# LeapConv: An Energy-Efficient Streaming Convolution Engine with Reconfigurable Stride

Dionysios Filippas
*Electrical and Computer Engineering*
*Democritus University of Thrace (DUTH)*
Xanthi, Greece

Chrysostomos Nicopoulos
*Electrical and Computer Engineering*
*University of Cyprus*
Nicosia, Cyprus

Giorgos Dimitrakopoulos
*Electrical and Computer Engineering*
*Democritus University of Thrace (DUTH)*
Xanthi, Greece

***Abstract*—Convolution is the central computation kernel for various machine learning applications. The convolution stride controls the number of pixels by which the kernel's window moves after each operation, thereby allowing for the reduction of the output's resolution. The streaming computation of strided convolution inherently involves large periods of inactivity interrupted by periods of actual computation. In this work, we propose LeapConv, a new streaming convolution engine that computes convolutions of arbitrary and reconfigurable stride using local buffering and by leveraging efficient data and memory reuse. The organization of LeapConv is based on the decomposition of strided convolutions into a set of parallel unity-stride convolution channels that are implemented by a merged hardware unit. The experimental results show that LeapConv reduces power consumption with increasing stride by eliminating redundant data movement. The incurred area overhead due to the additional multiplexing logic required to support reconfigurability is demonstrated to be marginal.**

***Index Terms*—Strided convolution, convolutional neural networks, low power design, machine learning accelerators.**

## I. Introduction

Convolutional Neural Networks (CNNs) have become the standard algorithms for many machine learning applications, especially in the fields of audio [1] and image processing [2]–[4]. Their widespread adoption has triggered the need to accelerate their computation directly in hardware [5], [6]. In this way, computational throughput requirements are satisfied in the most energy-efficient way. For datacenter-scale applications, vector or tensor processing units have been adopted [7]–[9]. In consumer devices, lower-cost streaming convolution engines have been proposed, which utilize convolution-specific memory architectures and unrolled hardware units. Such architectures take advantage of the "sliding window" approach that is intrinsic to visual processing and allows for local buffering and efficient data and memory reuse [10]. Streaming architectures have been enhanced for efficient parallelism [11] and functional safety [12].

Ideally, the scalability of streaming convolution engines should also be maintained when computing *strided* convolutions, and, especially, in environments where the stride can be dynamically reconfigured at runtime. The convolution stride controls the number of pixels by which the kernel's window moves after each operation. This feature facilitates the reduction of the output's resolution. Fig. 1 graphically depicts the application of strided convolution with stride length 2 of a 3×3 kernel on a 7×7 input image, resulting in a 3×3 output image.

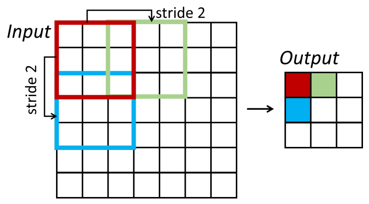


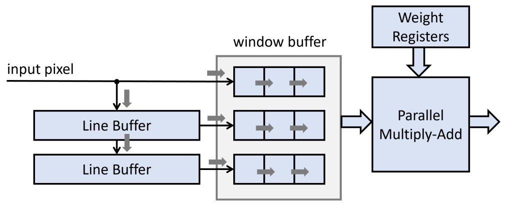
Fig. 1. The application of convolution with stride length 2 on a 7×7 input image using a 3×3 kernel.

Strided convolution can be computed easily using a streaming convolution engine by producing valid outputs only when output pixels align with the strided slide of the kernel. Although conceptually simple, this approach requires data movements that are equivalent to unity-stride convolutions, which are highly redundant when computing strided convolutions. In fact, strided convolution inherently involves large periods of inactivity interrupted by periods of actual computation. Other approaches that do not rely on streaming convolution engines try to reduce the complexity of strided convolutions by utilizing Winograd's algorithm [13], [14].

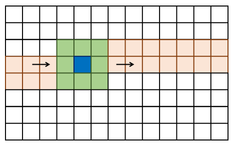
In this work, our goal is to design a streaming convolution engine that *efficiently* supports arbitrary and reconfigurable strides. The proposed LeapConv architecture exhibits the following key attributes:

- It decomposes strided convolutions into multiple independent unity-stride convolution channels to avoid redundancy in computation and data movements. Even though the computations of each channel are performed in parallel, the overall hardware implementation is merged in one unified structure to maximize efficiency and resource utilization.
- The parallel channels are mapped to the same window buffer, while the input is forwarded only to the registers of the active channel. This approach improves the clock-gating efficiency by limiting the data switching activity during the engine's operation to the absolute minimum required for the correct implementation of the algorithm.
- The aforementioned architectural features reduce the power consumption from 10% to 32%, for different ASIC configurations implemented using a 45-nm standard-cell library, without introducing any throughput penalty. The area is slightly increased, due to the multiplexing logic added to support the reconfigurability of the stride length.

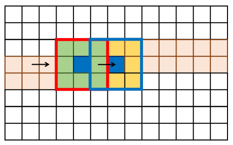
The rest of this paper is organised as follows: Section II



(a) Streaming convolution engine



(b) Standard sliding window



(c) Strided sliding window

Fig. 2. The organization of a streaming convolution engine and the slide of the kernel over the input depending on the stride length.

describes the architecture of streaming convolution engines used for strided convolutions. In Section III, we propose and analyze the overall architecture of LeapConv. The experimental results are presented in Section IV, while conclusions are drawn in Section V.

## II. Streaming Convolution Engines

In each clock cycle, a streaming convolution engine – such as the one shown in Fig. 2(a) – accepts one input pixel and computes one output pixel [15]. To do so, all pixels within the corresponding input window must be available, as highlighted in Fig. 2(b). For a $W \times W$ filter, the pixels from the last $W$ rows are required. This requires row buffers with the ability to store $W - 1$ rows, plus a window buffer that holds the currently active input pixels. This sliding-window memory architecture allows for data reuse and fine-grained parallelism. Window buffers are normally implemented with registers, while larger line buffers are mapped either to standard-cell-based memories [16], or SRAM blocks [17].

A set of parallel multipliers and an addition tree perform the actual computation by applying the weights of the kernel to the pixels stored in the window buffer. The engine manages to keep the input pixels properly aligned to the filter's coefficient by shifting the contents of the window buffer. In each clock cycle, the input pixel is pushed in the top left corner of the window buffer and in the top row of the line buffers. In parallel, the window buffer is filled with pixels that come from the line buffers and shifts its contents to the right to simulate the rightward sliding of the filter over the input. The pixels that belong to the same column with the incoming pixel are also moved downwards in the line buffers, to simulate the downward sliding of the filter. An algorithmic view of these operations, assuming a $W \times W$ window buffer (`window`) and $W - 1$ line buffers (`lb`), is shown in Fig. 3.

The ability to compute a strided convolution using this architecture would require minimal changes to the hardware. In strided convolutions, the filter slides over the image with a step $S$, in both dimensions. Fig. 2(c) illustrates this strided slide in the 'x' dimension. As the filter moves $S$ pixels away from its previous location in the same row, the engine needs to

```
foreach input pixel (i,j)
// read line buffers and shift window buffer
for (m = 0; m < W; m++) {
  tmp = (m < W-1) ? lb[m][j] : input;
  for (n = 0; n < W; n++)
    window[m][n] = (n < W-1)? window[m][n+1] : tmp;
}
// move data downwards in the line buffers
for (m=0; m < W-1; m++)
  lb[m][j] = (m < W-1)? lb[m+1][j] : input;
```

Fig. 3. An algorithmic description of the data movement involved in the window and line buffers of a streaming convolution engine.

receive $S$ new pixels in a row-wise manner to align the input pixels with the newly shifted location of the filter. This means that the engine can produce a new output only when the pixel that will be multiplied with the coefficient in the bottom right corner of the filter becomes available. Thus, a new output of the same row is computed every $S$ cycles, assuming that one new pixel arrives per clock cycle.

The same alignment should be performed when the filter moves downwards with a step $S$. This means that after the engine has produced the last output of a row, it must wait for $S-1$ complete rows to be read before it can restart to produce a valid output of the next active row.

Hence, the output of a strided convolution engine is characterized by bursts of useful output computations that are separated by large periods of inactivity. During the *active period*, where, for an incoming pixel $(i, j)$, it holds that $i \mod S = (W - 1) \mod S$, the engine produces one pixel every $S$ clock cycles. On the contrary, the output remains idle during *the inactive period*, i.e., when $i \mod S \neq (W - 1) \mod S$. During this time, the engine waits to read the next $S$ rows, before a new period of activity may resume. As the value of $S$ increases, the *inactive* period becomes significantly larger than the "active" one.

By enhancing its functionality, a streaming convolution engine can compute strided convolutions by computing an output only when needed. However, to keep the pixels aligned with the kernel, the data should always be shifted inside the window buffer, and the line buffers must still follow the data movement of standard unity-stride convolutions, even during the large periods of inactivity. This redundant data movement is effectively removed by LeapConv, thereby leading to significant power consumption benefits.

## III. LeapConv: Architectural Overview

The LeapConv architecture is based on the decomposition of strided convolutions into a set of parallel non-strided convolution channels [18]. To avoid redundancy in computation and data movement, instead of assigning the computation directly to the parallel channels, the hardware implementation of the latter is merged in one unified structure. Furthermore, the parallel channels are mapped to the same window buffer, while the input is forwarded only to the registers of the active channel, thus effectively reducing the data switching activity.

### A. Decomposition

Any strided convolution can be decomposed into the sum of multiple unity-stride convolutions [19]. Since the filter is

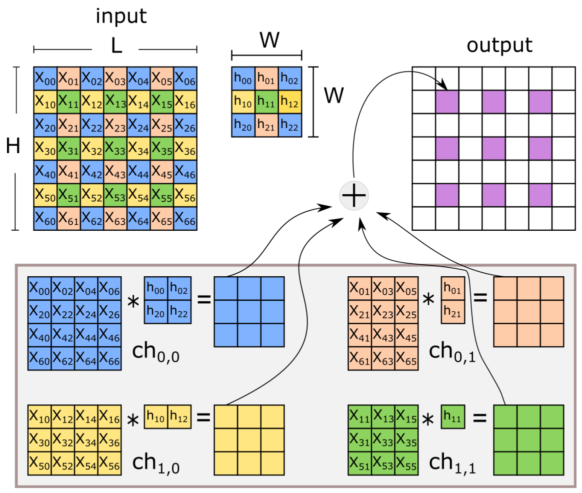


Fig. 4. The decomposition of a 2-stride convolution into 4 parallel unity-stride convolution channels.

applied with a stride $S$, each coefficient of the filter will be multiplied only with a subset of the input elements. Therefore, by grouping together the input pixels that each coefficient "touches," we can derive independent *channels* of sub-images and sub-filters. Fig. 4 depicts the transformation of a strided convolution with $S = 2$ into four independent channels. In the general case, the number of channels created is equal to $S^2$. A pixel $(i, j)$ belongs to the channel $Ch_{k,l}$ with $k = i \mod S$ and $l = j \mod S$. Similarly, the coefficient $h_{m,n}$ belongs to channel $Ch_{k,l}$ when $k = m \mod S$ and $l = n \mod S$.

The operation within each channel is, effectively, a unity-stride convolution, since every pixel of each sub-image is multiplied with every coefficient of the corresponding sub-filter. After computing the output of each channel, we can reconstruct the output of the original strided convolution by simply performing a pixel-wise addition between the individual outputs of each channel.

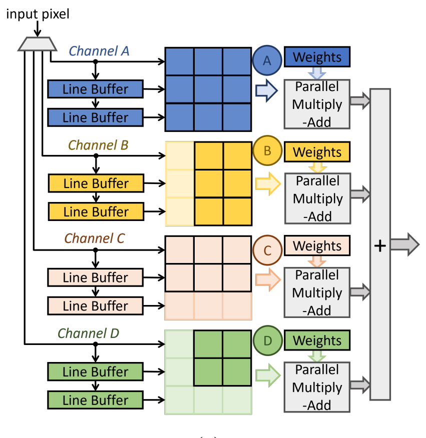
LeapConv utilizes this decomposition but time-shares the operation of each channel, thereby saving considerable amount of redundant data switching activity and offering a highly efficient overall hardware implementation.

## B. Merged Hardware Architecture

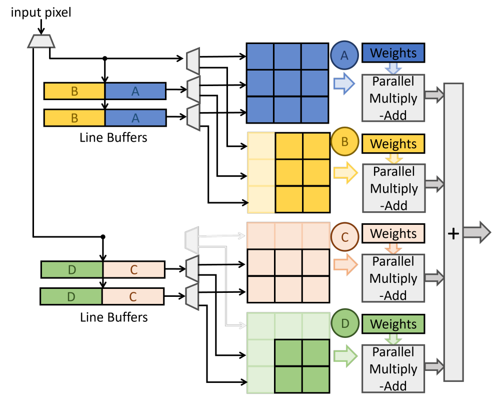
In a direct implementation of the decomposition transformation, the strided convolution can be computed using $S^2$ independent unity-stride convolution engines. The outputs of all engines are added to produce a valid result. Therefore, to produce the correct output, the operation of the engines should be aligned.

The hardware implementation of the decomposed strided convolution is shown in Fig. 5(a) for a 5×5 kernel applied with a stride of $S = 2$. The engine of each channel utilizes a smaller window buffer and requires fewer and shorter line buffers, as the input image of each channel is a subset of the original image.

The fragmentation of the line buffers can be avoided by taking advantage of the fact that each channel uses a different part of the input. For instance, channels A and B refer to pixels that belong to even-indexed rows, while channels C



(a)



(b)

Fig. 5. The (a) multi-channel architecture that allows the computation of a 2-stride convolution, and (b) the optimized architecture with shared line buffers.

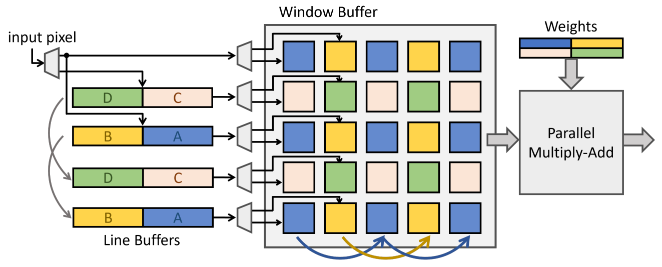


Fig. 6. Mapping in place the window buffers of the 4 channels. Note that no more registers are used here than in the unity-stride streaming architecture.

and D refer to odd-indexed rows. Therefore, the line buffers of channels A and B can be merged, resulting in larger line buffers that are equal to the size of the ones used in a unity-stride engine. Adding some de-multiplexing, as shown in Fig. 5(b), the line buffers can now push data either to channel A or B, depending on the column index of the current input pixel. Pixels from even columns will get pushed to channel A and the rest to channel B. In the same way, we can merge the line buffers for channels C and D. For uniform treatment, channels C and D are also equipped with two line buffers.

The window buffer of each channel can be viewed as a subset of the original window buffer. By rearranging the window buffers of the four channels, we can reconstruct the

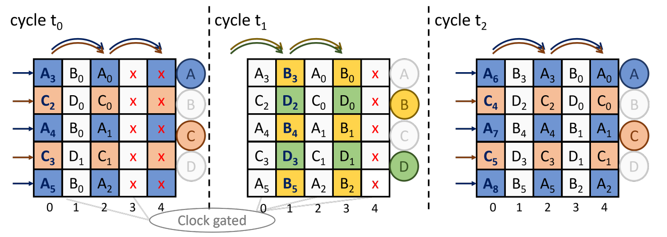


Fig. 7. The movement of data in the window buffer. Data moves between the registers of each channel, thus, effectively, mimicking the strided kernel movement.

window buffer of the unity-stride engine, as illustrated in Fig. 6. The only difference lies in the connectivity between the registers; each register connects only to the registers of the same channel.

To keep the operation of all channels aligned, data movement inside the window buffer occurs only during the active periods. During said periods, output computation occurs only when the window buffer has every pixel of the window of the input image that overlaps with the filter. On the contrary, during the inactive periods that arise naturally due to the strided movement of the window, the window buffer in LeapConv is completely inactive.

To understand the connectivity between registers and the involved data transfers, Fig. 7 highlights the movement of the data inside a $5 \times 5$ window buffer for a strided convolution with $S = 2$, during an active period. Assume that in cycle $t_0$, when pixels $\{A_3, C_2, A_4, C_3, A_5\}$ are being pushed in the first column (column 0) of the window buffer, channels A and C are activated. At that time, the first 3 columns of the window buffer are filled with data, while the last 2 have not received any input yet. This means that channel A and C have data in two of their columns, while channels B and D have data only in their first column.

In cycle $t_1$, pixel $B_3$ is pushed into the window buffer from the input. The rest of the second column of the window buffer is filled with pixels $D_2$, $B_4$, $D_3$ and $B_5$ that come from the line buffers. Pixels $B_3$, $B_4$ and $B_5$ belong to channel B, while $D_2$ and $D_3$ belong to channel D. These two channels are active and they should shift the corresponding columns 1 and 3 of the merged window buffer. The remaining registers of the window buffer are unaffected. In total, only two columns out of the five columns have experienced any data switching.

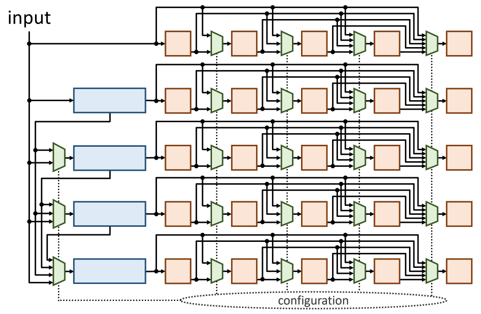
In the next cycle, $t_2$, three columns are being updated. Column 4 receives the data of column 2, and column 2 receives the data of column 0. The newly arrived pixels $\{A_6, C_4, A_7, C_5, A_8\}$ from the input and the line buffers are pushed into column 0. In all cases, data move two columns forward (to the right), following the stride length ($S = 2$) of the convolution. In this cycle, only three columns experience data switching activity.

Similar to the reduction in data movement inside the window buffer, LeapConv achieves an equal reduction to the movement of the data between the line buffers. In the case of the unity-stride convolution depicted in Fig. 3, in order to emulate the downward shifting of the filter over the image, the data of a line buffer is pushed to the next one, i.e., `lb[m][j]` receives data from `lb[m+1][j]`. In LeapConv, the data of the line buffers must move to the next active line buffer of the same channel. Since input pixels are streamed in a row-wise manner, until a complete row is read, only the $(W-1)/S$ line buffers that belong to the active channels are being used. As a result, instead of shifting all line buffers downwards in each clock cycle, only the line buffers of the active channels are being updated.

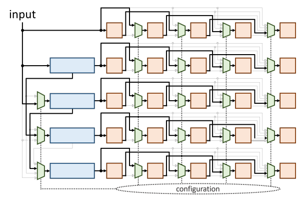
```
foreach input pixel (i,j)
// read line buffers and shift window buffer
if (active_row) // when on an active period
  for (m = 0; m < W; m++) {
    tmp = (m < W-1)? lb[m][j] : input;
    // shift only active columns
    for (n = j%S; n < W; n += S)
      window[m][n] = (n < W-S)? window[m][n+S] : tmp;
  }
// move data downwards in line buffers of active rows
for (m = i%S; m < W-1; m += S)
  lb[m][j] = (m < W-1-S)? lb[m+S][j] : input;
```

Fig. 8. An algorithmic description of the data movement involved in the window and line buffers of the proposed LeapConv architecture.

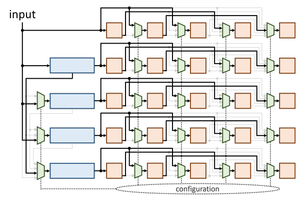
The update of the window buffer and the corresponding line buffers in LeapConv is detailed in the code segment shown in Fig. 8. For each input pixel, the windows buffers are shifted only in the active rows. The shifting does not involve all columns, but only the ones placed $S$ columns apart, i.e., `window[m][n]` is connected to `window[m][n+S]`. The same connectivity pattern is involved across line buffers, i.e., `lb[m][j]` receives a pixel from `lb[m+S][j]`.



(a) LeapConv with Reconfigurable Stride



(b) Configuration for $S = 2$



(c) Configuration for $S = 3$

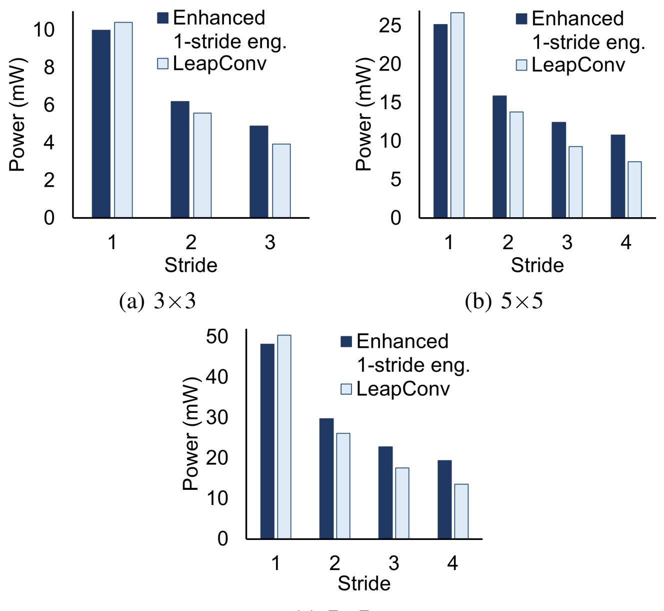
Fig. 9. The (a) overall architecture of LeapConv and the reconfigured design that computes strided convolutions with (b) $S = 2$ and (c) $S = 3$.

## C. *Support for Reconfigurability*

The purpose of LeapConv is to allow the computation of *any* strided convolution for a specified filter size. By generalizing the architecture of Fig. 6, we are able to design a streaming convolution engine with *reconfigurable* stride.

To enable this feature, each register of the window buffer is accompanied by a multiplexer, as shown in Fig. 9(a). These multiplexers enable the shifting of data from `window[m][n+S]` to `window[m][n]` for arbitrary values of $S$, assuming that the stride length $S$ is smaller than or equal to the window size $W$. By appropriately configuring the select signals of the multiplexers, a different stride length may be chosen. The connectivity for a specific stride length should be configured before the start of the computation and, for correctness, it should not change until the output data is computed.

To support an arbitrary stride, each column of the window buffer is connected to all previous columns. Therefore, the cost of multiplexing increases progressively from left to right. Multiplexers are also added to the write port of the line buffers to ensure that `lb[m][j]` receives an input from `lb[m+S][j]` (as shown in Fig. 8), for all possible values of $S$. The read port of each line buffer is connected directly to the window buffer. The column of the window buffer that receives the output of the line buffers – only during the active period – is also determined by $S$. Example configurations for $S = 2$ and $S = 3$ are shown in Figs. 9(b) and 9(c), respectively.



(a) 3×3 (b) 5×5 (c) 7×7

Fig. 10. The power consumption of the LeapConv and the enhanced unity-stride streaming engine implementations for different filter sizes and stride lengths.

## IV. Experimental Results

The goal of the experimental results is to highlight the effectiveness of LeapConv, as compared to current state-of-the-art approaches. To the best of our knowledge, LeapConv is the *first* streaming convolution engine that is also optimized for strided convolutions. Thus far, strided convolutions have been optimized only for large-scale *systolic arrays* using Winograd's algorithm to reduce the cost of multiplications in convolutions [13], [14], [19]. In these approaches, the strided convolution is computed via smaller convolution kernels mapped to Winograd-specific units. Since each Winograd unit supports a specified kernel size, the decomposed filters are zero-padded to be aligned with the predefined kernel [13]. Moreover, the systolic way of computing the final output does not allow for buffer sharing and regular data movements, as facilitated by streaming convolution engines. Thus, a comparison between a streaming convolution engine, such as LeapConv, and any systolic-array-based design would not provide meaningful insight, or be fair, since the two architectural approaches are very different in both concept and implementation.

Instead, the enhanced version of the unity-stride streaming architecture presented in Section II for computing the strided convolution remains cost efficient (even if it requires larger data transfers than LeapConv) and it does not suffer from irregularity in data accesses and lack of local buffer sharing, which afflict the systolic-architecture-based approaches [13], [14]. Hence, in the presented evaluation, we will compare LeapConv to the above-mentioned enhanced 1-stride streaming convolution engine, which serves as the benchmark for the simplest possible architectural alternative.

Both architectures were fully implemented in C++ and synthesized to Verilog RTL using Catapult HLS. The two architectures were designed for 16-bit input images and filters and are reconfigurable with respect to stride length. For each presented example, the size of the input images is assumed to be equal to 256×256. The designs follow the behavioral models shown in Figs. 3 and 8. However, the C++ models used were optimized for HLS using coding templates that favor efficient unrolling and reduced dependencies for efficient pipelining. The Verilog RTL for each case was synthesized with the Oasys RTL logic synthesis using a 45 nm standard-cell library and targeting a clock frequency of 500 MHz. The reported power is obtained from the PowerPro power analysis and optimization tool.

The power consumption of the two architectures is illustrated in Fig. 10 for different kernel sizes. For standard convolutions, where a unity stride is assumed, LeapConv incurs higher power consumption than the enhanced version of the unity-stride streaming architecture. This power overhead is a direct consequence of the reconfigurability provided by LeapConv and the extra multiplexing logic added to support it. Since the amount of the multiplexing is proportional to the size of the window buffer, the difference in power consumption between the two architectures increases as the size of the window buffer increases. This power overhead ranges from 4% to 6%, depending on the filter size.

However, LeapConv achieves a substantial reduction in power consumption when the engine computes convolutions with stride lengths greater than one, as shown in Fig. 10. The high inactivity of the window buffer and the efficient on-time activation of the line buffers allow LeapConv to significantly reduce the data switching activity. This translates to dynamic power savings that increase with increasing stride length. This result stems from the fact that the periods of inactivity of the window buffer and the number of active line buffers in each clock cycle are determined by the stride length. For the 3×3

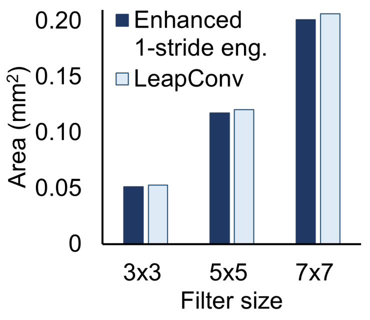


Fig. 11. The area consumed by LeapConv and the enhanced unity-stride streaming architectures.

kernel implementation, the reduction for a convolution with stride length 2 is around 10%, while it can reach up to 20% for longer strides. For the 5×5 and 7×7 kernel implementations, the power savings range from 13% to 32%. These savings are significant if we take into account the overhead of the multiplexing logic to support the desired reconfigurability.

Fig. 11 depicts the area of both designs under comparison for various kernel sizes. LeapConv is only marginally larger than the architecture of Section II, which – as previously mentioned – is the most efficient approach in implementing streaming convolutions, since it maximizes buffer sharing and relies on simple data access patterns. The source of this area overhead is the added multiplexing logic and wiring required to support the extra feature of stride reconfigurability. The area overhead increases slightly with the window buffer size, and ranges between 2.3% and 2.8%, for the different implementations.

Finally, it should be noted that the area cost of multiplexing does *not* translate to a delay overhead, since the multiplexers drive the input pins of the window's registers. The critical path in all cases starts from the output pins of the same registers and moves to the multiplication and addition logic that implements the arithmetic part of the convolution engine. In other words, LeapConv can achieve the *same* maximum possible operating frequency as the baseline design.

## V. Conclusions

Strided convolution can be inherently decomposed into a sum of multiple channels of unity-stride convolutions. The proposed LeapConv architecture takes advantage of this decomposed form of computing convolutions of arbitrary stride length to improve the power consumption of the streaming convolution engine. In LeapConv, the result of each channel is computed separately, albeit by using the same merged hardware unit. Both the window and line buffers of a baseline convolution engine built for unity-stride convolutions can hold the input of each channel. Using appropriately selected data movements, the data of each channel is properly aligned in the window buffer, to allow for direct computation of the desired result. The active and inactive periods of computation enable reduced data switching activity and increased clock-gating efficiency for the registers of the window buffer. Finally, with the addition of multiplexing logic, LeapConv can also support reconfigurable stride lengths.

## VI. Acknowledgements

This work was supported by a research grant of Siemens EDA to Democritus University of Thrace for "High Level Synthesis Research for Systems-on-Chip".

## References

[1] S. Hershey, S. Chaudhuri, D. P. W. Ellis, J. F. Gemmeke, A. Jansen, R. C. Moore, M. Plakal, D. Platt, R. A. Saurous, B. Seybold, M. Slaney, R. J. Weiss, and K. Wilson, "CNN architectures for large-scale audio classification," in *IEEE Intern. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, 2017, pp. 131–135.

[2] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[4] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A ConvNet for the 2020s," *arXiv preprint arXiv:2201.03545*, 2022.

[5] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," *ACM SIGARCH comp. arch. news*, vol. 44, no. 3, pp. 367–379, 2016.

[6] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs," in *Design Automation Conf. (DAC)*, 2017.

[7] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten Lessons From Three Generations Shaped Google's TPUv4i," in *Intern. Symp. on Computer Architecture (ISCA)*, 2021.

[8] K. Patsidis, C. Nicopoulos, G. C. Sirakoulis, and G. Dimitrakopoulos, "RISC-V$^2$: A Scalable RISC-V Vector Processor," in *IEEE Intern. Symp. on Circuits and Systems (ISCAS)*, 2020.

[9] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI," *IEEE Trans. on VLSI Systems*, vol. 28, no. 2, pp. 530–543, feb 2020.

[10] Y. Dong, Y. Dou, and J. Zhou, "Optimized Generation of Memory Structure in Compiling Window Operations onto Reconfigurable Hardware," in *Intern. Workshop on Applied Reconfigurable Computing*. Springer, 2007, pp. 110–121.

[11] G. Stitt, A. Gupta, M. N. Emas, D. Wilson, and A. Baylis, "Scalable Window Generation for the Intel Broadwell+Arria 10 and High-Bandwidth FPGA Systems," in *Proc. of the Intern. Symp. on Field-Programmable Gate Arrays*, 2018, p. 173–182.

[12] D. Filippas, N. Margomenos, N. Mitianoudis, C. Nicopoulos, and G. Dimitrakopoulos, "Low-Cost Online Convolution Checksum Checker," *IEEE Trans. on VLSI Systems*, vol. 30, no. 2, pp. 201–212, Feb. 2022.

[13] C. Yang, Y. Wang, X. Wang, and L. Geng, "A Stride-Based Convolution Decomposition Method to Stretch CNN Acceleration Algorithms for Efficient and Flexible Hardware Implementation," *IEEE Trans. on Circuits and Systems I*, vol. 67, no. 9, pp. 3007–3020, 2020.

[14] J. Yepez and S.-B. Ko, "Stride 2 1-D, 2-D, and 3-D Winograd for Convolutional Neural Networks," *IEEE Trans. on VLSI Systems*, vol. 28, no. 4, pp. 853–863, 2020.

[15] L. Ioannou, A. Al-Dujaili, and S. A. Fahmy, "High Throughput Spatial Convolution Filters on FPGAs," *IEEE Trans. VLSI Systems*, vol. 28, no. 6, pp. 1392–1402, 2020.

[16] P. Meinerzhagen, S. M. Y. Sherazi, A. Burg, and J. N. Rodrigues, "Benchmarking of Standard-Cell Based Memories in the Sub-$V_T$ Domain in 65-nm CMOS Technology," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 2, pp. 173–182, 2011.

[17] N. Weste and D. Harris, *CMOS VLSI Design a Circuits and Systems Perspective*. Addison Wesley (3rd Edition), 2010.

[18] C. Kong and S. Lucey, "Take it in your stride: Do we need striding in CNNs?" *arXiv preprint arXiv:1712.02502*, 2017.

[19] J. Pan and D. Chen, "Accelerate Non-Unit Stride Convolutions with Winograd Algorithms," in *Proc. of the Asia and South Pacific Design Automation Conf. (ASPDAC)*, 2021, p. 358–364.