Department of Electrical, Computer, and Software Engineering

Part IV Research Project

Final Report

Project Number: 20

Reconfigurable Hardware Accelerator for Embedded Artificial Intelligence

Nikhil Kumar

Project Partner: Cecil Symes

Project Supervisor: Dr Morteza Biglari-Abhari

14/10/2021

Declaration of Originality

This report is my own unaided work and was not copied from nor written in collaboration with any other person.

Name: Nikhil Kumar

Table of Contents

1. Abstract

In this report, an investigation into providing highly parallelized, automated, and reconfigurable FPGA accelerators for embedded neural network acceleration was conducted. Experimentation was conducted on the TensorFlow Lite machine learning framework and SYCL programming abstraction layer for heterogenous computation. The resulting implementation show a clear and simpler design workflow for generating portable accelerators for convolutional neural networks. The performance gains from parallelized computation of various neural network operations indicated benefits from using abstraction layers such as SYCL, with an emulated performance increase of x7.6 in comparison to CPU implementation. However, due to setbacks from external situations such as the COVID-19 Pandemic, the implementations were not able to be compiled and synthesized to an FPGA target device for real world testing and benchmarking.

2. List of Figures

3. List of Tables

4. Division of Work

Project work was divided between the both of us. Nikhil Kumar focused on the software elements for machine learning, automated accelerator design and integration of software to hardware. Cecil Symes focused on the hardware design of the accelerator, as well as the HPS and overall system integration

5. Introduction & Background

Machine Learning has become one of the most influential and prominent research areas in the last decade. The recent increase in interest has led to the research, development, and deployment of various machine learning algorithms in a wide variety of fields. This has been due to the improving performance of these algorithms, which have become increasing more accurate at various tasks. In areas such as Computer vision, machine learning algorithms such as Convolutional Neural Networks (CNNs) have shown to be effective in being able to identify and classify elements and objects inside images [1]. In turn, CNNs are being used in autonomous vehicles, facial recognition, and other areas where images are being processed for information and context.

With the utility of CNNs comes at the cost of high computational complexity. CNNs operate by convolving an input image using multiple convolution kernels [2]. This means there is a 100s of millions of multiplication and addition computations as well as high memory access and bandwidth usage.

Thus, CNN operation in software applications require efficient accelerator hardware to improve the performance on various platforms. These neural network accelerators can often be realized with general purpose graphics processing units (GPGPUs), due to the parallel processing capabilities inherent to GPGPUs [3]. By having more computations executed in parallel, these accelerators take advantage of Gustafson's law of parallel processing[4].

As NNs have become more prolific, more specialized hardware for machine learning has been developed in recent years. Application specific integrated circuits (ASICs) such as tensor processing units and neural processing units have been developed and deployed for various systems such as servers, desktops, and even mobile devices [5]. This is due to the inherent shortcomings of using GPGPUs for NN acceleration.

Due to the millions of computations associated with any network operation, state of the art NNs are difficult to implement and operate in embedded environments [6]. Embedded and edge systems are characterized by systems which have constrained computational resources. This is often because of the environments which these systems typically operate in, such as IoT devices. Embedded applications can derive immense utility discussed previously from CNNs [6]. The application of CNNs with high speed operation, without the need for large bandwidth connections would provide breakthroughs in a multitude of areas, such as autonomy in unmanned drones and vehicles.

Accelerators have yet to be as prevalent inside low power embedded systems, thus implementations of NNs and CNNs are not widely available. Field Programmable Gate Arrays (FPGAs) provide the ability to create accelerator ASICs for NNs without the requirement of manufacturing new silicon devices. In turn, accelerator development for low power embedded systems can be implemented using FPGAs.

6. Literature Review

Neural networks and accelerator design are a very active area of research, due to the immense utility NNs provide. The literature review conducted on the current state of research and development in the field of neural network design and accelerator design can be broken down into three research areas.

6.1. Neural Network design

Generally, Neural Networks have high computation and memory requirements, with often 100s of millions of parameters associated with a single network. Thus, research into neural networks have focused on providing high levels of accuracy for a smaller number of parameters with in the neural network model.

Smaller CNN architectures can provide constrained embedded systems the ability to operate CNNs as the smaller memory footprint of these architectures can fit embedded devices. Neural Networks design such as SqueezeNet [7] and MobileNet [8] are CNNs which have reduced memory footprints in comparison to larger networks, whilst providing comparable accuracy.

Optimization techniques such as quantization [9] for parameters inside a Neural network model is also used to further reduce the computational and memory costs associated with networks. Floating point operations can be costly operations inside low power devices; therefore, the floating-point operations are converted into lower resolution fixed point operations, resulting in a slight loss in accuracy, but reduction in computational complexity.

6.2. Accelerator Design

Convolutions are generally the most common and computationally intensive operation for a CNN [10]. Therefore, for more efficient operation of CNNs, accelerator design has been focused on providing efficient hardware units for calculation and memory access operations in convolution operations [11].

Designs such as Eyeriss [12], [13] make use of pipelined operation of systolic / spatial arrays of processing elements. Making use of processing units allows for parallel processing of multiple operations at a given time.

The use pipelined operation of processing elements, with dataflow in between each element then allows for memory efficient acceleration of neural network operation. Local memory reuse between processing elements allows for a reduction in the Von Neumann bottleneck associated with separated memory and computation elements. By leaving data flowing through multiple elements in the pipeline, the overall operation of the accelerator will require less memory accesses, thus less memory bandwidth [11]. This improves the overall performance of the accelerator.

Hardware accelerators are required to have specific efficient designs to operate neural networks. But due to the ever-growing number of neural network designs, accelerator design can become a difficult task from the perspective of hardware design. Efforts in research have been made to try and automate the accelerator design process to use NN architectures to automatically inform and generate the accelerator design.

6.3. Higher level implementations of accelerators

By using high level synthesis methods for accelerator hardware, the accelerator can be automatically generated to Register Transfer Level (RTL) elements from high level code implementing a neural network. This can then be used to operate networks from frameworks such as TensorFlow, which allows easier implementation of neural networks on FPGAs [14]. Thus, when creating a network and creating an accelerator for neural networks, this option for implementation provides a possibility for rapid development and deployment of NN and accelerators.

OpenCL is used to write code for heterogeneous systems, allowing to use C/C++ to operate CPUs, GPGPUs and any other OpenCL compliant device [15], [16]. FPGAs can implement OpenCL interfaces which in turn can be a pathway for integration of TensorFlow neural network frameworks to accelerators using the acceleration abstraction known as delegates [17]. There has also been work in automatically generating an entire accelerator from the neural network design itself, however, these are outside the scope of the project.

Due to the relatively new research into machine learning and neural networks, there has been less of a focus on implementation on low power devices. This leaves a lot of Neural Network frameworks targeting GPGPU parallel processing languages like CUDA and using x86 host accelerator architectures. These elements are not common on embedded platforms, which make use of RISC based processors such as ARM.

Because of this limitation, to implement Neural networks in embedded systems, entirely custom implementations for these networks must be implemented. This represents a barrier for NN operation inside embedded applications, as the development of NN and accelerator architecture must be done in tandem, such that the NN application can use the accelerator hardware. NVDLA [18], is a custom hardware and software stack for a NN accelerator. Its implementation requires an entire custom accelerator software stack to interface with NNs, and custom hardware acceleration unit design. Thus, to keep pace with the development of NN, a demand for customized, specialized NN accelerator designs which can be automatically generated based on the architecture of a NN is desirable, as it removes some of the work required to development an accelerator suitable for the chosen NN and target device.

7. Statement of Research Intent:

The purpose of the project was to provide an accelerator on a reconfigurable FPGA platform, which can improve performance of neural network convolution operations. Convolutions were mainly targeted as the operation is the most common in CNN operation. The result of developing an accelerator on an FPGA will open a pathway for more neural networks running on low power, low-cost devices. The focus on automated accelerator design based on Neural Network architecture allows for rapid development and deployment of NN to embedded systems, which is currently not prevalent as in other systems such as high-performance computing.

8. Methodology

For this project, the approach was decided to explore a multitude of technologies & strategies to implement a neural network and the corresponding accelerator hardware. This was done to explore implementations which would provide a smooth software to hardware pipeline. Doing so would allow us to rapidly implement and iterate designs, given the limited time and resources for executing the project.

To provide a platform which matches the low power embedded devices that are being targeted, a DE1-System on a Chip (SoC) development board from TerASIC was chosen. The board contains a Hard-Core processor in the form of an ARM Cortex-A9 processor, alongside an FPGA unit: a Cyclone-V FPGA from Intel.

The use of a SoC development board chosen as there has been growth in using heterogeneous multiprocessor SoC systems in many embedded applications, especially targeting embedded systems for machine learning applications. The use of a CPU for operating the main functionality of the system and the requirement of efficient computation necessitates neural networks operate on more optimized, parallelized processor architectures, which can be realized using an FPGA.

8.1. Software

8.1.1. TensorFlow

TensorFlow is a neural network framework which has been widely adopted by industry for creating, training, and deploying NNs. TensorFlow represents a framework which can allow developers to quickly generate and deploy neural networks without having to manage any hardware details for deployment. The use of TensorFlow has been realized in web, mobile and some limited IoT applications. The use of TensorFlow allows rapid development of neural network architectures, and thus TensorFlow Lite was chosen to implement a neural network suitable for edge devices. TensorFlow allows efficient deployment of neural networks and NN operation by targeting the OpenCL, CUDA, or similar standards for parallel processing. This is done via an acceleration delegate inside TensorFlow, as mentioned previously. Depending on the target machine, an acceleration device which supports a certain standard such as OpenCL or NNAPI can have TensorFlow operations offloaded on it, thus allowing various operations such as convolutions, accumulation, and activation functions to be performed on accelerator hardware, increasing performance and throughput of the neural network [17].

8.1.2. SYCL

Data Parallel C++ [19] is a new programming language for heterogeneous programming, which makes use of the SYCL standard. DPC++ is part of Intel's oneAPI offering, a cross architecture programming model for accelerated computing. SYCL [20] is a new accelerator programming abstraction layer standard from the Khronos group. The Khronos group is an industry consortium for interoperable standards for hardware and software, having introduced standards such as Vulkan, OpenGL and OpenCL. OpenCL being one of the acceleration targets for TensorFlow and TensorFlow Lite. As such, the SYCL standard allows for heterogeneous code to be written in C++ with optimized hardware acceleration on many various acceleration targets, spanning many different hardware vendors.

SYCL is similar in programming model to OpenCL. There is a host and kernel component of each DPC++ program. This would represent a host being a CPU where the main program is being executed, and a kernel component being an acceleration device. DPC++ works by having the acceleration details be abstracted away, and instead focusing on how computations over data should be parallelized. The use of SYCL allows the DPC++ compiler to compile for various parallel computation architectures, such as a GPGPU, and the implementation of hardware acceleration can be handled automatically by the compiler. SYCL allows many more acceleration devices to be targeted, such as GPGPUs, CPUs, FPGAs, AI/Tensor hardware, and any other OpenCL compliant hardware, using the same single source file. This in turn makes writing code which requires parallelism and hardware acceleration to be very portable.

NN operation is a prime target of implementation using SYCL, as the acceleration details can be abstracted away, allowing the network design to be as portable as possible, whilst being able to be accelerated in any heterogeneous system. Intel provides DPC++ compiler support which allows for compilation and synthesis of DPC++ code for FPGA target platforms. DPC++ for FPGAs allows for implementation of highly parallelized, pipelined architectures specific to operations contained in the program, in this case a NN, which can dramatically improve throughput of a neural network in comparison to GPGPUs or other targets. As such, DPC++ represents a pathway for low power devices, such as the DE1-SoC board to implement NN applications, without the need for implementing entirely custom accelerators and interfaces for hardware acceleration.

8.2. Hardware design

The focus of the hardware design element of this project was to operate convolutions in an efficient manner using RTL design techniques. The use of VHDL to create a convolution accelerator allows us to explore hardware accelerator designs. A systolic array accelerator design was chosen with a weight stationary implementation. Processing elements are generated depending on the requested size and a memory management unit was needed to convert the input data into the correct format. Weights and image data move through the systolic array at different rates to complete a single convolution. Once all computations are completed, the results are outputted and evaluated. More details can be found in the sister report for this project

9. Implementation Results

9.1. TensorFlow Lite

Due to the wide range of neural network designs and architectures, a subset of neural networks was chosen to be trained and tested. Models available for training and deployment for edge devices were chosen and tested by being trained, converted to the TensorFlow Lite (TFLite) format, and then optimized via quantized and then evaluated for accuracy, efficiency, and model size. The models chosen were Efficient-Net's lite0 and lite4, MobileNet-V2 and Inception V3.

Each model was trained using a dataset which is hosted by Google, in this case, images of flowers and the corresponding labels. This image classification task was trained locally on a CUDA enabled GPU, before being validated and tested on a separate validation dataset. The model is converted and exported to the TFLite format. This conversion was undertaken with and without post training quantization [21]. The use of automatic quantization allows the conversion of the floating-point 32-bit numbers to 16 bit floating or 8 bit fixed point numbers, further reducing the model size and improving the model operation on hardware. The quantized and unquantized TFLite formats are then evaluated against the validation data to obtain the error caused by quantization. Some models did not require quantization, whilst other larger networks had improved model size from the operation.

Table 1:Comparison of Model size, before and after quantization

Network	Parameters	TFlite Size (bytes)	TFlite Quant Size (bytes)	Byte reduction
Efficient - 01	3,419,429	4,009,837	3,926,829	83,008
Efficient - 04	13,123,941	14,635,837	14,409,973	225,864
Inception V3	21,813,029	22,397,885	22,325,309	72,576
MobileNetV2	2,264,389	2,782,165	2,714,797	67,368

This experimentation allowed us to obtain information about how the model may operate in an embedded system, and what the requirements would be to implement such a network. The training and accuracy measures for a given parameter count shows an accuracy limit given the number of parameters, which in turn provide a rule of thumb for network design efficiency, embedded size requirements and quantization error.

Table 2: Accuracy of models before and after quantization

Network	Accuracy	Accuracy Quantized	Accuracy Error %
Efficient - 01	0.8852459016	0.8852459016	0
Efficient - 04	0.825613079	0.8228882834	0.272479564
Inception V3	0.92915531	0.923705722	0.5449588
MobileNetV2	0.9182561308	0.9182561307	0

From these experiments, to maximize the accuracy whilst minimizing the size of the model, EfficientNet-01 or MobileNet-V2 would be the CNN of choice to run in an embedded system.

The TensorFlow Lite formats can be transferred and run on any low power processor, such as an ARM processor. The model requires the TensorFlow Lite Library and runtime on embedded Linux implementations common in embedded devices. The plan for the selected models was to be transferred to the DE1-SoC embedded Linux environment with the packaged runtimes required by the models to be installed as well. However, the packages available for TF-Lite are only available on Debian based Linux distributions, and the other alternative is to use a PIP package. This is not ideal for an embedded Linux distribution found in the DE1. Thus, investigation into the

use of CMake for building the TF-Lite libraries for the ARM processor was to be conducted. This was not able to be implemented in the given amount of time.

9.2.SYCL

The use of DPC++ to implement the same operations which are used in CNNs was implemented and tested. Experimentation and testing of various data operations on DPC++ were done to build experience, understand the operations themselves, and understand the internal logic of the abstraction layer, which could then be applied to create efficient kernel structures to run on an FPGA.

DPC++ has a focus on heterogeneous computing and parallel processing. Thus, testing vector and matrix calculations in various approaches was the first to be implemented. A simple vector addition using memory buffer structures was created. Using the same code structure, matrix multiplication using buffers was implemented. Vector addition and matrix multiplication are also two important operations in NN, so it was prudent to have these implementations tested as well.

Table 3:Timing of Matrix Multiplication for parallel and sequential implementations

Matrix Size (elements)	Parallel Time (ms)	Sequential Time (ms)
180000	7	111
720000	12	1114
2880000	168	7563
11520000	2865	56727
46080000	22017	453151

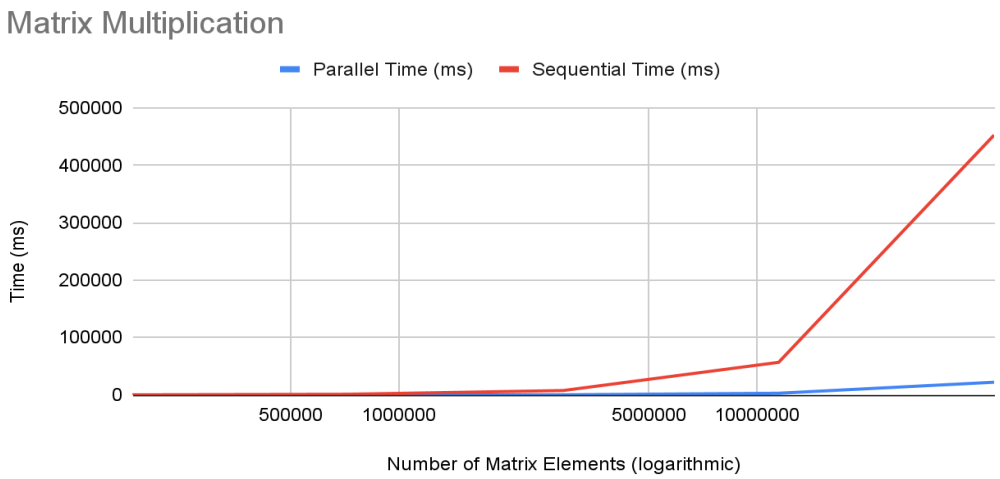


Figure 1: Matrix Multiplication comparison

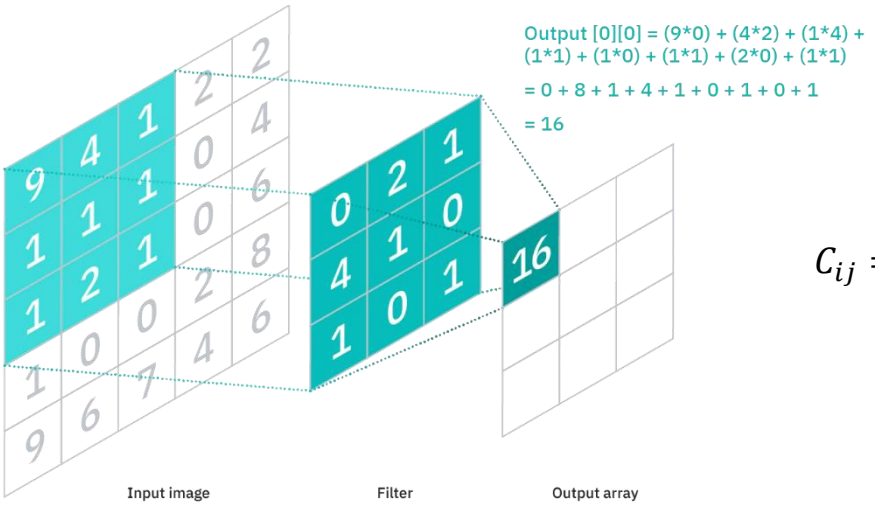
Experimentation was done to compare the time it takes for matrix multiplication sequentially using a naive implementation with no compiler optimization, and the parallel implementation on DPC++. The parallel implementation ran on a 6 core AMD processor, whilst the sequential ran on only 1 core of the processor. From the data, the sequential implementation for matrix multiplication begins to represent a power curve, with a sharp increase in computation time as soon as the matrix begins to grow. The parallel implementation, however, can conduct the operation in a shorter timeframe, with smaller increments in latency as the matrix elements grow. Naive Matrix Multiplication has a time complexity of $\Theta(n^3)$, where n is the length of the matrix. This power curve time complexity is not ideal for any software application. By having more processors operate in parallel to conduct the same matrix multiplication, through Gustafson's law, the time complexity of the parallel operation can be reduced to $\Theta\left(\frac{n^3}{6}\right)$. Gustafson's law states that a theoretical speedup of operation in a task can expect to improve given that there is an increase of resources. This speedup relates to how much of the task benefits from an increased in parallelized operation. The operation of matrix multiplication is highly parallelizable, thus benefits from more computation resources operating in parallel. Also, to note is the cache efficiency of parallel vs single core performance. In smaller matrices, the data elements of the matrices can be easily fit into the L3 and L2 caches of the single processor core. This means that the core has faster access to the data elements, due to the high speed of the cache memory. Although, once these matrices begin to increase in size, there is degraded performance as the core must start to access the RAM to retrieve more data elements. RAM is far slower than L3 cache, and thus there is more time taken to access memory elements. The parallel implementation can have more the matrix elements

distributed between all cores of the processor, thus there is less memory access issues due to the matrix being partitioned across the cores. This reduces the Von Neumann bottleneck common in single core computations.

The parallelizing nature of SYCL/DPC++ in matrix operations can be translated onto convolutions. Image convolution is similar in operation to matrix operations, and thus using a similar structure, an image convolution kernel was implemented and tested.

9.2.1. Formula for Convolution operation

Convolutions inside a CNN is done using a 2D matrix, often containing colour values. This input represents an image that is to be convoluted and information about the image extracted, such as what the image is of. The input layer is often of sizes 300x300; although this can be application dependent. The input layer then has a convolution operation undertaken, with the use of a matrix which contains weight values. This matrix is typically smaller than the input layer and "sweeps" across the entire input layer. This means that the weight matrix acts as a window which moves across the input layer during the convolution operation. The weight matrix values are matched with all input values, of which pointwise multiplication of the input values to the weight matrix are summed. This operation is often known as a MAC (multiply & accumulate) operation, where there are multiple multiplications conducted between the value of an input element and its corresponding weight, before being summed or accumulated all together to produce a single scalar value. This results in an output matrix which contains values from convolving the entire input matrix by the weight matrix. The equation for a convolution can be found below, where the output matrix of C is equal to the multiplication of the input matrix, a, and weight matrix, w. The values of these multiplications are then summated.



$$C_{ij} = \sum_{m=i}^{i+k} \sum_{n=j}^{j+k} a_{mn} \cdot w_{mn}$$

Figure 2: Convolution Operation [2]

9.2.2. Convolution operation in DPC++

The parallelization of the convolution operation was done via the abstraction of DPC++ Work Groups[22]. Work groups are an abstraction of a processing unit. It is analogous to a software thread, such that they represent a piece of work which can be operated independently by a single compute unit. This generally consists of operations over a slice of data from a data block such as a matrix, often with the data elements having spatial or temporal locality. This can be seen in *figure 2.*

By breaking down an operation, such as a convolution into smaller work group elements, the operation can become parallelized. Thus, the design of how the work groups should be arranged and operated over the larger data element will determine the efficiency and performance of the kernel operation itself, allowing for acceleration design without the requirement of necessarily dealing with hardware details. However, some code structures and work group formations are better suited for specific hardware designs and architectures, and some considerations are made as to the suitability of one kernel operation over another depending on the accelerator architecture.

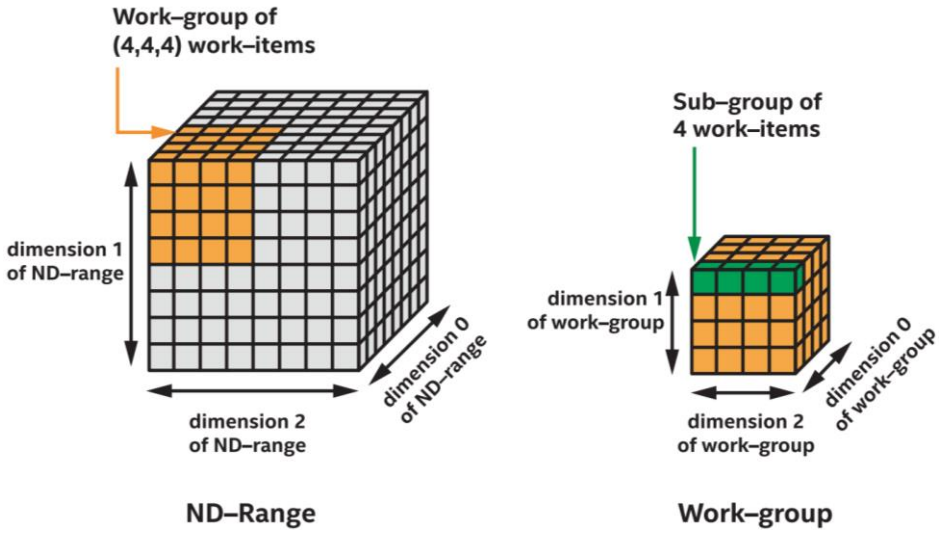


Figure 3: Work Group Abstraction

Due to the highly flexible nature of FPGAs, some kernel designs suitable for other architectures are not as optimized for FPGA architectures. An FPGA can be considered as implementing an entire program at once, with dedicated hardware for each program section, rather than having sections of hardware be reused, such as in a CPU/GPGPU architecture. This makes FPGAs incredibly suitable for implementing high levels of parallel processing, more so than traditional architectures. The ability to generate custom pipelines with various bit widths and custom operation hardware without the need to adhere to any specific architecture elements leads to highly efficient accelerator

performance. Single task kernel structures could be used for FPGA target kernels but for the application of matrix multiplication, the task benefits from multiple pipelines, rather than a larger pipeline implementing a single or multidimensional loop.

The use of buffers for memory management between the accelerator and the host device was done to simplify the management of data flow between the two devices. Unified Shared memory, whilst an option, may lead to degradation of performance due to unoptimized memory access across the bus. Thus, it is prudent to leave the management of data as automated as possible, to maximize local memory reuse during the convolution operation.

Like CPU implementations, the use of the "ND_Range" with a parallel for loop was used in splitting the input matrix into chunks and distributing the elements as different work groups. As such, the input layer is split up across multiple work groups, which is compiled down to multiple compute nodes implementing convolution over sections of the input layer. This simple implementation is pragmatic in its use of parallel processing and allows the code to be as portable as possible. It also allows a fairer comparison between CPU and FPGA. For a convolution operation running on a 6-core processor, the average timing for 1 convolution is

Table 4:Convolution timing on different hardware targets

Image Size	Single Core(ms)	CPU Parallelized (ms)	FPGA Emulation (ms)
Small (255x255)	50	102	11
Medium (1280 x720)	354	114	15
Large (1602x2298)	1585	130	29

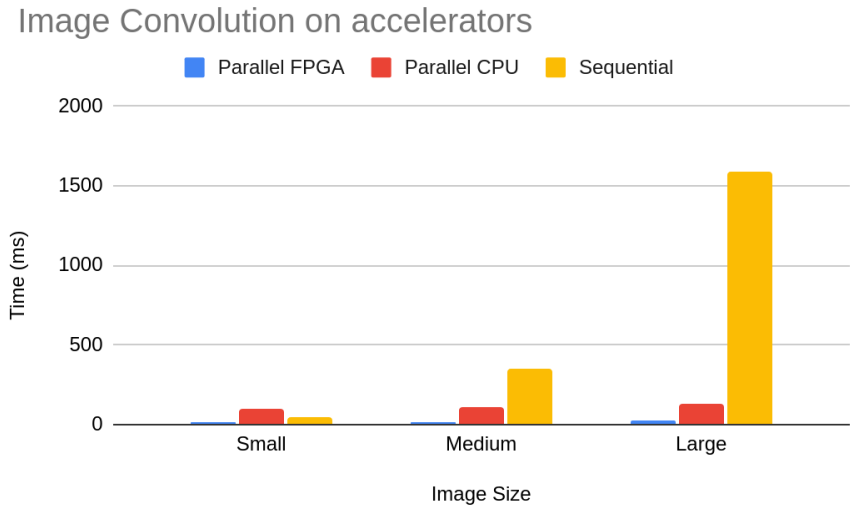


Figure 4: Graph of Convolution Timing

Emulation of FPGAs on a CPU is not completely transparent, and as such, cannot accurately ascertain the actual performance of an FPGA accelerator design. From the data presented, for efficient calculation and consistent timing, the use of highly parallelized kernels implemented in DPC++ shows clear signs of improving the acceleration of neural networks.

With highly parallelized implementations of convolution implemented on an FPGA target, there is a massive potential for computational efficiency in convolutional operations. As stated previously, a FPGA can provide extremely parallelized, pipelined designs. More so than having a multicore processor or GPGPU, due the FPGA implementing more compute cores acting in parallel. In Gustafson's law, given that convolutions can be parallelized, the potential increase in speed can be drastic. However, to note is the clock speeds at which FPGAs operate at. The Cyclone V and Arria 10 FPGAs have maximum clock frequencies of 125Mhz and 800 MHz depending on the application. These are surpassed by modern processors, where it is common to have 1GHz clock speeds and above.

For proper hardware evaluation, the use of a custom BSP would be required for compilation to Cyclone V. Another issue is the intel target for PAC applications, which is more focused on X86 host device systems. It is possible to target Cyclone V BSP, but another issue is of software licensing, with FPGA targets requiring Quartus Pro. The use of Quartus Pro was considered, though due to the COVID-19 Pandemic, the installation and licensing was not feasible, and due to the limited time, was not conducted. As such, a proper compilation and synthesis to the DE1-SoC to have the convolution code, and by extension, an entire neural network operation to work on the development board was not conducted.

Considering these values in a hypothetical scenario with Inception V3, which has 98 [23] in its architecture and DPC++ Convolution, taken as is for a single inference of the network is calculated at

Table 5: Calculated timings for convolution in Inception-V3

96 convs	Single Core (s)	6 Core CPU (s)	FPGA Emu (s)
255x255	4.8	9.79	1.056
1280 x720	33.984	10.944	1.44
1602x2298	152.16	12.48	2.784

The calculation does not consider the reduction of input sizes as the convolution occurs at different layers of the CNN. This calculation considers no parallelism between the convolution layers, all convolutions are done sequentially with no pipelined operation. It also factors in bus communication between each convolutional operation, which would not be the case in a proper implementation on an FPGA, in which memory accesses are optimized for local reuse.

To further achieve a more accurate number, Inception V3 has many of the convolution operations occur in parallel to one another. This means that for a single layer in the Inception V3 architecture, there are often 4 convolutions which are independent of each other operating at any one time.

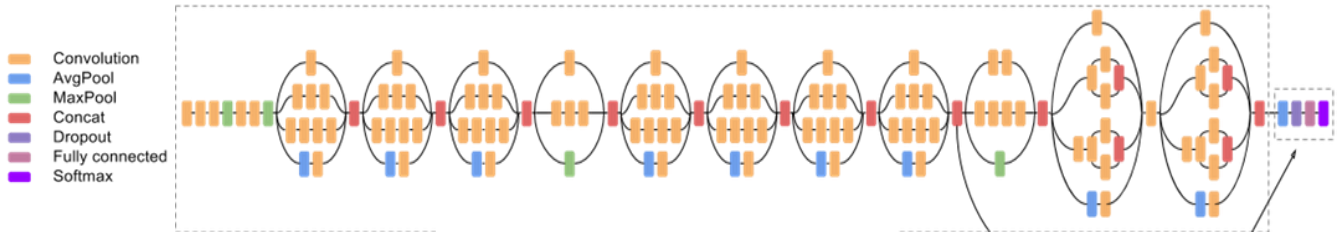


Figure 5: Architecture of Inception-V3

To account for this, the critical path of convolution operations which will provide the longest length of convolution operations sequentially can be obtained. The length of this convolution chain is 46 convolution operations. We can assume that at all stages of the network on an FPGA implementation, that all convolution operations will occur in parallel, and only the convolutions operating sequentially make up this critical path. For comparison purposes, an assumption that the CPU can also operate these parallel convolutions as well was made, which may not be the case for real world performance.

Table 6: Inception critical path convolution timing

No. of convolutions	CPU(s)	FPGA emulation(s)
46 convolutions	5.244	0.69

These numbers, whilst not real-world performance numbers, are indicative of the likely potential speed up of using high parallelizable FPGA designs using SYCL. The calculation shows that for an FPGA implementation there is sizable increase in performance of x7.6 over the CPU implementation. But despite that, due to the issues mentioned previously, such as clock frequency bus management, and the fact that this is purely an estimate gathered from the FPGA Emulation, this would not be the real-world performance. NN performance increases of x4.9~x15.4 on

FPGAs in comparison to CPU implementations are common, and some research has demonstrated upwards of x60 increase in performance [24].

9.3. Accelerator design

The accelerator has been designed to efficiently operate convolution operations. This is done via a systolic array design using PE arrays. More details can be found in the sister report for this project. Currently, the implementation of the VHDL accelerator can be run using the connection between the HPS and FPGA. Connection to the accelerator is done via an Avalon bus. There is no standard interface to provide a neural network with an accelerator interface. Standard accelerator interfaces for TensorFlow Lite are through using OpenCL or NNAPI. This entails providing the accelerator as a delegate for acceleration to specific kernels. Due to time constraints, limited manpower and the setbacks due to the Pandemic, this standard interface for a neural network was not able to be generated.

10. Analysis

From the exploration, understanding and experimentation of the various technologies that have been discussed a clearer understanding of implementing general CNNs to embedded hardware can be understood.

TensorFlow Lite and the acceleration infrastructure provided to developers allows for neural networks to be operated on edge devices. The interfaces to implement these accelerators can be complex as thus, to be able to implement an accelerator effectively and rapidly, an OpenCL or similar interface can be used.

The use of DPC++ can provide this standard implementation using SYCL and the hardware abstraction that it provides. Integration of TensorFlow NN operations to DPC++ kernels for operation of the networks can provide a pathway to generate specialized acceleration architecture for FPGAs, and in turn allow for low embedded systems to operate computationally expensive CNNs.

This workflow has been echoed in industry as well, as of writing, oneDNN, which is built of the SYCL standard and DPC++ runtime has been released by Intel. It provides a standardized way of generating NN which can be accelerated by CPU/GPGPU architectures. It's interoperability with DPC++ may also provide an avenue of CNN acceleration on FPGA targets, however this is yet to be supported by Intel. Intel TensorFlow[25] is also another oneAPI interpretation of TensorFlow, but it is targeted towards high performance computing applications.

11. Future Work

Because of the Pandemic, many planned implementations were not able to be realized. This is detrimental to the overall project goals, and as such future work would be to conduct the planned actions, provide metrics for accelerator designs and then to improve the accelerator designs.

Currently, the VHDL accelerator design can conduct fixed size convolutions, with some work to integrate the design into the DE1-SoC. The next course of action would be to fix the issues between the HPS and FPGA communication, such that the ARM can begin a convolution operation and retrieve the results from the accelerator. This will then allow rudimentary operation of convolution operations on the DE1-SoC.

The design of the accelerator and the interface between the two processors can then be further expanded. The ARM core can then have the TensorFlow Lite models operate solely on the ARM processor. From here, a translation layer can be created which can take any convolution operation required of the neural network to be translated to match the interface provided by the accelerator. This approach has been previously realized in previous research [26].

With further implementation, a TensorFlow Lite acceleration delegate interface can be realized between the processor and accelerator. This in turn will open the neural network implementation possibilities and allow for true operation of neural network acceleration without the need for translation layers between the CNN and hardware.

The DPC++ accelerator will require a Quartus Pro license to then synthesize the accelerator to a custom BSP package which represents the Cyclone V and ARM processor. This will allow accurate acceleration metrics and provide a platform to extend the DPC++ convolution accelerator to other operations contained inside the neural network. If this implementation using DPC++ or Intel FPGAs is unfeasible, there are other SYCL implementations which can support this workflow. CodePlay and Xilinx have provided similar SYCL implementations such as ComputeCPP[27] and triSYCL[28], respectfully. These implementations fulfill the same niche as DPC++ and the implementation that has been undertaken in this project can be ported over to a Xilinx based SoC [29]

From here, the project goals can then be truly realized, as the benchmarking of the accelerators on the development board will allow us to obtain metrics on accelerator performance, power usage, efficiency, and chip area usage. It

is then possible to determine the suitability of these accelerators designs in embedded applications and provide clear conclusions as to the suitability of automated accelerator design.

12. Conclusion

Through the investigation of several technologies implementing NN and accelerator infrastructures, an application of neural network architectures was conducted to investigate neural network acceleration. The use of SYCL was experimented with to try and provide an automated compilation and synthesis flow to low power embedded FPGA targets. As a result, SYCL can provide efficient acceleration kernels which can be implemented on to a multitude of acceleration architectures and found that through FPGA simulation that substantial acceleration can be achieved of x7.6 for NN on FPGAs. Whilst implementation to concrete accelerators to FPGA hardware was not able to be conducted, this project shows a clear pathway to implement accelerator architecture which can be utilized by industry standard neural networks, with a clear potential for accelerating neural network operation in embedded devices.

Acknowledgements

Thanks to everyone who had supported and helped over the course of this project.

Special thanks to Dr Morteza Biglari-Abhari for supervising the project and advising myself and my team member over what was a very difficult project to undertake, with many different avenues of research.

Thanks to Dr Maryam Hemmati for also providing guidance with regards to high level synthesis implementation.

Huge thanks to my team member, Cecil Symes, who faced the difficult task of a hardware accelerator implementation on top of having been tasked with the overall system integration. It's his hard work that made the next steps for this project clearer.

References

[1] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015, doi: 10.1007/s11263-015-0816-y.

[2] "What are Convolutional Neural Networks?," Jan. 06, 2021. https://www.ibm.com/cloud/learn/convolutional-neural-networks (accessed Oct. 14, 2021).

[3] "NVIDIA cuDNN | NVIDIA Developer." https://developer.nvidia.com/cudnn (accessed Apr. 19, 2021).

[4] J. L. Gustafson, "Reevaluating Amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988, doi: 10.1145/42411.42415.

[5] "Apple introduces eighth-generation iPad with a huge jump in performance," *Apple Newsroom*. https://www.apple.com/newsroom/2020/09/apple-introduces-eighth-generation-ipad-with-a-huge-jump-in-performance/ (accessed Oct. 15, 2021).

[6] L. Andrade, A. Prost-Boucle, and F. Pétrot, "Overview of the state of the art in embedded machine learning," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2018, pp. 1033–1038. doi: 10.23919/DATE.2018.8342164.

[7] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," *ArXiv160207360 Cs*, Nov. 2016, Accessed: Apr. 18, 2021. [Online]. Available: http://arxiv.org/abs/1602.07360

[8] A. G. Howard *et al.*, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *ArXiv170404861 Cs*, Apr. 2017, Accessed: Oct. 13, 2021. [Online]. Available: http://arxiv.org/abs/1704.04861

[9] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A White Paper on Neural Network Quantization," Jun. 2021, Accessed: Oct. 15, 2021. [Online]. Available: https://arxiv.org/abs/2106.08295v1

[10] Y. Chen, T. Krishna, J. Emer, and V. Sze, "14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, Jan. 2016, pp. 262–263. doi: 10.1109/ISSCC.2016.7418007.

[11] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017, doi: 10.1109/JPROC.2017.2761740.

[12] Y. Chen, T. Yang, J. Emer, and V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," *IEEE J. Emerg. Sel. Top. Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019, doi: 10.1109/JETCAS.2019.2910232.

[13] M. Sankaradas *et al.*, "A Massively Parallel Coprocessor for Convolutional Neural Networks," in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, Jul. 2009, pp. 53–60. doi: 10.1109/ASAP.2009.25.

[14] D. H. Noronha, B. Salehpour, and S. J. E. Wilton, “LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks,” Jul. 2018, Accessed: Oct. 15, 2021. [Online]. Available: https://arxiv.org/abs/1807.05317v1

[15] D. Wang, K. Xu, and D. Jiang, “PipeCNN: An OpenCL-based open-source FPGA accelerator for convolution neural networks,” in *2017 International Conference on Field Programmable Technology (ICFPT)*, Dec. 2017, pp. 279–282. doi: 10.1109/FPT.2017.8280160.

[16] Z. Bai, H. Fan, L. Liu, L. Liu, and D. Wang, “An OpenCL-Based FPGA Accelerator with the Winograd’s Minimal Filtering Algorithm for Convolution Neuron Networks,” in *2019 IEEE 5th International Conference on Computer and Communications (ICCC)*, Dec. 2019, pp. 277–282. doi: 10.1109/ICCC47050.2019.9064413.

[17] “TensorFlow Lite Delegates.” https://www.tensorflow.org/lite/performance/delegates (accessed Oct. 15, 2021).

[18] G. Zhou, J. Zhou, and H. Lin, “Research on NVIDIA Deep Learning Accelerator,” in *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, Nov. 2018, pp. 192–195. doi: 10.1109/ICASID.2018.8693202.

[19] “Data Parallel C++ Language,” *Intel*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/data-parallel-c-plus-plus.html (accessed Oct. 15, 2021).

[20] “SYCL - C++ Single-source Heterogeneous Programming for Acceleration Offload,” *The Khronos Group*, Jan. 20, 2014. https://www.khronos.org/sycl/ (accessed Oct. 15, 2021).

[21] “Post-training quantization | TensorFlow Lite.” https://www.tensorflow.org/lite/performance/post_training_quantization (accessed Oct. 15, 2021).

[22] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. 2021. Accessed: Oct. 15, 2021. [Online]. Available: https://doi.org/10.1007/978-1-4842-5574-2

[23] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception Architecture for Computer Vision,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, Jun. 2016, pp. 2818–2826. doi: 10.1109/CVPR.2016.308.

[24] D. Wu *et al.*, “A High-Performance CNN Processor Based on FPGA for MobileNets,” in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2019, pp. 136–143. doi: 10.1109/FPL.2019.00030.

[25] “TensorFlow* Optimizations on Modern Intel® Architecture,” *Intel*. https://www.intel.com/content/www/us/en/developer/articles/technical/tensorflow-optimizations-on-modern-intel-architecture.html (accessed Oct. 15, 2021).

[26] J. Luo and C. Lin, “Pure FPGA Implementation of an HOG Based Real-Time Pedestrian Detection System,” *Sensors*, vol. 18, no. 4, p. 1174, Apr. 2018, doi: 10.3390/s18041174.

[27] “Codeplay Developer - Products - ComputeCpp CE - Home.” https://developer.codeplay.com/products/computecpp/ce/home (accessed Oct. 15, 2021).

[28] *triSYCL*. triSYCL, 2021. Accessed: Oct. 15, 2021. [Online]. Available: https://github.com/triSYCL/triSYCL

[29] “Single-source SYCL C++ on a Xilinx FPGA,” *Xilinx*. https://www.xilinx.com/video/events/single-source-sycl-c-on-xilinx-fpga.html (accessed Oct. 15, 2021).