Co-tuning of Software Specializers and Hardware Accelerators within a CNN Application

Sean Roberts

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-53
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-53.html

May 11, 2016

Co-tuning of Software Specializers and Hardware Accelerators within a CNN Application

Sean Roberts
Department of Electrical Engineering and Computer Science
University of California, Berkeley
seanroberts@berkeley.edu

***Abstract*—Software specializers and hardware accelerators share the common goal of decreasing the runtime of an operation while being parameterizable and abstracting away underlying optimizations from users. The competition for reconfigurable hardware resources among candidate hardware accelerators means that tuning must take place at an application level and not at an operation level as is the case for software specializers. This paper presents a methodology for the co-tuning of software specializers and hardware accelerators so that both may be simultaneously used in applications. To explore the validity of this approach, experiments were carried with software specialized and hardware accelerated 2D stencils performing convolutions for trial convolutional neural networks. The results demonstrate that an application level co-tuner can discover which operations are best suited for software specializers and which merit the limited reconfigurable hardware resources required for hardware acceleration.**

I. Introduction

A. Motivation

Shrinking the size of transistors in computational devices has historically yielded performance benefits beyond what can solely be attributed to a Moore's law increase in the number of transistors on an integrated circuit. In 1974 it was demonstrated that shrinking the dimensions of a MOSFET by a factor of k would result in a equivalent factor of k decrease in the delay through the device, provided other parameters like doping concentration and voltage were scaled appropriately [1]. This Dennard scaling has for three decades allowed applications to become viable which were formerly prohibitively computationally intensive, simply by allowing sufficient time for the underlying technology to improve. This trend had been greatly inhibited since the mid to late 2000s due to supply voltage stagnation and the heightened impact of subthreshold leakage current [2].

Instead of relying on the diminishing benefits of device scaling to achieve performance improvements, an increasingly popular alternative is the use of specialized hardware circuitry like coprocessors. A notable example of this technique are graphics processing units, which despite their origins as visual accelerators for computer gaming are now employed widely in high performance scientific computing applications [3]. One important drawback of using hardware coprocessors is that programming them has proven to be a challenging departure from traditional software programming. It is frequently the case that existing software design methodologies are not transferable to heterogeneous systems without significant modification. Returning to the GPU example, new tools were required for effective utilization, like OpenCL, which is a framework for programs that execute on heterogeneous computing environments consisting of CPUs and GPUs [4].

The ASPIRE (Algorithms and Specializers for Provably optimal Implementations with Resilience and Efficiency) project is therefore interested in investigating strategies for the co-design and co-tuning of hardware and software within heterogeneous systems. Towards this end, an essential consideration for these systems is determining which of an application's operations are suitable to run on commodity processors and which might be worthwhile implementing with custom hardware. Two branches of the ASPIRE effort (SEJITS and Chisel, described below) are used in this report to investigate the combination of software and hardware representations of stencils, a computational pattern that underlies many applications. In addition to the initial results, limitations of the co-tuning methodology proposed are discussed along with challenges that are faced in general while orchestrating simultaneous software and hardware operations on heterogeneous systems.

B. SEJITS Software Specializers

SEJITS (Selective Embedded Just-In-Time Specialization) is a methodology used to encapsulate the performance benefits of operations expressed in efficiency level languages and make them available to users of productivity level languages [5]. SEJITS achieves this through the use of constructs called specializers which perform transformations on an abstract syntax tree of the PLL code to be specialized and then compiles to an ELL backend like C or OpenCL. An important benefit of this approach is that the users of a given specializer do not need to be knowledgeable of the low level optimizations being made, or the trade-offs of different choices, in order to benefit from the performance improvement. The optimal specializer is found just-in-time by applying different permutations of passes to the AST and tuning over them to find the best performing one for the provided operation. The JIT nature of SEJITS specializers means that specific knowledge of the data types for a given operation can be leveraged during transformations. This work upfront can be cached for later calls, provided that both the operation and data types match a specializer that has already been tuned.

C. Chisel Hardware Accelerators

Chisel (Constructing Hardware in a Scala Embedded Language) is a hardware construction language embedded in Scala as a DSL [6]. Circuits defined with Chisel can be emitted as C for efficient simulation and debugging or emitted as

Verilog for synthesis into physically realized circuits. Of great importance to this paper is that the full force of a modern programming language can be leveraged to produce highly parametric generators for hardware circuitry. In fact, entire processor generators have been made which require of users only architectural parameters (e.g. cache size) in order to produce a custom processor [7]. More modest designs like the hardware acceleration of a single operation are certainly within Chisel's parametric generation capabilities. Parallels can be drawn between Chisel hardware accelerators and SEJITS software specializers. Both can be customized based on the type and size of input and both can provide performance advantages to a user without requiring them to know how the advantages are gained.

D. Co-tuning Opportunity

There are two key differences between using software specializers and hardware accelerators to decrease the runtime of an operation. Firstly, JIT is assumed for this paper to be out of the question for hardware accelerators given current synthesis tools and reconfigurable hardware technology. Secondly, by only locally tuning to find an optimal software specializer for a given operation, an implicit assumption is made that this specializer won't be significantly impacted by the use of other possible software specializers within the same application. This assumption could be violated by contention over shared hardware resources like competing for cache space or threads of execution, but the simplifying assumption of independence is made regardless. Contrastingly, the different possible hardware accelerators of a given application are inextricably bound by their shared need for reconfigurable hardware resources. It will be the position of this paper that only one hardware accelerator can exist in reconfigurable hardware for the entire duration of an application, although alternative perspectives are considered at the end of this paper.

With the above differences in mind, a method co-tuning software specializer and hardware accelerators was devised which takes into consideration the operations of an entire application. First, operations which can be SEJITS specialized are tuned over locally (i.e. considering only the operation at hand) to derive optimal software specializers. Then operations which can be hardware accelerated are tuned over globally (i.e. running the entire application with one type of operation sent to hardware at a time) to find which candidate accelerator is best suited for being expressed on the physical hardware. Different factors which impact the worthiness of a hardware accelerator for a given application are the performance improvement relative to software alternatives, the number of times an accelerator will be exercised during the runtime of the application, and the ability of the main processor to perform useful work concurrently with the accelerator. If a hardware accelerator is found to be optimal for a given application and a software specializer also exist for the same operation, it means that the time spent deriving the specializer will have been wasted. Doing so is required in order to obtain an accurate understanding of the opportunity cost of using a specific hardware accelerator. Ultimately, an application could use an arbitrary number of software specializers an arbitrary number of times and 0 or 1 hardware accelerators an arbitrary number of times during the course of its execution, as shown in Fig. 1.

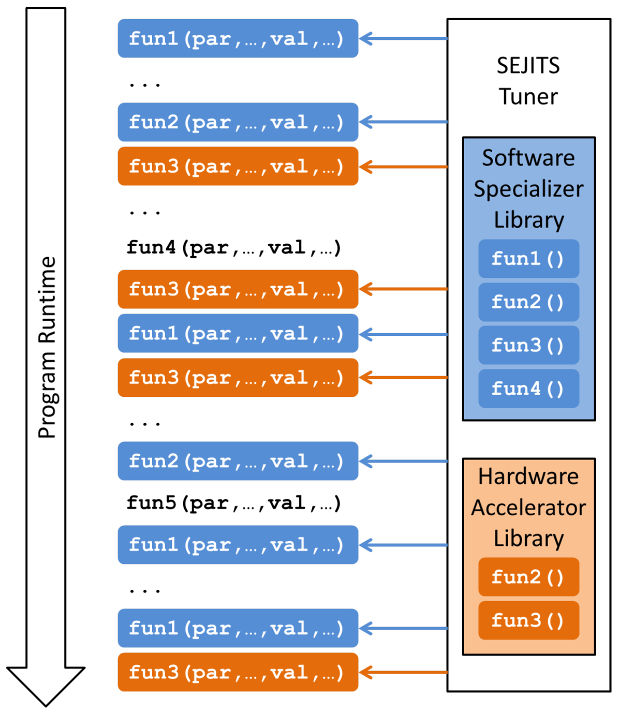

Fig. 1. Throughout the runtime of an application the same functions are called an arbitrary number of times. Software specializers (orange) may be used whenever they would improve performance. Only one type of hardware accelerator (blue) may be used during the entire execution of an application, even if unused accelerators would be an improvement over software alternatives.

II. Setup

A. Zynq SoC

The Zynq 7000 series SoCs manufactured by Xilinx incorporates an ARM Cortex A9 processor (called the Processing System, or PS) and FPGA fabric (called the Physical Logic, or PS) on a single chip [8]. The PS runs at 667 MHz and has 32KB L1 Instruction and Data caches as well as a 512KB L2 cache. The PL is equivalent to a standalone Atrix-7 FPGA with 85K logic cells, 560KB of block RAM, and 220 DSP slices. This SoC makes for a good candidate to perform software specializer/hardware accelerator co-tuning experiments because it has all of the requisite pieces of hardware.

Although it is able to run co-tuning experiments, the ARM is not capable of the computationally demanding task of synthesizing hardware. Therefore, a host computer with a desktop x86 processor is required to generate the accelerators in advance, as demonstrated in Fig. 2. This step is by no means a requirement for using software specializers and hardware accelerators together in an application. A desktop with an x86 processor could run both the software synthesis and application software with reconfigurable hardware attached via a daughter card communicating over PCIe. The IP used in this paper for communication between PS and PL is also available for communicating over PCIe so this modification would not even require changes to be made to application code or accelerator circuitry.

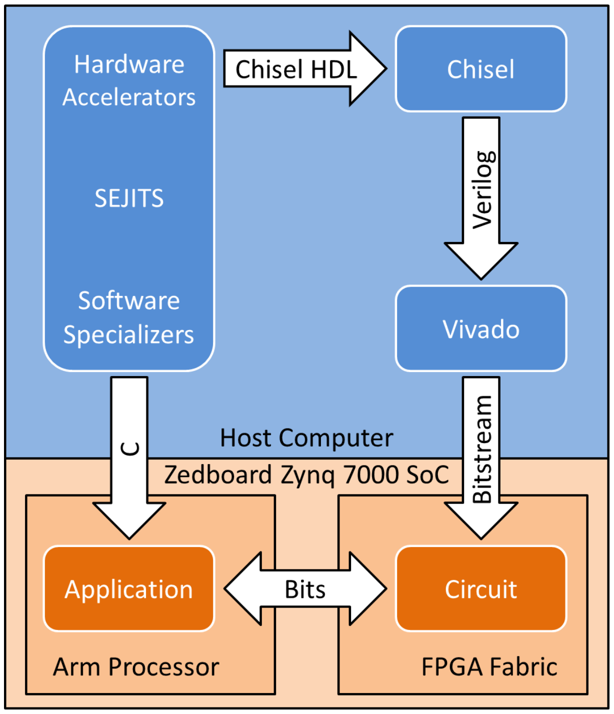

Fig. 2. The Zynq SoC (orange) contains the necessary components to run applications with software specializers and hardware accelerators. Synthesis of accelerator circuits is computationally intensive and requires a desktop x86 host computer (blue).

B. *Xillybus*

Xillybus is an IP core for FPGAs along with software drivers that is used for the communication of streaming data between the FPGA and a processor running Linux or Windows [9]. A version of this IP is available for Zynq SoCs which leverages the AXI interconnect to transmit data between PS and PL. On the processor side device files are made available to the programmer to read from and write to. A low level file I/O API is used to communicate with these device files in order to avoid unwanted buffering as shown in Fig. 3 and Fig. 4.

On the reconfigurable hardware side data streams are terminated with generic inbound and outbound FIFOs. One side of these FIFOs is connected to the Xillybus IP block and the other is connected to the user defined hardware accelerator circuitry (Fig. 5).

III. Experiments

A. *Convolutional Neural Nets*

CNNs are a machine learning technique that have been applied with great success to the task of classifying images [10]. In this setting, the CNN takes as input a 2D array of data values representing an image, and each internal layer sees a modified image derived from the previous layer. The output is a label classifying the image and its comparison with an expected value is used by backpropagation to update coefficients stored within the network. CNN layers are comprised of convolution layers which perform 2D convolutions on an image and pooling layers which downsample the images they receive. The net effect of these types of layers is that there are

```c
int write_xillybus(int fd, unsigned char *buf, int numbytes) {
    int rc = 0;
    int byteswritten = 0;
    while (byteswritten < numbytes) {
        rc = write(fd, buf+byteswritten, numbytes-byteswritten);
        if (rc < 0) {
            if (errno == EINTR){
                continue;
            } else {
                perror("write_xillybus failed to write");
                exit(1);
            }
        } else {
            byteswritten += rc;
        }

        if (rc == 0) {
            break;
        }
    }
    return byteswritten;
}
```

Fig. 3. C code for writing streaming data to device files which the Xillybus IP can read from.

```c
int read_xillybus(int fd, unsigned char *buf, int numbytes) {
    int rc = 0;
    int bytesread = 0;
    while (bytesread < numbytes) {
        rc = read(fd, buf+bytesread, numbytes-bytesread);
        if (rc < 0) {
            if (errno == EINTR) {
                continue;
            } else {
                perror("read_xillybus() failed to read");
                exit(1);
            }
        } else {
            bytesread += rc;
        }

        if (rc == 0) {
            break;
        }
    }
    return bytesread;
}
```

Fig. 4. C code for reading streaming data from device files which the Xillybus IP can write to.

typically a greater number of convolutions being performed at deeper stages, acting on smaller images and with smaller convolution windows.

CNNs are a good application to experiment with co-tuning because they contain a multitude of operations which have the same logical functionality but are parameterized differently. Furthermore, convolutions are conducive to being computed as software specializers or hardware accelerators. For the experiments performed in this paper a model was devised which captures the salient features of a CNN's convolutions while omitting other operations. Starting from a 9 by 9 convolution, the window size decays each new layer to 7 by 7, 5 by 5, and finally 3 by 3. The image size either decays by a power of two each layer or else remains constant, representing both extremes of pooling. Finally, deeper layers have more convolutions which in encoded by a branching factor β. Specifically, a layer L where the index starts at 0 would perform β times as many convolutions as the previous layer for a total of β^L convolutions at that layer. It is important to note that because other operations are missing which would occur in a full CNN, absolute changes in runtime due to the use of specializers and accelerators would be achieved by co-tuning the full CNN but

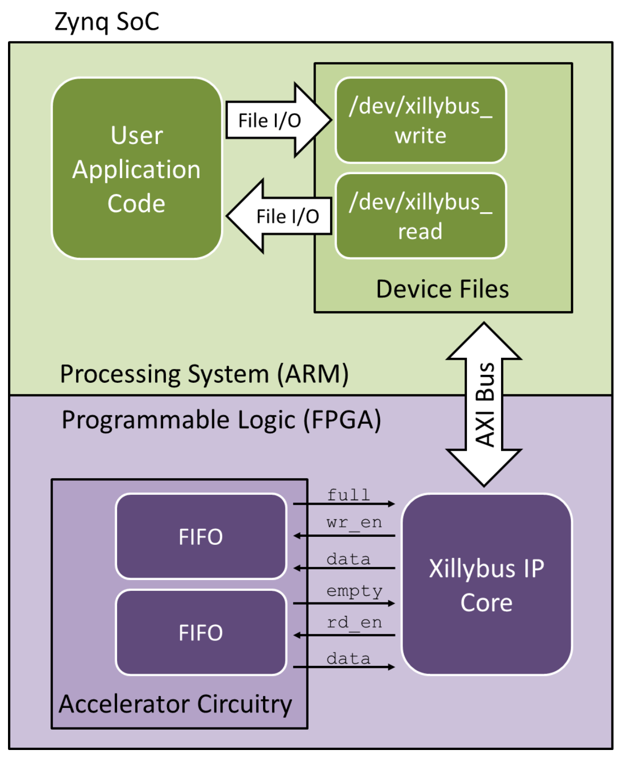

Fig. 5. Streaming data is sent from and received by user applications running on the ARM processor (green) to device files with low level file I/O API calls. This data is transmitted across to the programmable logic (purple) over the AXI bus. An Xillybus IP core resides in the reconfigurable hardware and exposes the data streams to accelerators as generic FIFOs.

relative changes need not be the same. Relative changes would necessarily be smaller for a full CNN unless the additional operations are also specialized and accelerated.

B. Streaming 2D Stencils

The 2D convolution operation required by the CNN can be realized as a stencil expressed in hardware on an FPGA. For the setup of hardware accelerators outlined previously, data enters and exits as a 1 element wide stream of 32 bit floating point numbers. The stencil operation is parameterized by the image size I and window size W which represent the number of elements along one axis of the square image and window, respectively. The first portion of the 2D streaming stencil in hardware introduces the appropriate delays so that all of the elements within a W by W box are made available on the same cycle. This is achieved by W rows of shift registers, nominally of length I (except for the last row), with specific intermediate values tapped and sent to the next segment (Fig. 6). The W^2 image elements received every cycle are then multiplied with the proper coefficient and the products are added together until a final result is generated to stream out.

The use of floating point units within the 2D stencil exposes a dilemma brought about by the assumption that software specializers and hardware accelerations need to be interchangeable with respect to functionality. Floating point units require more hardware resources than their fixed point counterparts, leading many past implementations of neural networks in hardware to employ the latter for data representation [11].

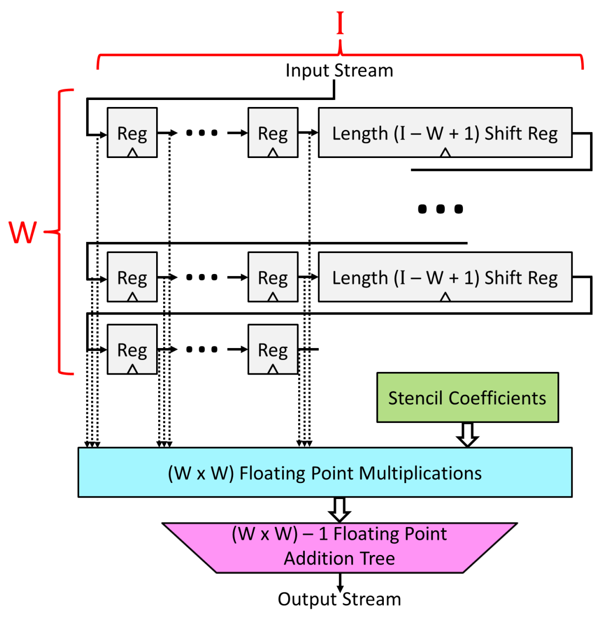
I
Input Stream
W
Reg
Reg
Length (I – W + 1) Shift Reg
Reg
Reg
Length (I – W + 1) Shift Reg
Reg
Reg
Stencil Coefficients
(W x W) Floating Point Multiplications
(W x W) – 1 Floating Point Addition Tree
Output Stream

Fig. 6. The hardware accelerator for a 2D stencil passes input streaming data through W rows of shift registers which are nominally of length I (gray). Specific intermediate values are tapped and sent to W^2 parallel multipliers (blue) which also take in the stencil coefficients (green). These products are added together with $W^2 - 1$ addition units (pink) and the result is streamed out.

The reduction in numerical precision brought about by a fixed point representation could be compensated for by changing the structure of the network to achieve comparable accuracy with an implementation that employs floating point units. For the co-tuning approach presented above, there is no mechanism available to modify an application's structure to accommodate variations in functionality between software accelerators and hardware specializers. Furthermore, such a mechanism would probably be highly application specific and consequently resistant to general automatic co-tuners. Therefore, floating point units were used in the hardware accelerators of the following experiments to maintain functional consistency with their respective software specializers.

IV. Results

For the following results, each data point represents the average across 10 trials unless otherwise noted and all times are in milliseconds. Absolute changes are calculated by $new - old$ and relative changes are calculated by $\frac{new-old}{old} \times 100\%$.

A. Standalone Convolutions

[ms]	Software Convolutions				Hardware Convolutions			
	W=3	W=5	W=7	W=9	W=3	W=5	W=7	W=9
I=32	0.418	0.968	1.631	2.262	0.241	0.238	0.186	0.232
I=64	1.753	4.367	8.025	12.17	0.295	0.291	0.288	0.288
I=128	7.200	18.60	35.38	55.75	0.558	0.560	0.518	0.573
I=256	29.21	76.65	148.4	238.3	1.768	1.768	1.766	1.792
I=512	118.3	313.4	603.7	990.7	5.688	5.602	5.618	5.614
I=1024	475.2	1285	2512	4280	22.05	22.02	22.13	22.05
I=2048	1962	5816	11399	18658	87.49	87.57	87.50	91.27

TABLE I. Time in [ms] to perform software and hardware convolutions across various window and image sizes.

The number of output image elements for which a stencil computation must be performed scales quadratically with the image size dimension I. As expected, a quadratic dependence was found in the standalone convolution data (Table I.) for both software and hardware convolutions. For each image element in the output, the number of floating point operations required scales quadratically with the window size W. Again this dependence can be found in the standalone convolution data for software convolutions. For the hardware convolutions all of the floating point operations can be mapped to distinct physical units which execute concurrently (provided there are sufficient hardware resources as is the case for the window sizes explored). This explains why there is no prominent quadratic increase in runtime for the hardware convolutions like was seen in software.

There are, however, theoretical reasons to suspect a modest increase in runtime given an increase in window size for hardware convolutions. Prior to streaming the image data, stencil coefficients must be sent to hardware. The amount of coefficients scales quadratically with the window size but the total number is nevertheless significantly smaller than amount of image data to be sent and received. Additionally, larger window sizes increase the number of shift register rows and the depth of the floating point addition tree. This deeper pipeline does not change the throughput but contributes a constant additional latency between streaming input and output data. Regardless, no definitive trend was observed that correlates window size to runtime for the standalone hardware convolutions.

There is a confounding factor introduced by the buffering of data streams sent through Xillybus which could introduce a variability in performance greater that the previously mentioned window size effects. Attempts were made to minimize this effect by using the low level API for file I/O to avoid an extra layer of buffering (e.g. `write()` instead of `fwrite()`). Also, a call to write with a zero length buffer, `write(fd, NULL, 0)`, is not well defined in general but Xillybus interprets it as an explicit flush and these instructions were placed after writing the coefficients and image data [12].

[ms]	Absolute Change from SW to HW				Relative Change from SW to HW			
	W=3	W=5	W=7	W=9	W=3	W=5	W=7	W=9
I=32	-0.177	-0.730	-1.445	-2.029	-42.29%	-75.4%	-88.6%	-89.7%
I=64	-1.458	-4.075	-7.738	-11.89	-83.18%	-93.3%	-96.4%	-97.6%
I=128	-6.641	-18.04	-34.86	-55.18	-92.24%	-97.0%	-98.5%	-99.0%
I=256	-27.44	-74.89	-146.7	-236.5	-93.95%	-97.7%	-98.8%	-99.3%
I=512	-112.6	-307.8	-598.1	-985.1	-95.19%	-98.2%	-99.1%	-99.4%
I=1024	-453.1	-1263	-2490	-4258	-95.36%	-98.3%	-99.1%	-99.5%
I=2048	-1874	-5728	-11311	-18567	-95.54%	-98.5%	-99.2%	-99.5%

TABLE II. ABSOLUTE AND RELATIVE CHANGE IN THE TIME TO PERFORM A CONVOLUTION IN HARDWARE VERSUS SOFTWARE.

Dividing the runtime of the software convolutions by the total number of image elements computed, I^2, yields the amortized runtime of computing a single image element. Across all choices of W this amortized single element runtime increased monotonically with increased image size. It is necessarily the case that the same amortized runtime does not increase as rapidly with image size for hardware convolutions because they become proportionally faster than software convolutions as image size increases (Table II.).

Software convolutions can leverage the fact that fewer floating point operations need to be carried out on border elements (where the stencil window extends beyond the edge of the image) than non-border elements. In contrast, identical throughput is achieved for both border and non-border elements calculated in hardware convolutions. For larger image sizes, proportionally fewer image elements are border elements and so software convolutions enjoy less of a benefit in handling this special case.

The image data exists in memory as a one dimensional array of floats. When a convolution accesses elements from different rows, the spatial distance between these accesses is a function of image size. Also, depending on the order in which the resulting image elements are calculated, a larger image size could increase the temporal distance between accesses of the same element. Both of these effects will cause the software specializer to experience degraded cache performance as the image size increases and in turn increases the amortize runtime of computing a single element. Conversely, the shift registers of the hardware convolution act as a bespoke cache for the streaming array of image data. A window's worth (W^2) of image values and coefficients are present each cycle. Provided there are sufficient hardware resources to build the shift register (as is the case for the image sizes explored), only one access to higher layers of the memory hierarchy is required for each value. For larger image sizes, software convolutions face worse caching behavior than experienced by hardware convolutions.

B. CNN Convolutions

[ms]		CNN with a HW Convolution				Absolute	Relative
β	SW Only	W=9 I=512	W=7 I=256	W=5 I=128	W=3 I=64	Change from SW Only to **Best** Result	
1	1173	**175**	1037	1155	1189	-998	-85.1%
2	1389	**391**	1106	1316	1394	-998	-71.9%
3	1663	**665**	1234	1500	1639	-998	-60.0%
4	2005	**1007**	1430	1715	1925	-998	-49.8%
5	2425	**1428**	1704	1973	2253	-997	-41.1%
6	2935	**1938**	2067	2283	2625	-997	-34.0%
7	3543	2547	**2530**	2656	3040	-1013	-28.6%
8	4261	3266	**3102**	3103	3502	-1159	-27.2%
9	5099	4104	3794	**3634**	4011	-1465	-28.7%
10	6067	5074	4617	**4259**	4568	-1808	-29.8%
11	7176	6184	5581	**4988**	5168	-2188	-30.5%
12	8435	7445	6696	5833	**5823**	-2612	-31.0%
13	9856	8868	7973	6803	**6533**	-3323	-33.7%
14	11448	10463	9422	7908	**7296**	-4152	-36.3%
15	13222	12239	11053	9160	**8109**	-5113	-38.7%

TABLE III. TIME IN [MS] TO PERFORM THE CONVOLUTIONS OF A CNN WHEN THE IMAGE SIZE I DECREASES BY A FACTOR OF 2 AT EVERY LAYER. EACH LAYER CONTAINS β TIMES MORE CONVOLUTIONS THAN THE PREVIOUS LAYER. UP TO ONE TYPE OF CONVOLUTION IS REPLACED BY A HARDWARE ACCELERATOR.

It is significantly more advantageous to run convolutions with larger window sizes on hardware than in software. Therefore, for small branching factors β, the convolution chosen by tuning to be hardware accelerated should be the one with the largest window size. This effect is compounded by the fact that latter stages in a CNN typically operate on shrinking image sizes. As the branching factor increases, the sheer number of instances of smaller window and image sizes tilts the scale towards hardware accelerators of the smaller instance being optimal for hardware acceleration.

The results in Table III. demonstrate that hardware convolutions which attain an inferior performance improvement relative to a software alternative can ultimately be the optimal choice when looking at an application on the whole. This is not surprising as there are global interactions between the set of all functions that could be sent to hardware. Namely, selecting one to accelerate with hardware uses physical resources and blocks the other functions from being accelerated in the same manner. Software specializers do not interact in the way and so focusing on the performance of software specialized functions in isolation remains a valid approach.

[ms]		CNN with a HW Convolution				Absolute	Relative
β	SW Only	W=9 I=512	W=7 I=512	W=5 I=512	W=3 I=512	Change from SW Only to **Best** Result	
1	2042	**1042**	1453	1740	1939	-1000	-49.0%
2	4407	3405	3219	**3182**	3523	-1225	-27.8%
3	8806	7800	7016	6038	**5800**	-3006	-34.1%
4	15930	14927	13547	11013	**8807**	-7123	-44.7%
5	26496	25488	23516	18807	**12575**	-13921	-52.5%

TABLE IV. Time in [ms] to perform the convolutions of a CNN when the image size I remains constant. Each layer contains β times more convolutions than the previous layer. Up to one type of convolution is replaced by a hardware accelerator.

The choice of which convolution is optimal to represent in hardware is not exclusively a function of the branching factor β. Supposing no pooling was performed between convolution layers in a CNN, each convolution would have the same image size (Table IV.). The runtimes of convolutions with smaller window sizes are larger in proportion to ones with larger window sizes than was the case when the image size decreased as well. Accordingly, smaller convolutions are found to be the best candidate for hardware resources at lower branching factors, and the transition occurs more rapidly.

Although it is possible to spend time analyzing the reasons why the optimal hardware accelerator varies across different CNN structures, in reality the goal is to find a way to improve runtime performance for a specific structure. Fortunately, the CNN examples show that it is feasible to locally tune software specializers to speed up those operations in isolation and then globally tune over a whole application to determine which operations should be accelerated in hardware. Furthermore, a programmer who uses these techniques is not required to be knowledgeable of the specific trade-offs between the software and hardware techniques or how they change with the structure of the application.

V. Concluding Remarks

A. Embedded Hardware Accelerators

Along the border of an image, the stencil window extends beyond the edge and so the typical calculation for output image elements is ill-defined. Suppose the operation for border elements is defined so that all of the locations where the stencil window extends beyond the edge are ignored and the others are multiplied by their respective coefficient and summed as usual. In this case, convolutions with smaller window sizes could be embedded inside of convolutions over larger window sizes by centering the coefficients of the smaller convolution within the window of the larger one and padding the difference

[ms]	CNN with ≥ 1 HW Convolutions		Change from **Best** Single HW to All HW	
β	Single HW	All HW	Absolute	Relative
1	1042	22.26	-1019	-97.9%
2	3182	83.25	-3099	-97.4%
3	5800	222.4	-5577	-96.2%
4	8807	478.1	-8328	-94.6%
5	12575	880.2	-11695	-93.0%

TABLE V. Time in [ms] to perform the convolutions of a CNN when the image size I remains constant. The best times when one hardware accelerator is permitted are compared against CNN convolutions where all are performed with hardware accelerators.

with zeros (Fig. 7). If locally tuning over hardware specializers, there would be no reason to suspect that a smaller convolution embedded in a larger one would outperform a smaller convolution that was deliberately sized. Broadening the scope to the entire application, the utility of embedded convolutions changes dramatically because hardware accelerators with different window sizes no longer block each other by competing for the same hardware resource.

An auxiliary experiment was run to compare the performance of CNN convolutions that were all hardware accelerated against the best previous results where only one type was allowed to be hardware accelerated (Table V.). Due to Amdahl's law, when one type of convolution is accelerated with hardware the contribution of the other software convolutions become a greater proportion of the total running time. Because these were all replaced with hardware in this experiment, the relative change in runtime is almost as dramatic as the change in runtime between the standalone hardware and software convolutions.

B. Concurrent Operations

Sequential Execution of HW and SW Convolutions			[ms]	Change from Average to Mixed Convolutions	
SW/HW	HW/SW	Average	Mixed	Absolute	Relative
2.4840	2.4849	2.4844	2.4607	-0.0237	-0.96%

TABLE VI. Time in [ms] to perform both a software and a hardware convolution. Either the hardware convolution and software convolution are run sequentially or else the software convolution is performed in between reading and writing data for the hardware convolution.

Hardware accelerators offer another benefit over their software specializer counterparts. The addition of physically separate circuitry to accelerate computations allows for the possibility of concurrent work to be performed on the main processor. Tuning at an application level could take advantage of this benefit by opting to perform some instances of operations in software even when the type of operation has been selected for hardware acceleration. Unfortunately, for this application the hardware convolutions are significantly faster than the software convolutions and most of the time is spent transmitting data which ties up the main processor as well.

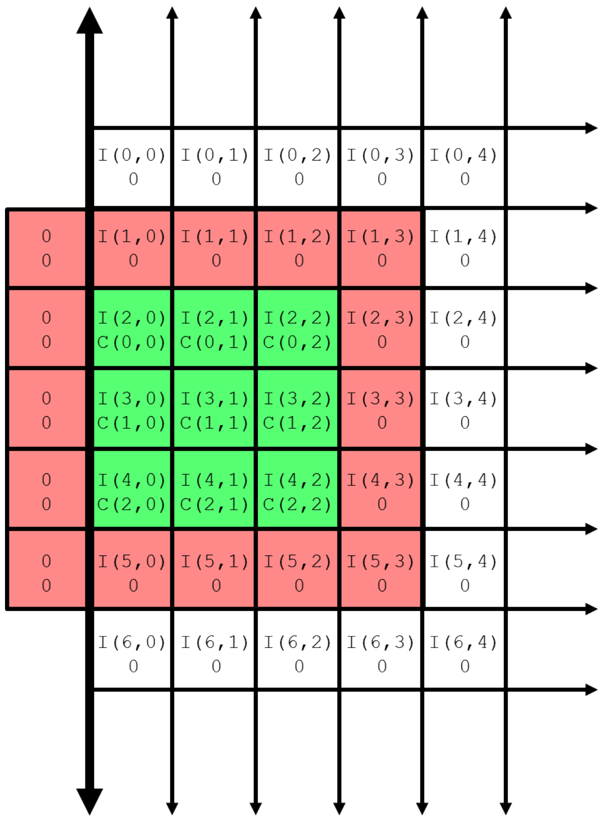

Fig. 7. A stencil with window size $W = 3$ (green) embedded in a stencil with larger window size $W = 5$ (red) extending beyond the left border of an image. If the border policy allows for the portion of the window which remain in the image to be summed, then padding the larger window with zeros yields the correct results. For brevity the row index of the image starts at 0 instead of an arbitrary n.

Therefore, selectively replacing some instances of convolutions that could be sent to the hardware by software convolutions is suboptimal in this instance but this need not be the case for different types of hardware accelerators.

Even so, the effects of concurrent work can still be demonstrated. An auxiliary experiment was performed to measure the runtime of two independent convolutions where one was performed in software and one in hardware. Either a software and a hardware convolution were issued sequentially, or else the two were mixed by calling a software convolution after the writing of data to hardware convolution but prior to reading back the results (Table VI.). Because the effect is small in this case and there is variability introduced by buffering the hardware convolution streams, 10,000 trials were performed.

Capitalizing on this effect presents many challenges for the co-tuner model proposed. Previously, the co-tuner only had to consider which type of convolution is best suited for hardware which is a modest enough space that exhaustive tuning is practical. Now the space explodes to every permutation of software or hardware for every instance of a given type of hardware convolution and a new tuning approach would be required. Heuristics could be applied in order to only consider changing hardware convolution instances to software when there is a good chance that concurrent processing is worthwhile, but this undermines the objective that a user need not have explicit knowledge of the trade-offs between the software specializer and hardware accelerator. Another difficulty is that the call to the hardware accelerator must be split up in a rational way to insert the concurrent software operations. In the above example this was done by hand but preferably it would be done automatically. Fortunately, there is a natural partition between the streaming write of data to the hardware accelerator and the streaming read of results back and this can serve as a general insertion point for concurrent software operations.

C. Alternative Reconfigurable Hardware

One of the fundamental assumptions about hardware accelerators which shaped the decisions about how to apply them to an application is that only one configuration can be resident in hardware at a given point in time. Extending this to the statement that only one configuration can be used during the runtime of an application makes the implicit assumption that the time to reconfigure the hardware is prohibitively large relative to the runtime of the application. Academic efforts have been made to explore reconfigurable accelerator architectures that are quicker to reconfigure on the fly. One example is the Garp coprocessor [13], which is built from an array of functional elements that communicate over a regular array of predefined interconnects. This allows for less data to be used in encoding the circuit, resulting in less time to transmit the configuration. It also allows for cached configurations to reside in situ which can be switched to in only a few cycles. Fast context switching that takes negligible time breaks the global interaction between different types of hardware accelerators in a given application. The best choice of acceleration for a specific operation could again be found by locally tuning over the appropriate software and hardware specializer. In between, when switching configurations takes considerable time but is still permissible, exhaustive tuning over different types of configurations becomes the more challenging problem of scheduling the use of scarce hardware resources.

For the setup outlined in this paper, applications are run on an ARM processor which is incapable of synthesizing the required hardware accelerators that had to be synthesized beforehand. Even if the main processor was more capable, hardware synthesis is a computationally intensive process that cannot currently be completed anywhere near as fast as the tree manipulations and c compilation performed by just-in-time software specializers. There have been academic efforts to decrease the time to synthesize hardware designs which could perhaps allow hardware accelerators to also be generated just-in-time instead of beforehand. Increasingly, the time required to synthesize a hardware design is dominated by routing. One proposed solution is to leverage the network structure of the target hardware itself to assist in distributed routing algorithms by augmenting the switches with additional hardware [14]. While an application is running, generating new configurations would prevent the reconfigurable hardware from being used elsewhere but this is entirely analogous to main processor cycles being spent to generate software specializers on the fly.

REFERENCES

[1] R. H. Dennard, V. Rideout, E. Bassous, and A. LeBlanc, "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, 1974.

[2] W. Haensch, E. J. Nowak, R. H. Dennard, P. M. Solomon, A. Bryant, O. H. Dokumaci, A. Kumar, X. Wang, J. B. Johnson, and M. V. Fischetti, "Silicon CMOS Devices Beyond Scaling," *IBM Journal of Research and Development*, vol. 50, no. 4.5, pp. 339–361, 2006.

[3] D. Luebke, "CUDA: Scalable Parallel Programming for High-Performance Scientific Computing," in *5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. IEEE, 2008, pp. 836–838.

[4] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010.

[5] B. Catanzaro, S. A. Kamil, Y. Lee, K. Asanovi, J. Demmel, K. Keutzer, J. Shalf, K. A. Yelick, and A. Fox, "SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-23, Mar 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-23.html

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1216–1225. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228584

[7] K. Asanovi, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[8] Xilinx. (2016, Jan.) Zynq-7000 All Programmable SoC First Generation Architecture. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[9] E. Billauer. (2016, Jan.) Xillybus Product Brief. [Online]. Available: http://xillybus.com/downloads/xillybus_product_brief.pdf

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[11] J. Misra and I. Saha, "Artificial Neural Networks in Hardware: A Survey of Two Decades of Progress," *Neurocomputing*, vol. 74, no. 1-3, pp. 239–255, Dec. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.neucom.2010.03.021

[12] E. Billauer. (2014, Jul.) Xillybus Host Application Programming Guide for Linux. [Online]. Available: http://xillybus.com/downloads/xillybus_product_brief.pdf

[13] J. R. Hauser, "Augmenting a Microprocessor with Reconfigurable Hardware," Ph.D. dissertation, University of California, Berkeley, Berkeley, CA, USA, 2000.

[14] R. R.-F. Huang, "Hardware-Assisted Fast Routing for Runtime Reconfigurable Computing," Ph.D. dissertation, University of California, Berkeley, Berkeley, CA, USA, 2004.