Hardware Convolution

Project Number: 17-1-2-1551

Project Report

by
Simcha Herschman ID: 938936036
and
Kobi Rappaport ID: 960284891

supervised by
Konstantin Berestizshevsky

Tel Aviv University
Tel Aviv, Israel

October 23, 2018

Abstract

There are many contemporary computer applications which could benefit from high-throughput hardware specially-designed for convolution calculations, including image processing and machine learning. The goal of this project is to develop a Field Programmable Gate Array (FPGA) for small scale applications where a cost-effective but efficient solution is needed.

MATLAB was used in order to investigate the performance of the convolutions on a Central Processing Unit (CPU), both with traditional execution and multi-threaded calculation. Then, after collecting data from MATLAB, a design was created for the FPGA using High Level Synthesis (HLS). The design was then loaded onto the FPGA and its performance was examined. The effects of image and kernel size and depth on algorithm runtime and power consumption were observed as part of this study.

Despite the promise of the approach, the FPGA developed as part of this project was not able to calculate convolutions more quickly than the CPU used as a benchmark. However, an FPGA similar to the one developed for this project may be a viable option where power envelope, price, and size are critical design configurations.

List of Abbreviations

2D Two-Dimensional.

3D Three-Dimensional.

ASIC Application-Specific Integrated Circuit.

CPU Central Processing Unit.

ELF Executable and Linkable Format.

FIFO First In First Out.

FPGA Field Programmable Gate Array.

FSM Finite State Machine.

GPU Graphical Processing Unit.

GUI Graphical User Interface.

HLS High Level Synthesis.

OOM out of memory.

RAM Random Access Memory.

RAPL Running Average Power Limit.

RISC Reduced Instruction Set Computing.

RTL Register Transfer Level.

SDK Software Development Kit.

SoC System on Chip.

Contents

List of Figures

Introduction

Machine learning is one of the fastest growing fields in technology today. Almost everybodys daily experience in enhanced by some interaction they have with a platform utilizing machine learning.

Machine learning typically involves the performance of a large number of convolutions. When it comes to image processing, it requires multiple two-dimensional convolutions per image. While there is a range of scales in which this process is applied, from smaller systems in an academic setting for research related to machine learning or in embedded systems to larger scale applications to Big Data, the processing needs are extremely high, and low latency and power consumption is a must.

While conventional CPUs are great for general computational needs, they fall short when it comes to operations such as image processing. The main reason for this shortcoming is the low throughput of the CPU which offers a low and non-ideal rate of operations and computations, restricting the speed of convolution which is simply a large mass of multiplications and additions. Furthermore, the CPU's cache size is likely too small to store all of the needed data,[1] thus requiring loading and storing from memory throughout the convolution; additional operations which are another large consumer of time and power.

The poor performance of CPUs lead to the use of graphical processing units (GPUs) to try to solve the problems. While GPUs have helped increase speed and power consumption due to their large throughput, they still have the problem of not being designed specifically to perform the operations needed for large scale convolution. Also the high price and spatial needs of GPUs are major obstacles for their use in the academic and embedded fields. Although this might not pose the same problems for large companies with many server farms, it can still be seen that hardware designed specifically for the task is of importance as it will significantly improve performance.

The project is to design hardware that will implement a type of Two-Dimensional (2D) convolution for image processing. While a CPU will be sufficient to implement such convolutions for small scale processing, for example touching up pictures from a personal collection, when dealing with large scale like machine learning, specialized hardware becomes necessary in order to make the process practical. Both machine learning and image processing are beyond the scope of this project; however, they provide the motivation for this project. The focus of the project is purely to implement the convolution and not to filter or classify actual images. The use of "image" and "kernel" throughout this report refer to what would be the image and filter, respectively, when used for image processing. The actual matrices used for testing were randomly generated.

We began by using MATLAB in order to investigate the performance of the convolutions on a CPU. After collecting data from MATLAB, we created a design for the FPGA using HLS. The design was then loaded onto the FPGA and its performance was examined. All of the code that we wrote and used for this project can be found in a GitHub repository located:
`https://github.com/KRappaport/Hardware-Convolution`.

[1] Between the actual image, the kernel, the results of the convolution (which depends on the size of the image), and other data needed throughout the process, the memory needs add up quickly.

The convolution performed for image processing is an operation on two matrices,[2] the image generally being the larger matrix and the kernel, which is the filter that is applied to the image, generally being a square matrix with odd dimensions and smaller than the image matrix. The convolution can be calculated by flipping the kernel horizontally and vertically, moving the center element of the kernel along each element of the image, and multiplying and summing the overlapping elements. The element of the image that is at the center of the kernel corresponds to the element of the resultant matrix. This can be represented by the equation

$$R[i_1, i_2] = \sum_{k_1} \sum_{k_2} I[k_1, k_2] \cdot K[c + i_1 - k_1, c + i_2 - k_2] \tag{1}$$

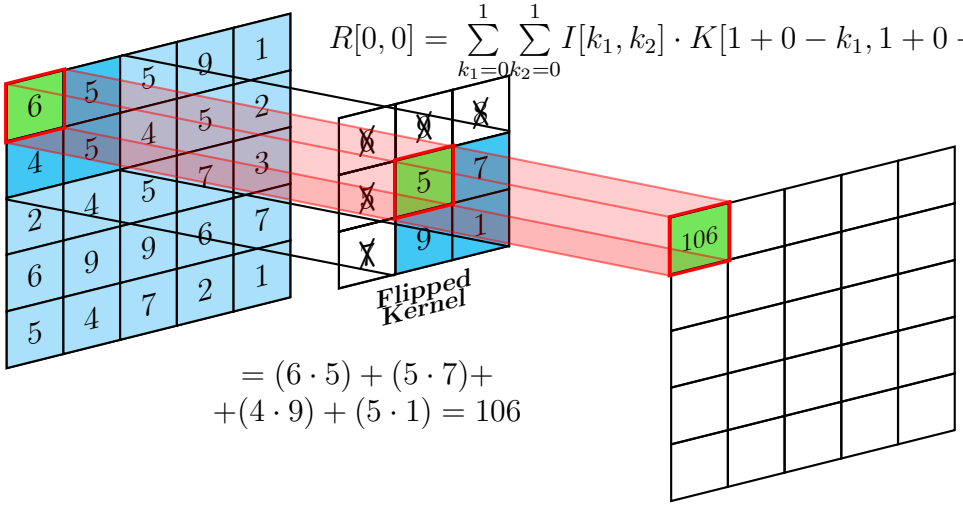
where c is the center element of the kernel and k_1 and k_2 are the integer indices of the image.[3] A visualization of this process can be seen in Figure 1. Figure 1a is the image and kernel that are convolved and the result of the convolution. In Figures 1b and 1c, the kernel has been flipped and aligned with a specific pixel of the image. Figure 1b shows the the center of the kernel set on the first pixel of the image to obtain the first pixel of the result. This is a border case as the only part of the kernel is used since part of the kernel is outside of the region of the image. Figure 1c can be considered a center case (the first one when moving the kernel along the image sequentially) since the entire kernel is within the bounds of the image and thus each element of the kernel contributes to the result for that pixel.

[2]The image and kernel will often be a set of 2D matrices with the same number of image and kernel matrices, and the convolution will only occur between corresponding matrices. The results of each convolution will be summed together providing a single 2D matrix as its final result.

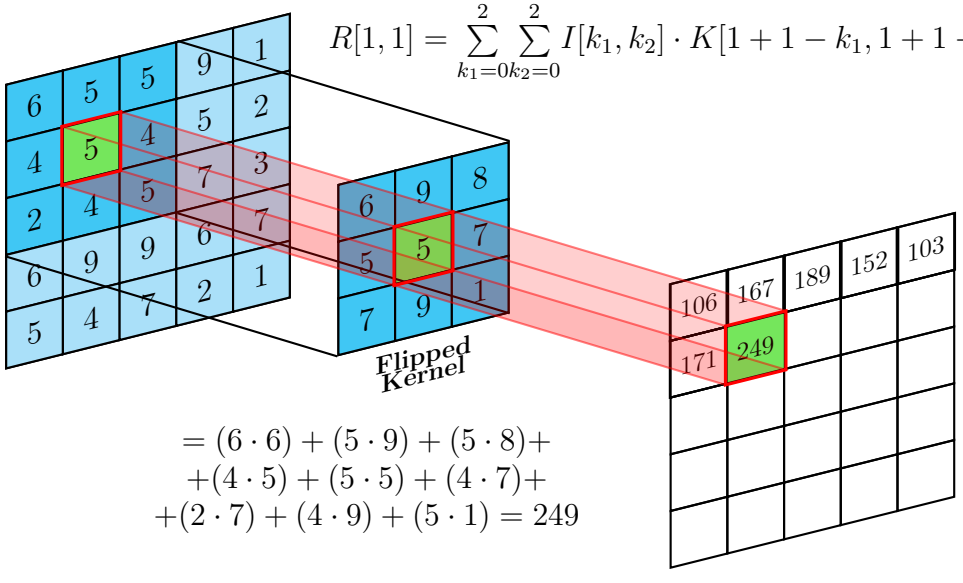
[3]The bounds of the summations will be updated as needed, in order to avoid the situation in which $c + i_1 - k_1$ or $c + i_2 - k_2$ provides an index that is beyond the dimensions of the kernel.

Image						Kernel				Result				
6	5	5	9	1		1	9	7		106	167	189	152	103
4	5	4	5	2		7	5	5		171	249	307	279	174
2	4	5	7	3	*	8	9	6	=	177	298	350	290	203
6	9	9	6	7						192	304	350	309	157
5	4	7	2	1						179	283	252	216	114

(a) Full Convolution



(b) Edge Case



(c) Center Case

Figure 1: 2D Convolution for Image Processing

Part I
Phase A (MATLAB)

Convolution in MATALB

The first phase of the project was to perform the convolutions on a CPU. This step served a dual purpose: first, allowing us to become familiar with the necessary operations to write an algorithm that can successfully perform the convolution, and second to collect data on the time it takes to perform the convolution on a CPU. For this process we used MATLAB.

MATLAB was chosen since it would allow us to perform all the tasks in this phase in a single environment. Using MATLAB gave us access to their well-developed functions to perform the convolutions and obtain timing data for a well-designed and efficient function. MATLAB also allowed for us to write our own functions to perform the convolutions. Although writing the code in C[4] would probably have led to better functions than what we could implement in MATLAB, it would also have required a larger investment of time to incorporate memory management etc. Because MATLAB's well-developed functions were available, this project was able to focus on the algorithm which would be used in subsequent stages of this project. MATLAB was used to collect the timing data for the MATLAB-provided convolution function as well as our own simpler and naive implementations. MATLAB was also used to output graphs[5] to analyze the results.

Proper testing required using a data set[6] which provided a good range of sizes of the image and kernel.[7] This meant that at times a large amount of memory would be needed and that the whole process would take a significant amount of time. Using personal computers[8] would not be practical for this step and so the University's server[9] was used instead. Using the University's server also meant that there were more cores available for multi-core testing of our functions.

The timing was tested for three different criteria: the time the convolution took as the image size increased, kernel size increased, and the depth increased. While the primary focus was on the time it took MATLAB's own function to perform the convolutions, for each of the three criteria MATLAB's time was compared to the time it took the simpler functions. The time-saving effect of distributing the processing of the

[4]The C language was designed by Dennis Ritchie at Bell Labs and was released in 1972. Although it is a relatively old language, it is still one of the most powerful languages and is used for writing operating systems, programs in embedded systems, and is one of the languages used to write MATLAB. See `https://en.wikipedia.org/wiki/C_(programming_language)` for more.

[5]Another downside to using C would be that we would need to export the timing data either to MATLAB or Excel in order to generate the graphs.

[6]The code used for creating and reading the data sets can be found `https://github.com/KRappaport/Hardware-Convolution/tree/master/misc`.

[7]Each size also had to be tested multiple times so that an averaged time could be obtained.

[8]Although personal computers would have enough memory, it would need a large majority of the available memory. More importantly, the runtime to receive a good averaged result could take up to a day which would monopolize the researchers personal computers for an impractically long period.

[9]See Appendix C.1 for some of the technical specifications of the server.

custom function among multiple cores was also checked. For the multiple core testing, the workload was distributed among 18 cores.

There were two convolution functions that were written in MATLAB. The first function, "conv_2d,"[10] uses a simplistic approach in which many nested for loops are used to pick the proper indices for the multiplications and additions, and similar to the description of the convolution process provided in the Introduction, the bounds of the for loop are updated as needed to make sure that the indices used refer to a valid element of the kernel. The other function, "conv2d,"[11] takes a slightly more complex approach in which the image is first padded with zeros,[12] then the kernel is flipped, and then using a few for loops a section the size of the kernel is taken from the padded image and goes through element-wise matrix multiplication with the flipped kernel and is then summed.[13] There are multi-core versions of both functions[14] which were written by replacing "for" of the outer for loop used for the convolution to "parfor".[15]

For a method that is built into MATLAB to do the convolution, MATLAB's neural network toolbox was used.[16] The first step in this process is to create the neural network, for which the function "matlab_NN_create" was used.[17] Once a neural network has been created, the needed convolution can easily be performed using:

```
net.predict(image) - net.Layers(2).Bias;
```

where "image" is a Three-Dimensional (3D) matrix of the size specified when creating the neural network.[18]

[10]The "conv_2d" function can be found at `https://github.com/KRappaport/Hardware-Convolution/blob/master/MATLAB_Code/conv_2d.m`.

[11]"conv2d" can be found at `https://github.com/KRappaport/Hardware-Convolution/blob/master/MATLAB_Code/conv2d.m`.

[12]The amount of padding needed on each side of the image can be determined by right bit-shifting the kernel dimension.

[13]Here the zero padding allows the entire kernel to be used each time by solving the boundary issue by adding in zero multiplication.

[14]The multi-core versions of "conv_2d" and "conv2d" are "conv_2d_mult_core" and "conv2d_mult_core," respectively. The code can be found at `https://github.com/KRappaport/Hardware-Convolution/blob/master/MATLAB_Code/conv_2d_mult_core.m` and `https://github.com/KRappaport/Hardware-Convolution/blob/master/MATLAB_Code/conv2d_mult_core.m` respectively.

[15]This is a multi-core version of a for loop in MATLAB which divides each iteration of the for loop among the different cores available in the parallel pool. See `https://www.mathworks.com/help/distcomp/parfor.html` for more details.

[16]While other standard MATLAB functions exist that can do the type of 2D convolution needed, there was no function that could take a set of 2D matrices, do the convolution between each pair of image and kernel from the set, and then sum the convolution results into a single 2D matrix. The neural network toolbox provided this functionality.

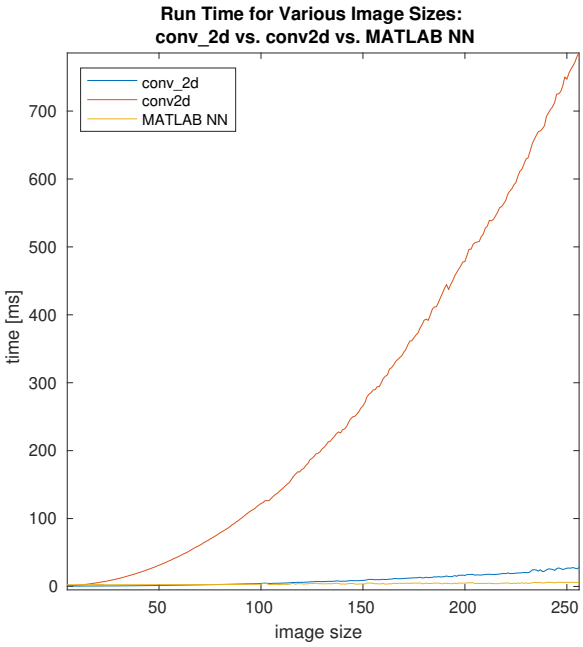
[17]This function only creates a neural network which can then be used to do the convolution, but it itself does not perform the convolution and is thus not included in the timing. See `https://github.com/KRappaport/Hardware-Convolution/blob/master/MATLAB_Code/matlab_NN_create.m` for the code.

[18]One does not provide a kernel since the kernel is created by MATLAB when the neural network is created.

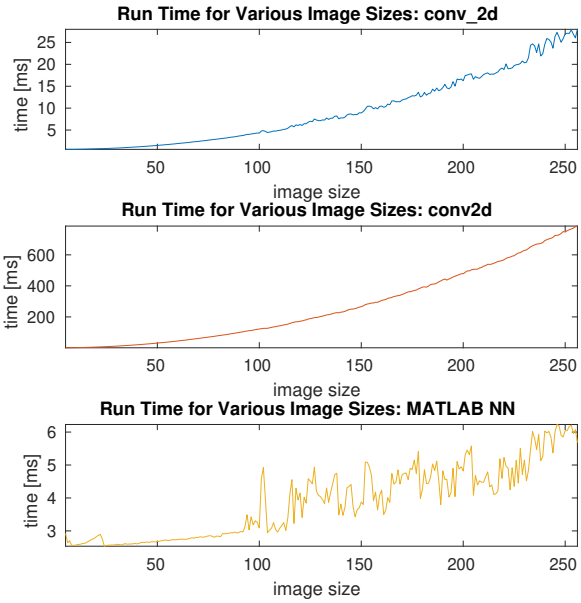
1 Increasing Image Size

The first test was to see how the processing time to do the convolution increases as the size of the image increases. For this test, images of square dimensions from a size of 5 to 256 and a depth of 3 were used. The kernels, also of square dimensions, were kept to a size of 3. There were 10 randomly generated images for each size and a different kernel for each image size.

1.1 MATLAB Compared to the Custom Functions[19]



(a)



(b)

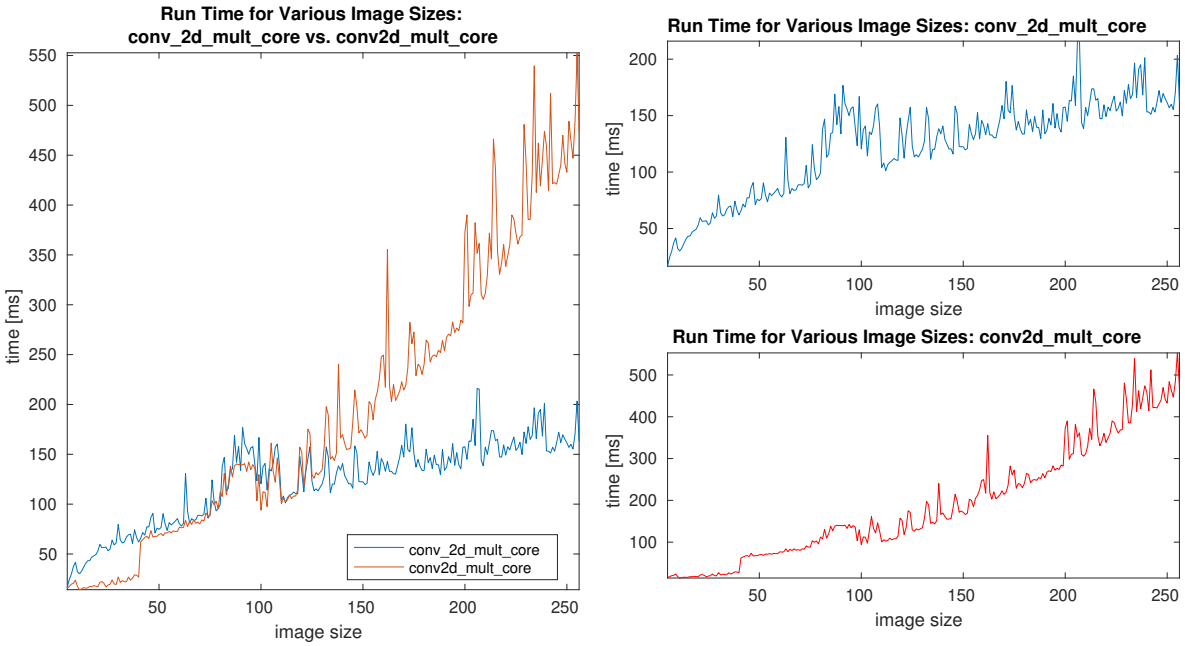
Figure 2: Runtime for Increasing Image Size: CPU

Figure 2a shows the significant time differences between the different functions and MATLAB's functions. It is clear from Figure 2a that the "conv2d" function takes around 100 times longer than the runtime of MATLAB's function and around 30 times as long as the "conv_2d" function.

Additionally, both the "conv_2d" and the "conv2d" function runtimes grow at an approximately quadratic rate, and although MATLAB's function has a much slower and not smooth growth trend, it too would appear to have a quadratic growth rate.

[19]The MATLAB script used for this test can be found at `https://github.com/KRappaport/Hardware-Convolution/blob/master/MATLAB_Code/time_cmpr_var_img_conv2.m`.

1.2 Multi-Core[20]

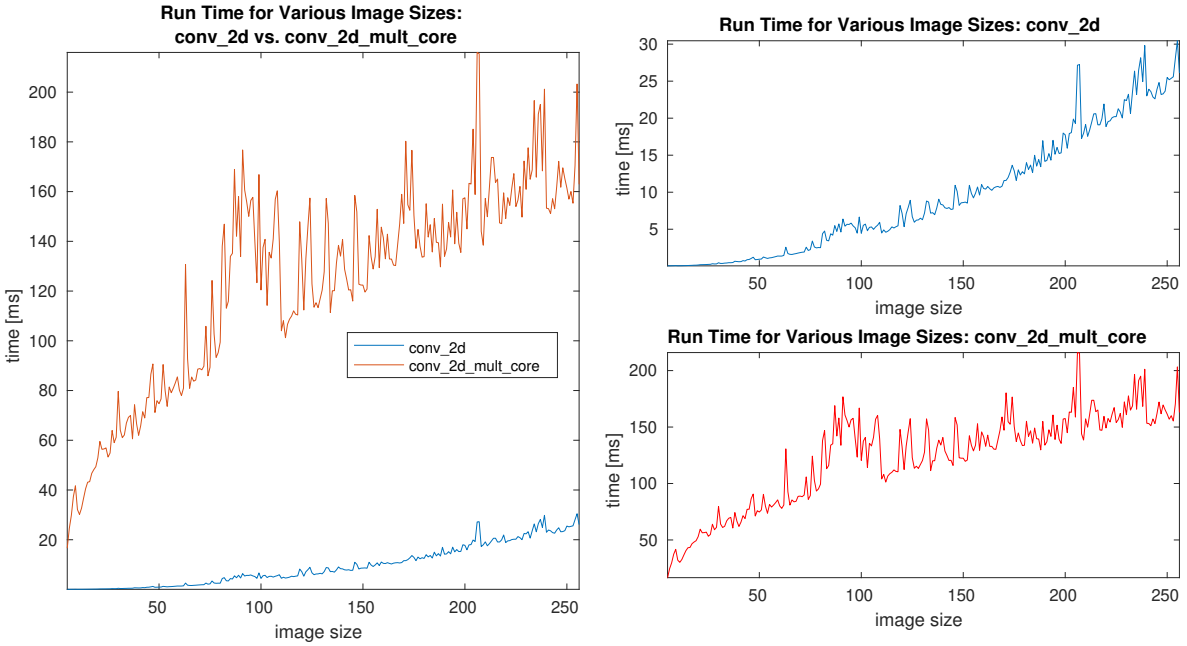


(a) (b)

Figure 3: Runtime for Increasing Image Size: Multi-Core

From Figure 3, two main observations are made. First, the ratio of the difference in runtime between the "conv2d" and the "conv_2d" has decreased. The ratio in the extreme case is now about 3.15:1 (conv_2d to conv2d) , whereas without the multi-core processing the ratio was about 30:1. The multi-core processing improved the runtime of the "conv2d" function, while it only improved the growth rate of the "conv_2d" function. Second, one can see that the growth rate of the conv_2d function is logarithmic, while without multi-core processing (Figure 4) it was quadratic. The "conv2d" function still has a quadratic trend.

[20]The MATLAB script used for this test can be found at `https://github.com/KRappaport/Hardware-Convolution/blob/master/MATLAB_Code/mcore_time_cmpr_var_img_conv2.m`.

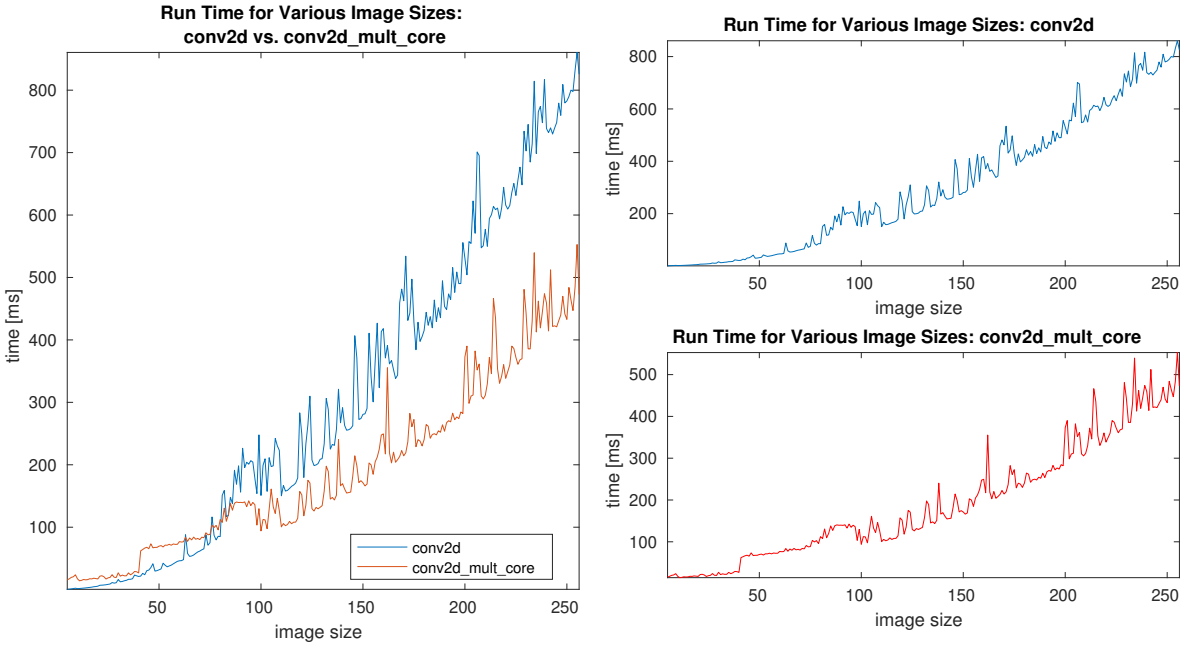


(a) (b)

Figure 4: conv_2d Runtime for Increasing Image Size: Single-Core vs. Multi-Core

Figure 4 leads to two main observations. First, the multi-core processing increased the runtime. This unexpected result may be caused by additional communication needed by the multiple cores and the repeated sending of information. The second is, as mentioned before (see Figure 3), that the multi-core processing of "conv_2d" has a logarithmic trend whereas the single-core processing of "conv_2d" has a quadratic trend.[21]

[21]For the image range used in testing the multi-core version of conv_2d the runtime is longer, however, it is likely due to the logarithmic growth rate that as the image size increases it will have better time.



(a) (b)

Figure 5: conv2d Runtime for Increasing Image Size: Single-Core vs. Multi-Core

Figure 5 shows that the multi-core processing of the "conv2d" function improved the runtime of the function in the most extreme case by 37.5%, from around 850 ms to around 500 ms. It also can be seen that both versions have a quadratic trend, however, with a slower growth rate for the multi-core version.

2 Increasing Kernel Size

The change in time for the convolution as the kernel dimensions increased was also tested. The images had a square dimension of 32 and a depth of 3. The kernels used were square matrices with dimensions at 3, 5, 7, and 9. For each kernel size, there were 10 randomly generated kernels with a corresponding image also random.

2.1 MATLAB Compared to the Custom Functions[22]

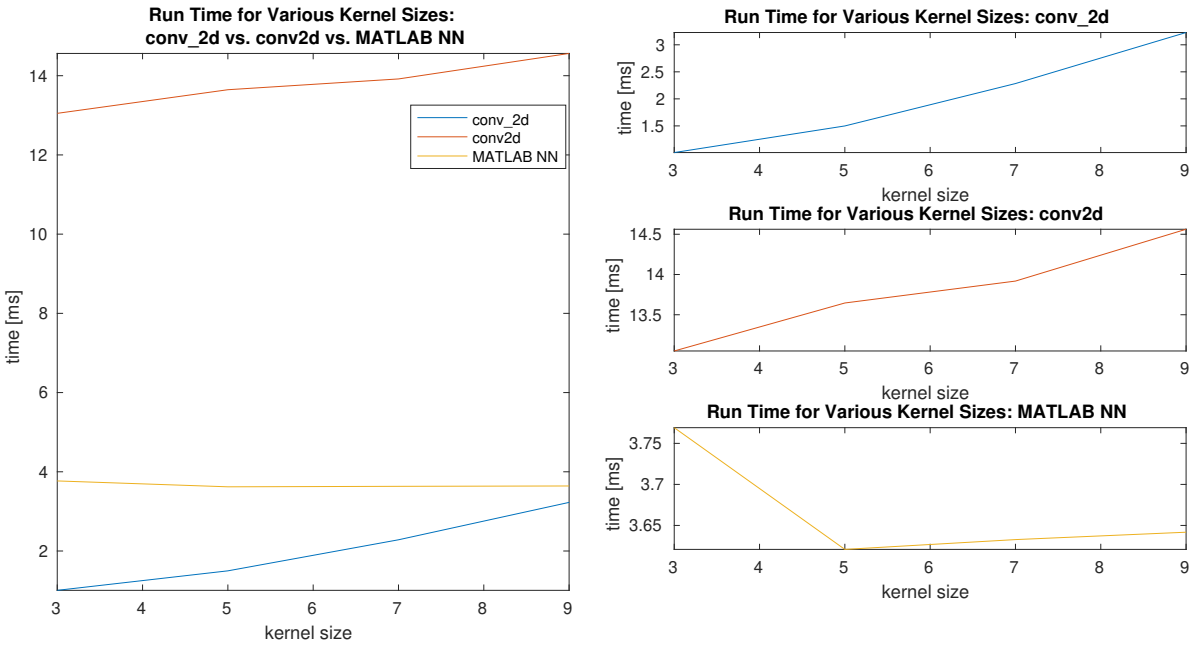


Figure 6: Runtime for Increasing Kernel Size: CPU

Figure 6 shows how the "conv_2d" function has better runtime than both the "conv2d" and the MATLAB functions. The "conv2d" function is not even close to the runtime of the other two functions as its runtime is more than ten times as long as that of the "conv_2d" at certain points and around three and a half times as long as that of MATALB's function. Though in Figure 6 the "conv_2d" shows a better runtime than that of MATLAB's function, one can argue that for kernels with a dimension greater than 9×9, MATLAB's function will have a shorter runtime. This argument is a result of the seemingly faster growth rate of the runtime for the "conv_2d" function as a function of kernel sizes compared to that of MATLAB's function. One other observation from Figure 6 is that MATLAB's function takes longer to process convolutions for the case of a 3×3 kernel than that of any other size tested. No satisfactory explanation is available for this strange result.

[22]The MATLAB script used for this test can be found at `https://github.com/KRappaport/Hardware-Convolution/blob/master/MATLAB_Code/time_cmpr_var_ker_conv2.m`.

2.2 Multi-Core[23]

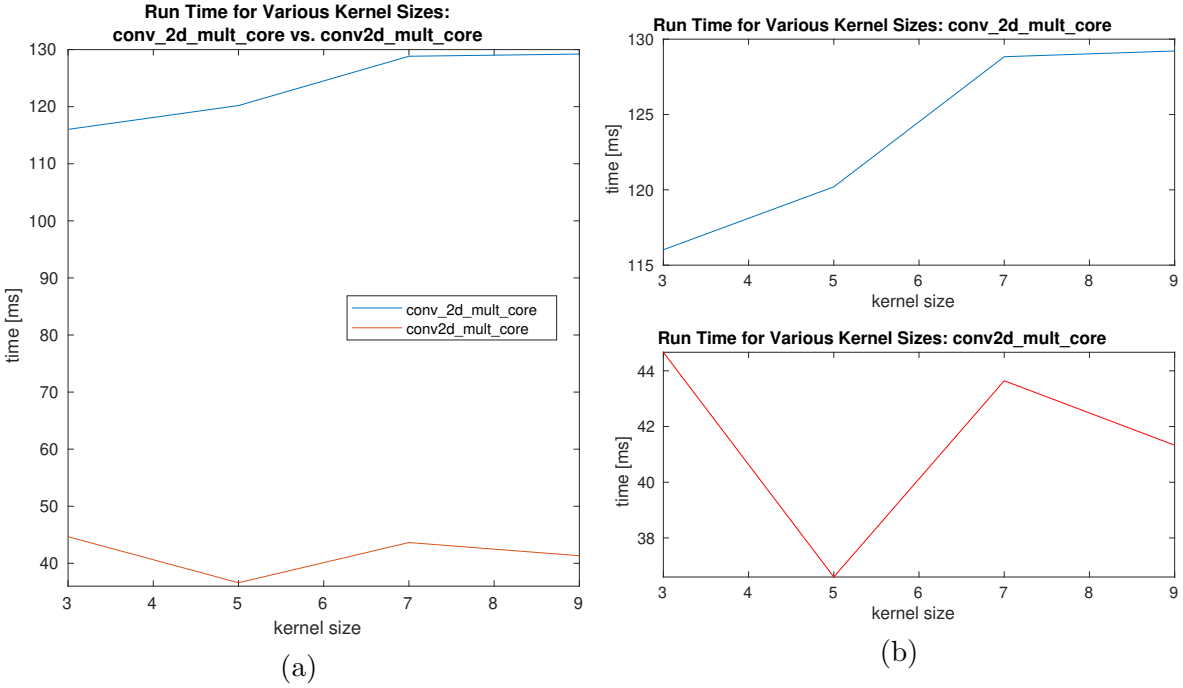
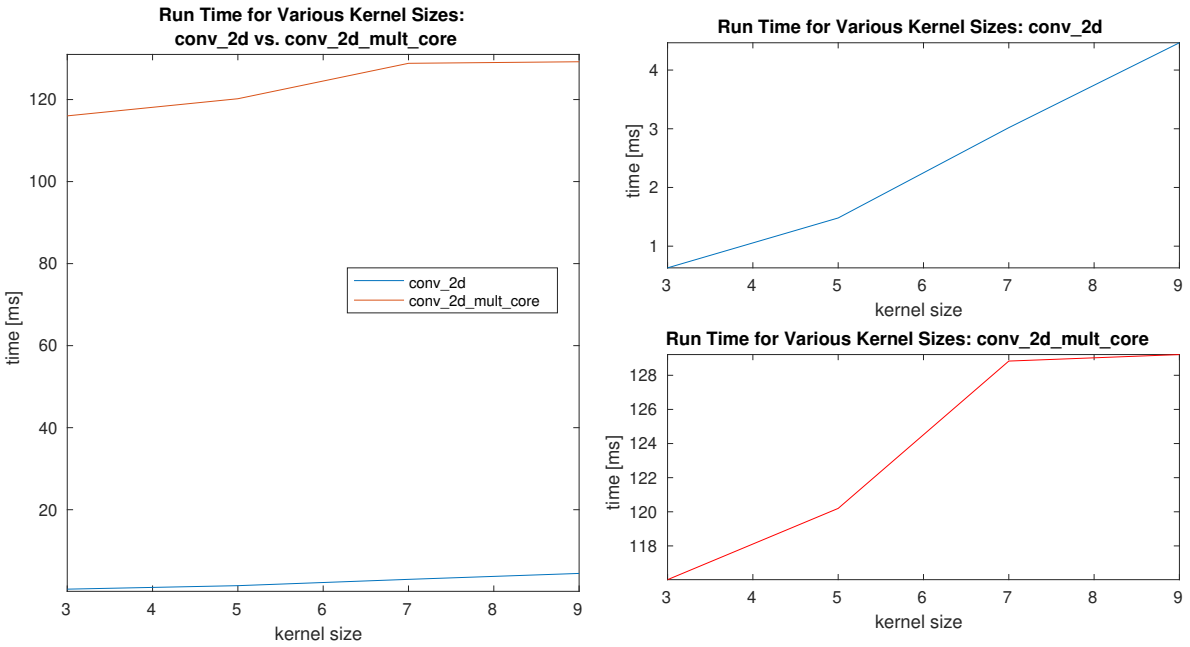


Figure 7: Runtime for Increasing Kernel Size: Multi-Core

Figure 7 shows how the "conv2d" function has shorter runtime than that of the "conv_2d" function when using multi-core processing. This is the first instance in which the "conv2d" function has better performance than the "conv_2d" function. The difference in performance is significant, with the "conv2d" function having about 3 times better runtime than the "conv_2d" function. There is no clear trend for the "conv2d" function, though for the "conv_2d" function there seems to be a linear trend in the increase of runtime. The reason for the high runtime for the dimension 3 kernel for the "conv2d" function, like that of the MATLAB function in Section 2.1, is not immediately known.

It is important to note that for both functions, "conv_2d" and "conv2d", the runtime has dramatically increased when processing via multi-core processing. The runtime of the "conv_2d" function has increased from around 3.5 ms to an astounding 130 ms, an increase by a factor of almost 40, and the "conv2d" function increased from around 14 ms to around 41 ms, an increase by around 3 times. A possible source for this increase in runtime is mentioned in Section 1.2 above.

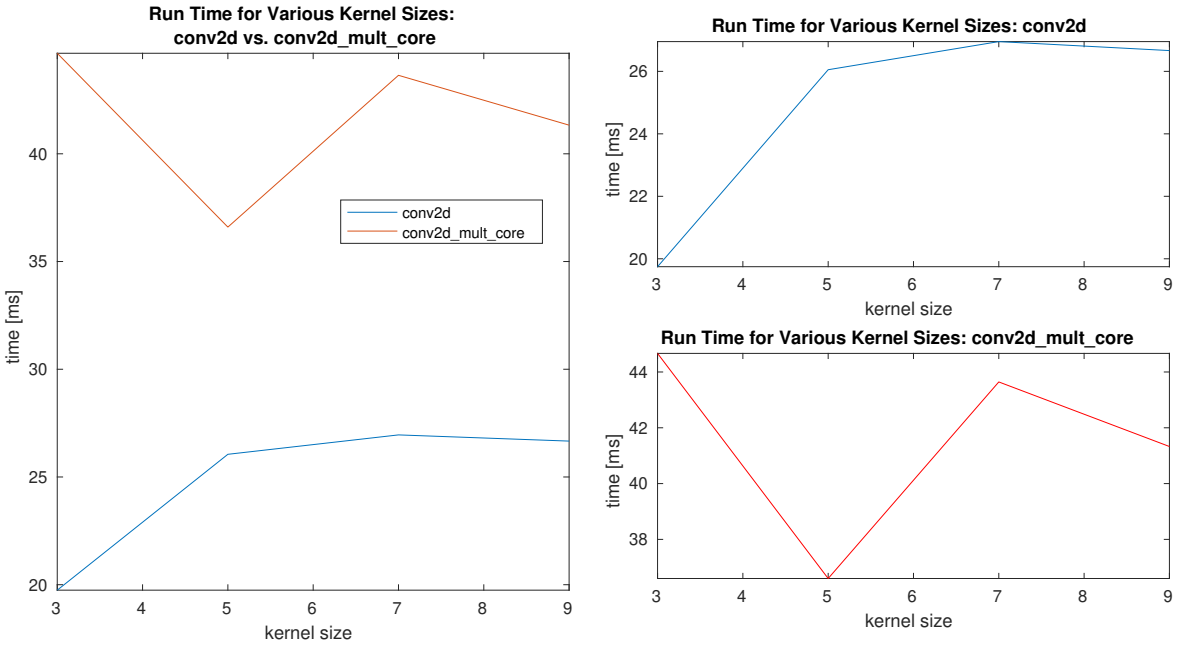
[23]The MATLAB script used for this test can be found at `https://github.com/KRappaport/Hardware-Convolution/blob/master/MATLAB_Code/mcore_time_cmpr_var_ker_conv2.m`.



(a) (b)

Figure 8: conv_2d Runtime for Increasing Kernel Size: Single-Core vs. Multi-Core

As noted above, it can be seen that the runtime for the "conv_2d" function increased nearly 40 times as a result of multi-core processing.



(a) (b)

Figure 9: conv2d Runtime for Increasing Kernel Size: Single-Core vs. Multi-Core

As noted above, the runtime for multi-core processing is greater than that of the regular processing. However, Figure 5 would indicate that as the image size grows the

multi-core processing will be able to save time.

3 Increasing Depth

In this test, the time for the convolutions as the depth of the image and kernel increased was checked. The depths varied from 1 to 11. The image dimensions were kept at a constant 32×32. For this test, two different kernel dimensions, a kernel set of dimension 3×3 and a kernel set of dimension 5×5, were used. The test set consisted of 10 randomly generated images for each depth and 10 randomly generated kernels for each dimension and depth.

3.1 MATLAB Compared to the Custom Functions[24]

3.1.1 Kernel of Dimension 3

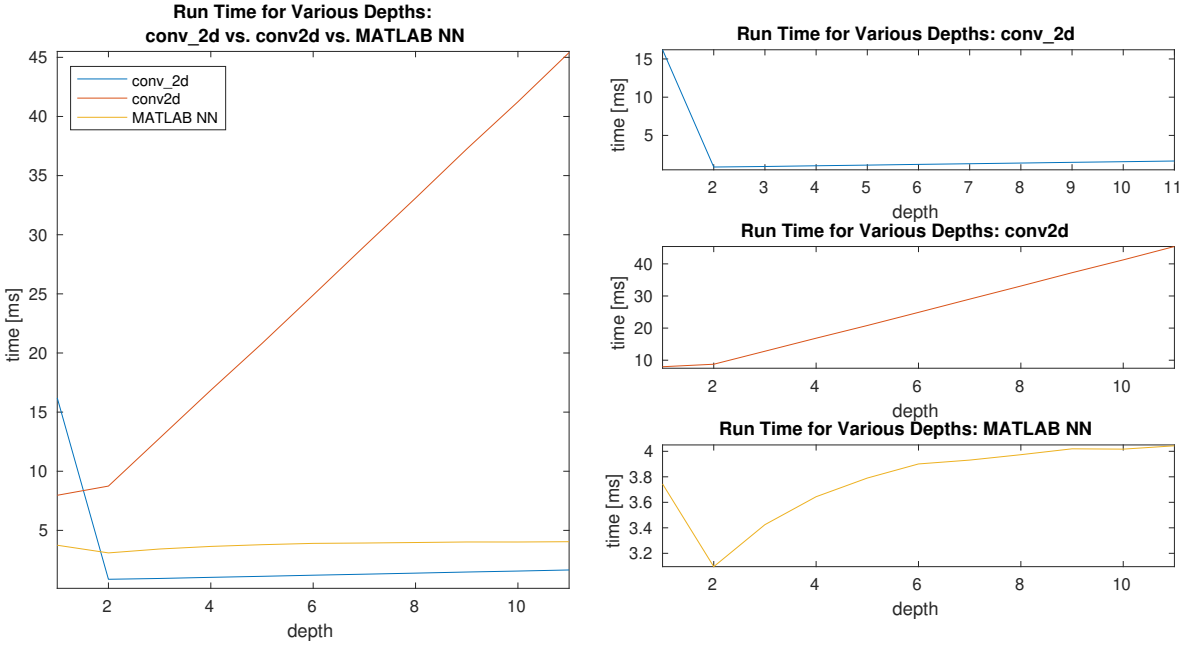


Figure 10: Runtime for Increasing Depth (Kernel 3×3): CPU

Figure 10 leads to a few observations. First, the runtime of the "conv2d" function is significantly greater than that of both the "conv_2d" function and that of the MATLAB function. The second observation is that the "conv_2d" function has better runtime than that of the MATLAB function. The runtime of "conv_2d" is about half of that of MATLAB for the case for varying depth. Furthermore, the "conv_2d" function seems to have very small linear trend, whereas the MATLAB function has a clear logarithmic trend. This suggests that for some large depth, the runtime of MATLAB will eventually be smaller than that of the "conv_2d" function. Lastly, the "conv_2d" function has the worst runtime for a depth of one, after which it takes over as the leader in runtime efficiency.

[24]The MATLAB script used for this test can be found at `https://github.com/KRappaport/Hardware-Convolution/blob/master/MATLAB_Code/time_cmpr_var_dep_conv2.m`. The same script was used for both the 3×3 and 5×5 kernels, but a few changes were made depending on which kernel was to be used at time of testing.

3.1.2 Kernel of Dimension 5

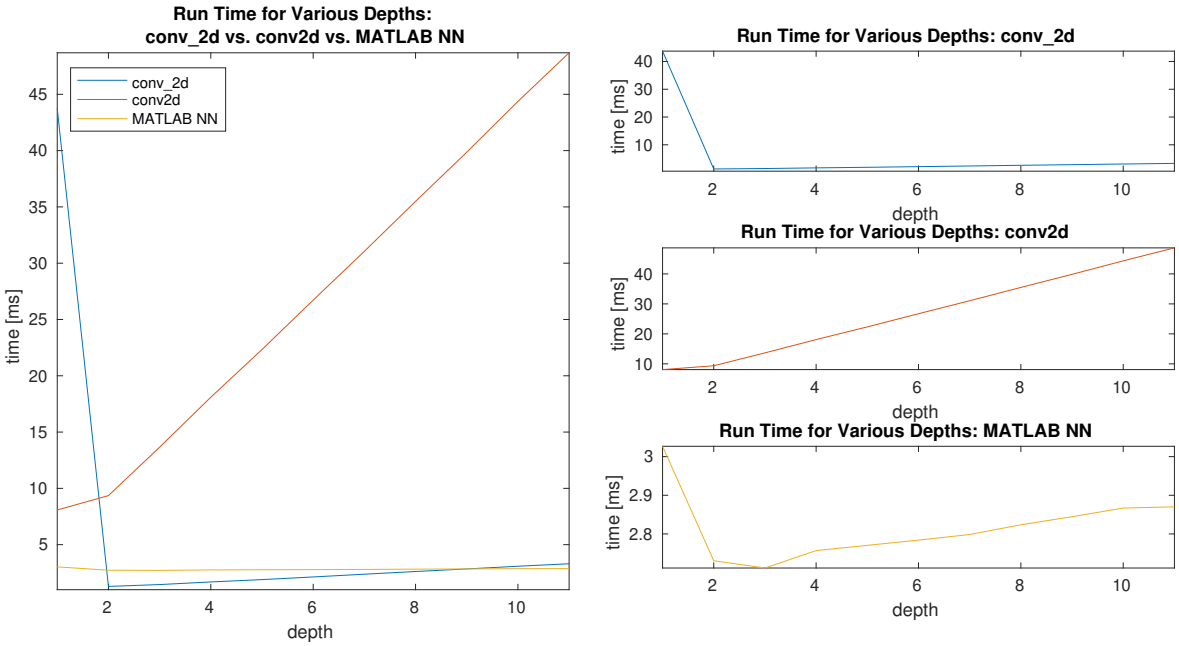


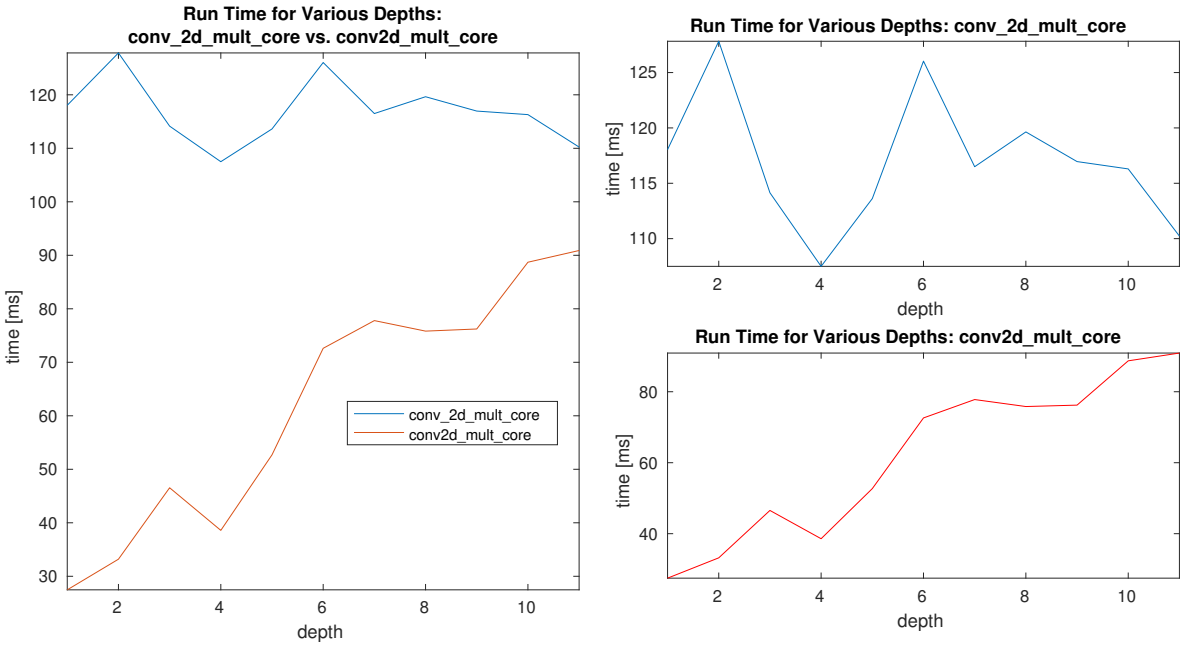
Figure 11: Runtime for Increasing Depth (Kernel 5×5): CPU

Figure 11 again shows how the "conv2d" function is dramatically worse in regards to runtime in all cases except for a depth of one. In this figure, it can be seen that the "conv_2d" function has the best runtime for depths 2-8, but MATLAB's function takes over as having the shortest runtime for larger depths. It is quite interesting to note that the runtime here is actually less than that of the 3×3 kernel seen in Figure 10. This is strange because there are significantly more computations required for the 5×5 kernel compared to the 3×3 kernel, especially considering that neither custom function followed this trend, rather they increased in their runtimes. The source of this anomaly is not immediately known.

Note that the fact that MATLAB's function overtook the "conv_2d" function in this section gives support to the idea proposed in Section 2.1 that given a large enough depth for the 3×3 kernel, MATLAB runtime would eventually be better.

3.2 Multi-Core[25]

3.2.1 Kernel of Dimension 3

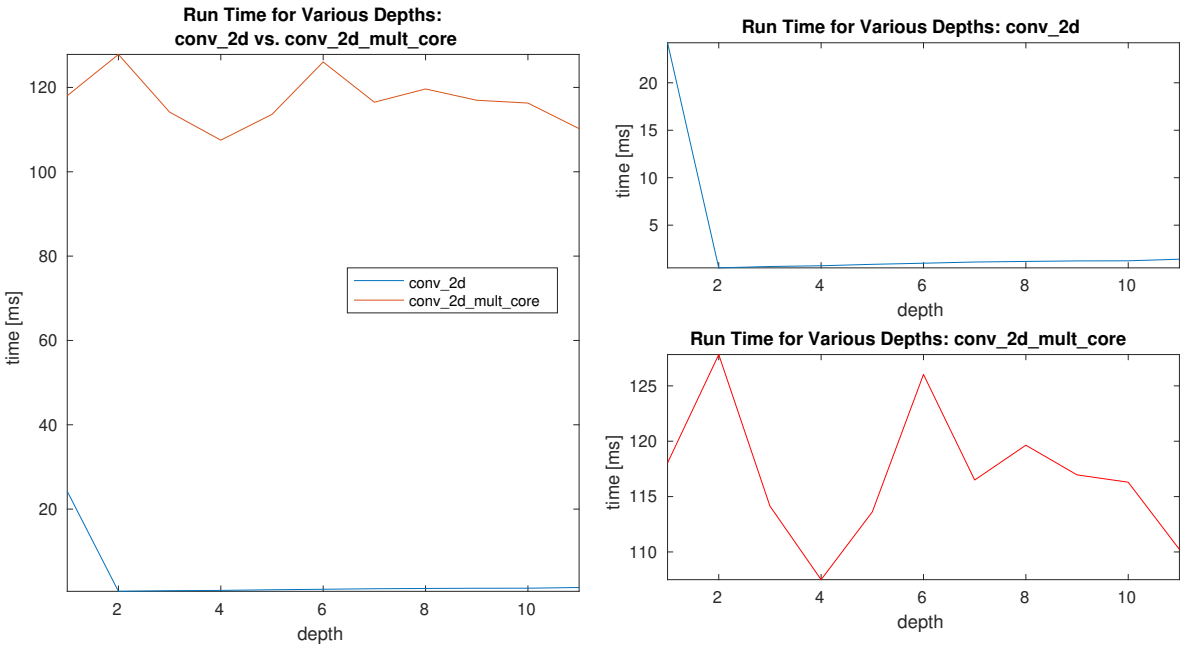


(a) (b)

Figure 12: Runtime for Increasing Depth (Kernel 3×3): Multi-Core

Figure 12 shows that the "conv2d" function has a better runtime than the "conv_2d" function for the given depths. This is consistent with the multi-core results in the multi-core processing for varying kernel sizes, as shown in Section 2.2. The trend of the "conv_2d" is not easily determined, though one can say that it is linearly decreasing (which is hard to understand given the fact that for greater depth there are a greater number of computations involved and hence a greater projected runtime). The trend of the "conv2d" function looks either linear or logarithmic, but in any case an increasing function with respect to runtime.

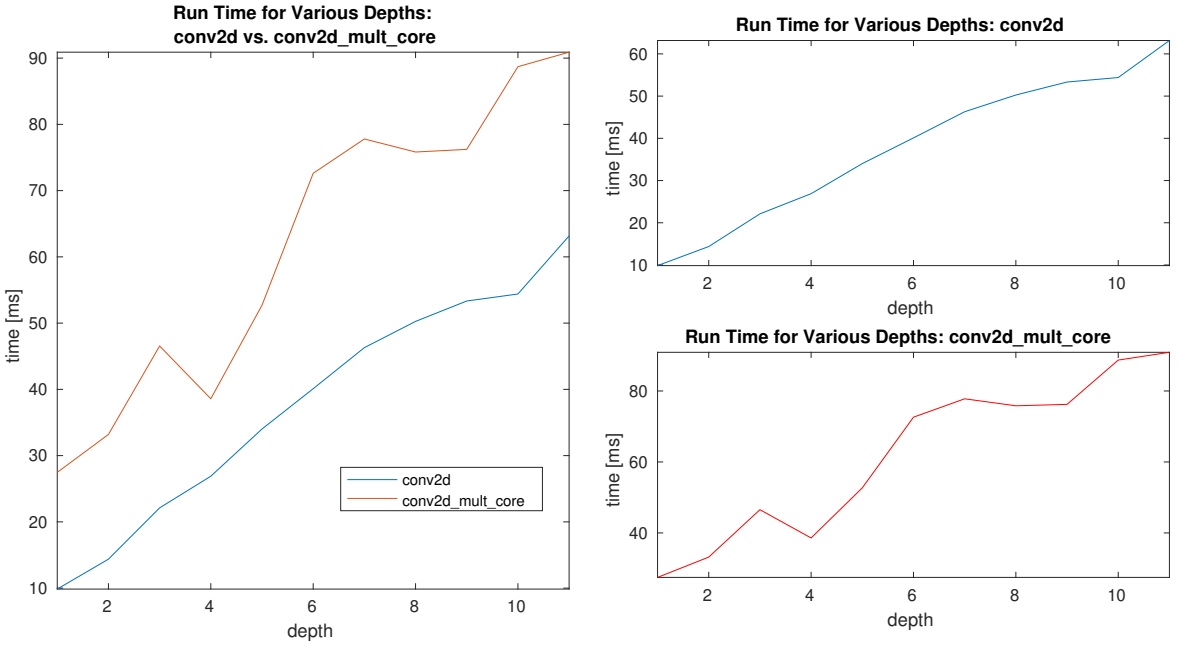
[25]The MATLAB script used for this test can be found at `https://github.com/KRappaport/Hardware-Convolution/blob/master/MATLAB_Code/mcore_time_cmpr_var_dep_conv2.m`. The same script was used for both the 3×3 and 5×5 kernels, but a few changes were made depending on which kernel was to be used at time of testing.



(a) (b)

Figure 13: conv_2d Runtime for Increasing Depth (Kernel 3×3): Single-Core vs. Multi-Core

Figure 13 shows how the multi-core processing increases the runtime of the "conv_2d" function by more than one-hundred times the original runtime. See Section 1.2 for possible explanation.

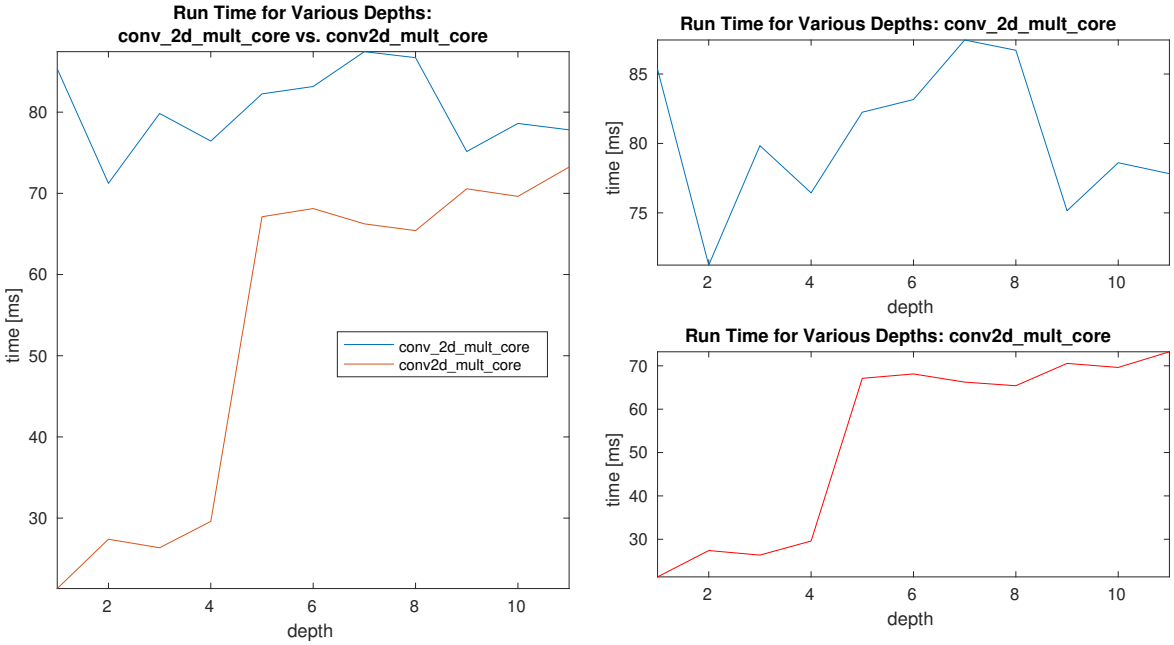


(a) (b)

Figure 14: conv2d Runtime for Increasing Depth (Kernel 3×3): Single-Core vs. Multi-Core

Figure 14 shows how the multi-core processing increases the actual runtime of the "conv2d" function but not as drastically as that of the "conv_2d" function. The increase in runtime for the multi-core processing is between 1.5 to 3 times that of the regular processing. Though the ratio of runtime does change, the difference in runtime remains fairly constant at roughly 20 ms, due to the fact that the two implementations have a similar trend at the values of depth tested.[26]

3.2.2 Kernel of Dimension 5



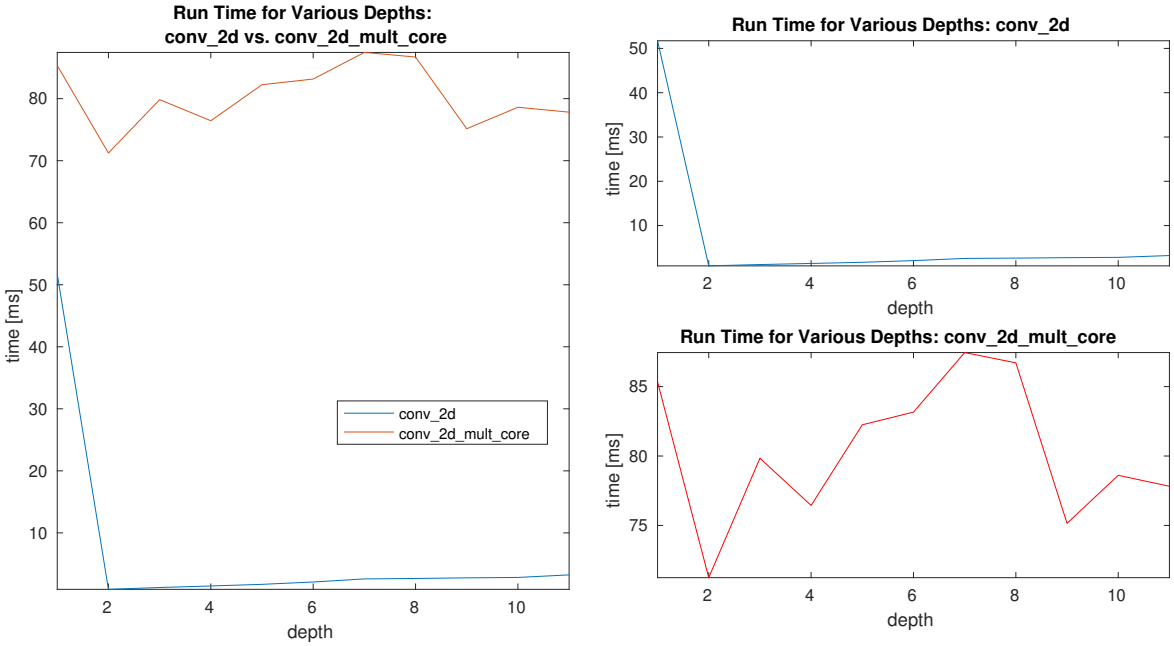
(a) (b)

Figure 15: Runtime for Increasing Depth (Kernel 5×5): Multi-Core

Figure 15 shows how the "conv2d" function has a shorter runtime than that of the "conv2d" function when processing on multiple cores. This is consistent with the previous section which analyzed the results of using a 3×3 kernel with varying depths, where the "conv2d" function was the one with better runtime as well.

Although the "conv2d" function has better runtime, the difference between the two is much less than for a 3×3 kernel with varying depths. Furthermore, one can predict from the trend of the graphs that eventually the "conv_2d" function will have better a runtime for a sufficiently large depth.

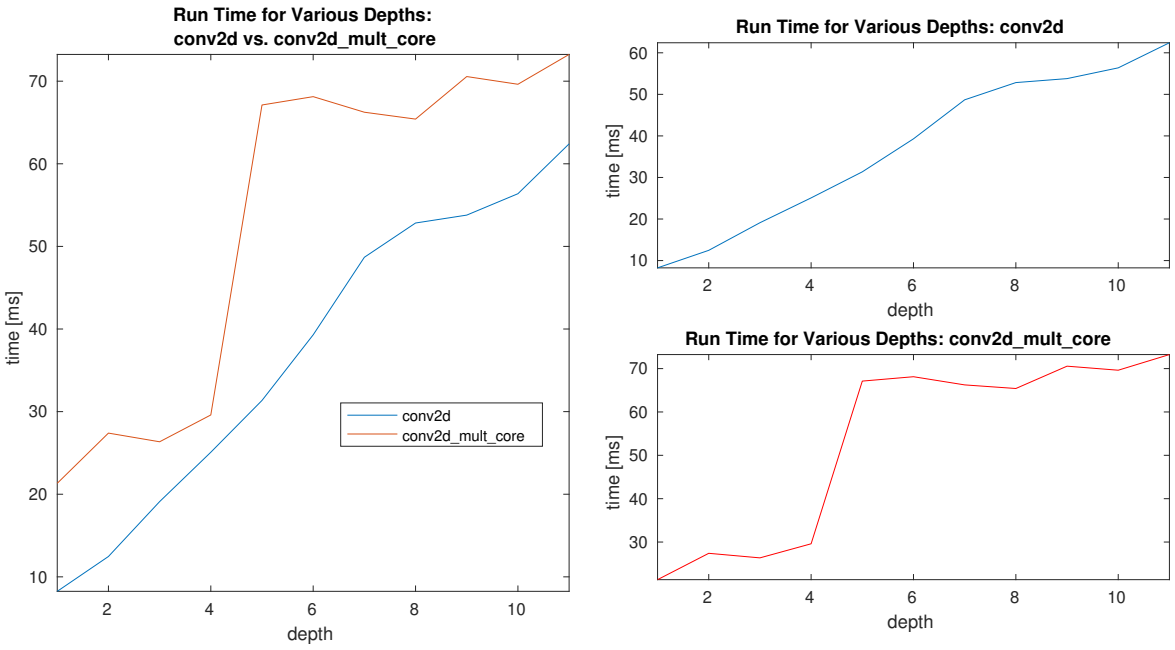
[26] One can argue that the multi-core processing has a more logarithmic trend than that of the regular processing which seems more linear, but the difference for the values tested is negligible.



(a) (b)

Figure 16: conv_2d Runtime for Increasing Depth (Kernel 5×5): Single-Core vs. Multi-Core

Figure 16 shows that the "conv_2d" function has a significant increase in runtime for performing the multi-core processing. Though the runtime is about one-third less than the runtime needed for the 3×3 kernel, nonetheless the multi-core processing takes around 40 times as long as that of regular processing.



(a) (b)

Figure 17: conv2d Runtime for Increasing Depth (Kernel 5×5): Single-Core vs. Multi-Core

Figure 17 shows that the runtime for the "conv2d" function for multi-core processing takes longer than that of regular processing. Although the multi-core processing takes longer, unlike the case of the 3×3 kernel, the difference between multi-core and regular processing is much less here with the 5×5 kernel. Similarly, it seems from the trend of the curves that the "conv2d" with multi-core processing will eventually be faster than that of regular processing, given a large enough depth.

Part II
Phase B (FPGA)

FPGA and HLS Environment

The second phase of the project was to design a method to perform convolutions on an FPGA. The design was made and tested for the ZedBoard,[27] though there are likely many other devices that are capable of running our design. This was a three stage process which began with creating a method to perform the convolution and write the code for the FPGA. The second stage was to take our design and connect it to the necessary interfaces needed to transfer data to it and compile the bitstream that programs the gates of the FPGA, and the final stage was to write code for an ARM[28] CPU which handles interfacing with the FPGA to perform the convolutions.

Once we understood the general process and operations for the convolution from the function we wrote in the first phase of the project (as described in Part I), we were ready to create a method to perform it on an FPGA. Due to a relatively short time frame for the project and our limited knowledge of Verilog and VHDL, the decision was made to write the code in C/C++[29] and to use Vivado HLS to create the Finite State Machine (FSM), generate the Register Transfer Level (RTL), and translate it down to Verilog and VHDL. This method comes with the obvious downside of having to hand over control of how things should be scheduled[30] as well as a loss of complete understanding to the internal operations of the design. However, after getting familiar with the directives[31] as well as how to structure the operations needed for the convolution, we were able to generally receive an acceptable RTL. Overall there were definitely many advantages to using HLS and it allowed us to get through the design process at a much faster rate than possible had we used Verilog or VHDL. In addition, Vivado HLS provides two forms of verification: C simulation and cosimulation. The C simulation is the similar to running a standard C/C++ program and has the advantage of speed and allows for initial checking even before the RTL is synthesized. Cosimulation is useful after the RTL is synthesized. It will first do a standard C simulation and then perform a simulation of the RTL in order to verify that the

[27] See Appendix C.2 for some of the technical specifications of the ZedBoard.

[28] ARM is a Reduced Instruction Set Computing (RISC) architecture developed by Arm Holdings and licensed out to other companies.

[29] Although we are familiar with C/C++, the coding style was very different from what we were used to since it needs to be as static as possible. Dynamic memory allocation is not an option and loops with dynamic stop conditions cannot be unrolled and scheduled in parallel.

[30] This lack of ability to control scheduling has at times caused serial scheduling as opposed to the desired parallel scheduling, while other times it has caused operations to be executed in an incorrect order and produced incorrect results. Also, at times it is incapable of synthesizing it and will keep using more and more memory until it runs out and is killed by out of memory (OOM) Killer (this was ran on a server which had at least 180GB of physical memory available).

[31] The directives placed throughout the code are used to give directions for the synthesis. The directives deal with defining port types, setting what resources should be used for arrays and how they should be structured, unrolling loops, pipelining operations, and other configuration items in order to help the synthesis provide an efficient implementation.

synthesis is correct; however, this can be a timely process. Once the design has been synthesized, tested, and deemed satisfactory, the RTL is exported so it can be used in conjunction with the other devices necessary for proper function.

The exported design from Vivado HLS is then imported to Vivado to be used in a block diagram along with other items.[32] With the design imported from Vivado HLS, it can then be connected to the necessary Xilinx IP's so that the interfaces for the convolution block that we designed can be properly accessed and used from the ARM CPU. Most connections are able to be automated, however, some still needed to be done manually and can be confusing without the help of a clear example showing the devices and connections that need to be made. After everything is connected and setup, the block design is synthesized and a bitstream[33] that can program the gates of the FPGA is generated. Once the bitstream is compiled, Vivado also provides a design report which gives resource usage, a timing report, and a power consumption breakdown for the design.

With the bitstream successfully compiled, the design is exported to the Software Development Kit (SDK) for the final step in the process; the ARM CPU side of the ZedBoard's System on Chip (SoC). The SDK provides many useful and needed C libraries, many of which are standardized libraries and some are customized for the FPGA design. Using the provided libraries, C code can be written, without too much difficulty, in order to use the CPU to interface with the FPGA and perform the convolutions. Included in the tools of the SDK are the cross-compilation toolchain for ARM in order to compile the C code and generate an Executable and Linkable Format (ELF) file for the ARM architecture of the ZedBoard's CPU, a tool to download the bitstream to the FPGA as well as the ELF file to run on the CPU, and a serial port interface in order to monitor standard output.[34] Using these tools along with the C code, we were able to test the timing for the convolutions on the same set of test data used in MATLAB.

4 The FPGA Convolution Algorithm

There were many considerations to be made throughout the design of the convolution block for the FPGA. The first decision that needed to be made was the overall algorithm for the convolution. We had two basic options: the first was a windowing method in which the kernel is moved along the image and image pixels are multiplied with a kernel element as needed for a particular result,[35] and the second option is called Shoup's method[2] where each pixel of the image is first multiplied by each element of

[32] In our case, the other items were Xilinx's blocks needed for proper interfacing between our FPGA design and the ARM CPU. However, one can also import other designs from Vivado HLS as well as use blocks created using Verilog or VHDL.

[33] While there are tools for simulating the block design, it requires writing a testbench in either Verilog or VHDL. Writing the testbench, however, is difficult since there are many signals that must be handled and set at the correct times, but without fully knowing how our design works it would require reading computer generated Verilog or VHDL code to know what needs to be done.

[34] There are many other tools included in the SDK, however, the listed ones are those that were most commonly used in this project and which were necessary for the design and testing.

[35] This is very similar to what is shown in Figure 1.

the kernel and then placed in a pipeline. As the function transverses the image, it will accumulate the rest of the image pixel-kernel element pairings needed for a result. Both of these approaches easily allow for the multiplications and additions to be done in parallel so that many operations can happen simultaneously,[36] cutting down on the time for each convolution and also significantly lowering the time growth rate as the kernel dimensions increase.[37] The main difference between the windowing method and Shoup's method is that windowing requires having a proper section of the image available[38] and Shoup's method requires only a single pixel of the image at a time.[39]

The sequential access of the pixels that is used in Shoup's method had a very significant advantage of allowing for the image to be streamed[40] to the FPGA as opposed to using a memory-mapped method for transferring the image. Being able to stream the image allows for a smoother flow of data from CPU to FPGA and also means that the convolution can start the moment the first pixel arrives to the FPGA. Due to this, we decided to make Shoup's method our method of choice. Also, along with streaming the image into the FPGA, the results are streamed out. This idea of streaming is also used for moving data internally since it allows for a constant flow through the convolution block, cutting down on time spent on memory access calls.

Another major design choice made was not to create an implementation in which a single bitstream could be compiled and handle the convolution for all desired kernel sizes and depths.[41] Instead, we decided to take advantage of the re-programmable nature of an FPGA and designed it such that each different kernel size and depth would need its own bitstream.[42] This does not mean that there are many different HLS codes, each corresponding to a different kernel size and depth. In reality there is one set of HLS code, but in the code the kernel size and depth are set as constants and all that needs to be done to synthesize for a particular kernel size and depth is to have certain defines properly set.[43] While the HLS code could be easily changed so that the kernel size and depth is a variable that is set through a port, allowing for a single bitstream to be capable of handling convolution for all kernel sizes and depths, this would significantly limit the amount of parallelism achievable. However, the ports to the convolution block were not changed and remained the same for all kernel sizes and depths. Although this would mean that the resources used for the ports could be more than needed,[44] this cost

[36]For a 3×3 kernel, 9 multiplication operations can occur at once.

[37]It is important to note that due to sizing constraints of the FPGA, at a certain point not all multiplications will be able to be done at the same time. A 3×3 kernel is small enough, but for a 7×7 it would require more resources than are physically available.

[38]The requirement is for pixels that are next to each other but from different rows and columns. Thus the order in which the image data is required is not how the data in a standard image is ordered.

[39]Each image pixel is only needed once for each convolution and in a sequential manner, corresponding to the structure of a standard image file.

[40]Streaming the data makes use of the First In First Out (FIFO) method instead of using addresses to access the data.

[41]The desired kernel sizes are 3×3 to 9×9. The desired depths are 1-11.

[42]In practice, we only tested kernel sizes of 3×3 and 5×5 with depths from 1-11 and kernel sizes 7×7 and 9×9 with a depth of 3. For a 3×3 and 5×5 kernel it is likely that depths greater than 11 (up to some finite limit) can be successfully compiled. For the 7×7 and 9×9 kernels, depths less than 3 should be fine and depths greater than 3 may be possible, but without testing it cannot be guaranteed and there may not be resource available on the FPGA for depths greater than 3.

[43]See Appendix B for details on setting the kernel size and depth.

[44]For example the port for the kernel is memory-mapped and even for a 3×3 kernel and a depth of 1

was very minimal compared to the benefits. Setting the ports this way meant that the address space used to access the convolution block from the ARM CPU was the same for all kernel sizes and depths. This meant that the same code for the ARM CPU to handle the interaction with the FPGA could be used regardless of which bitstream the FPGA was reprogrammed with. The functions and addresses used to interact with the FPGA did not need to be changed, but changes to ensure the correct images and kernels are used may be necessary since configuring the ports this way does not make it possible to use the bitstream of 3×3 kernel to perform convolution with a 9×9 kernel.

4.1 Shoup's Algorithm

Shoup's algorithm takes an approach of creating a pipeline through which all the data flows and is operated upon. This means that each image pixel is used in a sequential manner, as it would be stored in a standard image file, and only needs to be accessed once, allowing for the image pixe×s to be loaded into a FIFO and streamed into the FPGA. Streaming the data is advantageous over a memory-mapped interface since it cuts down on memory access time. Each pixel enters the convolution block and is routed to a set of multipliers where it will be multiplied with each element of the kernel. All the pixels for a given row and column undergo the multiplications with their respective kernels. In other words, if the image has a depth of three, then the three pixels that make up a certain row and column are removed from the FIFO and then each pixel is multiplied against the kernel corresponding to its depth. Then the results of the multiplications are summed along the depth, so that there is a single 2D matrix that has the dimensions of the kernel. The multiplication results are then moved to an adding stage where they are added with a set of partial sums that have accumulated from the multiplication results of previous image pixels. Eventually these partial sums, as they move down the pipeline, will become a full result and streamed out. For this to work properly, the partial results need to be moved down the pipeline correctly so that the correct image pixel and kernel element pairings will be summed together. To facilitate this, there is a network of holds and delay lines. Each kernel row contains $K - 1$ holds, where K is the kernel dimension, and then between each kernel row is a delay line of size $W - (K - 1)$, where W is the width of the image.[45] The direction in which the data flows is from the last kernel element to the first kernel element and from there it leaves the convolution block. This can be best seen from Figure 18,[46] which shows a pipeline for a 3×3 kernel. In Figure 18, the boxes labeled Δ represent a "hold" and the boxes labeled $\Delta(W - 2)$ are the delay lines.

the port is set to handle the maximum (9×9 kernel and depth of 11), thus requiring a larger memory space than is actually needed.

[45]The total storage amount between the holds for a kernel and row and the delay line between that row and next is equal to the width of an image. The delay line is there in order to make sure there is proper alignment from one image row to the next.

[46]Figure 18 is adapted from one of the figures provided by Shoup in his paper.

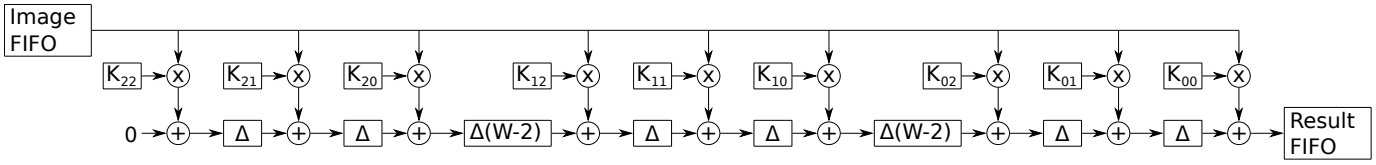


Figure 18: Pipeline for Shoup's Method for a 3×3 Kernel

The diagram in Figure 18 provides a basic representation of the pipeline, however, it does not show everything and implementing this exactly as shown will lead to the wrong results. The main element missing in the block diagram is the filtering that is needed to prevent pixels on the edge of the image[47] from being multiplied with elements of the kernel that it should not. The first part of the filtering is to make sure that collecting of the outputs from the pipeline starts at the correct time, as the first number of outputs from the pipeline will be purely from the top edge of the image multiplied with kernel elements it should not be. Then the left and right image edge pixels need to be prevented from being multiplied with kernel elements they should not be, otherwise the results of those multiplications will continue further down the line creating incorrect results. The final step is flushing out the remainder of the results that are in the holds and delay lines after all the image pixels have been processed and there are no more multiplications and additions to be performed. The number of results that remain in the pipeline at that point is the same number of initial outputs from the pipeline that are ignored.

4.2 Window Algorithm

The Window Algorithm is relatively straight forward. The algorithm works by fixing a "window" of the kernel on a section of the image which match the dimensions of the kernel. The window is then shifted along the image in a zig-zag fashion going left to right along the x axis and top to bottom along the y axis. This algorithm is split into two cases: center cases and edge cases.

In center cases, the window of the kernel fits entirely onto the image in the x, y, and z dimensions. This means that every pixel in the kernel will be used for this set of multiplications and additions for a single pixel of the convolution result. This is best demonstrated by Figure 1c.

In edge cases, part of the kernel does not overlap with any part of the image. This hanging edge can be in entirely in the x or y dimension, or a both in a corner-case. The z dimension is always the same for the image and kernel. This edge is zero-ed out by using non-static logic in the algorithm based on the kernel and image size. The remaining kernel pixels are multiplied with their proper image counterparts and added as required for the convolution. This is best demonstrated in Figure 1b.

The way The Window method was implemented was to memory-map a single 3D block of the image containing the necessary 2D slices for the result of a single desired 2D slice. For example, for a 5×5×3 image and a 3×3×3 kernel, to process the first line, line-0, both 2D slices of line-0 (in the y dimension) and line-1 are necessary for the required convolution. To process line-1, the 2D lines are required to be mapped in and

[47]The size of the image edge is dependent on the kernel size.

stored: line-0, line-1, and line-2. Each block of required lines are mapped and stored on the FPGA.

Due to the need to store large amounts of data, the restricting factor in the Window Algorithm is space. The plan of the project included testing convolution of kernel sizes ranging from 3×3×3 to 9×9×3 (maximum depth 11. Example: 7×7×11), and image sizes ranging from 3×3×3 to 256×256×11. This meant that for the worst-case scenario of a kernel of 7×7×11 and an image of 256×256×11, there was a need to store 256×7×11 pixels multiplied by 32 bits for each pixel. This amounts to 630,784 bits of memory for each line in the worst-case scenario. Considering that the ZedBoard contains 140 units of 36 kb of BRAM, which is a total of 4.9 Mb, this allows for a maximum of four lines to be stored, and hence processed, at once. Due to this consideration, we transferred four blocks of 2D slices at a time to be processed for the convolution.

The final result of the convolutions was streamed out.

5 Timing

The next step, after completing the design process, was to test the time it takes to perform the convolutions and then to compare it to the time it took on a CPU using MATLAB's neural network toolbox. As in Part I, the timing was tested for three criteria: increasing image size, increasing kernel size, and increasing depth. The test set used is the same data set that was used for testing on the CPU.

5.1 Increasing Image Size[48]

The image set used to test the timing the FPGA takes to perform convolution on increasing image size, consisted of 10 images for each size with square dimensions ranging from 5 to 256 making a total of 2,520 different images. For each image size there was a corresponding 3×3 kernel (a total of 252 kernels). Both the images and the kernels had a depth of 3.

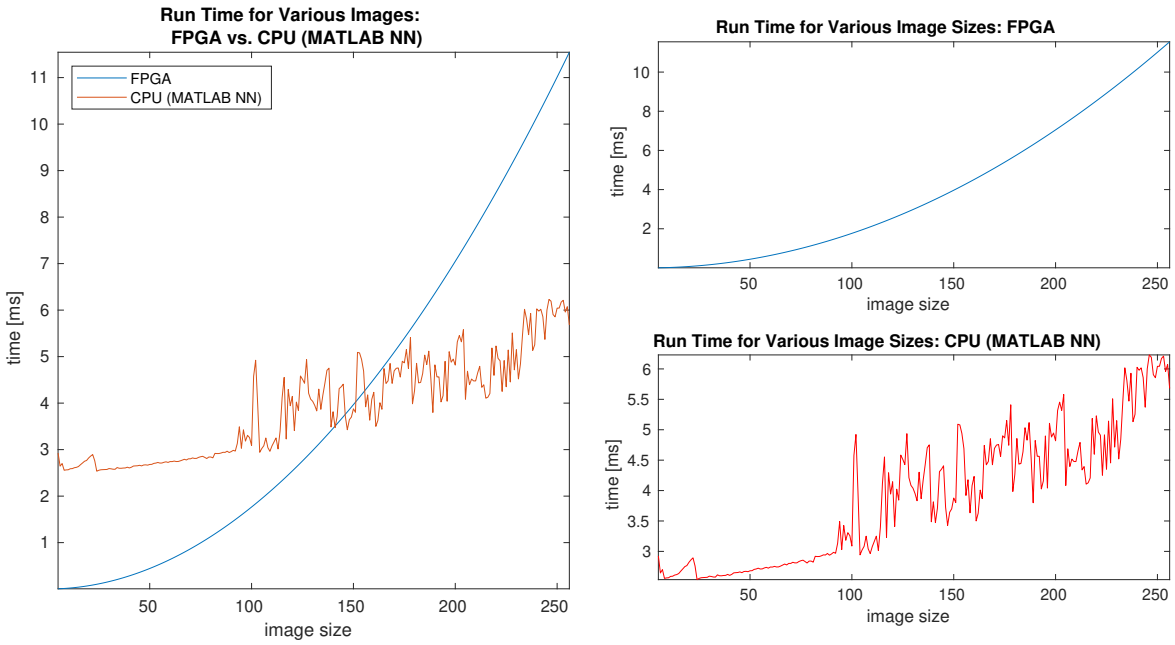


Figure 19: Runtime for Increasing Image Size: FPGA vs. CPU

From Figure 19, it can be seen that the growth rate for the time it takes to perform the convolutions has a bounding of $O(x^2)$, where x is the dimension of the image. While both of their rates are quadratic, the FPGA time growth is much faster. So even though the performance obtained from the FPGA starts off strong with smaller image sizes, it is quickly overtaken by the CPU. Thus at the maximum image size tested (256×256) the FPGA is around 5 ms slower than the CPU.

[48]The main function used for this test is located at `https://github.com/KRappaport/Hardware-Convolution/blob/master/FPGA_Code/SDK_main_testing_files/main(varimg).c`.

5.2 Increasing Kernel Size[49]

For this test, a set of 40 kernels with square dimensions of 3, 5, 7, and 9 (10 kernels for each dimension) were used. There were 10 different images used, all with dimension of 32×32. Both the images and kernels had a depth of 3. This is the same test set that was used on the CPU.

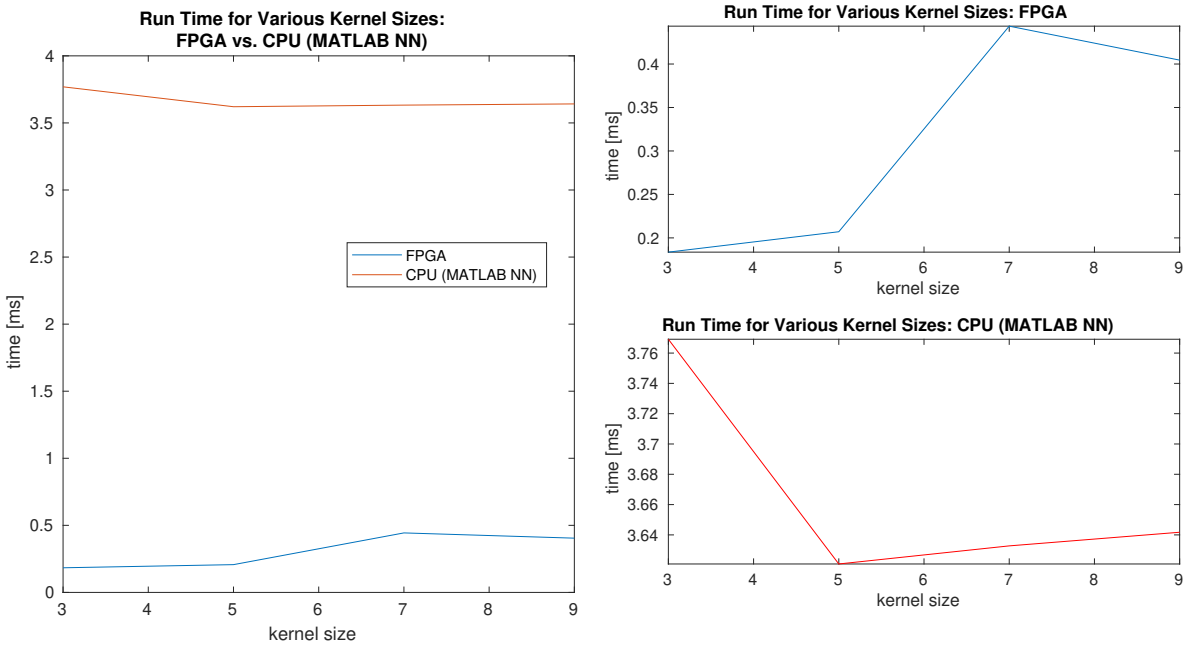


Figure 20: Runtime for Increasing Kernel Size: FPGA vs. CPU

Figure 20 clearly shows that the CPU for each of the kernel sizes tested take about 3 ms longer. However, as seen from Figure 19, for larger images this will not continue to be the case and it will take longer for the FPGA to compute the convolutions than the CPU. More importantly, the CPU has a linear growth rate[50] which is much slower than the increase experienced by the FPGA from the 5×5 kernel to the 7×7. This jump in time seen between the 5×5 and 7×7 is due mainly to the limited resources of the FPGA,[51] so due to an insufficient amount of DSPs needed for the multiplications and additions the parallelism achieved[52] by the smaller kernels could not be obtained.[53]

[49]The main function used for this test is located at `https://github.com/KRappaport/Hardware-Convolution/blob/master/FPGA_Code/SDK_main_testing_files/main(varker).c`. Minor changes were made to the main function throughout the testing, to reflect which kernel size was being tested.

[50]This is overlooking the fact that the 3×3 kernel strangely takes significantly more time than the other kernel sizes tested.

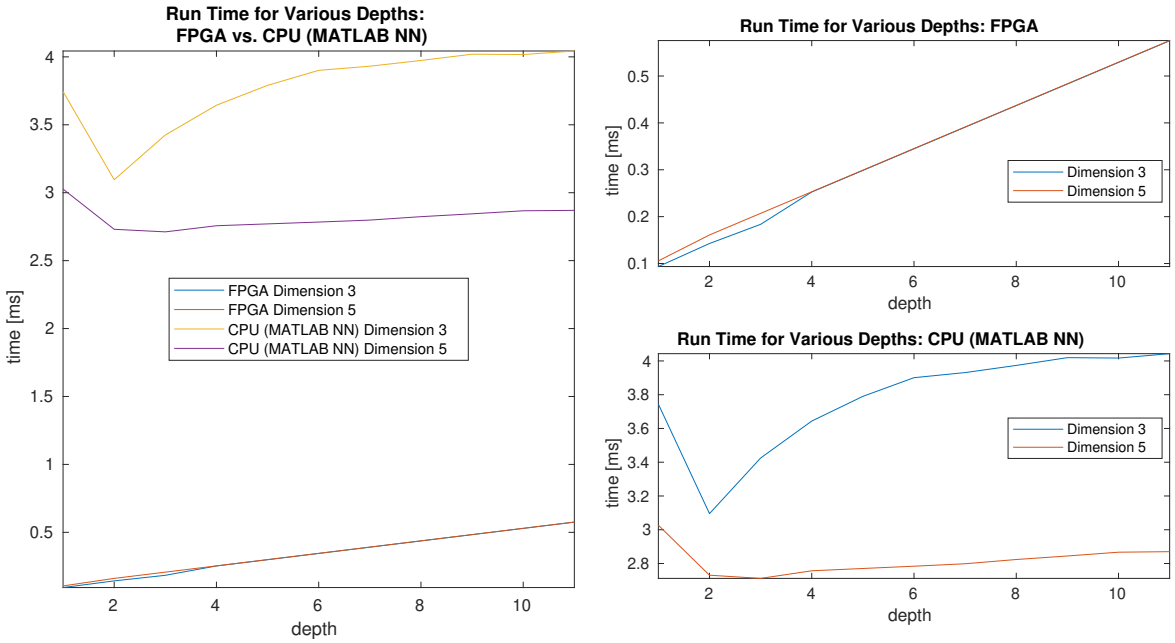
[51]The other cause for the jump is due to the difference in the frequency that Vivado was able to guarantee during synthesis. The 5×5 was able to be guaranteed at 111 MHz, while the 7×7 could only achieve verification at 100 MHz.

[52]While more operations may still be performed in parallel, it cannot, for example, perform all multiplications in the same clock cycle and thus requires additional cycles as opposed to smaller kernels.

[53]The slight decrease in time seen by the 9×9 kernel from that of the 7×7 kernel is because the 9×9 kernel implementation is from a slightly improved version. However, Vivado HLS was incapable of synthesizing this improved version for the 7×7 kernel and would consume all available physical memory

5.3 Increasing Depth[54]

In this test the timing is tested as the depth of the images and kernels are increased from 1 to 11. There were also 2 sets of kernels and images used, one for kernel size of 3×3 and the other for kernel size of 5×5. The images used had a dimension of 32×32. For each depth there were 10 different images and kernels used in each data set.



(a) (b)

Figure 21: Runtime for Increasing Depth: FPGA vs. CPU

The first thing to note, as can be easily seen from Figure 21b, is that for the FPGA starting at a depth of 4, both the implementation for the 3×3 and 5×5 kernel are able to complete the convolutions in the same amount of time. So even though there are more floating point operations required to perform the convolution with a 5×5 kernel than a 3×3 kernel, the resources available on the FPGA allow for enough parallelism to be obtained that there is no increase in time required for the increased number of floating point operations. However, for depths lower than 4 the 3×3 kernel implementation is faster. This is most likely due to the fact that Vivado could verify a 125 MHz implementation for the 3×3 kernel below a depth of 4, but only could verify it at 111 MHz for depths 4 and above. All implementations for 5×5 kernels were only able to verified at 111 MHz.

Figure 21 shows similar results obtained in the other testings. For the 32×32 image, the FPGA shows a time savings of a few milliseconds. However, the linear time growth

(at least 180 GB) before being killed by the operating system.

[54]Two main functions were used for this test. One for 3×3 kernel and the other for a 5×5 kernel. They are located at `https://github.com/KRappaport/Hardware-Convolution/blob/master/FPGA_Code/SDK_main_testing_files/main(vardep3).c` and `https://github.com/KRappaport/Hardware-Convolution/blob/master/FPGA_Code/SDK_main_testing_files/main(vardep5).c`, respectively. Minor changes were made to the main functions throughout the testing to reflect which depth was being tested.

rate of the FPGA is faster than the rate of increase of the CPU for the 5×5 kernel. Also, even though the total time increase seen by the CPU for the 3×3 kernel is more than that of the FPGA, the growth rate of the CPU appears to be logarithmic and tapers off.

6 Power

Power consumption is an important design consideration as an intended application is for embedded uses. Thus, we needed to measure the power consumption for both the CPU and FPGA.

Measuring the power required to perform convolution on the CPU is not a simple process. The CPU is only a part of a greater system. While it is arguably the most important part of the system, the CPU needs access to a sufficient amount of physical memory in order to operate correctly and this must also be included in the measurement. We relied on Running Average Power Limit (RAPL)[55] which is a feature available on some of Intel's CPU's to obtain the power data for the CPU and physical memory. The system measured was the university server[56] that was used for collecting timing data. The server has two CPUs installed, and since only one of them was used to perform the convolutions we only took into consideration the power consumption for that CPU.[57] One of the more difficult problems we had was that there are many different processes running on the server at any given point. Since the kernel and other operating system-related processes are necessary to be able to use the server we did not want to ignore them, but the processes from other users on the server should not be included. At the time of measuring the power it was observed that there was only one other active user and so using a purely empirical method it was decided to subtract 6 W from the results. While the power was not exactly stable while performing the convolutions, we were able to see that for all convolutions the average power fell within a 5 W range. Thus, the determined power consumption used to perform the convolutions on the CPU was 49 W - 54 W.[58]

The power measurements for the FPGA were significantly simpler. Here we needed the power consumption of the entire ZedBoard and at any given point the only processes running were ours for performing the convolutions. Luckily there are pins provided on the ZedBoard to measure the voltage across a small 10 $m\Omega$ resistor that is in series with the 12 V supply in order to measure the ZedBoard's current and thus power.[59] Using a digital multimeter, we measured the voltages across the resistor for the different convolutions and used

$$P = \frac{V_{resistor}}{10m\Omega} \cdot 12V \tag{2}$$

to calculate the power. We found that the power varied from 3.8 W to 4.0 W and while

[55]This tool provides sensors that allow for measuring power. The data is placed into specific registers that can be accessed by the kernel.

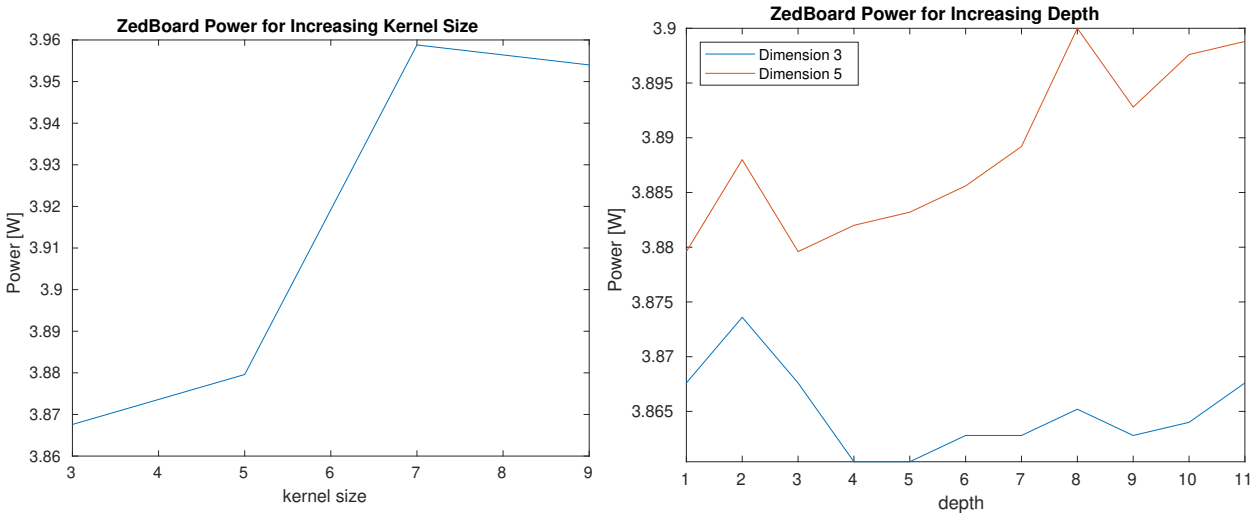
[56]See Appendix C.1 for specs.

[57]This was not difficult since the program used provided the power data for each one separately. Also, it was easy to determine which one was being used by simply observing which CPU showed increases in its power consumption when the convolutions were started.

[58]The averages would always fall within that range. While we tested for power changes when using different data sizes (as was done for timing) it did not appear to change the power consumption and performing convolutions on the same data size multiple times would produce a different average each time, but it always was in that 5 W range.

[59]See page 28 of the "ZedBoard Hardware User's Guide" available at `http://www.zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf`) and page 5 of "Technical report - Implementation of Hardware Accelerators on Zynq" by Jakob Toft availble at `http://orbit.dtu.dk/files/125849853/tr16_07_Nannarelli_A.pdf` for more information on this measurement.

the image dimensions did not matter, the power usage did change depending on the kernel size and depth. This is most likely due to the fact that the resource needs are the same for all images (except for time so it does affect needed energy), but different kernel sizes and depth have different bitfiles which change the resource usage of the FPGA.



(a) FPGA Power: Increasing Kernel Size

(b) FPGA Power: Increasing Depth

Figure 22: FPGA Power Consumption

Figure 22a shows the power of the ZedBoard for the same set of kernels and images used for testing the time for increasing kernel size as described above in Section 5.2. Interestingly, the power trend as the kernel size increases is very similar to that of its timing trend seen in Figure 20. The general trend would appear to indicate that as more of the FPGA is being used, the power it requires also increases.[60]

Figure 22b shows the power of the ZedBoard for the same set of kernels and images used for testing the time for increasing the depth as described above in Section 5.3. It is clear that for a 5×5 kernel, which uses more of the FPGA, it uses more power than for a 3×3 kernel. As the depth increases, the amount of resources on the FPGA also increases. This should lead to an increase of power consumption as the depth increases. From the 5×5 kernel that trend can be seen with the exception of a depth of 2 and 8. The reason for those spikes are not clear, however, it would seem unlikely to be noise since the 3×3 kernel also experiences spikes at those same depths. The 3×3 kernel with depths below 4 operate at 125 MHz while for a depth of 4 and above it operates at only 111 MHz, the increased clock frequency could be the reason that it uses more power even though it uses less resources. So taking into account the extra power used by the 3×3 kernel at depths of 1, 2, and 3 due to the higher clock frequency, the power consumption trend is remarkably similar to that of the 5×5 kernel.[61]

[60] Though the 9×9 kernel uses more of the board than the 7×7 kernel, but is using less power. This maybe due to noise or some other unknown factor.

[61] While the trends might have some minor differences, still they are both generally increasing with the depth except at the depths of 2 and 8 where they both experience a spike in power.

7 Conclusion

The most disappointing results were from timing. One may be mislead by Figure 20 and Figure 21 into thinking that the FPGA provided a significant improvement in the time it takes to perform convolutions compared to the CPU. However, Figure 19 quickly corrects that by showing the FPGA only out-performs the CPU for very small images. The FPGA's rate of increase as a function of the image size is too fast, so for an image of 150×150 the time it needs to complete the convolutions are similar to the CPU and at 256×256 the FPGA is significantly slower and looks like it will only get worse. To make matters worse, a standard image is significantly larger than 256×256, so at the point that the size is in the range of an actual image the time difference may be too big to ignore. Also, it is important to keep in mind that the CPU can switch between performing convolutions using one kernel size and depth to another size and depth with no time lost, while the FPGA needs to be reprogrammed for each different kernel size and depth costing more time. One possible positive that does exist is that for the 3×3 kernels (for all depths 1 to 11) there should be enough room on the FPGA to fit four of the convolution blocks based off of the resource utilization used with only one convolution block.[62]

The power consumption of the FPGA stayed under 4 W while the CPU used around 50 W. However, this 46 W difference needs to be taken with a grain of salt. Energy, which is a function of time, is also a very important factor. So for a standard-sized image the energy used by the FPGA could end up being greater than that used by the CPU. Still, if the energy consumption of the FPGA does not get too high, its low power needs can be very advantageous for an embedded use where low power consumption is an important design factor.

There are some other factors to consider as well. Price is an important factor that needs to be considered. The ZedBoard costs about $475[63] while the CPU on the server used, the Intel Xeon Processor E5-2680 v2, has a recommended price of over $1700.[64] Of course for the CPU option there is more than just the CPU, it also requires at least Random Access Memory (RAM) and a hard drive which further increases the price. Another factor to consider is the sizing. The ZedBoard is fairly compact and will not need much space, as opposed to the CPU and the other necessary components will easily need too much space for an embedded solution.[65]

Our design will not save on time or even match the time of the CPU when dealing with standard sized images. Though the power looks promising, the FPGA may still fall short in regards to energy for a standard sized image. The only real undeniable advantages of our design is the power envelope, price, and size factor. Power, price, and size may be an important enough factor making it the right choice for embedded needs.

[62] See Appendix D.

[63] The price is from AVNET `https://www.avnet.com/shop/us/products/avnet-engineering-services/aes-z7ev-7z020-g-3074457345635221599/?aka_re=1`.

[64] Based off of Intel's recommended pricing found here: `https://ark.intel.com/products/75277/Intel-Xeon-Processor-E5-2680-v2-25M-Cache-2-80-GHz-`.

[65] Also due to the high power dissipation of the CPU, steps need to be taken to ensure proper temperature control, further adding to the CPU's required space.

8 Further Work

Although the design may not have been successful, there is still potential for it. The first step before going further would be to make improvements to the design in order to cut down on the time. While one option might be to find a way to divide the image and use multiple convolution blocks and then combine their outputs to obtain the results, this will only work for 3×3 kernels since larger kernels require too many resources for multiple concurrent calculations. Also this would mean that the entire FPGA could only work on one convolution at a time, but it may be more important for multiple different convolutions to be able to be happening at the same time.[66] The best way to cut down on time would be to split the design into two different HLS designs and create better parallelism. Currently, each pixel first undergoes a set of multiplications and then additions, but before the multiplications can start for the next pixel the additions from the previous pixel need to finish. By creating two separate HLS blocks, additional pipelining can be achieved by having one block take in an image pixel, perform the multiplications, and then output the results into a FIFO buffer which the second block takes from and performs the additions and outputs the final results. This would mean that as soon as one pixel is finished being multiplied the multiplications for the next pixel can start right away and allow the additions and multiplications to occur simultaneously. This change could save on time without a significant increase in resource needs.

Once some time improvements have been made, the next step would be to use the design for a practical use. This would most likely be machine learning. Since machine learning usually uses multiple convolutional layers, it may only be practical to continue with 3×3 kernels due to resource availability. However, due to the popularity of 3×3 kernels this is not necessarily a downside.[3, 1]

[66] That would allow for multiple convolutional layers. That would mean that to convolve the results it would not have to first go from the FPGA to the ARM CPU and then back to the FPGA wasting time. Also, all layers could be in use at the same time and not need to wait for one layer to finish before the next layer can begin.

Appendices

Appendix A Our Thoughts and Experiences Using Vivado

The majority of the work for this project was completed using the Vivado Design Suite.[67] The Vivado Design Suite included: Vivado HLS, Vivado, and the SDK. These applications allowed us to create and RTL for the convolution block, setup the interfaces needed by the convolution block, compile the bitstream, and then deploy the design on the ZedBoard and perform the convolutions.[68] The design suite was used in both Linux[69] and Microsoft Windows 10.

The first step of the entire process was to create a project in Vivado HLS. This appeared to be a simple process, however, when attempted from the Graphical User Interface (GUI) version on Ubuntu 18.04 the "Next" button to move on from the "Add/remove C-based testbench files (design test)" does not do anything when pressed and thus the project creation cannot be completed. Fortunately Vivado HLS can also be run as an interactive shell through which a project was able to be created and could then be opened in the GUI version.

With HLS it was possible to learn the directives to place within the C/C++ code and create a design for the FPGA to perform convolution with a decent amount of parallelism faster than it would have taken to do so in VHDL/Verilog.[70] When time is limited and the needed design is not overly complex, HLS definitely has a major advantage over VHDL/Verilog, but with sufficient time and a decent knowledge of VHDL/Verilog a more sophisticated design can be obtained by writing it in VHDL/Verilog. While Vivado HLS will generally be able to successfully synthesize a decent RTL, its immaturity will at times show through. When Vivado HLS fails it will at best be able to synthesize an RTL that has very bad scheduling, but at times it will be unable to synthesize an RTL at all. When this occurs Vivado HLS will spend a long time trying to synthesize the design and keep allocating memory until there is no more left to be given.[71] After the synthesis is complete, HLS provides a table with the estimated resource usage. However, it was discovered that these estimates were not always reliable and at times it overestimated the LUT usage by more than 25%. This made it difficult to properly determine if the amount of parallelism needed to be decreased, in order for the design to fit on the FPGA. The only way to obtain an accurate report on the resource utilization was by compiling the bitstream in Vivado, however, compiling the bitstream requires extra steps and is a lengthy process. So in the event too many resources are needed for the design the process requires synthesizing

[67] The version used was Vivado HLx 2018.1 WebPack. This can be found and downloaded from `https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2018-1.html`

[68] For more details on the use of these applications, see the Introduction to Part II.

[69] Ubuntu 18.04 using kernel version 4.15.0, and Ubuntu 14.04 using kernel version 4.4.0

[70] At the start of the project we already had a decent familiarity with C/C++, but our knowledge of VHDL/Verilog was fairly limited.

[71] After having this problem on a particular design, it was attempted to synthesize it on a server that had over 180GB of physical memory available and it still was not enough memory.

in HLS, compiling the bitstream in Vivado, making changes to the HLS code, and then repeating the process until the resources needed are within the limits of the FPGA, thus significantly increasing the development time needed than would be needed if the estimates provided by HLS were more accurate.

Setting up the exported design from HLS in Vivado can be difficult. In Vivado, the imported HLS block needs to have its interfaces correctly connected so that it can be accessed by the ARM CPU and function. For proper communication between the HLS block and the CPU, Xilinx provides many IP blocks for the different interface types to handle the transactions on the FPGA side. However, without finding a straightforward example showing which of Xilinx's block are needed and how to properly connect and configure them, this process is extremely challenging at best. Sadly it was not always so easy to find a decent guide for all the different interfaces. Though the setup was difficult, once it was ready, being able to compile the bitstream was a reliable process. The compilation process, however, can be long and requires a lot of CPU time.[72]

The SDK provided similar difficulties to Vivado. Figuring out how to write code to access the FPGA from the ARM CPU required either going through the provided libraries and source code trying to follow what it did and how or if a function could be used for what was needed, or finding a manual that would be able to help. In the end it took a mixture of reading through a lot of the source code for Xilinx's provided functions and some of the manual that provided useful examples. The SDK also had some flaws that caused issues. On a handful of occasions libraries would disappear, which usually occurred when a board support package was modified. The other problem faced was with using the library to use an SD card as a file system on the ZedBoard. This library needed a specifier to link it during compilation, and although it would be added automatically to the linker script, it would sometimes be automatically removed from the linker script and was difficult to manually add it back in.

[72]Thankfully we were able to complete this task on a server that had many threads available to divide the work, cutting down on the needed time significantly.

Appendix B User Manual

This user manual is a very basic guide that mainly points out important changes that will need to be made to some the code, in the FPGA_code directory of the GitHub repository, necessary to running the hardware convolution design on the FPGA. This guide assumes that the user knows how to use the Vivado Design Suite, therefore instructions for using it will not be provided. In addition to a computer with the Vivado Design Suite, a ZedBoard (or similar hardware) and an SD card are needed. All paths used here are relative to the GitHub repository.

The files found in ./FPGA_Code/HLS/src should be added to the source files of the HLS project and the files in ./FPGA_Code/HLS/tb to the testbench files.[73] The first change to make is to the desired kernel size and depth to be used. In ./FPGA_Code/HLS/src/conv2d.h on lines 4-10 there are two defines that should be changed: "KERNEL_DIM" should be set for the desired kernel size, and "DEPTH" for the desired depth. Then in ./FPGA_Code/HLS/src/conv2d.cpp on line 31 the depth specified in the pragma "HLS stream variable=delay_line" should be changed to reflect the maximum width of the images to be convolved. Then in ./FPGA_Code/HLS/tb/test_bench.h there are multiple defines to be changed: "KERNEL_FILE", "IMAGE_FILE", and "RESULT_FILE" should be given paths to the files[74] containing the kernel, image, and result (to be used for comparison to ensure correctness) respectively, and "IMAGE_HEIGHT", "IMAGE_WIDTH", and "RESULT_SIZE" should be set to the image height and width of the given test file and the size of the results (the total number of expected outputs from the convolution) respectively.[75]

Once the necessary changes to the HLS code have been made and the RTL has been synthesized and exported, ./FPGA_Code/block_design_creation.tcl can be used to setup the block design and synthesize the bitstream. There are multiple changes that will need to be made. The easier changes are to the FPGA clock frequency on line 76 (currently it is set at 100), and to jobs on lines 100 and 109 (currently set to 16 but should be set to number of CPU that can be used for the process). The other changes to be made are to the paths. Each place where paths may need to be changed there is a comment by that location and can be found by searching for "path".[76] The block diagram that is produced should look like Figure 23.

[73] The top function is "conv2d" found in ./FPGA_Code/HLS/src/conv2d.cpp.

[74] The code found in ./misc/ can provide some assistance in creating these files.

[75] The image height and width should be kept small in order to keep the simulation time from being long, also for sizes larger than 10×10 it will be unlikely that the cosimulation will be able to finish as frequently has difficulty completing for the 10×10.

[76] There are many spots where the paths will need to be changed and one should exercise care to ensure that they are all set correctly.

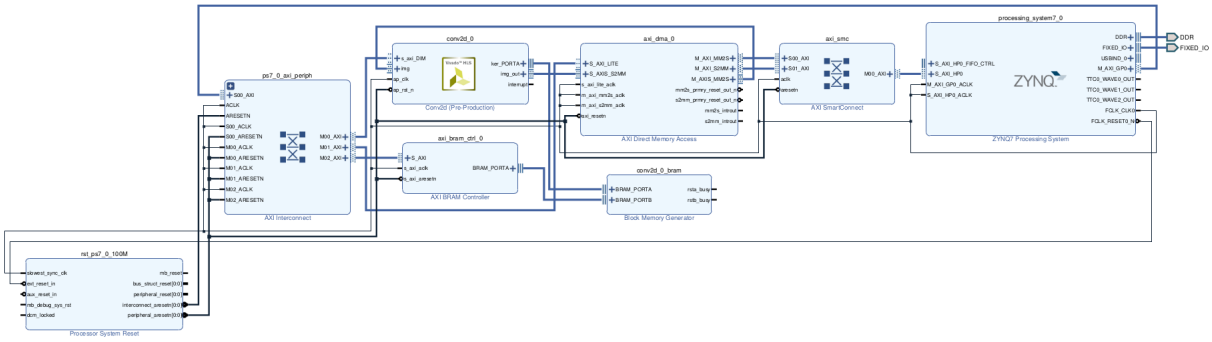


Figure 23: FPGA Block Design

After the bitstream is completed and the hardware is exported, the SDK project can be setup. The code in ./FPGA_Code/SDK can be added to a blank application. A link to "ps7_init.c" should be made to the new application.[77] The board support package for the application should be modified to include the "xilffs" library.[78] In ./FPGA_Code/SDK/main.c on lines 19-21 decide whether to have "TEST" defined or not. "TEST" should be defined when checking to make sure that everything is working correctly and the correct results of the convolution are produced. When used to obtain results for a convolution "TEST" should not be defined. On lines 37-39 are variables to specify the paths to the kernel, image, and result[79] files.[80] After everything is setup and the necessary changes are made the application is ready to be built.

[77]This file should exist in the "design_1_wrapper_hw_platform_0" folder that was created by Vivado and should be in the "Project Explorer" panel on the left of the SDK.

[78]The following changes to the default settings of "xilffs" are recommended: Set "use_lfn" to true and "set_fs_rpath" to 2.

[79]When "TEST" is defined the result file should contain results for comparison, if "TEST" is not defined then it is the location where the convolution results will be written.

[80]These files need to be on the SD card that will be inserted in to the SD card slot of the ZedBoard when used.

Appendix C Technical Specifications

Provided are the specifications for the hardware used to perform convolutions: the University's server and the ZedBoard. This is not all inclusive, and will only provide data on the more pertinent parts of the hardware used.

C.1 The University's Server

OS	CentOS 6.7 Kernel 2.6.32
CPU	Intel® Xeon® Processor E5-2680 v2 (×2) 2.80 GHz 25 MB cache 10 cores, 20 threads $1723.00 - $1727.00
RAM	258,285 MB
Storage	1.8 TB

C.2 The ZedBoard

OS	Standalone
Zynq®-7000 SoC XC7Z020	
CPU	ARM Cortex-A9 666.67 MHz 32 KB L1 and 512 KB L2 cache Dual-core
FPGA	Artix-7 FPGA 53,200 LUTs 106,400 FF 4.9 Mb BRAM 220 DSP Slices
RAM	512 MB
Storage	4 GB SD Card
Price	$474.99

Appendix D Resource Utilization

The following table provides the resource utilization and frequency for the compiled bitstreams for each of the different kernel sizes and depths that were tested.

Kernel	LUT	LUTRAM	FF	BRAM	DSP	BUFG	Frequency
3×3×1	8761 (16.30%)	694 (3.99%)	11691 (10.99%)	5 (3.57%)	43 (19.55%)	1 (3.13%)	125MHz
3×3×2	9454 (17.77%)	694 (3.99%)	13037 (12.25%)	5 (3.57%)	46 (20.91%)	1 (3.13%)	125MHz
3×3×3	9932 (18.67%)	693 (3.98%)	13946 (13.11%)	5 (3.57%)	46 (20.91%)	1 (3.13%)	125MHz
3×3×4	10091 (18.97%)	694 (3.99%)	14531 (13.66%)	5 (3.57%)	46 (20.91%)	1 (3.13%)	111MHz
3×3×5	10250 (19.27%)	692 (3.98%)	15113 (14.20%)	5 (3.57%)	46 (20.91%)	1 (3.13%)	111MHz
3×3×6	10546 (19.82%)	692 (3.98%)	15708 (14.76%)	5 (3.57%)	46 (20.91%)	1 (3.13%)	111MHz
3×3×7	10574 (19.88%)	692 (3.98%)	16281 (15.30%)	5 (3.57%)	46 (20.91%)	1 (3.13%)	111MHz
3×3×8	10734 (20.18%)	692 (3.98%)	16869 (15.85%)	5 (3.57%)	46 (20.91%)	1 (3.13%)	111MHz
3×3×9	10879 (20.45%)	692 (3.98%)	17450 (16.40%)	5 (3.57%)	46 (20.91%)	1 (3.13%)	111MHz
3×3×10	11089 (20.84%)	693 (3.98%)	18033 (16.95%)	5 (3.57%)	46 (20.91%)	1 (3.13%)	111MHz
3×3×11	11207 (21.07%)	692 (3.98%)	18605 (17.49%)	5 (3.57%)	46 (20.91%)	1 (3.13%)	111MHz

Kernel	LUT	LUTRAM	FF	BRAM	DSP	BUFG	Frequency
5×5×1	13948 (26.22%)	693 (3.98%)	19033 (17.89%)	6 (4.29%)	123 (55.91%)	1 (3.13%)	111MHz
5×5×2	15807 (29.71%)	695 (3.99%)	22430 (21.08%)	6 (4.29%)	126 (57.27%)	1 (3.13%)	111MHz
5×5×3	17050 (32.05%)	693 (3.98%)	24906 (23.41%)	6 (4.29%)	126 (57.27%)	1 (3.13%)	111MHz
5×5×4	17570 (33.03%)	693 (3.98%)	26521 (24.93%)	6 (4.29%)	126 (57.27%)	1 (3.13%)	111MHz
5×5×5	17968 (33.77%)	692 (3.98%)	28125 (26.43%)	6 (4.29%)	126 (57.27%)	1 (3.13%)	111MHz
5×5×6	18769 (35.28%)	692 (3.98%)	29701 (27.91%)	6 (4.29%)	126 (57.27%)	1 (3.13%)	111MHz
5×5×7	18852 (35.44%)	693 (3.98%)	31349 (29.46%)	6 (4.29%)	126 (57.27%)	1 (3.13%)	111MHz
5×5×8	19615 (36.87%)	692 (3.98%)	32965 (30.98%)	6 (4.29%)	126 (57.27%)	1 (3.13%)	111MHz
5×5×9	19750 (37.12%)	693 (3.98%)	34572 (32.49%)	6 (4.29%)	126 (57.27%)	1 (3.13%)	111MHz
5×5×10	20144 (37.86%)	694 (3.99%)	36195 (34.02%)	6 (4.29%)	126 (57.27%)	1 (3.13%)	111MHz
5×5×11	20393 (38.33%)	693 (3.98%)	37765 (35.49%)	6 (4.29%)	126 (57.27%)	1 (3.13%)	111MHz
7×7×3	36693 (68.97%)	692 (3.98%)	50567 (47.53%)	7 (5.00%)	165 (75.00%)	1 (3.13%)	100MHz
9×9×3	36998 (69.55%)	692 (3.98%)	52569 (49.41%)	8 (5.71%)	188 (85.45%)	1 (3.13%)	100MHz

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[2] Richard G. Shoup. Parameterized convolution filtering in an fpga. In *Selected Papers from the Oxford 1993 International Workshop on Field Programmable Logic and Applications on More FPGAs*, pages 274–280, Oxford, UK, UK, 1994. Abingdon EE&CS Books.

[3] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.