	1n.04	1n.74	15.04
		2x2	18.n4	15.64	1n.24	1n.54	1×.×4
		×××	17.24	15.84	1n.×4	87.24	1×.84
		nxn	5.14	12.34	16.n4	15.64	10.×4
		Avg	11.54	15.74	15.n4	1n.24	
	38-bit Float	3x3	10.64	1n.04	1n.74	15.64	1×.n4
		2x2	18.84	1×.54	15.64	15.×4	16.54
		×××	n.n4	12.84	16.n4	1×.34	10.54
		nxn	2.54	n.74	5.n4	n.04	5.34
		Avg	17.64	12.04	12.54	16.74	



(a) 3×3 Filter



(b) 5×5 Filter

Figure 7: A CNN performance comparison of the Xeon Broadwell and P6000 GPU with projections of the Arria 10 and Stratix 10 using the presented window generation with hypothetical increases in memory bandwidth.

existing pipelines for this strategy, using memory and communication latencies obtained from the BDW+A10 experiments. Due to space constraints, these experiments only use 16-bit fixed-point kernels, which are common for CNNs [11]. GPU examples use single-precision floating point due to DeepBench not providing fixed-point implementations. We also evaluated half precision, but the results are omitted due to worse performance than single precision, which is a known issue on GPUs [12]. DeepBench can potentially be optimized for half precision, but is outside the scope of this paper. We omit power and energy in this section due to the use of projections for envisioned optimizations and the lack of a Stratix 10 to physically measure.

Figure 7 compares CNN performance, again including GPU results both with and without PCIe transfers. For 3×3 filters and 32 filters per image, the Stratix 10 outperforms all other devices. The Arria 10 outperforms the GPU when including PCIe transfers, and is comparable to the GPU excluding PCIe transfers. For 64 filters per image, the Arria 10 performance falls behind the GPU excluding PCIe transfers, but the Stratix 10 is still 2× faster without GPU PCIe transfers, and 7× faster than the GPU when including PCIe transfers. Trends are similar at 256 and 512 filters per image, but with reduced FPGA speedup.

For 5×5 filter sizes, the GPU has significantly better performance due to extra parallelism from the larger filter. However, when including PCIe transfers, the Stratix 10 projections are better or comparable up to 256 filters per image. At 512 filters per image, the GPU begins to outperform the Stratix 10 both with and without PCIe transfers, achieving speedups of 1.9× and 1.3×, respectively.

Table 6 shows the parallelism strategy used by each FPGA example, where p is the number of replicated pipelines, and k is the filters per pipeline. BW is the required bandwidth in GB/s to achieve this parallelism without stalls. *Perf* is the resulting performance in tera-operations per second (TOPS). In general, most examples used both replicated pipelines and performed multiple filters in each pipeline. For the larger number of filters per image, some examples did not use pipeline replication and instead used all available resources to maximize the number of parallel filters. On average, the Arria 10 and Stratix 10 achieved a sustained performance of 1.1 TOPS and 4.2 TOPS, respectively, which required bandwidth ranging from 18 GB/s to 286 GB/s. Required bandwidth was calculated by multiplying the number of inputs and outputs by the clock frequency (271 MHz). For example, the 3×3 Stratix 10 example for 32 filters/image had 32 inputs each cycle and $32 \cdot 32$ outputs each cycle for a total of $(32 + 32 \cdot 32)271 = 286$ GB/s. Performance was calculated as the number of multiplies each cycle ($pkw_r w_c$) added with the number of adds each cycle ($pkw_r w_c - p$), multiplied by the clock frequency.

The reason for the large differences in required bandwidth is due to lower resource utilization for a particular parallelization strategy. For example, for the 5×5 Stratix 10 examples, all of the circuits used 6400 multipliers, which is only 54% of the 11,721 available multipliers. The reason for this underutilization is that the existing version of the convolution code only supports replication in powers of two, where the next highest power would exceed 11,721

Table 6: FPGA parameters from Figure 7, where p is pipeline replication, k is filters per pipeline, *BW* is required memory bandwidth in GB/s, and *Perf* is performance in tera-ops/s.

		Filters Per Image															
		32				64				256				512			
Device	*Filter*	*p*	*k*	*BW*	***Perf***	*p*	*k*	*BW*	***Perf***	*p*	*k*	*BW*	***Perf***	*p*	*k*	*BW*	***Perf***
Arria 10	3x3	8	32	72	**1.2**	4	64	70	**1.2**	1	256	70	**1.2**	1	256	70	**1.2**
	5x5	2	32	18	**0.9**	2	32	18	**0.9**	2	32	18	**0.9**	2	32	18	**0.9**
Stratix 10	3x3	32	32	286	**5.0**	16	64	282	**5.0**	4	256	279	**5.0**	2	512	278	**5.0**
	5x5	8	32	72	**3.5**	4	64	70	**3.5**	1	256	70	**3.5**	1	256	70	**3.5**

multipliers. Ideally, we would replicate by non-powers of two to ensure that all examples achieve closer to 100% utilization of DSP resources. We will investigate such optimization in future work, but even without this optimization, these projections show that the presented window generator enables emerging FPGA systems to achieve performance that is better or comparable to the P6000 GPU for many CNN use cases.

6 CONCLUSIONS

In this paper, we introduced a sliding-window generator architecture that enables scalable pipeline replication to over 330 GB/s of memory bandwidth, while also eliminating software pre-processing and PCIe overheads from input padding. We evaluated the window generator for 2D convolution on the Intel Broadwell+Arria 10 and demonstrated order-of-magnitude speedup over software running on the Xeon and a high-end P6000 GPU. Although the GPU outperforms any existing FPGA for CNN usage, we demonstrate that the presented window generator running on a Stratix 10 system with sufficient memory bandwidth can outperform the GPU for many common CNN use cases.

ACKNOWLEDGMENTS

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422 and IIP-1161022. We would like to thank and acknowledge the donations and support from Intel, and the help provided by Ken Hill and Pawel Cieslewski.

REFERENCES

[1] S. Asano, T. Maruyama, and Y. Yamaguchi. 2009. Performance comparison of FPGA, GPU and CPU in image processing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. 126–131. https://doi.org/10.1109/FPL.2009.5272532

[2] Patrick Cooke, Jeremy Fowers, Greg Brown, and Greg Stitt. 2015. A Tradeoff Analysis of FPGAs, GPUs, and Multicores for Sliding-Window Applications. *ACM Trans. Reconfigurable Technol. Syst.* 8, 1, Article 2 (March 2015), 24 pages. https://doi.org/10.1145/2659000

[3] B. Cope, P.Y.K. Cheung, W. Luk, and S. Witt. 2005. Have GPUs made FPGAs redundant in the field of video processing?. In *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*. 111 –118. https://doi.org/10.1109/FPT.2005.1568533

[4] Yazhuo Dong, Yong Dou, and Jie Zhou. 2007. Optimized Generation of Memory Structure in Compiling Window Operations onto Reconfigurable Hardware. In *Proc. of the Int. Symp. on Applied Reconfigurable Computing*. 110–121.

[5] Mouser Electronics. 2017. Intel Arria 10 GX 1150 Series FPGA - Field Programmable Gate Array. (September 2017). http://www.mouser.com/Intel/Semiconductors/Programmable-Logic-ICs/FPGA-Field-Programmable-Gate-Array/Arria-10-GX-1150-Series/_/N-3oh9p?P=1ypc7usZ1yy6lwu

[6] Jeremy Fowers, Greg Brown, John Wernsing, and Greg Stitt. 2013. A Performance and Energy Comparison of Convolution on GPUs, FPGAs, and Multicore Processors. *ACM Trans. Archit. Code Optim.* 9, 4, Article 25 (Jan. 2013), 21 pages. https://doi.org/10.1145/2400682.2400684

[7] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S. Chung, and Greg Stitt. 2014. A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. 36–43. https://doi.org/10.1109/FCCM.2014.23

[8] Gidel. 2017. Proc10A PCIe Arria 10 Accelerator Boards. (2017). http://www.gidel.com/HPC-RC/Proc10A_HPC.asp

[9] Zhi Guo, Betul Buyukkurt, and Walid Najjar. 2004. Input data reuse in compiling window operations onto reconfigurable hardware. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '04)*. ACM, New York, NY, USA, 249–256. https://doi.org/10.1145/997163.997199

[10] PK Gupta. 2016. Accelerating Datacenter Workloads. (2016). http://www.fpl2016.org/slides/Gupta%20--%20Accelerating%20Datacenter%20Workloads.pdf FPL 2016 Keynote.

[11] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 1737–1746. http://dl.acm.org/citation.cfm?id=3045118.3045303

[12] Nhut-Minh Ho and Weng-Fai Wong. 2017. Exploiting half precision arithmetic in Nvidia GPUs. In *IEEE High Performance Extreme Computing Conference*.

[13] S. Kestur, J.D. Davis, and O. Williams. 2010. BLAS Comparison on FPGA, CPU and GPU. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*. 288–293. https://doi.org/10.1109/ISVLSI.2010.84

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[15] Nallatech. 2017. Nallatech 385A FPGA Accelerator Card. (2017). http://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-385a-arria10-1150-fpga/

[16] Nallatech. 2017. Nallatech 510T Compute Acceleration Card. (2017). http://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-510t-fpga-computing-acceleration-card/

[17] E. Nurvitadhi et al. 2017. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/3020078.3021740

[18] E. Nurvitadhi, D. Sheffield, Jaewoong Sim, A. Mishra, G. Venkatesh, and D. Marr. 2016. Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC. In *2016 International Conference on Field-Programmable Technology (FPT)*. 77–84. https://doi.org/10.1109/FPT.2016.7929192

[19] Nvidia. 2017. Tesla P100: The Most Advanced Data Center GPU Ever Built. (2017). http://www.nvidia.com/object/tesla-p100.html

[20] M. A. Ozkan, O. Reiche, F. Hannig, and J. Teich. 2017. Hardware design and analysis of efficient loop coarsening and border handling for image processing. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 155–163. https://doi.org/10.1109/ASAP.2017.7995273

[21] A. Putnam et al. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 13–24. http://dl.acm.org/citation.cfm?id=2665671.2665678

[22] Baidu Research. 2017. DeepBench. (2017). https://svail.github.io/DeepBench/

[23] Greg Stitt, Eric Schwartz, and Patrick Cooke. 2016. A Parallel Sliding-Window Generator for High-Performance Digital-Signal Processing on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 9, 3, Article 23 (May 2016), 22 pages. https://doi.org/10.1145/2800789

[24] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. *Intel Math Kernel Library*. Springer International Publishing, Cham, 167–188. https://doi.org/10.1007/978-3-319-06486-4_7

[25] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore. 2012. Measuring Energy and Power with PAPI. In *2012 41st International Conference on Parallel Processing Workshops*. 262–268. https://doi.org/10.1109/ICPPW.2012.39

[26] Haiqian Yu and M. Leeser. 2006. Automatic Sliding Window Operation Optimization for FPGA-Based Computing Boards. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*. 76 –88. https://doi.org/10.1109/FCCM.2006.29

[27] Chi Zhang and Viktor Prasanna. 2017. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 35–44. https://doi.org/10.1145/3020078.3021727