CS/ECE 752 Project Checklists

Final Project Report

Please use this checklist to confirm the completeness of your final project report. This checklist should be submitted with your final report.

Group Number: 15

Project Title: Building SqueezeNet CNN Accelerator on Gem5-SALAM

Project Team: Balaji Adithya Venkataramana, bvenkatarama@wisc.edu
Subhadip Kundu, kundu8@wisc.edu

	Final Report Writing Requirements	Feedback
☐	**Scope.** The report is 15-20 pages, double-spaced, 10-point font.	
☐	**Development and Organization.** The report includes the required sections, and each section has been revised with a reader's need for coherence in mind.	
☐	**Editing.** The report has been edited to limit mechanical errors (punctuation, spelling, capitalization) and grammatical errors.	

	Final Report Components	Feedback
☐	**Abstract.** Abstract includes: • statement of a focal problem • explanation of the relevance of the problem for engineers and researchers • explanation of the proposed solution to the problem • explanation of the impact of the proposed solution on the subfield	
☐	**Background.** Background section includes: • discussion of relevant background information (with citations), • thematic organization (organization around themes or concepts rather than sources?) organized around techniques and concepts • explanation of how each theme is related to the project	
☐	**Methodology.** Methodology section includes:	

	• description of the methods used • explanation of the setup for each method • justification for the setup for each method (e.g., why this method is appropriate)	
☐	**Related Work.** Related Work section includes: • Discussion of important publication on the project topic • Explanation of the ways in which the project is related to prior research	
☐	**Conclusion.** Conclusion section includes: • Description of project takeaways	
☐	**Future Work.** Future Work section includes: • Explanation of next steps for research on the topic	
☐	**Style.** The writing has been edited for organization and flow of sentences, paragraphs, and sections appropriate word choice grammar and mechanics format (double-spaced, 10-point font) length (15-20 pages)	

Comments:

ABSTRACT

It is now widely admitted that the increasing complexity and demands of modern AI/ML workloads necessitate using hardware accelerators like GPUs, TPUs, FPGAs, and ASICs. These accelerators provide massive parallelism for computationally intensive tasks and high throughput for large datasets. Recent research has shown that it is a challenge to meet the power and area constraints of Artificial intelligence (AI) and Machine Learning (ML) workloads as they grow in complexity.

Image classification is a core task in many AI/ML applications, and it is particularly demanding in terms of accelerator applications due to several factors related to the complexity of image data, the required computations, and the need for real-time performance. MobileNetV2 and SqueezeNet are two popular CNN architectures designed specifically for mobile and edge devices, where computational resources (CPU, memory, and power) are limited.

For simple tasks or smaller datasets, CPUs are often sufficient. For large-scale machine learning models, accelerators are typically the better option. In our report, we try to evaluate the idea that multiple heterogeneous accelerators are advantageous when the workload is highly diverse, with different layers or tasks requiring different types of acceleration. In AI and ML workloads, accelerators are often used alongside traditional CPUs. GEM5 -SALAM helps simulate such heterogeneous systems to optimize performance in terms of area and power.

The power-area tradeoff in accelerators for image classification is an ongoing challenge and lies in achieving high performance while staying within the power and area constraints of the system.

We have chosen to model the SqueezeNet architecture within the GEM5 SALAM simulator and benchmark specific layers, such as convolution, against a modern out-of-order CPU. Our results indicate a performance improvement of approximately 500 times relative to the CPU (Deriv-O3CPU provided by default which is a state-of-the-art ARM-based out-of-order processor) working at 100MHz operating frequency. Furthermore, we illustrate how performance gains can be achieved by adjusting parameters, such as the system's clock frequency, with minimal impact on power consumption and area.

INTRODUCTION

In recent years, the trend in modern system-on-chips (SoCs) has shifted toward heterogeneous architectures that combine CPUs with multiple domain-specific accelerators. This approach addresses the limitations of homogeneous systems by leveraging the strengths of different hardware components for specific tasks. A recent Harvard study has determined that the use of accelerators in the modern SoCs has increased significantly to complete compute intensive tasks with high performance and low power utilization [1]. The paper suggests that modern SoCs like Qualcomm's Snapdragon and Apple's A12 Bionic have shown remarkable improvements in performance and energy efficiency for deep learning applications.

For this project, we selected the SqueezeNet architecture, a lightweight CNN model widely used for real-time image classification in mobile and edge devices. SqueezeNet's compact design makes it an excellent candidate for exploring the power and performance tradeoffs in heterogeneous accelerator systems. To evaluate its performance, we utilized the GEM5-SALAM simulator, which offers features like LLVM-based accelerator modeling, configurable communication interfaces, and support for scratchpad memory. These capabilities allowed us to simulate heterogeneous systems and analyze interactions between CPU and accelerators efficiently.

Instead of designing a unified accelerator system, we opted to divide the model into multiple accelerator clusters. This approach enhances scalability, reduces communication overhead, and enables parallel execution. By categorizing layers into clusters, such as the head cluster for convolution and max pool accelerators and body clusters for subsequent layers, the model becomes more modular and reusable. This clustering also minimizes off-chip memory transfers and allows for optimized dataflows using techniques like streaming DMA. Through this project, we aim to demonstrate how heterogeneous accelerators can deliver significant performance improvements for AI/ML workloads while balancing power and area constraints. The main contributions of this work are as follows:

- Cluster-based accelerators built to have more reusability in simulating large models.
- Use of stream buffers and stream DMA and other techniques like convolution windowing and loop unrolling inside convolution accelerators to exploit accelerator-level parallelism.

- Increasing the clock frequency results in significant throughput gains while maintaining an efficient power tradeoff.

BACKGROUND

Modern AI/ML workloads are increasingly pushing the boundaries of performance, efficiency, and scalability in computational systems. Hardware accelerators are now integral to handling these workloads due to their ability to provide high throughput for computationally intensive tasks. The shift toward heterogeneous architectures, combining CPUs with accelerators, addresses the limitations of traditional homogeneous systems. One of the most common applications for using such accelerators is in Deep Learning heterogeneous high-performance platforms, as can be seen from an extensive survey conducted recently in 2024, by Cristina et. al. [3]. Hence, for this project, we selected one such CNN model – SqueezeNet, which is commonly used in real-time image classification applications aimed for mobile and edge devices with low power requirements. For the next step, an appropriate simulator model was chosen among the available options to be able to model the heterogeneous CPU-accelerator interactions – and chose gem5-SALAM over gem5-ALADDIN and gem5, because of its faster simulation time and LLVM-based Accelerator Modeling which enables more configurability in terms of accelerator clusters, communication interfaces, DMA options and scratch-pad memory sizes, which can be run on different clock frequencies [4]. Following this, the option of designing the model as a collection of accelerator clusters seemed to be a better option as compared to a unified cluster of accelerators [5]. This is because of – *a) Improved scalability:* By splitting the accelerators into clusters allows for better scaling if the input/ output sizes have to scaled up or down; *b) Reduced Communication Overhead:* Having clusters enable local communication between the accelerators within, hence reducing the overhead of global memory accesses and leads to reduced latencies; *c) Energy Efficiency:* By clustering the accelerators, there is reduced power consumption due to the limited long-distance data transfers from the off-chip memory; *d) Parallel Execution:* Each cluster can handle tasks in parallel, hence the throughput of the system increases; *e) Reconfigurability and Reusability:* By making the clusters generic, they can be reused for different layers in the model and hence reduce the burden on redesigning each accelerator. Finally, as described by studies on streaming dataflow architecture by Tony et. al. [6], using streaming DMA for data transfer from the host to the accelerator cluster will help in reducing overall memory accesses and also parallelize the data flow through accelerators.

METHODOLOGY

The project can be split up into 4 sections/ phases describing the entire accelerator model. Section-I gives an overview of the SqueezeNet CNN and the corresponding functions that will be built to model this CNN as an accelerator. Section-II will describe the intuition behind opting for a cluster mechanism employed in this design, to be able to exploit Accelerator Level Parallelism (ALP) as coined in [1]. Section-III dives into the details on how the accelerators and accelerator clusters are interconnected using Stream-buffers and Stream-DMAs, in essence, describing the data movement throughout the model. Finally in Section-IV, the configuration setup on gem5-SALAM is described followed by which the evaluation details are laid out.

Section-I: The SqueezeNet CNN

SqueezeNet is a lightweight Convolutional Neural Network (CNN) architecture which was designed to achieve high accuracy with considerably small model size [2]. The SqueezeNet CNN architecture broadly consists of the the following layers:

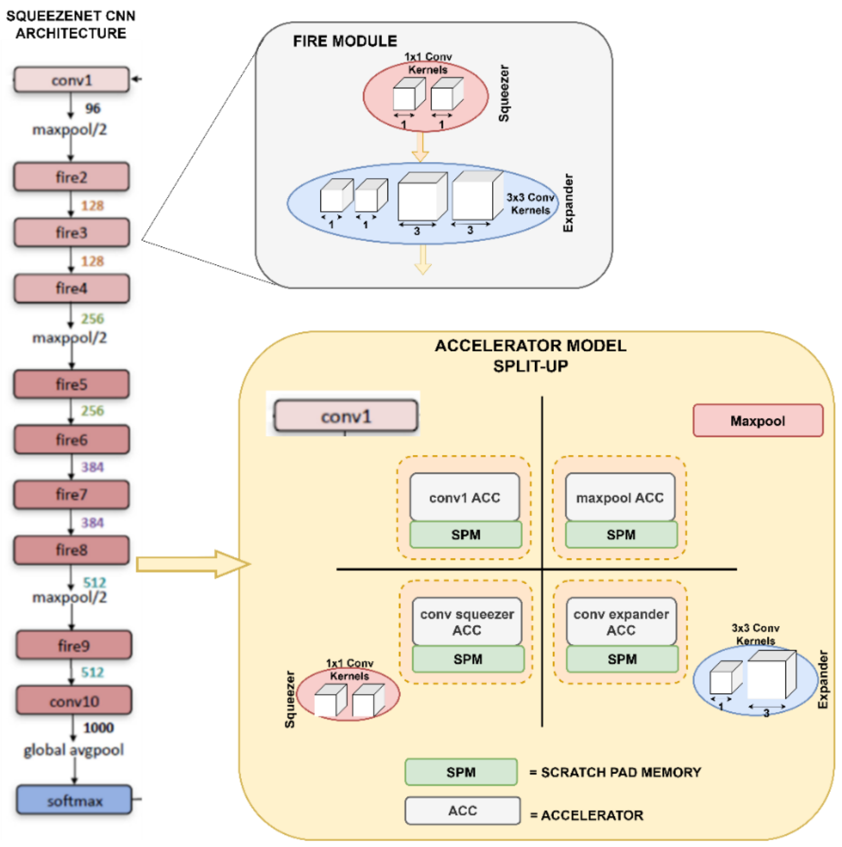


Fig-1: (a) The SqueezeNet CNN Architecture – contains the initial convolution layers, fire-modules and maxpool layers; (b) The break-up of fire-module into constituent convolution operations; (c) Accelerator models in gem5-SALAM corresponding to each function from SqueezeNet CNN.

– a) Input Convolution Layer with stride 2, b) Fire Modules – which contain 1x1 and 3x3 convolution operations with stride 1 and c) Maxpool and Average Pool layers. Each of these layers serve their own purpose – the fire modules consist of a squeeze layer (1x1 convolutions) followed by an expand layer (a mix of 1x1 and 3x3 convolutions), which helps in reducing the total number of parameters and ensure compact model size, with very small weight and bias sizes – ~0.5 MB. Its compact size makes it an ideal candidate for deployment in environments with low power and area requirements such as mobile devices and embedded systems. Since these convolution operations are highly parallelizable, they can be the perfect candidates to be implemented in an accelerator. *Image-1* describes the major accelerator models which have been used to model the entire SqueezeNet CNN. As it can be seen, each accelerator consists of 2 components – a) *The execute unit*, which is used to perform the convolution operation in a parallel manner and b) *the scratchpad memory (SPM)*, which is used to store the weight and bias values which will be operated along with the input feature maps (inputs) to generate the appropriate ouptut feature maps (outputs).

$$fo(m,y,x) = \sum_{n=0}^{N-1}\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} w(m,n,i,j)\times fi(n, S\times y+i, S\times x+j) + bias(m),$$

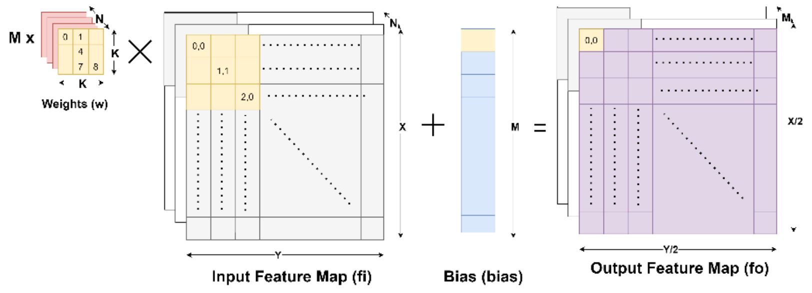


Fig-2: (a) The convolution operation equation; (b) The required elements from input matrix [size = (N,X,Y)], weights[size = (M,N,K,K)] and bias[size = (M)] parameter values to form the output matrix. [size = (M,X/2,Y/2)] (the required elements per cycle is highlighted in yellow)

Section-II: Accelerator-Host Interactions

In this section, firstly, the general flow of control transfer from host CPU to the accelerator will be explored.

System Level Overview

The accelerator model is made of the following components in general: a) Processing Elements/ compute engine; b) Scratch Pad Memory (SPM); c) Communication Interface

The processing elements (PEs) are responsible for carrying out the parallel execution of instruction with the use of multiple functional units for the specific task. (for example, multipliers for multiplication, adders/ subtractors, etc – collectively can be used for the convolution operation).

The scratch pad memory (SPM) is used for storing the necessary data that is going to enter into the accelerator. In case of a convolution operation, this includes – the input feature vector window, the weight matrix which contains the kernels with which the input is getting convolved and the bias matrix which is a parameter added to the convolution output to add an offset for each channel. There is a separate SPM for each of these data points.

The communication interface is responsible for sending/ receiving the data to/from the accelerator/ DMA. There are two variants to this comms interface – a) Direct DMA and b) Streaming DMA. For the purposes of this project, using direct DMA is not relevant as it is not feasible to transfer such huge data from the global memory to the SPM for each computation. Hence the model described in this project, uses streaming DMA.

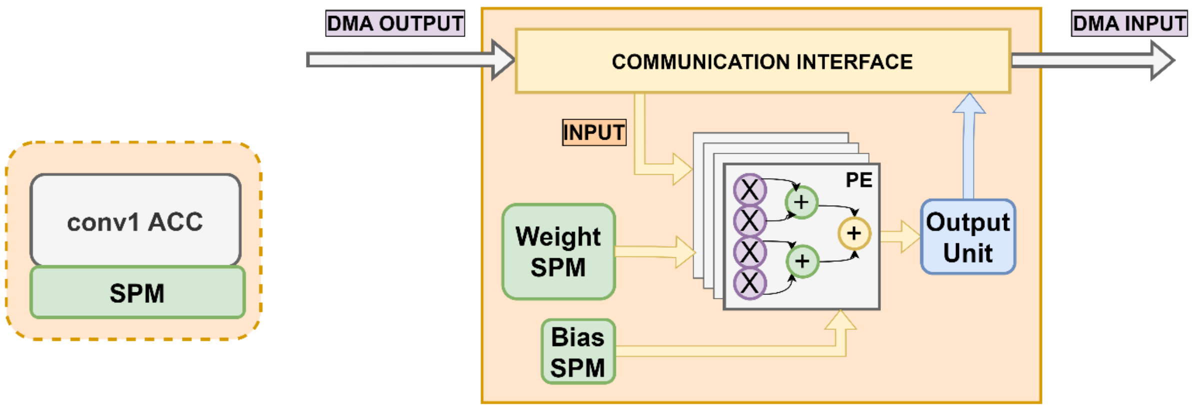


Fig-3: Detailed view of the Accelerator Model and its components – The a) communication interface, b) Weight, Bias SPM and c) The processing Elements/ Functional Units.

The control transfer from host to device consists of multiple steps until the desired execution is done and the host receives back the data. The steps are as follows:

- **Step-1:** Set the appropriate Memory Mapped Registers (MMRs) with correct values to initiate control transfer from O3CPU (host) to the accelerator (device)
- **Step-2:** MMR set done. Control Transfer successful. Initiate DMA data transfer to get the input, weight and bias values.
- **Step-3:** Transfer of data from DRAM (global memory) to the accelerator's respective SPMs (Scratchpad Memories) using the DMA controller through the system bus.

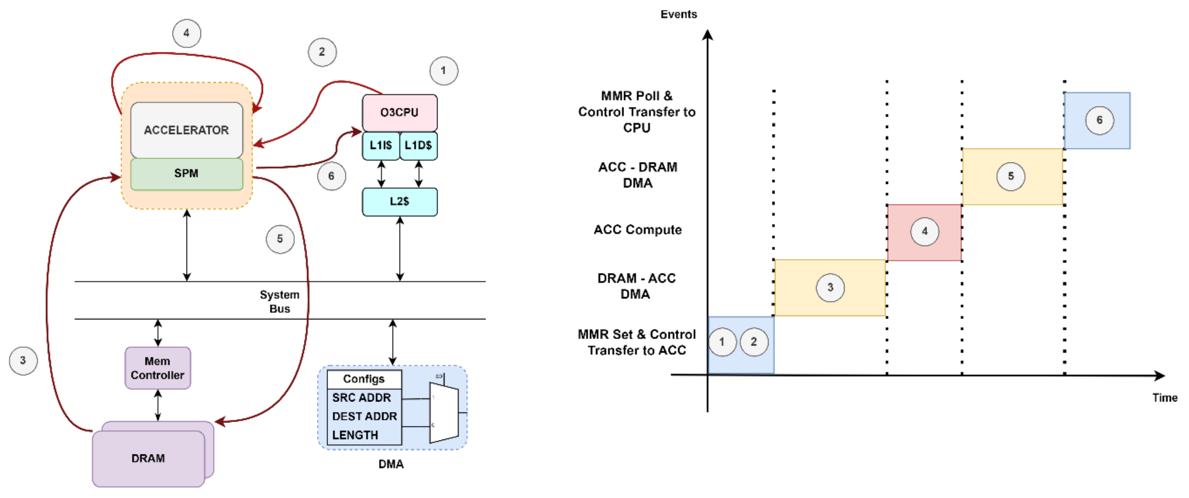


Fig-4: (a) System-level step-by-step interaction between the host (O3CPU) and the accelerator to transfer data and perform accelerated execution. (b) Timing Diagram for the sequence of steps described above.

- **Step-4:** Accelerator execution step. This is the step in which the parallel execution of the instruction occurs within the Processing Elements (PEs) or the functional units.
- **Step-5:** Transfer output data back to DRAM using DMA.
- **Step-6**: Reset the appropriate MMRs and transfer control back to host (where the host is continuously polling the updated MMR values).

However, it can be noted from the timing diagram from *Fig-4* that each of these steps are performed in a sequential manner. This is because the basic accelerator configuration is such that each of these steps are dependent on the previous step to be completed. Especially, it can be noted that Step-3 and Step-5 take up most time on an average as it requires the entire data to be transferred before the accelerator's processing elements can start execution. This is causing severe bottleneck in the throughput of the system because there's more scope for parallelization, as the entire data is not required to start the convolution operation. Rather, it is sufficient to send enough data for the kernel to operate on each cycle before sending in the next set of data on which the accelerator can execute upon. In this manner we can perform "windowed execution" of the input data to get the required output. In order to facilitate this windowed operation, this project moves towards the use of streaming DMA to be able to send only the required window of data to begin execution (or the Processing Elements' convolution operation).

Section-III: Stream DMA and Stream Buffer

As mentioned in the previous section, in order to facilitate the “windowed execution” operation to both improve the overall accelerator model’s performance as well as reduced power consumption, the communication interface between the memory and the accelerator employs stream DMA [6]. In this type of data transaction, there is a constant stream of input data that enters the accelerator. This data over time fills up the initial values – such as weights and biases (which remain unchanged throughout the execution process for a given accelerator). After this, the stream of data from DMA starts filling in within the convolution window buffer. The necessary window portion is picked from this buffer (SPM) and the convolution operation is performed over this data. The stream of data continues to be sent via the system bus until all the required “tokens” (or data) of a given “frame” (which has total number of tokens based on the size of the input) are received by the accelerator. Similarly, once the execution is completed in the accelerator, the data is sent back in a stream back to the DRAM to store the results.

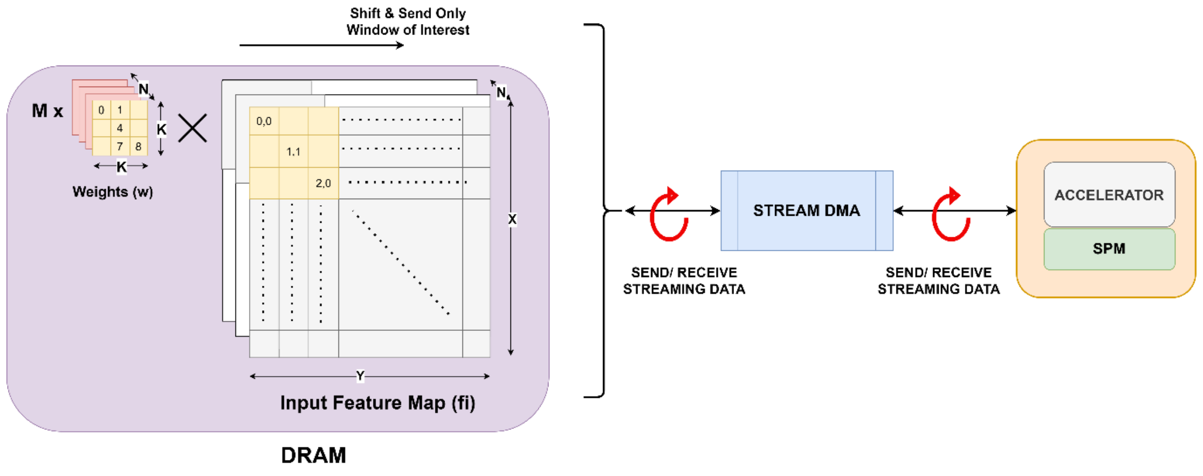

Shift & Send Only Window of Interest
M x
N
K
Weights (w)
0,0
1,1
2,0
X
Y
Input Feature Map (fi)
DRAM
SEND/ RECEIVE STREAMING DATA
STREAM DMA
SEND/ RECEIVE STREAMING DATA
ACCELERATOR
SPM

Fig-5: A broad overview of the Stream-DMA model integrated with a generic accelerator model. The image gives an overview of how the data from DRAM is send and received back from the accelerator in a repeated fashion (a pipelined fashion) until all the tokens of the frame complete transaction.

By adding stream DMA support to an accelerator, we can unlock further parallelism across accelerators as now, each successive accelerator does not have to wait for the entire output data from the previous layer to be ready. Rather, it would take in the data in windowed fashion and store it in a buffer until sufficient data is received after which it can begin execution in parallel to the previous accelerator. In this manner, the execution can happen in a pipelined fashion and hence lead to “accelerator level parallelism”. This buffer in which the data is stored is known as the Stream Buffer.

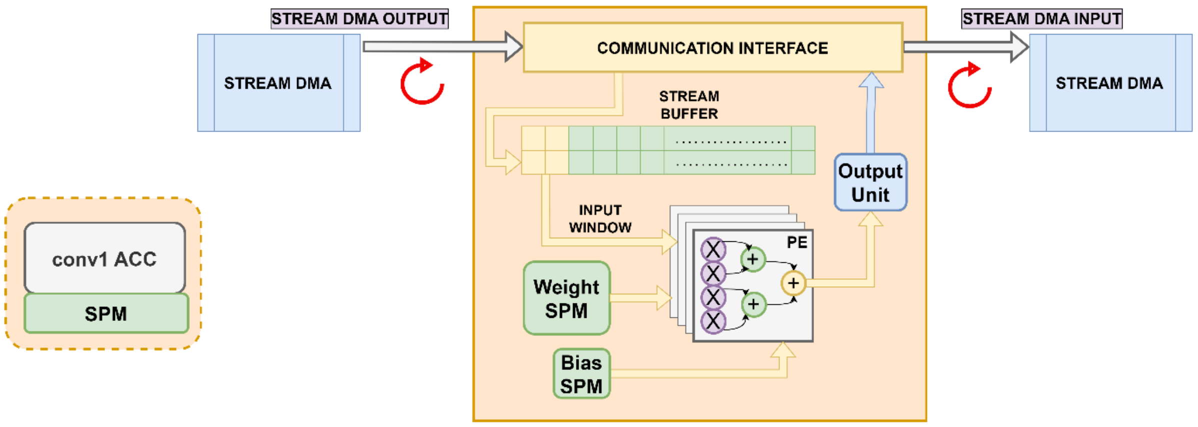
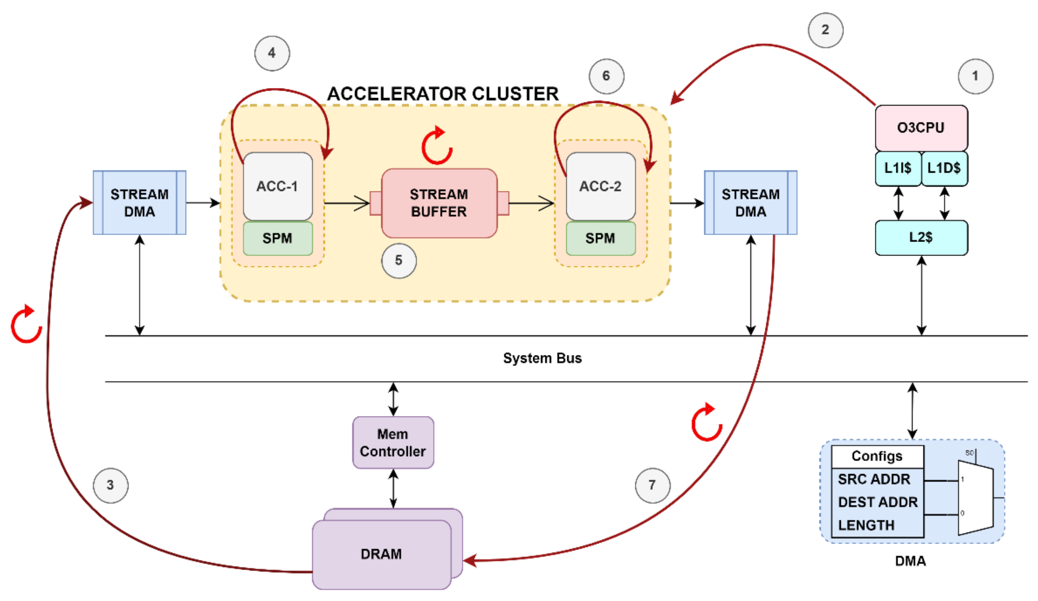


Fig-6: Updated Accelerator module which supports Stream DMA and has stream buffer feature added to support accelerator level parallelism.

Section-IV: Accelerator Clusters

With the Stream-DMA feature enabled, it would be beneficial to encapsulate a bunch of accelerators into one group (known as a cluster). This cluster formation is done to – a) improve the organization of the accelerators, as it is better organized to group a bunch of accelerators and plug them together based on their input size and reusability/ modularity; b) reduce overhead of MMR polling from host, as now fewer MMR registers are to be kept under consideration from the host side; c) reduce interaction with the DMA and have accelerator inter-communication using stream buffers, and hence effectively reducing costly off-chip data transfer costs; d) introduce pipelining within each cluster and hence exploit accelerator level parallelism, by making use of stream buffers and e) use a divide-and-conquer approach to simulate parts of the accelerator to obtain power and area estimate which can then be extrapolated to the entire model by re-using the same clusters.



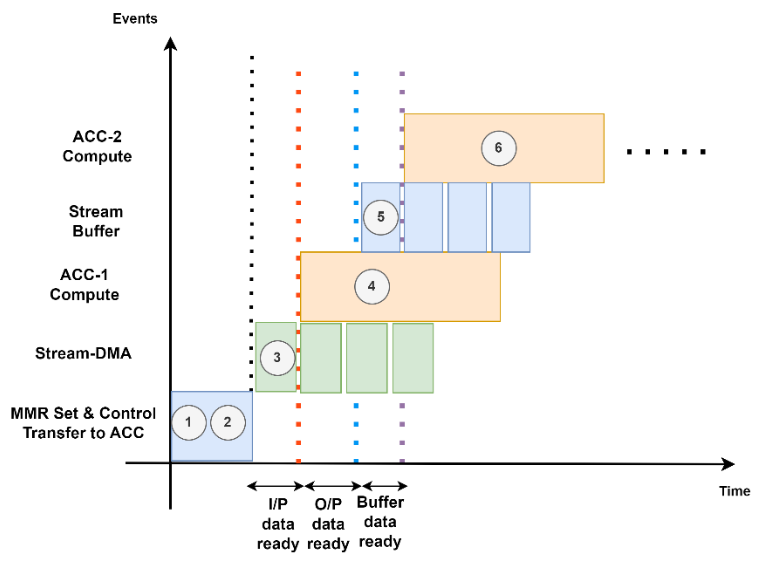


Fig-7: (a) System-level step-by-step interaction between the host (O3CPU) and the accelerator-cluster to transfer data and perform accelerated execution with Stream DMA. (b) Timing Diagram for the sequence of steps described above. It can be noted that now the steps are overlapping and pipelined in nature and hence increased throughput.

The control transfer from host to device consists of modified set of steps until the desired execution is done and the host receives back the data. The steps are as follows:

- **Step-1:** Set the appropriate Memory Mapped Registers (MMRs) with correct values to initiate control transfer from O3CPU (host) to the accelerator (device)
- **Step-2:** MMR set done. Control Transfer successful. Initiate Streaming DMA data transfer to get the input, weight and bias values.

- **Step-3:** Transfer of data from DRAM (global memory) to the first accelerator's respective SPMs (Scratchpad Memories) using the DMA controller through the system bus.
- **Step-4:** Accelerator-1 execution step. This is the step in which the parallel execution of the instruction occurs within the Processing Elements (PEs) or the functional units.
- **Step-5:** Streaming buffer used to transfer intermediate results from accelerator-1 to accelerator-2 in a streaming fashion.
- **Step-6:** Accelerator-2 execution step. This is the step in which parallel execution of instruction occurs in the second accelerator's PEs. This execution is done in parallel to Accelerator-1.
- **Step-7:** Transfer final output data back to DRAM using Streaming DMA.
- **Step-8**: Reset the appropriate MMRs and transfer control back to host (where the host is continuously polling the updated MMR values).

It is evident from the timing diagram from Fig-7 that, owing to addition of streaming DMA support and making accelerator clusters, the data transfer step and the accelerator execution steps have become pipelined in nature. This leads to better throughput and overall performance for the system.

Top Manager Accelerator and SqueezeNet Clusters

For large models such as SqueezeNet – it is hence beneficial to split up into multiple accelerator clusters. However, there is a need for a "top manager" cluster which manages all the subsequent clusters in the design which arbitrates – a) Stream DMA transactions between the DRAM and clusters, b) Data transfer within clusters between the accelerators via stream buffers and c) denote process completion by setting appropriate accelerator MMR registers.

Hence, the accelerator design for SqueezeNet model contains the following clusters in it:

- ***Top Manager Cluster:*** Responsible for managing other clusters within the design
- ***Head Cluster:*** Contains one convolution and Maxpool operation which prepares the input data for further layers in the model.
- ***Body-1 Cluster:*** Contains 3x fire-modules (with corresponding squeeze and expand convolution layers) and a Maxpool layer (all of which have the same input dimensions – 3x3)
- ***Body-2 Cluster:*** Contains 4x fire-modules and 1 Maxpool Layer. (all of which have the same input dimensions – 1x1)
- ***Tail Cluster:*** Contains 1x fire-module, a convolution layer and global average pool layer.

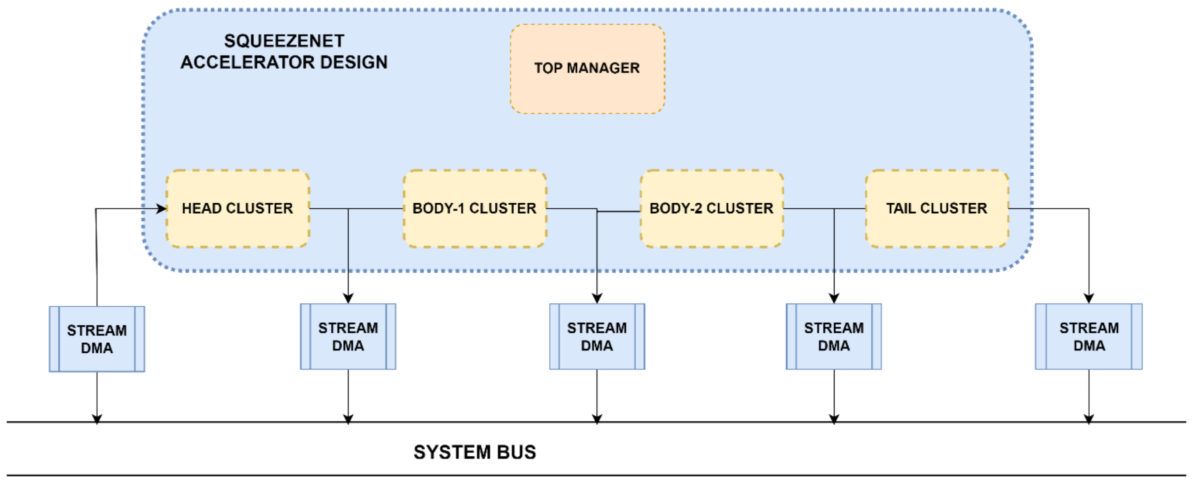
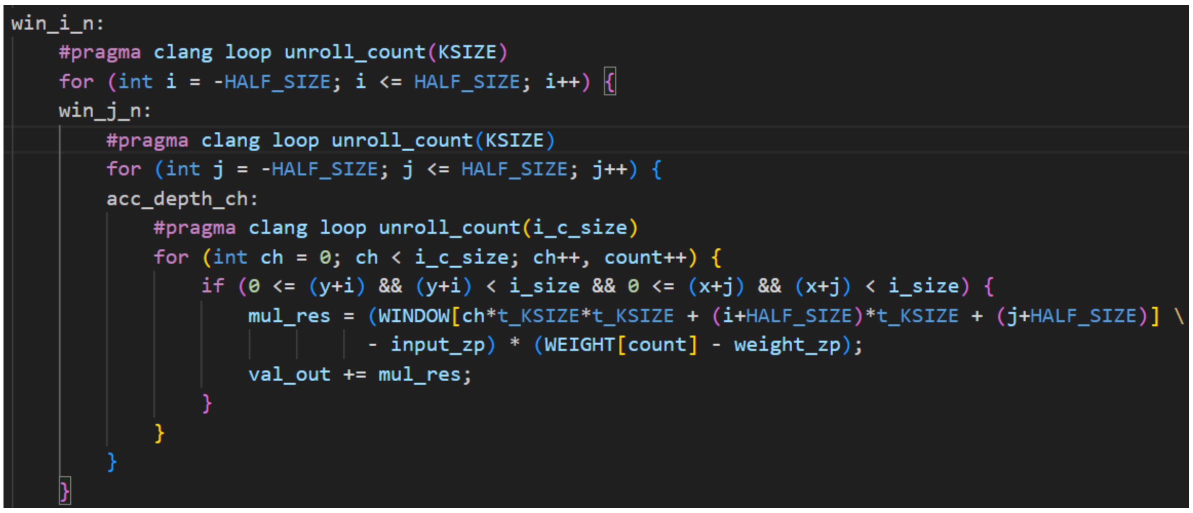


Fig-8: The SqueezeNet Accelerator Design has the following components which are connected via Stream-DMA to the system-bus & DRAM and managed by the Top manager Accelerator (which monitors and arbitrates the data transfer between the accelerators within each cluster as well as DMA transfers from clusters to DRAM)

Section-V: Evaluation and Results

In the gem5-SALAM setup, we are able to achieve parallelization in the operations using the pragma compiler directive option provided by the clang compiler for unrolling loops. The use of the #pragma unroll enables unrolling the loop during the LLVM-IR generation step. Once the unrolled loop is generated, each of the instructions are mapped to corresponding functional unit (which is supported by gem5-SALAM).



```
win_i_n:
    #pragma clang loop unroll_count(KSIZE)
    for (int i = -HALF_SIZE; i <= HALF_SIZE; i++) {
    win_j_n:
        #pragma clang loop unroll_count(KSIZE)
        for (int j = -HALF_SIZE; j <= HALF_SIZE; j++) {
        acc_depth_ch:
            #pragma clang loop unroll_count(i_c_size)
            for (int ch = 0; ch < i_c_size; ch++, count++) {
                if (0 <= (y+i) && (y+i) < i_size && 0 <= (x+j) && (x+j) < i_size) {
                    mul_res = (WINDOW[ch*t_KSIZE*t_KSIZE + (i+HALF_SIZE)*t_KSIZE + (j+HALF_SIZE)] \
                            - input_zp) * (WEIGHT[count] - weight_zp);
                    val_out += mul_res;
                }
            }
        }
    }
```

Fig-9: Code-snipped showing the convolution for-loop being loop unrolled to generate multiple functional units in the simulated accelerator model.

We conducted a study with different loop unroll values on the convolution loop present in the head cluster and the performance was compared with the same convolution loop which was run on the O3CPU. We noticed around 500x improvement in performance as compared to O3CPU when there was no limit placed on the functional unit. While we can see that with limited number of functional units, the performance reduces as expected as the system is stalled for the units to free up to continue with the next operation. An interesting point to note is that, running the same code with no unrolling on the accelerator actually improved the performance. This is because, the required data is offloaded to SPM (ScratchPad Memory) and hence there is minimal latency for data access (as there is never a data access miss) as compared to reading from the memory in O3CPU (which can occasionally lead to cache-miss and lead to large memory access latency).

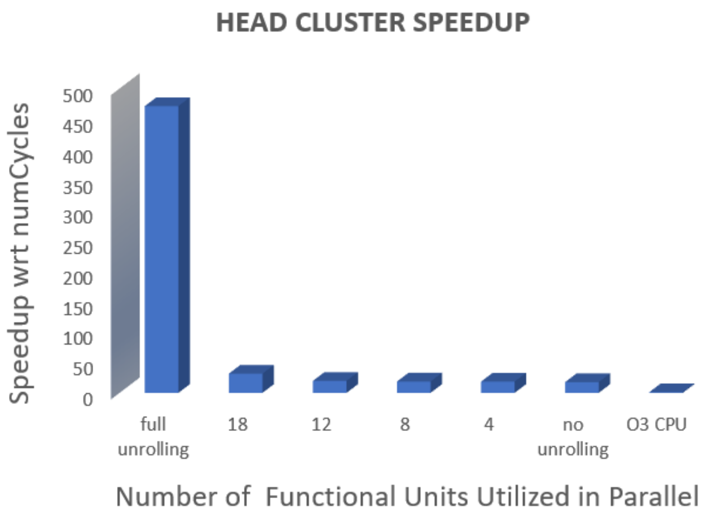
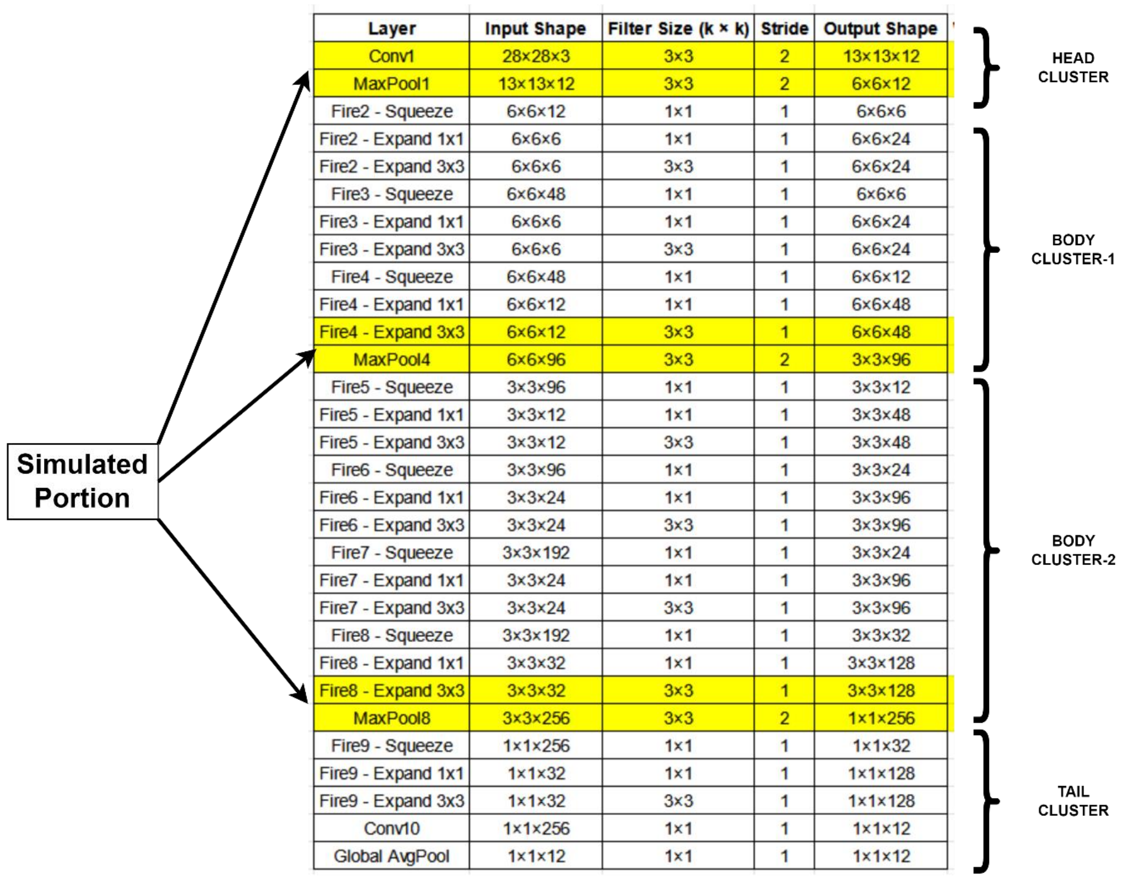


Fig-10: Performance speedup when using the accelerator as compared to running the convolution loop on O3-CPU. It can be seen that there's 500x improvement in performance when full unrolling is done.

Due to the large image size of the default SqueezeNet model (input size fi=[3,224,224]fi = [3, 224, 224]fi=[3,224,224] and kernel size (weights) =[96,3,7,7]= [96, 3, 7, 7]=[96,3,7,7]), completing the simulation by scoreboarding all the transactions through the DMA takes approximately 116 hours, even for a single convolution layer. To expedite the simulation and evaluation process, the input image size was scaled down by a factor of 8x. The new input image size is [3,224/8,224/8]=[3,28,28][3, 224/8, 224/8] = [3, 28, 28][3,224/8,224/8]=[3,28,28], and the kernel size becomes [96/8,3,3,3]=[12,3,3,3][96/8, 3, 3, 3] = [12, 3, 3, 3][96/8,3,3,3]=[12,3,3,3]. Consequently, the output channels at the first layer are reduced from 96 to 12. The scaled-down version of the SqueezeNet accelerator model is shown in Fig.-.

By scaling down the model, we could significantly speed up the simulation and evaluate the head cluster of accelerators (convolution + max pool accelerator) under different clock frequencies. This cluster was also reused for partial evaluations of subsequent layers, where the Fire module interacts with the max pool layer. As SqueezeNet contains a large number of convolution and max pool layers, we employed a divide-and-conquer approach to stabilize the model and obtain evaluation results for different sections. This allowed us to estimate the power and area requirements of the model.

Upon examining Fig.-, it becomes evident that the first set of Fire layers share a similar input feature size ([6,6][6, 6][6,6]), while the subsequent Fire modules share another common input feature size ([3,3][3, 3][3,3]). Based on this observation, we categorized these layers into two clusters: **Body Cluster 1** and **Body Cluster 2**. The tail end of the model layers was grouped into a separate **Tail Cluster**. This clustering approach organizes the parameters of each section more effectively and makes the model more scalable. For instance, if additional Fire layers need to be introduced, a new body cluster can be added to the model without requiring significant modifications.


Simulated Portion
HEAD CLUSTER
BODY CLUSTER-1
BODY CLUSTER-2
TAIL CLUSTER

Layer	Input Shape	Filter Size (k × k)	Stride	Output Shape
Conv1	28×28×3	3x3	2	13×13×12
MaxPool1	13×13×12	3x3	2	6×6×12
Fire2 - Squeeze	6x6x12	1x1	1	6x6x6
Fire2 - Expand 1x1	6x6x6	1x1	1	6x6x24
Fire2 - Expand 3x3	6x6x6	3x3	1	6x6x24
Fire3 - Squeeze	6x6x48	1x1	1	6x6x6
Fire3 - Expand 1x1	6x6x6	1x1	1	6x6x24
Fire3 - Expand 3x3	6x6x6	3x3	1	6x6x24
Fire4 - Squeeze	6x6x48	1x1	1	6x6x12
Fire4 - Expand 1x1	6x6x12	1x1	1	6x6x48
Fire4 - Expand 3x3	6x6x12	3x3	1	6x6x48
MaxPool4	6x6x96	3x3	2	3x3x96
Fire5 - Squeeze	3x3x96	1x1	1	3x3x12
Fire5 - Expand 1x1	3x3x12	1x1	1	3x3x48
Fire5 - Expand 3x3	3x3x12	3x3	1	3x3x48
Fire6 - Squeeze	3x3x96	1x1	1	3x3x24
Fire6 - Expand 1x1	3x3x24	1x1	1	3x3x96
Fire6 - Expand 3x3	3x3x24	3x3	1	3x3x96
Fire7 - Squeeze	3x3x192	1x1	1	3x3x24
Fire7 - Expand 1x1	3x3x24	1x1	1	3x3x96
Fire7 - Expand 3x3	3x3x24	3x3	1	3x3x96
Fire8 - Squeeze	3x3x192	1x1	1	3x3x32
Fire8 - Expand 1x1	3x3x32	1x1	1	3x3x128
Fire8 - Expand 3x3	3x3x32	3x3	1	3x3x128
MaxPool8	3x3x256	3x3	2	1x1x256
Fire9 - Squeeze	1x1x256	1x1	1	1x1x32
Fire9 - Expand 1x1	1x1x32	1x1	1	1x1x128
Fire9 - Expand 3x3	1x1x32	3x3	1	1x1x128
Conv10	1x1x256	1x1	1	1x1x12
Global AvgPool	1x1x12	1x1	1	1x1x12

Fig-11: SqueezeNet model parameters – Scaled down by a factor of 8x to account for reasonable simulation time.

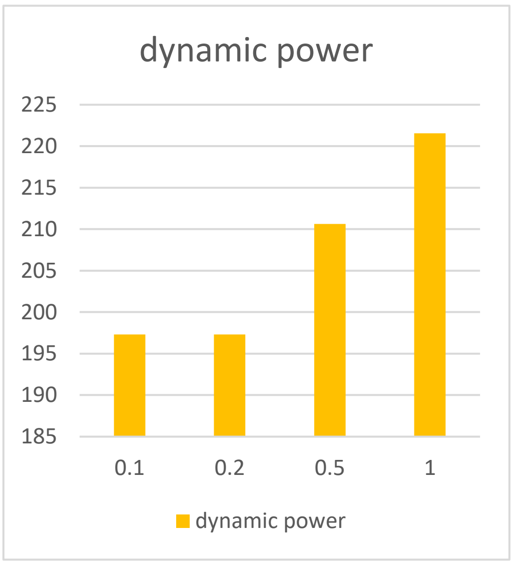
Conceptually, two important aspects should be noted here:

1. **Accelerator-Level Parallelism**: Each cluster exhibits accelerator-level parallelism through the use of stream buffers between accelerators. For example, the convolution accelerator communicates with the max pool accelerator via stream buffers. Convolution windowing is also employed to transfer convolution results to the max pool accelerator in a tiled fashion.
2. **Intra-Cluster Handshaking**: Intra-cluster communication is managed through stream DMA. For instance, when a complete frame (one full image frame) is read from main memory via stream DMA, the head cluster processes it entirely. Afterward, the processed frame is written back to the main memory, allowing the next cluster (e.g., a body cluster) to retrieve the data and start processing. This process continues in a pipelined fashion across clusters.

This handshaking mechanism is controlled through the writing of the appropriate MMRs of each top accelerator from the software layer and monitoring the frame written into the main memory (DRAM) through the top accelerator of each cluster.

Based on the simulation results in the head cluster by running different frequencies. We tested for four different clock frequencies here. We observed that the increase of static power is almost negligible and dynamic power increases gradually (**Fig E.1**), but the biggest gain here is the number of cycles taken by the cluster (**Fig E.2**). There is a straightforward comparison presented in (**Fig E.3**), where the improvement of the number of cycles is multifold where total power increase is not that much. So, it is a good trade-off between throughput and power utilized by these accelerators.

The main contributor to this throughput improvement is the usage of scratchpad memory in the heterogeneous system. Scratchpad memory is small, embedded inside the accelerator, and eliminates the need for frequent off-chip memory accesses (e.g., DRAM), which are slower and consume significantly more power. It also facilitates data reuse where the same data (e.g., weights or input feature maps) is reused multiple times for computations without fetching it repeatedly from DRAM. Unlike traditional caches, scratchpad memory has also deterministic access patterns. This also improves the timing and energy efficiencies as we don’t have to deal with cache miss penalties. In addition to these, having multiple ports, allows efficient parallel access for multiple processing elements supporting techniques like loop unrolling and block/windowing. This boosts the computational speed, particularly in parallel operations. It also supports fast input/output buffering and streaming, enabling seamless data flow between operations in pipelined accelerators. By allowing allowing faster data access, it ensures that processing elements remain fully utilized, maximizing throughput. These all factors are summed up in our experiment to achieve this throughput improvement.



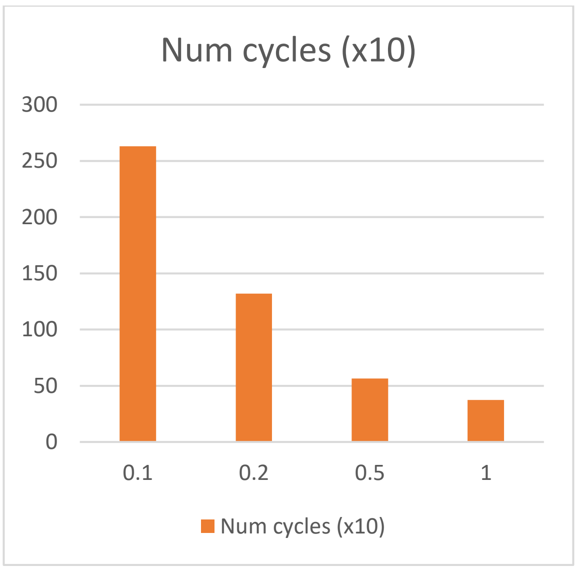


Fig E.1: Dynamic power vs clock frequency in GHz

Fig E.2: Num cycles vs clock frequency in GHz

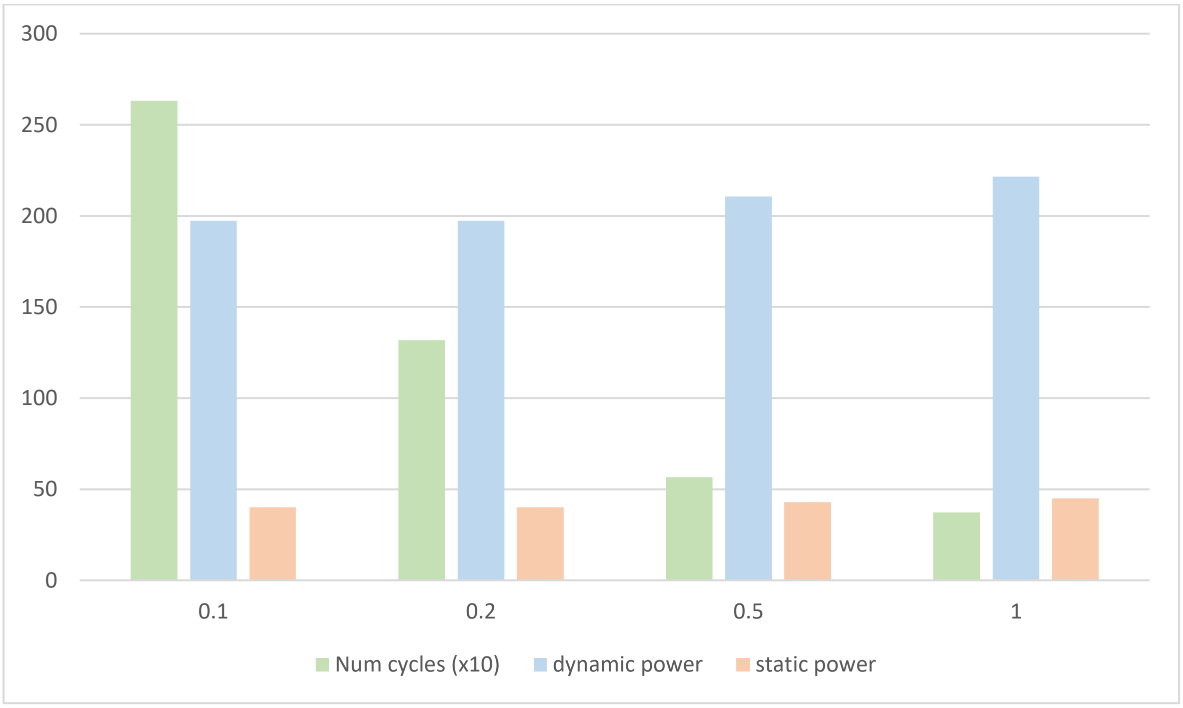


Fig E.3: consolidated plot of all parameters (Num cycles, dynamic power, static power) concerning clk frequency in GHz

Accelerator	Area
TOP	1300.17
Convolution	490633
Maxpool	10278.6

Table 1: Area statistics of different accelerators in a cluster

Clk freq (GHZ)	Num cycles	static power	dynamic power
1	373.447	45.0815	221.551
0.5	565.55	42.8606	210.637
0.2	1318.36	40.1494	197.314
0.1	2631.31	40.1494	197.314

Table 2: Head cluster statistics

Clk Freq (GHZ)	Num cycles	static power	dynamic power
1	558.244	64.3057	316.022
0.5	849.814	62.5296	307.294
0.2	1875.32	56.4357	277.348
0.1	3717.35	56.4192	277.267

Table 3: Body cluster (part 1) statistics

Clk Freq (GHZ)	Time cycles	static power	dynamic power
1	405.664	26.0517	128.036
0.5	565.55	25.344	124.558
0.2	803.6	22.1546	108.486
0.1	1588.15	22.0535	108.388

Table 4: Body cluster (part 2) statistics

Another observation we present here is the area utilized by each accelerator in the head cluster, as shown in Table 1. Since the top accelerator does not handle any functional units or buffers, it occupies the least area. The convolution accelerator, responsible for the majority of computations (such as addition and multiplication), consumes the largest area. In contrast, the maxpool accelerator involves fewer computations and, therefore, occupies less area compared to the convolution accelerator.

As illustrated in *Fig-11***,** we also independently simulated a small part of the body clusters using the same cluster model, modifying parameters such as input dimensions and channel size. The results are tabulated in Table 2, Table 3, and Table 4.

CONCLUSION

To handle the varying layer shapes and sizes in the SqueezeNet model, we divided it into distinct clusters. A head cluster was implemented, consisting of three accelerators: a top accelerator, a convolution accelerator, and a maxpool accelerator. The top accelerator is responsible for managing communication between the accelerators, initiating stream DMA transfers from memory, and detecting task completion by monitoring the output stream from the final accelerator in the cluster. Accelerator-level parallelism was achieved by using stream buffers between the convolution and maxpool accelerators.

In addition to simulating the head cluster, we tested a portion of the Fire module involving a convolution layer followed by a maxpool layer. The power consumption (static + dynamic) and execution time were measured while varying the clock frequency. A notable reduction in execution time was observed, with only a slight increase in power consumption. This performance gain was attributed to techniques such as block/windowing convolution, input/output parallelism enabled by stream buffers, and loop unrolling.

Furthermore, we benchmarked the convolution accelerator against a traditional out-of-order processor, achieving an approximate 500x speedup due to loop unrolling. We also analyzed the area usage of each accelerator within the cluster. The convolution accelerator occupied the largest area, given its computational intensity and significant hardware requirements, followed by the maxpool accelerator and the top accelerator.

RELATED WORK

There is already a lot of ongoing research going on the ML accelerators to improve efficiency. There is a significant improvement in efficiency compared to previous accelerator models [11]. It shows how the MobileNetV2 accelerator can achieve real-time performance and operate effectively within the resource constraints of an FPGA by using a technique called adaptive data flow scheduling technique which is broadly categorized into four steps: 1. Block-Convolution Strategy: The feature maps are divided into smaller blocks, allowing the computations to be performed on these blocks independently; 2. Dynamic Reconfiguration: the accelerator dynamically reconfigures between depth-wise convolution (DWC) and pointwise convolution (PWC) modes ; 3. Intra-Channel and Cross-Channel Data Scheduling: data is arranged in a row-by-row manner within each channel for depth-wise convolution and in case of pointwise convolutions, data is stored across channels, allowing efficient parallel computation and reducing the need for repeated memory accesses; 4. Pipeline Parallelism: the computations for different blocks are pipelined, allowing DWC and PWC to be processed concurrently, thus reducing overall latency. A related study [12] illustrates a comparable methodology for developing a hardware accelerator aimed at improving the performance of Depth-wise Separable Convolution (DSC)-based Convolutional Neural Networks (CNNs). This approach employs a technique known as Low-Precision Numeric Computing, which integrates quantization and batch normalization to optimize hardware performance while minimizing accuracy degradation. Specifically, the design utilizes 16-bit precision to achieve a balance between computational efficiency and classification accuracy. Another interesting work on building a framework for deploying heterogeneous models is observed

These innovations make the accelerator highly effective for real-time image classification in embedded and resource-constrained environments, achieving a competitive balance of speed, efficiency, and power consumption.

FUTURE WORK

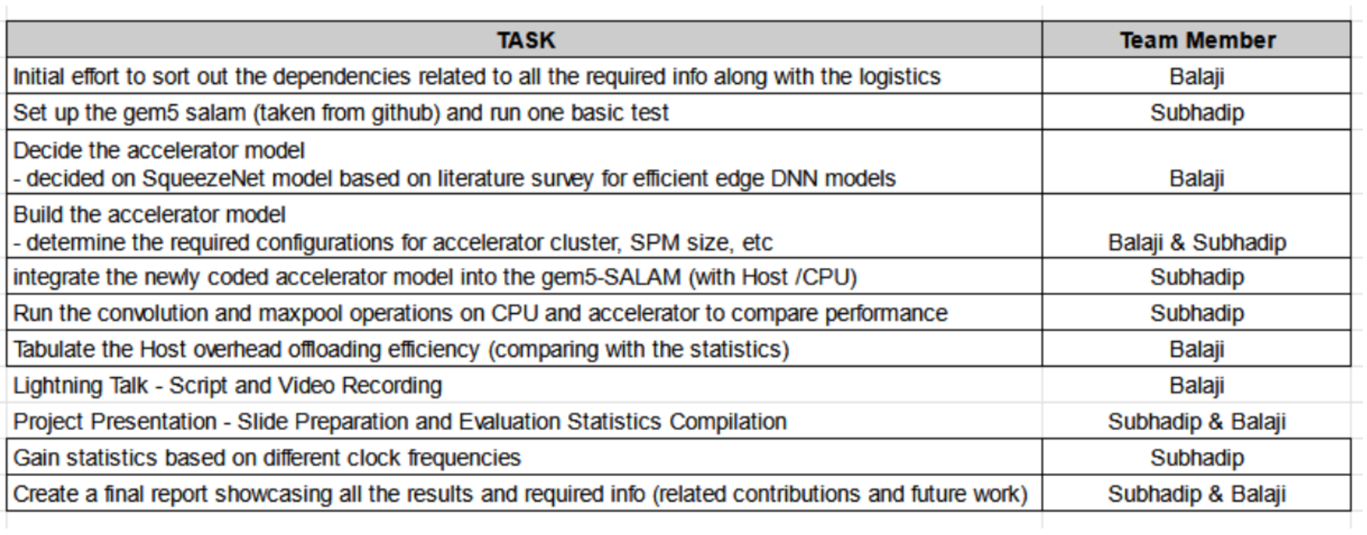
The first objective would be to complete other clusters, which has series of convolution and max pools (Fire modules). The ultimate goal is to simulate the full model (i.e. with the SoftMax layer at the last).

The simulation was done piece-wise and was not done end-to-end due to lack of time. So our next objective would be to get the entire model working in the same simulation.

Thirdly, optimize the convolution accelerator model so the SPM usage is less. Adding Cluster Cache to the Accelerator Cluster and view the change in performance (in terms of numCycles).

Running a sweep over all the possible number of functional units using the pragma directive and find optimal number of functional units. Finding the optimal point beyond which there is no gain in adding more FUs to the accelerator.

TEAM CONTRIBUTIONS



TASK	Team Member
Initial effort to sort out the dependencies related to all the required info along with the logistics	Balaji
Set up the gem5 salam (taken from github) and run one basic test	Subhadip
Decide the accelerator model - decided on SqueezeNet model based on literature survey for efficient edge DNN models	Balaji
Build the accelerator model - determine the required configurations for accelerator cluster, SPM size, etc	Balaji & Subhadip
integrate the newly coded accelerator model into the gem5-SALAM (with Host /CPU)	Subhadip
Run the convolution and maxpool operations on CPU and accelerator to compare performance	Subhadip
Tabulate the Host overhead offloading efficiency (comparing with the statistics)	Balaji
Lightning Talk - Script and Video Recording	Balaji
Project Presentation - Slide Preparation and Evaluation Statistics Compilation	Subhadip & Balaji
Gain statistics based on different clock frequencies	Subhadip
Create a final report showcasing all the results and required info (related contributions and future work)	Subhadip & Balaji

REFERENCES

[1] M. D. Hill and V. J. Reddi, “Accelerator-Level Parallelism,” *arXiv*, vol. arXiv:1907.02064v4 [cs.DC], 2019.

[2] Iandola, Forrest N. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size." *arXiv preprint arXiv:1602.07360* (2016).

[3] Silvano, Cristina, et al. "A survey on deep learning hardware accelerators for heterogeneous hpc platforms." *arXiv preprint arXiv:2306.15552* (2023)

[4] S. Rogers, J. Slycord, M. Baharani and H. Tabkhi, "gem5-SALAM: A System Architecture for LLVM-based Accelerator Modeling," 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, 2020, pp. 471-482, doi: 10.1109/MICRO50266.2020.00047.

[5] F. Conti, A. Marongiu and L. Benini, "Synthesis-friendly techniques for tightly-coupled integration of hardware accelerators into shared-memory multi-core clusters," 2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Montreal, QC, Canada, 2013, pp. 1-10, doi: 10.1109/CODES-ISSS.2013.6658992.

[6] Nowatzki, Tony, et al. "Stream-dataflow acceleration." *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017.

[7] Y. S. Shao, S. L. Xi, V. Srinivasan, G. -Y. Wei and D. Brooks, "Co-designing accelerators and SoC interfaces using gem5-Aladdin," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 2016, pp. 1-12, doi: 10.1109/MICRO.2016.7783751.

[8] Zephaniah Spencer, S. Rogers, J. Slycord, H. Tabkhi, “Expanding hardware accelerator system design space exploration with gem5-SALAMv2,” in Journal of Systems Architecture, Volume 154, Sept. 2024, 103211

[9] https://www.cs.sfu.ca/~ashriram/Courses/CS7ARCH/tutorials/gem5-acc/index.html#accelerator – Simon Fraser University: CS7ARCH course gem5-SALAM accelerators tutorial.

[10] https://github.com/TeCSAR-UNCC/gem5-SALAM - Official gem5-SALAM GitHub repository.

[11] S. Mujawar, D. Kiran and H. Ramasangu, "An Efficient CNN Architecture for Image Classification on FPGA Accelerator," 2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAECC), Bangalore, India, 2018, pp. 1-4, doi: 10.1109/ICAECC.2018.8479517

[12] Sang, X., Ruan, T., Li, C. *et al.* A real-time and high-performance MobileNet accelerator based on adaptive dataflow scheduling for image classification. *J Real-Time Image Proc* **21**, 4 (2024).