Synchronization Constraints for Interconnect Synthesis

Alex Rodionov and Jonathan Rose
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada
{arod, Jonathan.Rose}@ece.utoronto.ca

ABSTRACT

Interconnect synthesis tools ease the burden on the designer by automatically generating and optimizing communication hardware. In this paper we propose a novel capability for FPGA interconnect synthesis tools that further simplifies the designer's effort: automatic cycle-level synchronization of data delivery. This capability enables the creation of interconnect with significantly reduced hardware cost, provided that communicating modules have fixed latency and do not apply upstream backpressure. To do so, the designer specifies constraints on the lengths, in clock cycles, of multi-hop logical communication paths. The tool then uses an integer programming-based method to insert balancing registers into optimal locations, satisfying the designer's constraints while minimizing register usage. On an example convolutional neural network application, the new approach uses 43% less area than a FIFO-based synchronization scheme.

Keywords

Convolutional Neural Networks; FPGA Interconnect Synthesis

1. INTRODUCTION

Interconnect synthesis and system integration tools, such as Altera Qsys[2], Xilinx IPI[21], and LatticeMico[16], provide a higher-level design entry method than manually writing HDL: they automate the creation of the hardware that connects functional modules together. Similarly, Network-on-Chip (NoC) architectures like CONNECT[18], Split+Merge[12], and Hoplite[13] provide ready-made interconnect solutions for FPGA designs. These tools and architectures simplify the creation of large, complex hardware systems.

A consequence of designing at this level of abstraction is that the implementation of the interconnect is hidden from the designer behind a standardized signaling protocol like Avalon[2] or AXI[4]. As a result, a functional module *can not* assume, in general, that the interconnect will provide a specific end-to-end latency, or that the latency will even be constant during system operation.

This presents a challenge when functional modules require synchronized data arrival from two or more sources. To guarantee synchronization even in the face of unknown interconnect latency, FIFOs or similar constructs can be inserted just before the functional module inputs, and dequeued when the module sees fit.

The alternative would be to manually create the interconnect with explicit, fixed, known latencies, such that the data arrives at each functional module input at the correct clock cycle by design. While this removes the area penalty incurred by FIFOs, it requires significantly more effort for the designer. They must either create the interconnect in HDL, giving up

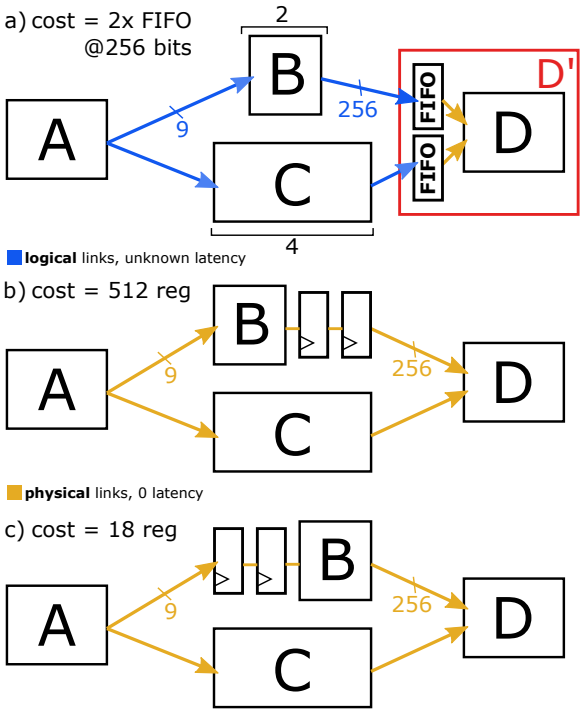

Figure 1: Three solutions for ensuring that inputs at module D arrive at the same time: a) Using an automated system-building tool and modifying module D (as D′) by adding FIFOs at its inputs, or building the interconnect manually and adding two pipeline stages b) after or c) before module B.

FPGA '17, February 22 - 24, 2017, Monterey, CA, USA

ISBN 978-1-4503-4354-1/17/02…$15.00

DOI: http://dx.doi.org/10.1145/3020078.3021729

the productivity advantage of automated tooling, or (if their tools allow) manually specify the locations of registers in the interconnect. Not only must they add the correct *number* of registers, but they could potentially select among many equally-valid *locations* to insert them, with some yielding higher area usage than others.

Figure 1 illustrates a motivating example, containing four functional modules, of such a synchronization problem. Here, modules B and C are internally fully pipelined with fixed input-to-output latencies of 2 and 4 clock cycles, respectively. Each takes a 9-bit input and produces a 256-bit output, as a block RAM might commonly do, for example. Module D requires matching inputs to arrive during the same clock cycle.

If a tool is used to build this system, the four modules would be connected with abstract logical links that are synthesized to an implementation with an unknown latency. The designer may employ the solution shown in Figure 1(a), where module D is wrapped inside a new module D′ that adds two 256 bit wide FIFOs to synchronize the data arrival at the inputs.

However, if the designer had full control over the design of the interconnect, they may opt instead to use balancing registers to add the correct, fixed amount of extra latency to synchronize data arrival, and avoid the unnecessary hardware complexity of FIFOs. Two equally-valid solutions are shown in Figures 1(b) and 1(c), with the latter having the lower area usage of 18 (versus 512) registers. The choice of (c) over (b) may be trivial to see in this example, but a larger more complex system would present the designer with less-obvious choices.

In this paper, we propose augmenting an interconnect synthesis tool with the ability to automatically create area-optimal, fixed-latency interconnect in response to the synchronization needs of the designer's application, effectively enabling solutions such as Figure 1(c) to be generated automatically.

This is accomplished by accepting, from the designer, a set of *synchronization constraints*, which take the form of equations or inequalities that relate the end-to-end latencies of one or more logical links and a constant. The tool then satisfies these constraints during interconnect creation by inserting the correct number of balancing registers, favouring solutions that use the minimal amount of total registers.

The problem of synchronizing pipelined systems with delay buffers is itself not novel, as will be discussed in Section 2. However, existing system-building tools for FPGAs lack this capability; adding it would allow their use in constructing new classes of applications, such as systolic arrays[15], beyond the traditional use cases of "processor plus memory-mapped IP cores" or streaming dataflow pipelines.

To this end, we will demonstrate the creation of an FPGA-based implementation of a convolutional neural network (CNN) accelerator using our new synchronization constraint enabled system building methodology. Its area and performance will be compared against a similar system built using a FIFO-based synchronization approach resembling Figure 1(a).

We will present a review of previous work on the synchronization problem in Section 2 and our own interconnect synthesis-specific formulation in Section 3, where we augment our own open-source GENIE interconnect synthesis tool[19, 20] with the ability to apply synchronization constraints. This is followed, in Section 4, by the description of the CNN accelerator example design. Section 5 evaluates and compares its clock frequency and area against other design approaches. Finally, we conclude in Section 6.

2. PREVIOUS WORK

The optimization problem of inserting the minimal amount of delay elements to satisfy the synchronization of fixed-latency pipelined computation blocks is a form of *buffer minimization problem* and has been well-studied[11]. The integer programming based approach that we will use to solve it in Section 3.4 is a basic approach that has had refinements made by others to improve its asymptotic runtime complexity through decomposition approaches[7] or graph-theoretic reformulations[5].

The buffer minimization problem has also found use in High-Level Synthesis (HLS) tools[9, 6]. There, hardware modules representing operations in a control/data flow graph are scheduled to begin at a certain clock cycle in order to satisfy dependencies, which naturally leads to the same problem of determining the optimal locations for delay element insertion.

Our contribution is to apply this existing buffer minimization problem in the context of FPGA interconnect synthesis tools, thereby exposing synchronization directly to the end user. Existing tools[2, 21, 16] do not have this capability; individual components can stall upstream communication via backpressure, but this complicates the interconnect implementation. Alternatively, the designer could buffer incoming signals manually, as part of their functional modules.

FPGA-based network on chip architectures[18, 12, 13] are not complete system-building tools like the software provided by the FPGA vendors. Furthermore, contention in packet-switched networks makes packet latency variable, preventing global synchronization of transmissions from taking place. Our approach could theoretically be applied to circuit-switched networks[10], however.

Previously, we introduced our own system-building and interconnect synthesis software, GENIE[19, 20], which is based on Split and Merge[12] routing primitives. Although it is capable of creating backpressure-free interconnect, it currently suffers from the same lack of global synchronization capability as the existing vendor-provided tools. In this paper, we augment it with synchronization constraints. This will extend the automation afforded by interconnect synthesis software to the creation of tightly-scheduled pipelined systems, such as systolic arrays, in addition to (and co-existing with) traditional memory-mapped and streaming components.

3. PROBLEM DEFINITION

In this section, we begin with a review of the traditional system representation seen by interconnect synthesis tools, and proceed to augment this with our formalization of synchronization constraints. An integer programming based method will be provided to solve the constraints and generate the necessary interconnect. Two additional post-processing optimizations will be described that further reduce interconnect area usage.

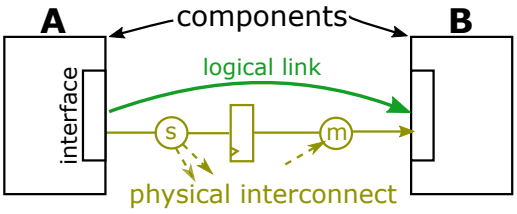

Figure 2: Elements of GENIE's representation of a system, with example generated interconnect.

3.1 System Representation

First, we review the terminology used to represent a system as seen by a generic interconnect synthesis tool. Although we are extending GENIE in this paper, the concepts presented here should be applicable and relevant to other tools as well.

The designer creates a system containing instances of *components* that represent functional modules. The individual input/output ports of each component are grouped into higher-level connection points called *interfaces* which adhere to some protocol and have well-defined roles for each constituent signal. The designer defines *logical links* between the interfaces of instantiated components to specify that they should communicate. The tool then accepts this system representation as input, and realizes it by instantiating the components and connecting them with physical interconnect according to the logical links.

The structure and details of the generated hardware will differ depending on the tool and interconnect architecture, but in general will include functions for distribution, arbitration, buffering, and conversion. For example, a network-on-chip architecture will instantiate routers that perform most of those functions, joined by wires. The GENIE tool we are extending uses Split nodes for distribution, Merge nodes for arbitration, and FIFOs or registers for buffering depending on whether backpressure is required. The exact arrangement of the primitives used by GENIE to realize the logical specification is defined by an optionally-customizable *topology*. More details can be found in previous work[20].

Figure 2 illustrates an example of the basic elements in a system representation: two component instances A and B are connected with a logical link, which is turned into physical hardware consisting of a split node, a register, and a merge node, which connect to other components in the system that are not shown. This is the baseline which we seek to extend next.

3.2 Internal Links and Chains

In the existing representation, logical links originate and terminate at the interfaces of components. In order to capture the type of global synchronization requirements depicted in the opening example shown in Figure 1, we must first extend the basic system representation of the previous section with the ability to specify communication *through* components.

Internal links serve this purpose – they define a communication path from one of a component's receiving interfaces to one of its transmitting ones. Each internal link has an associated fixed latency, in clock cycles, and is explicitly specified by the designer as part of a component's definition. It is also possible for an interface to participate in multiple internal links within a component, each with a different latency.

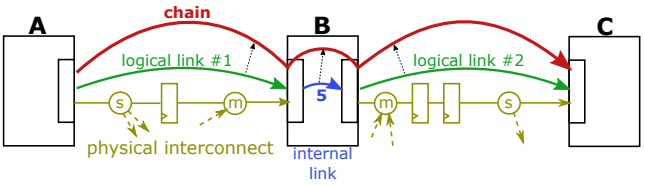

Figure 3: A chain spanning three components A, B, and C, with its constituent two logical links and one internal link within B that has a latency of 5 clock cycles. Each logical link will be realized into example interconnect

We can now define a higher-level type of construct called a *chain*, which captures a transmission beginning at one component, through zero or more intermediate components, and terminating at an ultimate destination. A chain defines a contiguous set of one or more logical links and internal links. Figure 3 illustrates an example of a chain spanning three components - A, B, and C. The intermediate component B has an internal latency of 5 clock cycles.

3.3 Synchronization Constraints

Recall that the goal of this work is to automatically generate interconnect that obeys user-specified synchronization constraints. Now, with the ability to capture multi-component transmissions using chains, we are ready to introduce the formulation of the constraints proper. Given a set of $N \geq 1$ chains $h_1, h_2, \ldots, h_N$, a synchronization constraint takes the form:

$$h_1 \pm h_2 \pm \cdots \pm h_N \ \mathbf{op}\ K \tag{1}$$

where **op** is a comparison operator (one of $<, \leq, =, \geq, >$), and K is an integer. Each term h_i represents the end-to-end latency, in clock cycles, of that chain. This general form allows the designer to specify arbitrary latency relationships between chains, or to bound the latency of an individual chain. A chain (and its constituent logical links) can participate in multiple constraints.

Figure 4 restates the example system in Figure 1 as an input to GENIE using chains, logical links, and synchronization constraints. The explicitly-specified physical interconnect in the original example has been replaced with logical links between components A, B, C, and D, whose interfaces have been named 'in' and 'out'. The latencies of B and C are captured with internal links. The requirement for D's inputs to arrive simultaneously has been captured as a synchronization constraint between two chains $h_0 = \{A.out \rightarrow B.in, B.out \rightarrow D.in\}$ and $h_1 = \{A.out \rightarrow C.in, C.out \rightarrow D.in\}$, with the constraint being that $h_0 = h_1$.

3.4 Optimization Problem Formulation

The synchronization constraints are used as an input to the tool. The goal is to use them to guide the generation of interconnect. However, in general, there may be *many* legal solutions, differing in the number of total inserted registers; ideally, we would like to find the solution that yields the fewest. This is the well-known buffer minimization problem[11], and this section formalizes a version of it using the system representation terminology introduced previously.

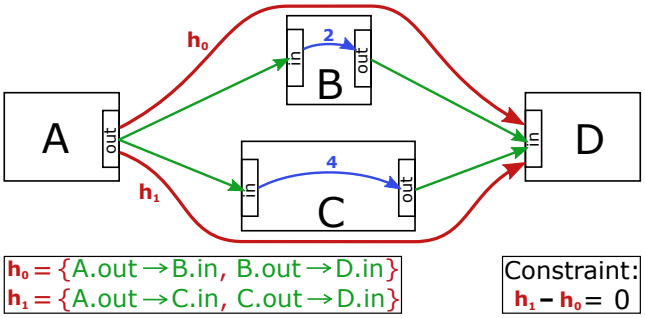

Figure 4: The example in Figure 1 restated using a synchronization constraint on an interconnect synthesis problem. The two chains from A to D are constrained by the user to have equal latency.

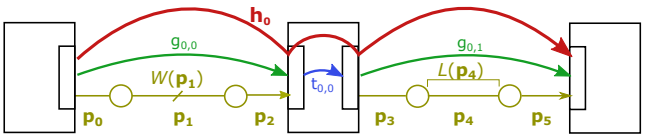

Figure 5: A single chain consisting of one internal link and two logical links, which are synthesized into interconnect containing a total of four primitives and six physical links p_0 through p_5. $W(p_1)$ is the width in bits of link p_1 and $L(p_4)$ is the necessary extra latency, in cycles, of p_4.

Let $\mathbf{C}$ be the set of all user-provided constraints, each taking the form of Equation 1. For a constraint $c \in \mathbf{C}$, let $\mathbf{H}_c$ represent the set of chains that appear on the left hand side. A chain $h \in \mathbf{H_c}$ has an associated set of logical links, $\mathbf{G}_h$, which is a subset of all logical links $\mathbf{G}$. Chains also traverse internal links, that are represented by the set $\mathbf{T}$.

The tool realizes logical links into physical interconnect consisting of hardware primitives connected by *physical links*, which directly represent RTL nets. $\mathbf{P}$ is the set of all physical links. By splicing registers into physical links, cycles of delay can be added in appropriate places to satisfy the overall set of synchronization constraints. If we define $L(p)$ as the number of registers to insert into physical link p, then the goal of the overall optimization problem is to solve $L(p)$ for all $p \in \mathbf{P}$.

We also wish to satisfy the constraints using the minimum total amount of registers. If $W(p)$ represents each physical link's width in bits, then this objective can be codified as the minimization of the following cost function:

$$\#\ of\ registers = \sum_{p \in \mathbf{P}} W(p)L(p) \tag{2}$$

Figure 5 illustrates the relationship between an example chain h_0 and its constituent logical, internal, and physical links, as well as the properties W and L of physical links. The latency L of a physical link is a numerical annotation that is only later realized as extra registers.

To solve the set of constraints $\mathbf{C}$, each constraint $c \in \mathbf{C}$ is first converted from the form of Equation 1, as provided by the user, into that of Equation 3 by expanding each chain term h_i into its constituent physical links p_i and internal links t_i:

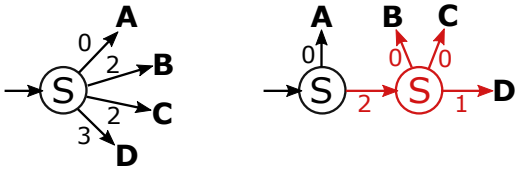

Figure 6: Left: a Split node feeding four physical links with differing latencies. Right: applying systolic retiming, the number of required register stages is reduced from 7 down to 3.

$$h_1 \pm h_2 \pm \cdots \pm h_N \ \mathbf{op}\ K_c \tag{1}$$

$$L(p_0) \pm \cdots \pm L(p_N)\ \mathbf{op}\ K_c \pm L(t_0) \pm \ldots \pm L(t_M) \tag{3}$$

The left-hand side consists of unknowns (the latency of physical links to solve for), and the right-hand side of constants (the user's constraint constant K_c together with the fixed latencies of internal links denoted by $L(t_i)$). Interconnect primitives in GENIE have zero latency, but for general applicability, the latencies of interconnect primitives should also be included on the right-hand side.

The resulting system of inequalities is in a canonical form suitable for solving using integer programming: the (nonnegative, integer) unknown variables $L(p)$ are on the left-hand side, and constants are on the right-hand side. A solution to the IP problem yields the values of $L(p)$ for all for all p, subject to the optimization criterion of minimizing the cost function of Equation 2, which is linear with respect to the unknown variables $L(p)$.

Note that additional techniques[7, 5] can be used to improve the asymptotic performance of solving this optimization problem.

3.5 Systolic Retiming Transform

The solution to the optimization problem can lead to scenarios like the left side of Figure 6. Here a Split node (used for distributing signals to different destinations) fans out to four physical links that were assigned three distinct latency values. Realizing this assignment requires 7 register stages - the sum of the 2, 2, and 3 register stages for destinations B, C, and D. However, if we allow for the freedom to change the topology of interconnect primitives *after* initial latency assignment, a less-costly solution can be found, as shown in the right side of the figure where an extra Split node is used to reduce the cost to three register stages. We call this optimization the *systolic retiming transform*.

It is performed after the initial $L(p)$ values have been assigned by the solution of the optimization problem. The overall process is:

1. Sort the Split node's fanout physical links into bins by their latency assignments $L(p)$.
2. Remove the initial Split node.
3. Create a Split node for each bin from step 1.
4. Connect each Split node from step 3 to the original destinations in the respective latency bins, but reset the latency on these new physical links to 0.

5. Connect the Split nodes together with physical links whose latencies are the differences between successive bins.

6. The last Split node can be removed and its sole fanout reassigned to the second-last Split node.

After performing the systolic retiming transform, the updated latency values $L(p)$ on each physical link are used to insert the corresponding number of register stages.

3.6 Long Register Chain Optimization

Current FPGAs can repurpose logic blocks to be used as distributed RAM resources. Although they are not as wide or deep as larger, traditional block RAMs, they still offer more bits of storage per logic block than registers do.

After satisfying the design's synchronization constraints and performing the systolic retiming transform, we perform one final post-processing optimization: long chains of balancing registers are replaced with a single distributed RAM based implementation to reduce area usage. Unlike a FIFO, this memory-based delay element is functionally equivalent to a chain of registers (for example, it preserves non-valid bubble cycles rather than compacting them).

Chains of registers are replaced with memory-based delay elements when the estimated cost is lower. This depends on the length of the chain, the data width, and FPGA-specific architecture parameters such as the maximum depth and width of distributed RAM elements.

As an aside, the introduction of registers embedded directly in the routing fabric of the latest FPGAs[3] will change the cost/benefit analysis of performing this optimization. It may be preferable to leave chains of registers untouched when targeting such architectures.

3.7 Limitations and Scope

Backpressure-free synchronization with balancing registers requires that components have internal links with fixed, static latencies. This will exclude any part of the system that communicates with off-chip peripherals that have non-deterministic latency, such as DRAM controllers. A less-obvious limitation of the presented approach is that it excludes communication patterns in which $N > 1$ sources wish to communicate with 1 sink and the N transmissions are not guaranteed to be mutually-exclusive in time. This scenario implies one or more of the sources would have to be stalled to avoid data loss, implying the need for backpressure and thus non-fixed latency due to the resulting stalls.

A real system may contain a mixture of both types of communication: latency-insensitive with backpressure, and backpressure-free. These regions may even interact with one another. For example: an off-chip memory controller feeding returned read data into an adjoining network of fixed-latency processing modules. It's not possible to determine when valid read data will be returned from memory, but our synchronization constraints could still be applied to the latter region to correctly distribute that data *relative to* its entry into the region.

4. DESIGN EXAMPLE

In this section, we describe a concrete example application for which synchronization constraints can be used effectively: a Convolutional Neural Network (CNN) accelerator targeted towards image classification workloads. There exist many FPGA implementations of CNNs (for example, [23]), and so our goal is not to improve on the state of the art of CNN architecture. Rather, we wish to illustrate the power of synchronization constraints in aiding the creation of a functionally correct, high-performance implementation using an interconnect-centric design flow. We will begin with an overview of the CNN context and then provide a description of the overall design, followed by the implementation details and results in the next section.

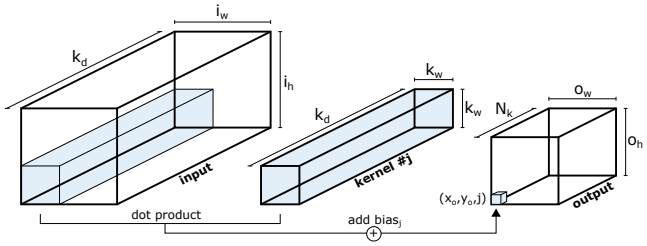

Figure 7: Visualization of the dot product between a kernel and a kernel-sized subvolume of the input. This produces a single output voxel.

4.1 CNN Background

CNNs are a machine learning technique[17] and have been effectively used in applications ranging from image classification [14] and speech recognition[1], to playing the game of Go[22]. CNNs operate in two modes: training and inference. Training 'teaches' the network to classify inputs, and involves feedback. Inference is a feed-forward process that uses the trained neural network to classify inputs. As we have chosen the problem of image classification, training can be performed offline, so we focus only on building hardware to perform inference.

Each input image is split into color channels and stacked together to form a 3-dimensional volume. This image undergoes a chain of different computation stages, each producing an intermediate volume that represents higher-order features of the original image. The final output is a low-dimensional array that directly represents the probabilities of different image categories. The most time-consuming[8] processing stages are the *convolutional layers* from which CNNs derive their name. Our accelerator only implements these.

A convolutional layer convolves its input image with N_k different kernels to produce the output. Each kernel contains weights from off-line training that are constant during inference. The image, kernels, and output can be visualized as 3-dimensional volumes, and the convolution process as a repeated dot product of volumes: Each voxel of the output volume is produced by calculating the dot product of a kernel volume, of width/height k_w and depth k_d, with an equally-sized sub-volume of the image, and then adding a kernel-specific bias constant. This dot product operation, restated as Equation 4, is also visualized in Figure 7.

$$out(x_o, y_o, j) = \sum_{x=0}^{k_w-1} \sum_{y=0}^{k_w-1} \sum_{z=0}^{k_d-1} kern_j(x, y, z) \cdot inp(x + x_0, y + y_0, z) + bias_j \quad (4)$$

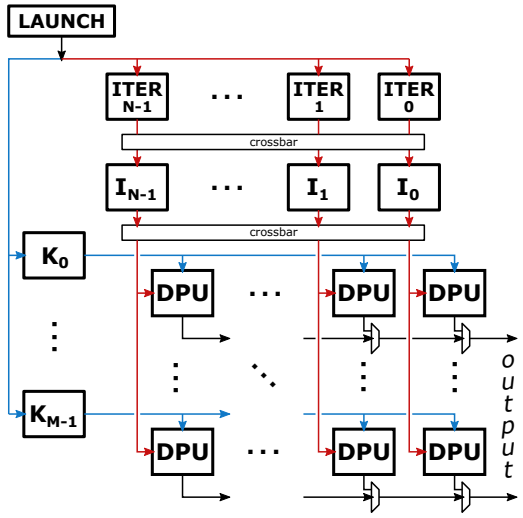

Figure 8: Block diagram of the dot product array. It contains $N \times M$ Dot Product Units, fed by N on-chip image buffers and M kernel buffers.

To generate the entire output volume (and complete the convolution), multiple such dot products must be performed. Each time, a different kernel and/or a different image sub-volume is chosen. The former corresponds to the z coordinate of the output voxel, and the latter to the x and y coordinates of the output. Additionally, When the sub-volume is swept across the input image in the x and y dimensions, the amount it moves within the input is called the *stride*. Equation 4 assumes a stride of 1 for simplicity.

Note that in the overall convolution, many dot product operations share either the same kernel or the same region of the input volume. This is exploited to achieve high parallelism in our hardware implementation, which is described next

4.2 Hardware Design

Our CNN design is able to process a single convolutional stage at a time, with size parameters that are configurable at runtime. The input image volume and kernels are initially stored in off-chip memory and are streamed into on-chip buffers, and the output volume is written back into off-chip memory as well. Voxels are represented using 16-bit fixed-point integers.

We accelerate the convolution by performing many of its constituent dot product operations in parallel. This is implemented with an array of $N \times M$ *dot product units* (DPUs), as shown in Figure 8. Each DPU consumes 16 input voxels and 16 matching kernel voxels per clock cycle, using hardened FPGA DSP blocks to do voxel-wise multiplication and accumulation. The image and kernel data are supplied by N on-chip image and M kernel buffers, each providing 256 bits of data per cycle. They are labeled I_0 through I_{N-1} and K_0 through K_{M-1} respectively, in Figure 8.

Since many DP operations share either kernel or image data, the buffer outputs can be broadcast to many DPUs simultaneously, efficiently utilizing on-chip read bandwidth. Each kernel buffer is broadcast to a row of DPUs, and each image buffer can broadcast to one of the N columns (targeting a different column every clock cycle).

The blocks marked $ITER$ in Figure 8 sequentially generate addresses to read the image buffers, and each $ITER$ unit reads a different image buffer every clock cycle. The kernel buffers also have address-generating logic, but it is built-in, feeding only its associated buffer with a straightforward access pattern, and is thus omitted from the figure for simplicity.

Once all image and kernel buffers have been filled, control logic issues a $LAUNCH$ signal that begins address generation by the $ITER$ units and the kernel buffers' built-in equivalents. The addresses are used to read the respective kernel and image buffers, feeding DPUs with data, which eventually produce a result.

This process is repeated many times, as neither the entire image (nor all the kernels) will necessarily fit into the available on-chip buffers. The synchronization of the delivery of kernel and image data streams to DPUs is the problem which we aim to help solve using the synchronization constraints described in this paper.

5. IMPLEMENTATION AND RESULTS

In this section, we seek to measure the difference in quality of results (area, clock frequency) when building the CNN accelerator system using an interconnect synthesis flow augmented with synchronization constraints. To this end, we build three different versions of the system using GENIE:

1. A basic, unpipelined implementation with no significant synchronization requirements and low performance.

2. A pipelined implementation, which introduces synchronization requirements that are addressed by the manual addition of FIFOs to the functional modules.

3. The same pipelined implementation as Version 2, but with the synchronization requirements solved using the flow described in Section 3.

The first naïve unpipelined version is easy enough to make functionally correct with minimal manual effort, but suffers from low performance as a result.

The second version improves on this by inserting pipeline registers to increase clock frequency, at the cost of complicating the problem of achieving functional correctness, which is solved with the insertion of FIFOs and adds a small amount of extra effort for the designer. This implementation represents a realistic higher-performance solution that a designer may choose, if they do not have access to the synchronization constraint flow presented in this paper.

The third version keeps the performance-increasing pipeline registers from the second version, but uses the synchronization constraint flow to allow GENIE to automatically insert balancing registers, rather than relying on manually-inserted FIFOs. We will demonstrate that this version maintains high performance and uses significantly less area than the FIFO-based design.

5.1 Methodology

All three versions are generated with GENIE. In each, we also vary the number of DPUs in the accelerator by changing the number of columns (N) of the DPU array. This increases the size of the accelerator, adding computational power, while also increasing the distances that signals need to travel.

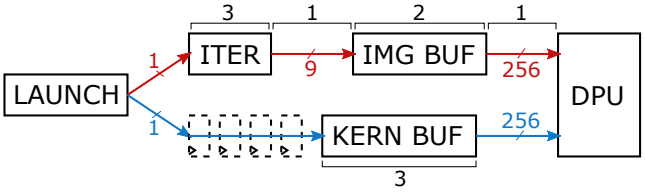

Figure 9: The paths taken by image data (top), and kernel data (bottom) to reach the inputs of a single DPU. Also shown are bit widths of each logical link, and the fixed latencies of design modules and interconnect. Four extra registers are required on the kernel path to achieve balance.

The design was synthesized with Altera Quartus Prime Pro 16.0 and targets an Arria 10 `10AX115S2F45I2SGES` device. We measure clock frequency, register usage, and the number of Adaptive Logic Modules (ALMs). ALMs represent post-packing logic, register, and distributed RAM usage, and thus the overall consumed area (not including DSP blocks).

5.2 Version 1: Unpipelined Implementation

The first version achieves correct operation without optimization of the clock frequency. It represents a realistic first design iteration. The key simplification made in this initial version is that kernel and image data are broadcast combinationally to all DPUs simultaneously, rather than distributed in a systolic pipelined fashion.

For correct operation, kernel and image data must arrive simultaneously at each DPU. In this version, this synchronization is achieved by manual insertion of registers to balance arrival times. The location, and number, of such registers are determined by inspection of the signal paths and existing fixed latencies.

Figure 9 shows the paths taken by the image and kernel data through the computation array toward a single DPU, as well as the fixed latencies of the intervening blocks and generated interconnect. From inspection, it is trivial to balance the two paths by manually inserting four registers between the $LAUNCH$ signal generator and a kernel buffer.

Canonically, this register insertion is performed for all kernel buffers individually. Fortunately, since this version of the array broadcasts $LAUNCH$ to all kernel buffers simultaneously, a single instance of the four registers can be used, and the output broadcast to *all* of the kernel buffers. In the two other versions of the array, synchronization will be more complicated once this broadcast is removed.

Table 1: Area and Frequency for Unpipelined Array

N	DSP	ALM	REG	F_{max} (MHz)
4	512	14421	40335	333
6	768	21290	56603	265
8	1024	28977	72819	230
11	1408	48349	97250	169

Table 1 shows the speed and area results for various parameterizations of the Version 1 CNN hardware: changing N increases the number of DPUs, and by extension, DSP block usage, ranging from a third of the 1520 available DSPs at $N = 4$, to 93% of the DSPs at $N = 11$. Since the circuit broadcasts data combinationally to all DPUs, it is no surprise that the observed clock frequency drops (from 333 to 169 MHz) as the number of DPUs increases – the signals must travel longer distances to reach their destinations. Examination of the most-critical paths from timing analysis confirms this intuition.

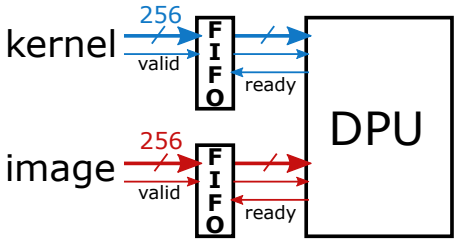

Figure 10: FIFOs are inserted at each DPU to synchronize kernel and image data arrival times. A DPU asserts the *ready* signals to simultaneously dequeue both FIFOs.

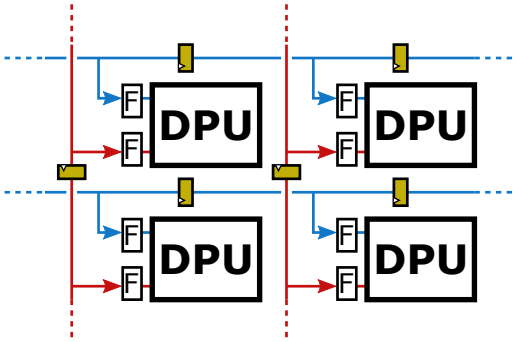

Figure 11: A section of the modified Version 2 DPU array, showing the locations of manually-added pipeline registers as well as the pairs of FIFOs added in front of each DPU.

5.3 Version 2: FIFO-based Implementation

To achieve higher clock frequencies, we can break up the broadcast of the image and kernel data with pipeline registers. However, this affects the kernel and image data arrival times at each DPU and complicates their synchronization. One solution is to place FIFOs in front of every DPU, as shown in Figure 10.

This requires a manual change to the DPU module, but can be performed once and then replicated aross the entire array. As long as the FIFOs are deep enough to cover the worst case arrival disparity, the designer can independently vary the amount of delivery pipeline stages without concern about functional correctness. Figure 11 illustrates a portion of the array after the insertion of FIFOs and pipeline registers.

Table 2 gives the area and frequency measurements for the Version 2 FIFO-based implementation. The Δ column compares the results with the Version 1 unpipelined system described previously. The clock frequency improves by up to 40% but the average ALM and register usage is increased by 167% and 84% respectively. ALM usage is affected by the FIFOs, as they are configured to use distributed RAM rather than more expensive block RAM.

While using FIFOs to maintain synchronization is simple for the designer to implement, it is not an elegant solution for

Table 2: Area and Frequency of Pipelined+FIFO Implementation

N	DSP	ALM	Δ	REG	Δ	F_{max} (MHz)	Δ
4	512	39329	2.73x	40335	1.74x	378	1.13x
6	768	59160	2.78x	103645	1.83x	343	1.30x
8	1024	79976	2.76x	136955	1.88x	321	1.40x
11	1408	117982	2.44x	187181	1.92x	236	1.40x

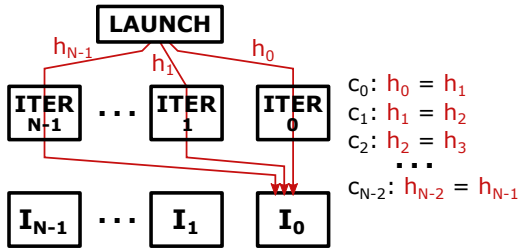

Figure 12: $N-1$ synchronization constraints, c_0 through c_{N-2}, that force all the possible N chains feeding image buffer I_0's address input to have equal latency, avoiding collision of read addresses as they arrive at the buffer.

a system which originally had no backpressure and completely deterministic data arrival times. Ideally, we would like to insert the correct amount of registers in the correct locations automatically, using registers to balance delays instead of FIFOs. This is done in Version 3 below, using synchronization constraints.

5.4 Version 3: Synchronization Constraint Implementation

In this version, we begin with the kernel and image buffer broadcasts manually pipelined in the same systolic fashion as in Figure 11, but *without* the FIFOs for synchronization. Instead, to achieve correct data delivery in the presence of the pipeline registers, we define several sets of synchronization constraints, which GENIE will use to automatically insert the necessary registers to balance delays.

The $ITER$ units have been carefully designed such that, as long as they receive the $LAUNCH$ signal simultaneously, all N of them will *never* send a transmission to the same destination image buffer during the same clock cycle. This temporal mutual-exclusivity of transmissions is critical to avoiding the need for backpressure within the design, and must be preserved all the way from the output of the $ITER$ units to the inputs of the image buffers. Therefore, the first set of constraints ensure exactly this scenario, by demanding that all possible $LAUNCH$ to image buffer chains are of the same length. Figure 12 illustrates these constraints for the first image buffer, I_0. Similar sets of constraints are needed for the remaining image buffers, I_1 through I_{N-1}.

The next set of constraints is intended to fulfill the role previously performed by the FIFOs: to ensure that kernel and image data arrive synchronized at each DPU. This requires the chain supplying the kernel data to a DPU to have the same latency as all chains (from all image buffers) supplying the image data.

Figure 13 illustrates these constraints. Note that we choose a single arbitrary $ITER$ unit for the N image data chains,

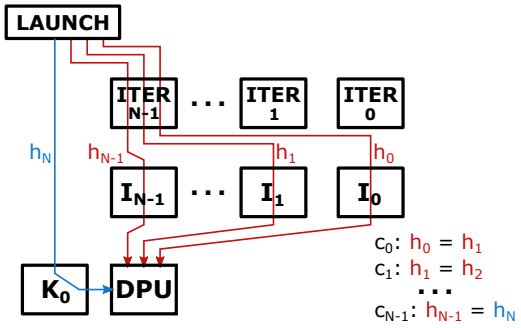

Figure 13: N synchronization constraints, c_0 through c_{N-1}, that force kernel and image data to arrive simultaneously at a DPU.

rather than the quadratic-in-N-sized exhaustive set that traverses through all possible $ITER$s. We are allowed to perform this simplification due to the first set of constraints of Figure 12, which forced all $ITER$-to-image-buffer chains to be equal.

Figure 14 shows the hardware that results from applying this total set of synchronization constraints. As before, four registers have been added to delay the $LAUNCH$ signal before it reaches the first kernel buffer. These four registers were inserted manually in the original unpipelined implementation, but now they are inferred automatically from the constraints.

In addition to these registers, more have been automatically added to balance data arrival. Proceeding down the column of kernel buffers, each buffer receives its 1-bit $LAUNCH$ signal one cycle after the previous – the optimizer naturally favoured delaying the signal here, at the input of the kernel buffers, rather than at the more expensive 256-bit outputs.

Our constraints forbid the optimizer from employing a similar elegant solution for delaying the 1-bit $LAUNCH$ signal to the $ITER$s for image data. This is because each $ITER$ ultimately feeds all DPU columns, and thus can't be individually delayed at the source. The best legal solution was instead to add an increasing number of expensive 256-bit registers at the top of every DPU column to delay the image data.

Table 3: Area and Frequency for Synchronization Constraint-based Array

N	DSP	ALM	REG	F_{max} (MHz)
4	512	21835	68673	376
6	768	32867	101206	343
8	1024	45024	133617	330
11	1408	69218	182658	248

The clock frequency and area usage of the Version 3 hardware are given in Table 3. As with the other two versions, area usage increases and clock frequency decreases with increasing array size.

Figure 15 compares these results to that of the other two versions. Clock frequency remains within 5% of Version 2, which is expected since it uses the same arrangement of pipeline registers. However, this same level of performance is achieved using 43-45% fewer ALMs than Version 2 (Figure 15b). Given that the total number of registers remains

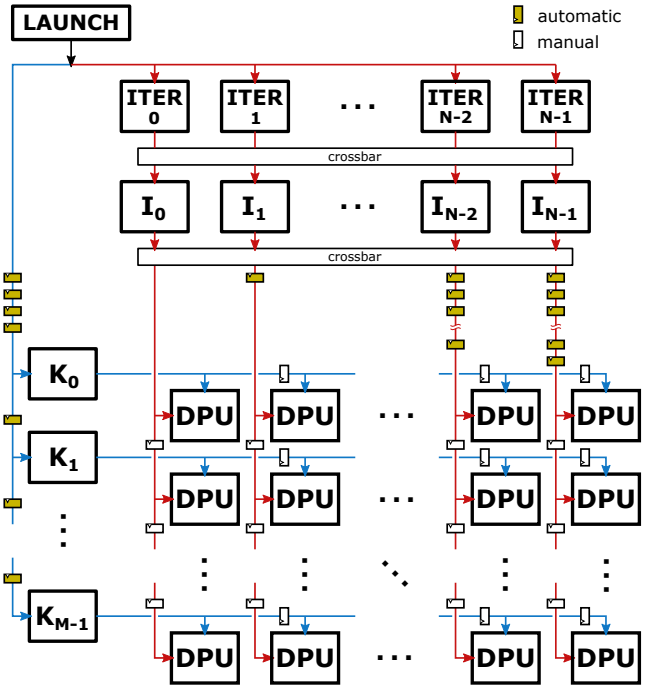

Figure 14: Version 3 DPU array resulting from the application of the synchronization constraints. Automatically-inserted balancing registers are distinguished from the manually-inserted broadcast distribution registers.

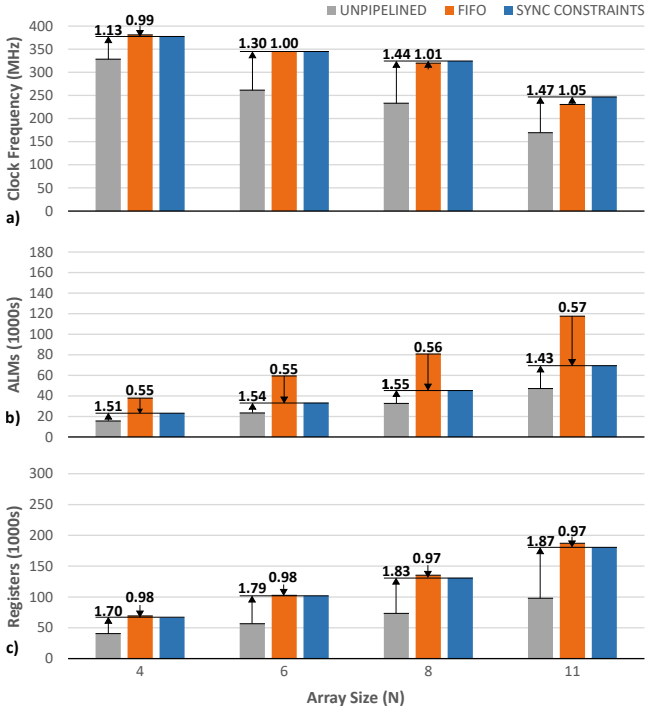

Figure 15: Absolute and relative a) clock frequency, b) ALM usage, and c) register usage of the three array implementations, with varying array size from $N = 4$ to $N = 11$. The relative values compare the synchronization constraint implementation versus the unpipelined and FIFO implementations.

roughly the same (Figure 15c), this observed difference in ALM usage can be attributed to Version 2's per-DPU FIFOs.

Compared to Version 1, Version 3 uses 43% more area (ALMs) to achieve a 47% increase in clock frequency at the largest array size of $N = 11$. Using synchronization constraints, this result was achieved automatically by GENIE and provided an easy way to achieve a correct pipelined implementation of the neural network array without the need to modify any of the functional blocks of the system.

6. CONCLUSION

We have created a means for an interconnect synthesis tool to satisfy data synchronization requirements by automatically inserting the correct number of balancing registers into area-optimal locations within the interconnect. The requirements are represented as inquality-based constraints provided by the designer. For portions of a system that lack backpressure, this automates and therefore simplifies the creation of globally, rather than locally, synchronized interconnect for correct circuit operation.

To illustrate the utility and potential gains of this approach, we applied it to the design of an FPGA-based convolutional neural network accelerator. Here, synchronization constraints were used to help solve a realistic design problem: maintaining correct data synchronization in spite of the insertion of performance-enhancing pipeline registers. The backpressure-free interconnect generated with our approach used less area than a more traditional FIFO-based method of synchronization, which would be a more representative implementation given existing interconnect synthesis tools and their communication protocols. The end result is that system building tools can more efficiently build interconnect for a new class of applications that contain fixed-latency pipelined modules.

Although the general form of our constraints allow a designer to use them to explicitly add performance-enhancing pipeline registers to a design, rather than manually add them as we have done in our example, this still requires manual intervention by the designer to know *which* paths are critical. In the future, we envision synchronization constraints propagating to the place-and-route stage of synthesis, where back-end tools are already capable of inserting pipeline registers embedded into the FPGA interconnect[3]. With the presence of synchronization constraints, such late-stage physical modifications could be performed on the design while ensuring that the resulting circuit will continue to operate correctly.

7. REFERENCES

[1] O. Abdel-Hamid, A.-R. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu. Convolutional Neural Networks for Speech Recognition. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 22(10):1533–1545, Oct. 2014.

[2] Altera. QSys - Altera's System Integration Tool. http://www.altera.com/products/software/quartus-ii/subscription-edition/qsys/qts-qsys.html.

[3] Altera. Stratix 10 FPGA and SOC. https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html, 2016.

[4] ARM Ltd. AMBA Open Specifications. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

[5] E. Boros, P. L. Hammer, and R. Shamir. A Polynomial Algorithm for Balancing Acyclic Data Flow Graphs. *IEEE Trans. Comput.*, 41(11):1380–1385, Nov. 1992.

[6] A. Canis, S. D. Brown, and J. H. Anderson. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[7] P. R. Chang and C. S. G. Lee. A Decomposition Approach for Balancing Large-Scale Acyclic Data Flow Graphs. *IEEE Trans. Comput.*, 39(1):34–46, Jan. 1990.

[8] J. Cong and B. Xiao. *Artificial Neural Networks and Machine Learning – ICANN 2014: 24th International Conference on Artificial Neural Networks, Hamburg, Germany, September 15-19, 2014. Proceedings*, chapter Minimizing Computation in Convolutional Neural Networks, pages 281–290. Springer International Publishing, Cham, 2014.

[9] J. Cong and Z. Zhang. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 433–438, New York, NY, USA, 2006. ACM.

[10] C. Hilton and B. Nelson. PNoC: a flexible circuit-switched NoC for FPGA-based systems. *IEE Proceedings - Computers and Digital Techniques*, 153(3):181–188, May 2006.

[11] X. Hu, S. C. Bass, and R. G. Harber. Minimizing the Number of Delay Buffers in the Synchronization of Pipelined Systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 13(12):1441–1449, Nov. 2006.

[12] Y. Huan and A. DeHon. FPGA Optimized Packet-Switched NoC using Split and Merge Primitives. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 47–52, Dec 2012.

[13] N. Kapre and J. Gray. Hoplite: Building austere overlay NoCs for FPGAs. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–8, Sept 2015.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012.

[15] H. T. Kung. Why systolic architectures? *IEEE Computer Magazine*, 15(1):37–46, Jan 1982.

[16] Lattice Semiconductor. LatticeMico System Development Tools. http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/EmbeddedDesignSoftware/LatticeMicoSystem.aspx.

[17] Y. Lecun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

[18] M. K. Papamichael and J. C. Hoe. CONNECT: Re-examining Conventional Wisdom for Designing Nocs in the Context of FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 37–46, New York, NY, USA, 2012. ACM.

[19] A. Rodionov, D. Biancolin, and J. Rose. Fine-Grained Interconnect Synthesis. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 46–55, New York, NY, USA, 2015. ACM.

[20] A. Rodionov and J. Rose. Automatic FPGA system and interconnect construction with multicast and customizable topology. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 72–79, Dec 2015.

[21] Xilinx Corporation. Accelerating Integration. http://www.xilinx.com/products/design-tools/vivado/integration/.

[22] M. Zastrow. Machine outsmarts man in battle of the decade. *New Scientist*, 229(3065):21 –, 2016.

[23] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM.