Assignment 2 SCALE-SIM

Name: Rahul Vigneswaran K

Roll: CS23MTECH02002

Subject: Hardware Architecture for Deep Learning (CS6490)

The objective of this assignment is to use an estimation tool SCALE-SIM to understand the effect of various configurations on the execution cycles and the DRAM access bandwidth. The suggested steps are some guidelines, but you are free to explore more to develop deeper understanding. The source paper is: https://ieeexplore.ieee.org/document/9238602

1. Download/setup the scale-sim repo as per instructions from https://github.com/scalesim-project/scale-sim-v2
2. Understand what it does, how it does, and write a short summary of your understanding
3. Pickup different CNN architectures (mobilenet, alexnet (modify CONV2 layer input size from 207 to 27), resnet18, Googlenet, FasterRCNN, yolo_tiny (pick the yolo_tiny model from link)) and run the tool for three different configs (eyeriss, google, scale) as given in the repo. Tabulate various metrics, study the behavior and summarize the observations.
4. For three CNNs (mobilenet, FasterRCNN, and resnet18), and eyeriss.cfg, study and summarize your observations for the following:
   1. Change the dataflow architecture between WS, IS, and OS and observe the effect
   2. Change the size of different SRAM and observe the effect
   3. Change the array size and observe the effect

Reader's Notes:

- I didn't dig deep into the scale-up vs scale-out scenarios in this writeup as the instructions to emulate them are not provided in the repo. Instead I have focused mainly on the overall goal and working of SCALE-Sim.
- I didn't compare energy as they need some more external simulators to obtain the metric. With the help of external simulators such as CACTI or DRAMPower, we can obtain, power consumption and area estimates.
- I couldn't understand the difference between Overall utilization and compute utilization anywhere in the documentation or the paper itself. So I have stuck to using the overall utilization.
- Similarly I couldn't find from anywhere what the meaning of mapping efficiency is. I have assumed it to be how much can we efficiently fill the PEs.
- While the use of tables to find trend is very difficult, I have added it in some parts for the sake of completion. I mainly used the plots to observe the trends I have discussed in the assignment.

---

1. I will push the code to this repo once the submission deadline is over.

---

2. Understand what it does, how it does, and write a short summary of your understanding

Let us try to understand the following things step by step,

- Gap in the literature
- How SCALE-Sim fills the gap?
- How is everything implemented?

Gap in the literature

When we attempt to design a CNN accelerator, there are typically alot of design permutation and combinations that has to be made through rigorous iterative steps depending the scenario of deployment. These design choices are often expensive to arrive at based on a trial and error strategy. It is simply not feasible to create an accelerator with a set of design points, benchmark it and go back to the drawing board to iterate on those earlier design choices.

Consider an example of building a house. Imagine we randomly take a design decision to build a house with 5 rooms. Build the house and later understand that there is no need for 5 rooms and instead we need 2 rooms and play area. Now we have to go demolish those remaining 3 rooms and reconstruct them into a play area. There is wastage of resources and this gets expensive very fast if we decide to change our design choices repeatedly. Turns out, this is a very common problem and civil engineers employ Computer Aided Designing (CAD) in order to get a virtual model of the house, iterate upon it virtually and then begin the actual construction only after the design iteration is finalized. Doing this on a CAD software is very inexpensive compared to doing it for real.

Similarly, instead of starting from random design parameter choice, invest resources into building an actual accelerator, benchmark it, then change the design parameters accordingly and rebuild the accelerator once again, it would be very helpful if we could have a similar CAD software but for building and testing CNN accelerators virtually. This is exactly what SCALE-Sim aims to achieve. SCALE-Sim is a configurable systolic array based, cycle accurate DNN accelerator simulator. We will now dig deep into how exactly SCALE-Sim fills this gap.

From this point on we will only talk about CNNs as they comprise the major aspect of most of the modern architectures and is the most common DNN layer type.

How SCALE-Sim fills the gap?

When it comes to design of a CNN accelerator, there are a few key aspects that every accelerator considers. They are,

- Compute
- Dataflow
- Memory

If we are able to simulate the interplay between the above elements with knobs that can be controlled, then we can simulate most of the existing accelerators with little to no changes using SCALE-Sim.

Compute

SCALE-Sim can natively simulate convolutions and Matrix-Matrix (MM) multiplications. It also supports Matrix-Vector and Vector-Vector multiplications as a special case of MM. SCALE-Sim uses systolic arrays to do the MM multiplications. SCALE-Sim models compute units as systolic arrays for the multiplication operations. This is because most accelerators use systolic array in their design choice, enabling the user to emulate as many widely used accelerators as possible.

Why systolic array? There is no surprise that most accelerators use systolic arrays for the MAC operations as they are extremely efficient. Systolic arrays take the values as inputs from the edges. Each PE at each step, takes the input from all the sides, multiply the inputs, add them with what it had earlier and pass it onto the next PE. They store the computes locally and pass it on to the neighbouring PE depending on the design choice. This significantly saves SRAM read bandwidth and could effectively exploit the reuse opportunities provided by conv operations.

The accelerators typically use a square shaped arrangement of the PEs. SCALE-Sim, exposes the user with an endpoint that decided the length and breadth of the systolic array. Later we will see how these design choices would impact the outcome of the metrics.

Dataflow

Typically convolution operations are filled with opportunities for reuse. CNN accelerators exploit this reusability to accelerate the CNNs. This specific reusability is called as dataflow. To be precise, a dataflow is a mapping scheme that provides the order in which inputs are fetched, outputs are generated and intermediate results are stored and reused,

Each accelerators have their own reasoning as to which dataflow they lean towards. SCALE-Sim provides a common benchmarking solution to test out various dataflows without the downside of development cost. SCALE-Sim supports the following dataflows,

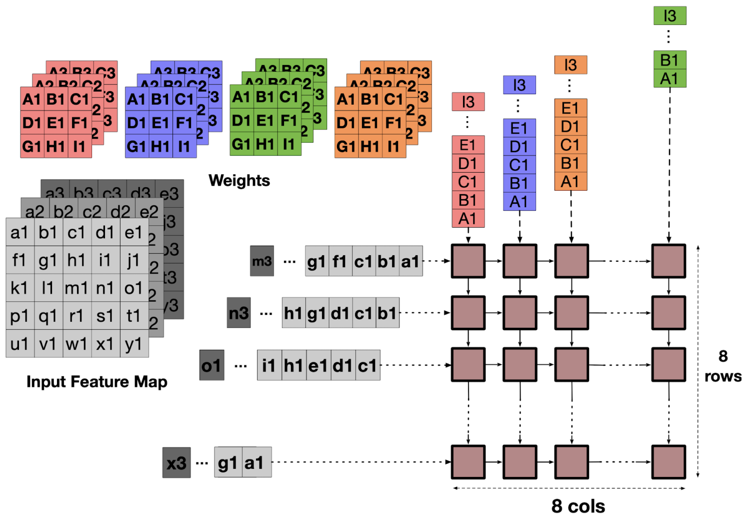
- Output Stationary (OS)
- Weight Stationary (WS)
- Row Stationary (RS)

Eyeriss has the following additional features, but they are not explicitly taken into consideration by SCALE-Sim because of the following reasons.

- No Local Reuse (NLR): NLR is a special case of any of the above dataflows with an additional buffer memory.
- Row Stationary (RS): RS is tied to the specific PE design chosen in Eyeriss and we dont benefit from simulating it with generic components.

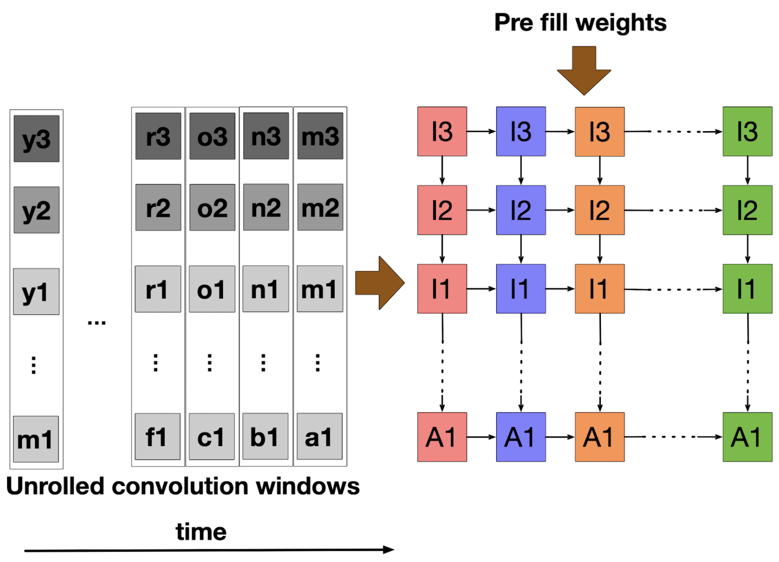
Now lets get s brief overview of the dataflows handled by SCALE-Sim. Our main goal is not to explain the dataflows throughly but instead understand how the said dataflow is mapped onto a systolic array of fixed size.

Outputs Stationary (OS)



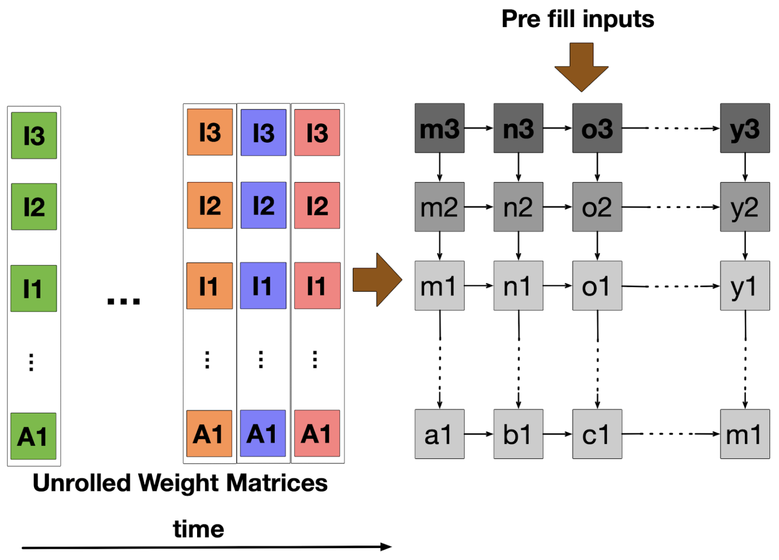
In OS, the output matrix is reused as much as possible. As we can see from the above figure, the input feature is fed through the left and the weights of filters are fed through the top. Each PE does MAC operations based on the inputs and passes it onto the adjacent PE. Everything is done in-place and there is no external SRAM reads done from the PE directly.

Weight Stationary (WS)



In WS, the weights are first pre-filled into the array, unlike in OS where both top and left are fed together. Doing so maximizes the reuse of the weights. After filling the array with the weights, the next phase starts where the unrolled conv windows are fed through the left. The unrolled conv windows are just inputs which are unrolled beforehand to make the mapping with the corresponding weights work properly. During the second phase the partial sums are produced and accumulated in preset cycle intervals. This reduction is done over the columns.

Input Stationary (IS)



IS is similar to WS, but instead of weights being prefilled and input being in an unrolled fashion, the exact opposite happens. The inputs are prefilled and the weights are fed after unrolling to match the inputs. This increases the reuse of the inputs. Similar to WS, the reduction is performed over the columns.

Memory

While the reuse opportunities are being exploited, we do need some SRAM buffer to access these from. All accelerators have some kind of scratchpad memory, but determining its optimal size is a design choice. Therefore SCALE-Sim models this memory into three parts,

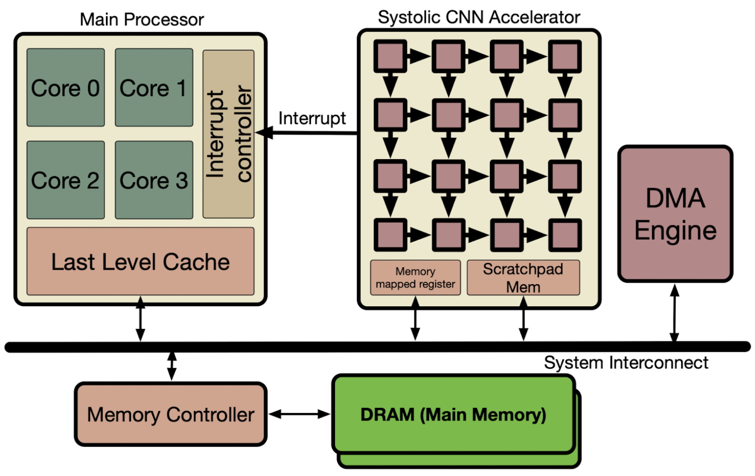
- IFMAP: The reasoning for this is obvious as we have to store the inputs.
- Filters: The reasoning for this is too is obvious as we have to store filter weights.
- OFMAP: This is not so obvious but it serves 2 tasks. It stores the outputs till there are enough elements to do a bursty efficient transfer. Then, if we use WS or IS, we have to store the partial sums till the cycle is done.

The memory is modelled as double buffers to eliminate the access latency on SRAM as this is not what we are trying to study.

What is double buffered? There would be two sets of memories - working set and idle set. Working set is used to feed the array and idle set is us populated by fetching data from the off-chip memory simultaneously. This reduces the access latency to SRAM.

Now that we have understood how the individual components are modelled, lets understand how this all comes together.

How does SCALE-Sim work in conjunction with the system?



Like shown above, the CNN accelerator is connected with the system in a slave interface. The master creates the mappings and instructions and offloads the task to the accelerator, then it switches to doing other jobs while the accelerator wakes up and does it things independently. When the calculations are done, the results are copied to the memory and the master is notified of the end of job.

How is everything implemented?

1. First, SCALE-Sim generated cycle accurate address for all the elements that goes in and out of the systolic array in such a way that the PE never stalls based on the user provided config.
2. Now SCALE-Sim produces traffic traces for the provided config. The address generated earlier are effectively the SRAM read traffic for filters and inputs depending on the chosen dataflow. The output traces are effectively the SRAM write traffic.
3. Now SCALE-Sim parses all the traffic traces that were generated earlier. Based on this, the total runtime for compute and data transfer time are derived. Data transfer time is the cycle count of the last output trace entry. Now that we have the SRAM traffic traces, SCALE-Sim can generate the DRAM traffic traces for the input and outputs.
4. Finally these DRAM traces are used to get an estimate of the bandwidth and the power consumption of the memory.

One specific assumption that SCALE-Sim makes is that the generated outputs can be pushed out of the systolic array without an stalls in compute which might not be the reality. So we can think of the metrics derived as a lower bound wrt the reality. Even in the output files we can note that there are not stalls in compute.

---

3. Pickup different CNN architectures (mobilenet, alexnet (modify CONV2 layer input size from 207 to 27), resnet18, Googlenet, FasterRCNN, yolo_tiny (pick the yolo_tiny model from link)) and run the tool for three different configs (eyeriss, google, scale) as given in the repo. Tabulate various metrics, study the behavior and summarize the observations.

Metrics that am considering for analysis:

- Average Bandwidth (Averaging across layers)
  - SRAM
    - IFMAP
    - FILTER
    - OF
  - DRAM
    - IFMAP
    - FILTER
    - OF
- Runtime in cycles (Adding across layers)
- Overall utilization (Averaging across layers)
- Mapping efficiency (Averaging across layers)

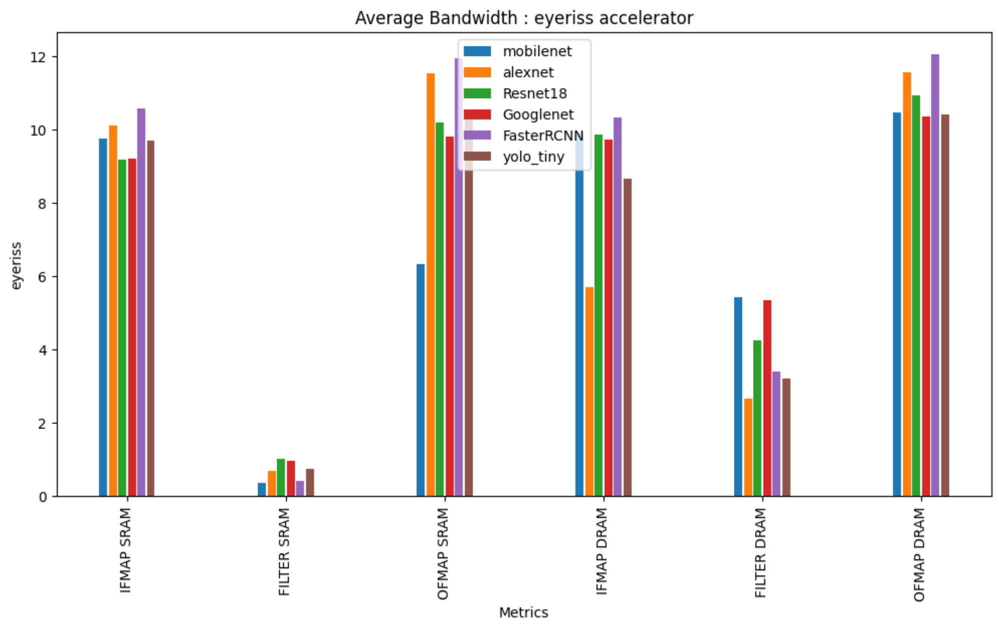
Average Bandwidth

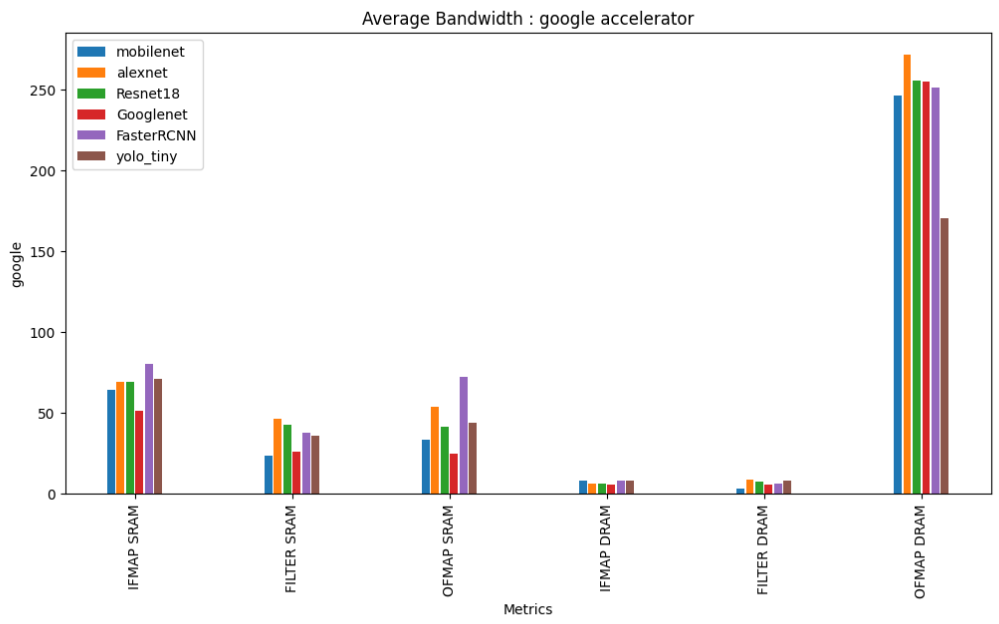
Tables

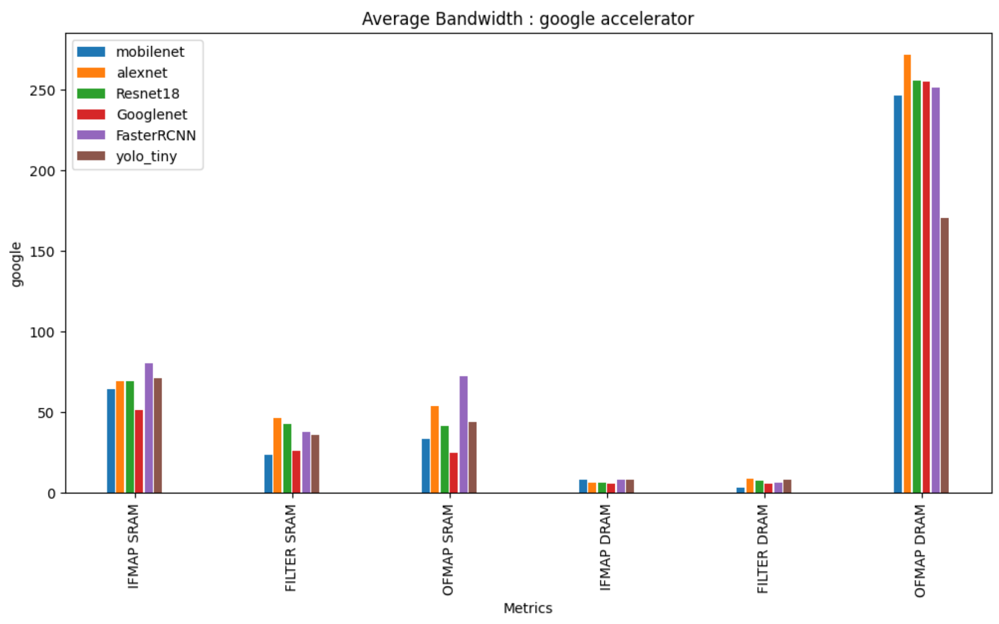
eyeriss	mobilenet	alexnet	Resnet18	Googlenet	FasterRCNN	yolo_tiny
IFMAP SRAM	9.76936	10.1105	9.18959	9.20477	10.5706	9.71193
FILTER SRAM	0.349855	0.692984	1.01918	0.966143	0.419639	0.746595
OFMAP SRAM	6.336	11.5383	10.1841	9.80542	11.9527	10.3266
IFMAP DRAM	9.91444	5.69707	9.87003	9.72661	10.3283	8.65283
FILTER DRAM	5.43437	2.65338	4.25372	5.35457	3.41015	3.22125
OFMAP DRAM	10.4835	11.5639	10.9475	10.3689	12.0531	10.4082
google	**mobilenet**	**alexnet**	**Resnet18**	**Googlenet**	**FasterRCNN**	**yolo_tiny**
IFMAP SRAM	64.0754	69.0341	69.4573	51.6176	80.2014	70.9263
FILTER SRAM	23.7949	46.4301	42.7787	26.0276	37.5754	35.932
OFMAP SRAM	33.4751	53.7012	41.2582	24.961	72.3497	44.1209
IFMAP DRAM	8.1066	6.14554	6.55471	5.91371	8.12937	8.37802
FILTER DRAM	3.31682	8.7285	7.31892	5.86187	6.45471	7.86638
OFMAP DRAM	246.666	271.7	255.7	255.398	251.644	170.471
scale	**mobilenet**	**alexnet**	**Resnet18**	**Googlenet**	**FasterRCNN**	**yolo_tiny**
IFMAP SRAM	23.8728	29.2846	24.8907	23.4463	24.7953	24.5391
FILTER SRAM	12.7933	30.5121	28.5358	26.773	26.0918	26.3115
OFMAP SRAM	2.28013	0.800293	1.72624	1.9095	2.80296	2.10339
IFMAP DRAM	14.621	5.27299	7.65491	15.1331	16.4709	7.35778

scale	mobilenet	alexnet	Resnet18	Googlenet	FasterRCNN	yolo_tiny
FILTER DRAM	9.10909	24.6524	18.7707	11.1252	13.6755	19.1517
OFMAP DRAM	16.0499	7.50842	11.6332	24.6573	4.99732	5.57156

Plots







Observations

Across Metrics

1. **IFMAP SRAM:**
    - Googlenet consistently shows the lowest bandwidth across all the plots for IFMAP SRAM. FasterRCNN consistently shows the highest bandwidth.
2. **FILTER SRAM:**
    - Resnet18 generally exhibits the lowest bandwidth for FILTER SRAM across all the plots. alexnet consistently shows the highest bandwidth.
3. **OFMAP SRAM:**
    - mobilenet generally exhibits the lowest bandwidth for OFMAP SRAM across all the plots. FasterRCNN consistently shows the highest bandwidth.
4. **IFMAP DRAM:**
    - alexnet consistently exhibits the lowest bandwidth for IFMAP DRAM across all the plots. FasterRCNN or mobilenet consistently show the highest bandwidth.
5. **FILTER DRAM:**
    - Resnet18 generally exhibits the lowest bandwidth for FILTER DRAM across all the plots. alexnet consistently shows the highest bandwidth.
6. **OFMAP DRAM:**
    - mobilenet generally exhibits the lowest bandwidth for OFMAP DRAM across all the plots. FasterRCNN consistently shows the highest bandwidth.

Summary: These trends suggest that Resnet18 and mobilenet often show lower bandwidth requirements across different layers and accelerators, while alexnet and FasterRCNN tend to have higher bandwidth requirements. Googlenet also frequently exhibits lower bandwidth for certain metrics, especially in IFMAP SRAM and OFMAP SRAM. Understanding these trends can be valuable for designing efficient hardware architectures and optimizing neural network models.

Across Accelerators

Note the y-axis for each accelerators. The max values from Eyeriss is much lower when compared to the other accelerators.

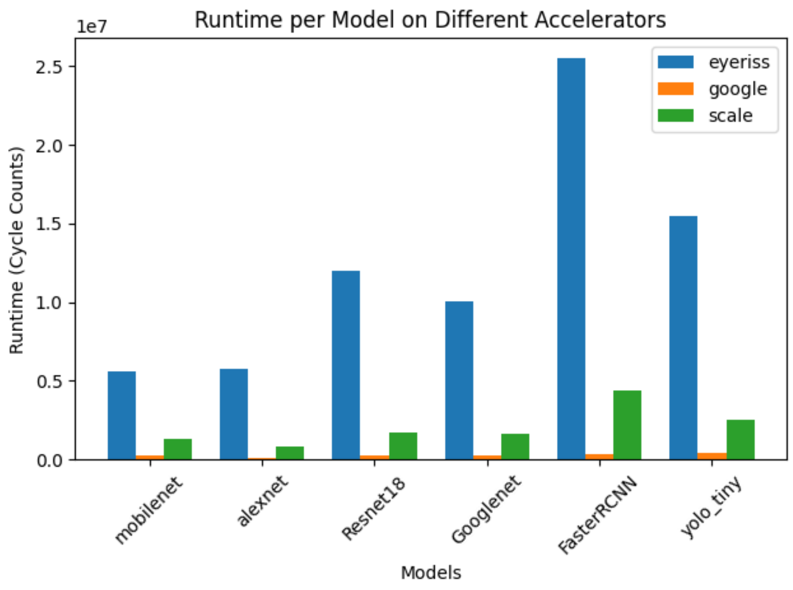
1. **Eyeriss:**
    - Demonstrates consistent and moderate to low bandwidth requirements across various layers and neural network architectures.
    - Potentially optimised for a broad range of neural network models without significant variations in bandwidth.
    - **Best Model to run:** Resnet18 generally exhibits lower bandwidth requirements across different layers.
2. **Google Accelerator:**
    - Exhibits variable bandwidth requirements, with both lower and higher values depending on the layer and neural network model.
    - Shows competitive or lower bandwidth in certain instances, especially in IFMAP SRAM and OFMAP SRAM.
    - **Best Model to run:** Googlenet and mobilenet tend to have lower bandwidth requirements across various layers. This shouldn't come as a surprise as both the TPU and Googlenet are from the same company.
3. **Scale Accelerator:**
    - Displays varying bandwidth requirements across layers and neural network architectures, similar to Google's accelerator.
    - May show lower bandwidth requirements in specific cases, such as OFMAP DRAM for certain models.
    - **Best Model to run:** Scale's bandwidth falls between Eyeriss and Google in many cases, showing moderate to low values for various models.

Runtime in cycles

Tables

Model/Accelerator	eyeriss	google	scale
mobilenet	5.60077e+06	287925	1.33852e+06
alexnet	5.78042e+06	73747	850960
Resnet18	1.1967e+07	223181	1.71835e+06
Googlenet	1.00663e+07	216967	1.65999e+06
FasterRCNN	2.54973e+07	299379	4.36757e+06
yolo_tiny	1.54413e+07	450956	2.5494e+06

Plots



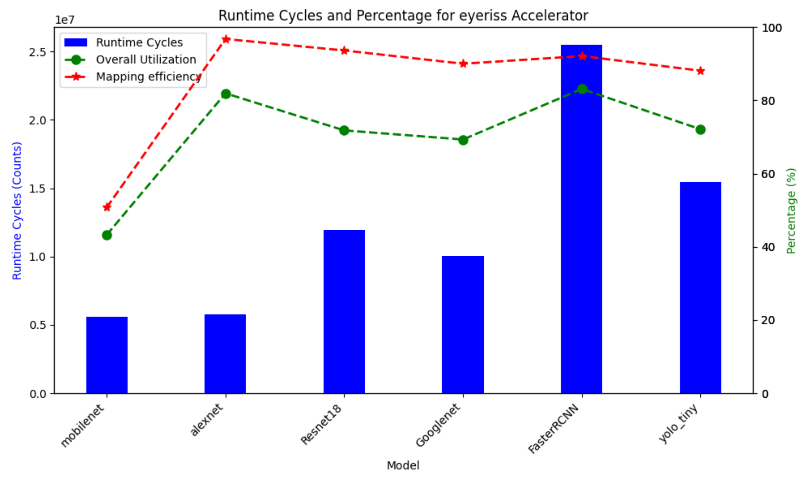
Observations

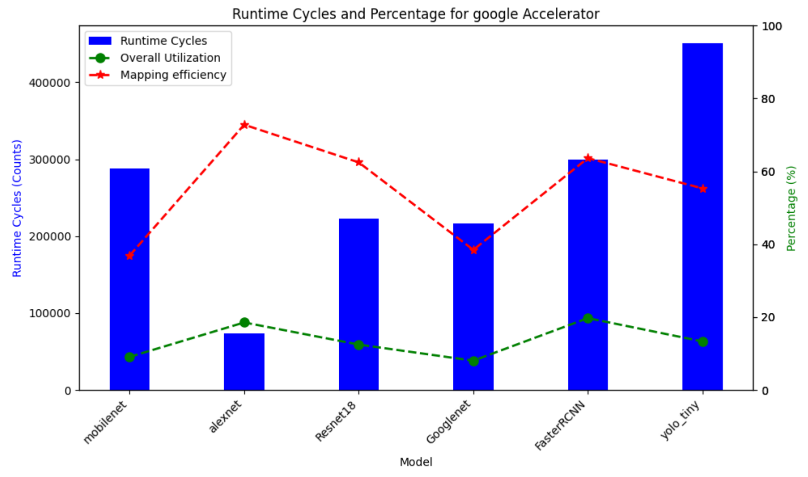
Note the y-axis. Everything is multiplied by 1e7.

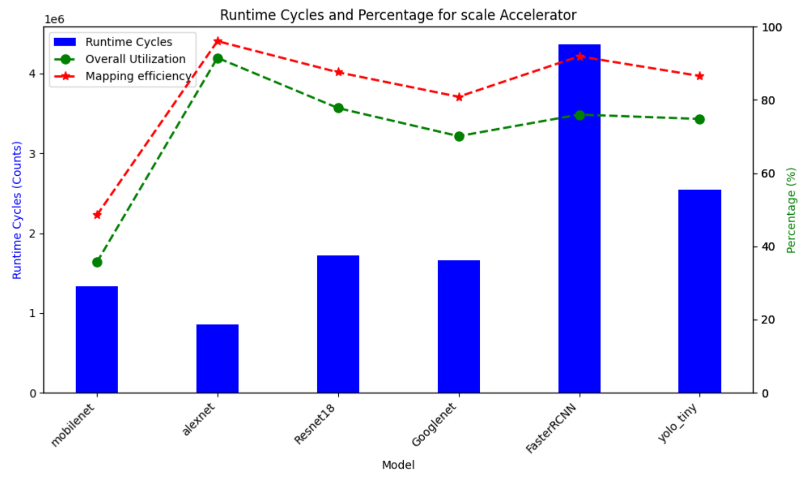
1. The eyeriss accelerator shows significantly higher cycle counts across all models when compared to the google and scale accelerators. This suggests that eyeriss may have a higher computation time or is less efficient in handling these models compared to the other two accelerators.
2. Models like FasterRCNN and yolo_tiny have notably higher cycle counts compared to models like mobilenet and alexnet indicating the difference in complexity of the models.
3. The google's TPU accelerator shows significantly lower cycle count across all the models showing its consistent efficiency and effectiveness across a wide range of models while eyeriss is the exact opposite. Scale provides a middle ground for between both.
4. In application prioritising speed, google's TPU should be opted by these models. Eyeriss with its significantly higher runtime should be avoided for time-sensitive applications. Again, scale provides a middle ground between both.
5. Energy efficiency, though not directly measured here, often correlates with cycle counts, suggesting that google's TPU might also be the most energy-efficient, followed by scale, then eyeriss.

Overall utilization and mapping efficiency

Plots







Observations

1. As we saw earlier, FasterRCNN and yolo_tiny have the highest runtime across all accelerators but we can see that eyeriss and scale has close to 100% utilization and mapping efficiency indicating that they are being utilized to their limits. This could be due to the complexity of the models themselves. This could mean that these accelerators could benefit from increase in PE size in order to reduce the runtime.
2. But we can see that already google's TPU has the lowest runtime for those models compared to other accelerators while also have very low mapping efficiency and utilization. This could mean two things. One, Google could benefit from a different shaped PE to increase the mapping efficiency further. Two, the bottleneck is not the MAC units and instead lies somewhere else. Further investigation is required to understand the reason better and make a conclusion.
3. We can see that in most cases higher utilization and mapping efficieny are good for the runtime. Also we can see that mapping efficiency correlated nicely with the utilization, except in some cases of google's TPU.

We have successfully used SCALE-Sim to gain insights into different accelerators across various models and metrics.

**4. For three CNNs (mobilenet, FasterRCNN, and resnet18), and eyeriss.cfg, study and summarize your observations for the following: 4.1. Change the dataflow architecture between WS, IS, and OS and observe the effect 4.2. Change the size of different SRAM and observe the effect

4.3. Change the array size and observe the effect

4.1. Change the dataflow architecture between WS, IS, and OS and observe the effect

- **Fixed parameters:**
  - configs = ["eyeriss"]
  - ifram = [108]
  - fram = [108]
  - ofram = [108]
  - arrayH = [12]
  - arrayW = [14]
- **Ablation parameters:**
  - models = ["mobilenet", "Resnet18","FasterRCNN"]
  - df = ["ws", "is", "os"]

Average Bandwidth

Tables

mobilenet

Data Flow	IFMAP SRAM	FILTER SRAM	OFMAP SRAM	IFMAP DRAM	FILTER DRAM	OFMAP DRAM
ws	9.76936	0.349855	6.336	9.91444	5.43437	10.4835
is	9.18959	1.01918	10.1841	9.87003	4.25372	10.9475
os	10.5706	0.419639	11.9527	10.3283	3.41015	12.0531

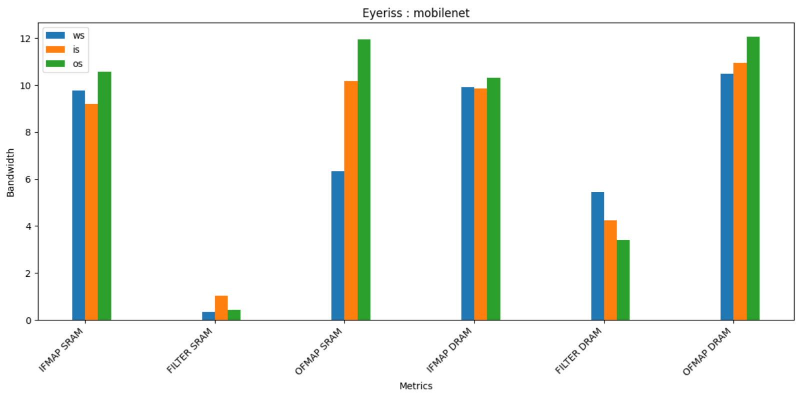
Resnet18

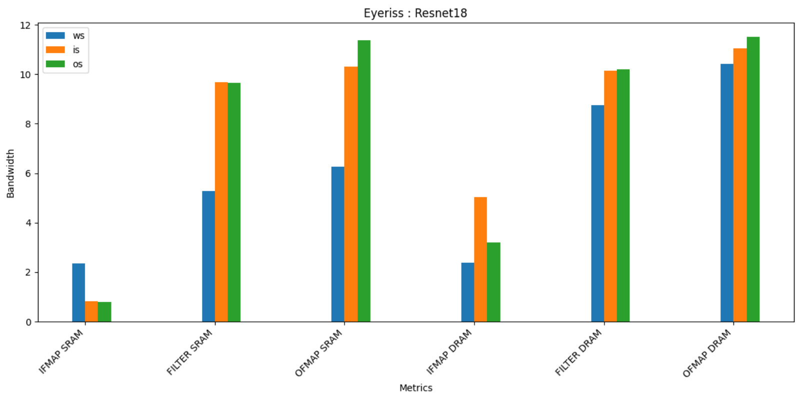
Data Flow	IFMAP SRAM	FILTER SRAM	OFMAP SRAM	IFMAP DRAM	FILTER DRAM	OFMAP DRAM
ws	2.36067	5.28113	6.2484	2.38848	8.74813	10.4268
is	0.824655	9.67821	10.3036	5.04182	10.1374	11.051
os	0.789803	9.65286	11.3739	3.19499	10.2119	11.5022

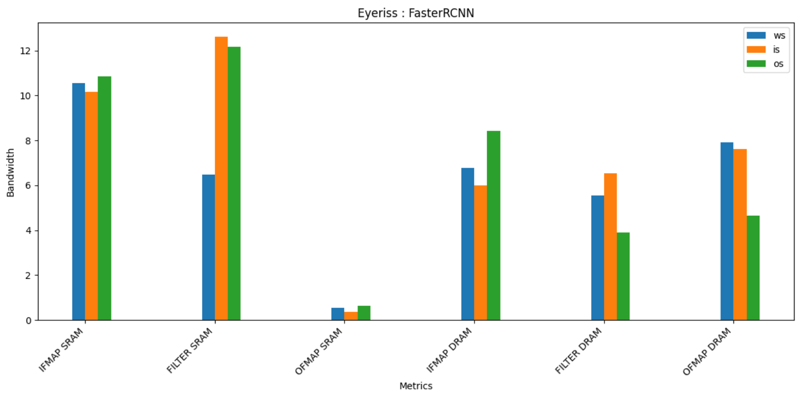
FasterRCNN

Data Flow	IFMAP SRAM	FILTER SRAM	OFMAP SRAM	IFMAP DRAM	FILTER DRAM	OFMAP DRAM
ws	10.5523	6.48094	0.537448	6.79041	5.53585	7.92032
is	10.1635	12.611	0.356572	5.9933	6.53797	7.61888
os	10.8678	12.1582	0.615115	8.41358	3.89369	4.65496

Plots







Observations

Across metrics

1. **FMAP SRAM:**
   - WS generally has lower IFMAP SRAM requirements.
   - IS and OS tend to have comparable or higher IFMAP SRAM requirements.
2. **FILTER SRAM:**
   - WS consistently has lower FILTER SRAM requirements.
   - IS and OS often have comparable or higher FILTER SRAM requirements.
3. **OFMAP SRAM:**
   - WS typically has lower OFMAP SRAM requirements.
   - IS and OS can have comparable or higher OFMAP SRAM requirements.
4. **IFMAP DRAM:**
   - WS generally exhibits higher IFMAP DRAM requirements.
   - IS and OS tend to have comparable or lower IFMAP DRAM requirements.
5. **FILTER DRAM:**
   - WS often has higher FILTER DRAM requirements compared to IS and OS.
   - OS tends to have the lowest FILTER DRAM requirements.
6. **OFMAP DRAM:**
   - WS usually has higher OFMAP DRAM requirements compared to IS and OS.
   - OS tends to have the lowest OFMAP DRAM requirements.

Across models

While making the following choices, am not taking into account the energy and cost difference.

1. **MobileNet:**
   - Limited DRAM? WS looks like the better option as it generally has less DRAM usage.
   - Limited SRAM? WS looks like the better option as it generally has less SRAM usage.
2. **Resnet18:**
   - Limited DRAM? WS looks like the better option as it generally has less DRAM usage.
   - Limited SRAM? WS looks like the better option as it generally has less SRAM usage.
3. **FasterRCNN:**
   - Limited DRAM? OS looks like the better option as it generally has less DRAM usage for FasterRCNN.
   - Limited SRAM? WS looks like the better option as it generally has less SRAM usage.

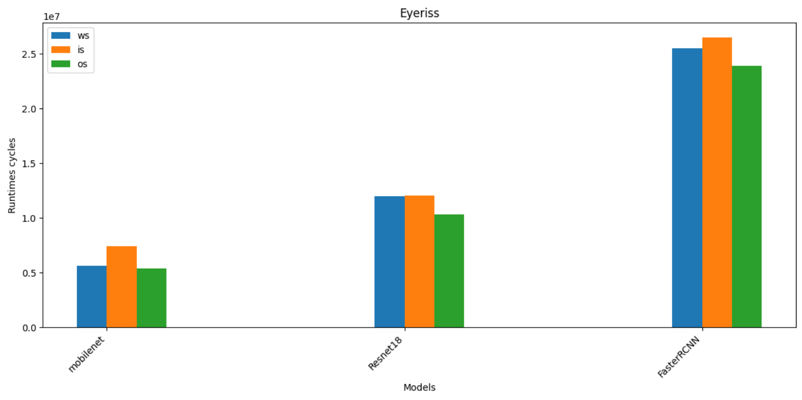
Summary: Inside the given set of models, WS looks like a overall good choice across the board if we were to invest in just a single accelerator.

Runtime in cycles

Tables

Model	mobilenet	Resnet18	FasterRCNN
ws	5600771	11967012	25497279
is	7402373	12037951	26510794
os	5389323	10349074	23898337

Plots



Observations

- On an average IS has the highest runtime while OS has the lowest runtime. If I were to take design decision based on runtime alone, then I would use OS but we know that WS is better for the bandwidth and in runtime WS provides a middle ground between the given dataflows.

4.2. Change the size of different SRAM and observe the effect

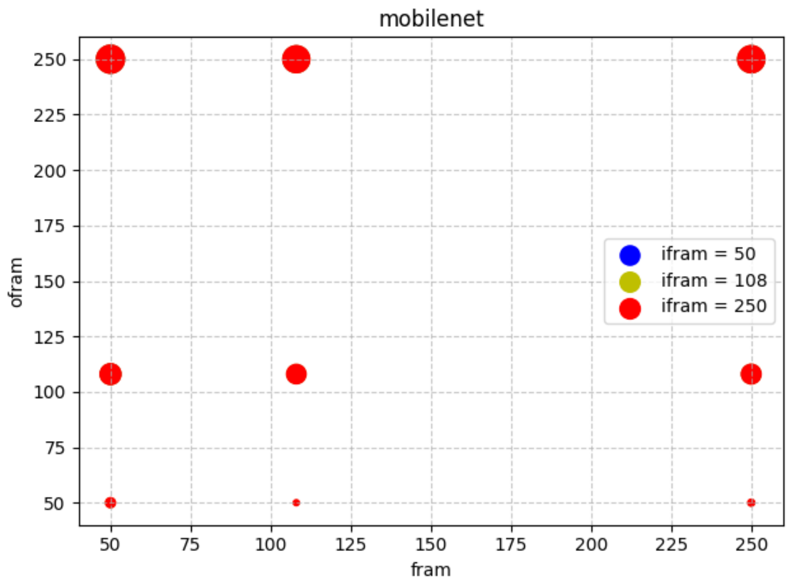
- **Fixed parameters:**
  - configs = ["eyeriss"]
  - df = ["ws"]
  - arrayH = [12]
  - arrayW = [14]
- **Ablation parameters:**
  - models = ["mobilenet", "Resnet18","FasterRCNN"]
  - ifram = [50, 108, 250]
  - fram = [50, 108, 250]
  - ofram = [50, 108, 250]

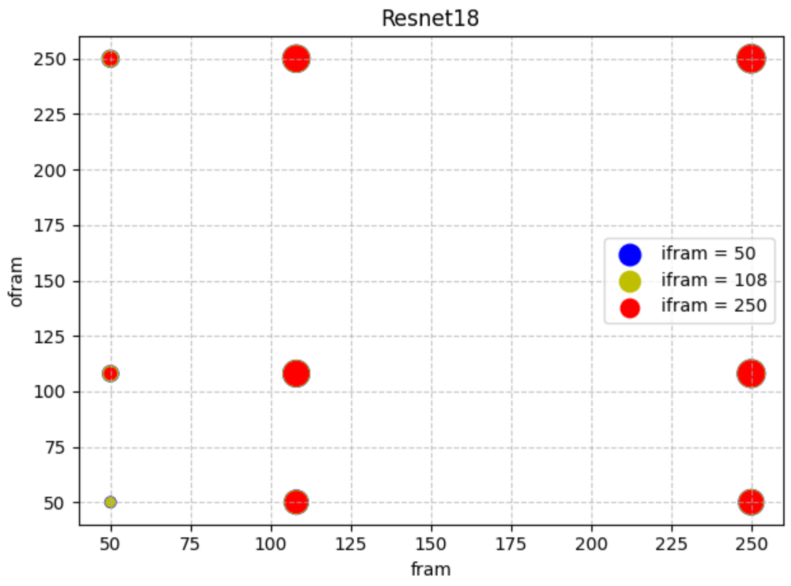
Because there are alot of combinations of parameters am making the following simplifications for ease of understanding and analysis
  - Bandwidth
    - Am taking the average of all the SRAM and DRAM usage and calling it simply as Avg Bandwidth.
  - There are simply too many values, so am sticking to only plots.

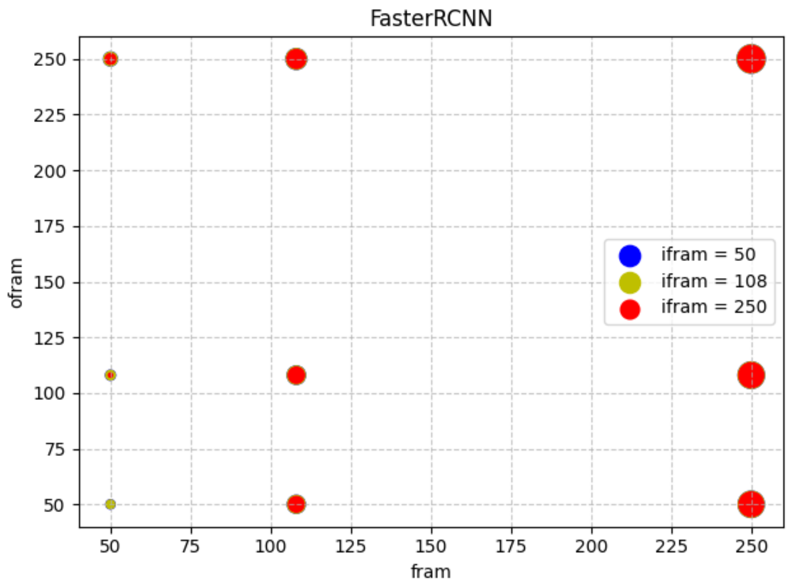
Average Bandwidth

Plots

- Y-axis: fram value
- X-axis: ofram value
- Different marker colors: ifram value
- Size of the marker: Normalized Avg bandwidth







Observations

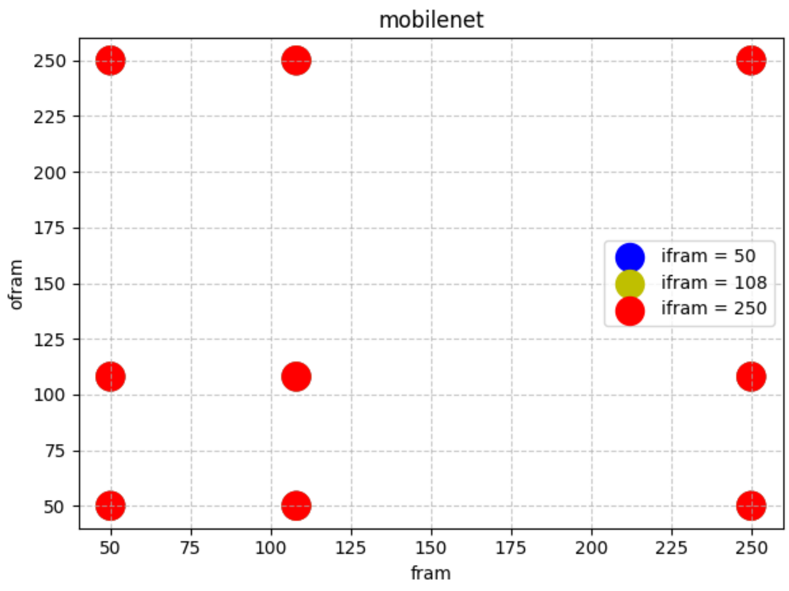
- Irrespective of the ifram, the following seem to be the best options for ofram and fram.
    - mobilenet
        - ofram : [50]
        - fram : [50, 108, 250]
    - Resnet18
        - ofram : [50]
        - fram : [50]
    - FasterRCNN
        - ofram : [50]
        - fram : [50]

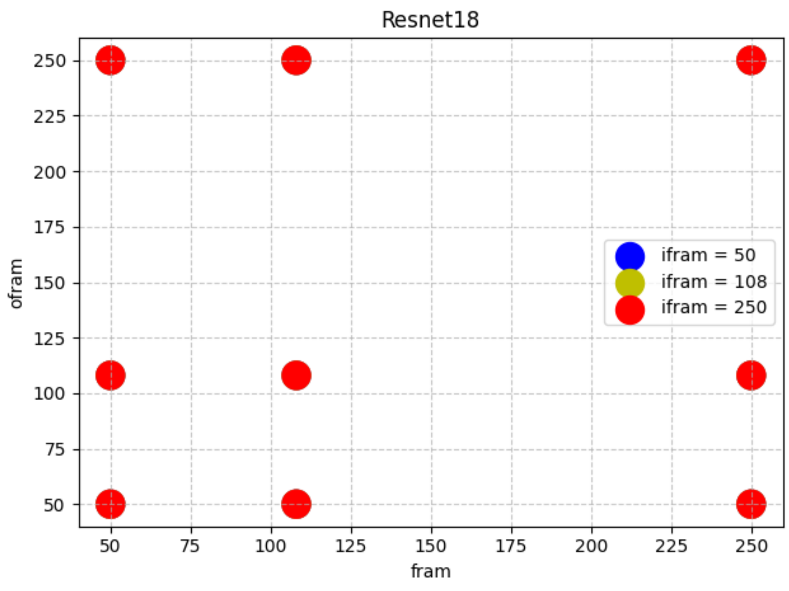
Summary: If I were to make a decision choice based on these observations alone, then to accomodate all the models, I would opt for a [50, 50, 50] setup. But this small buffers will definitely come at a cost of runtime. We will see next if our intuition is correct.

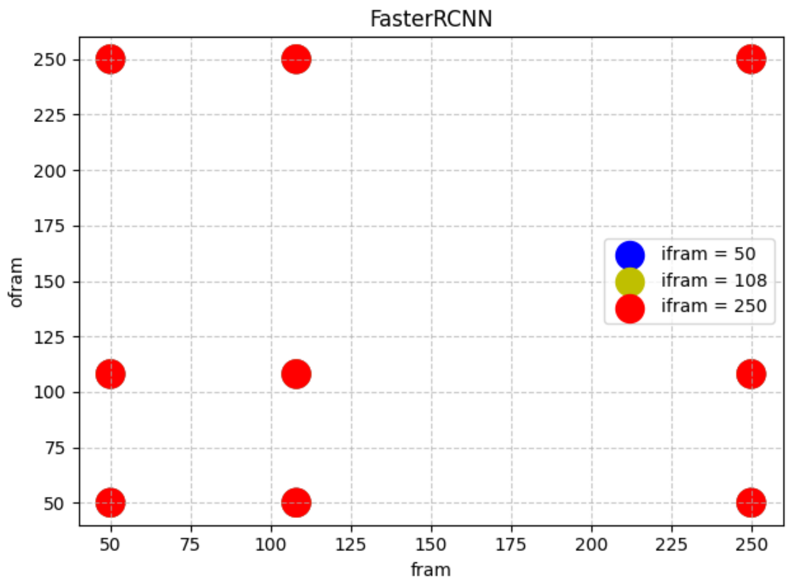
Runtime as cycles

Plots

- Y-axis: fram value
- X-axis: ofram value
- Different marker colors: ifram value
- Size of the marker: Normalized Avg bandwidth







Observations

- Interestingly the choice of the memory size doesnt seem to be affecting the runtime for the respective models. This could mean that we could get away with using the earlier choice of [50,50,50] without any runtime overheads.
- This might be interesting to probe further down the road.

4.3. Change the array size and observe the effect

- **Fixed parameters:**
    - configs = ["eyeriss"]
    - df = ["ws"]
    - ifram = [108]
    - fram = [108]
    - ofram = [108]
- **Ablation parameters:**
    - models = ["mobilenet", "Resnet18","FasterRCNN"]
    - arrayH = [5, 12, 30, 100]
    - arrayW = [5, 14, 30, 100]

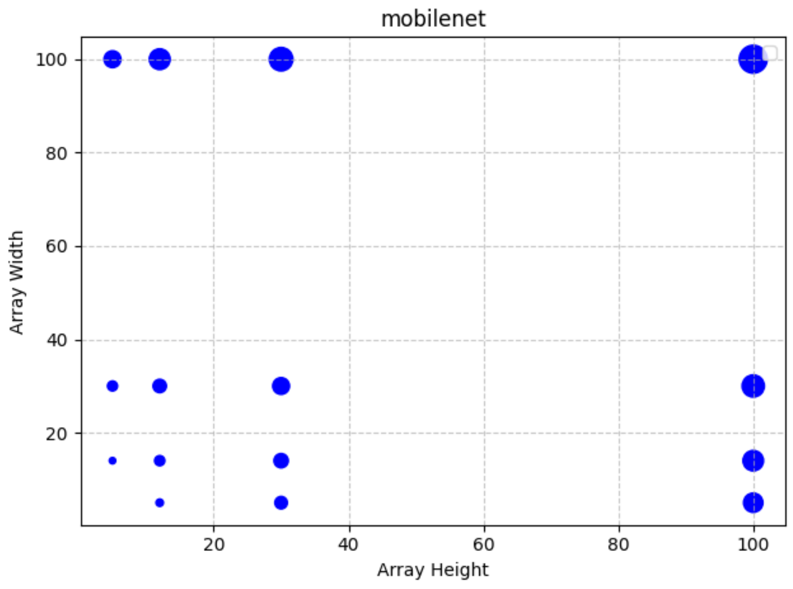
Because there are alot of combinations of parameters am making the following simplifications for ease of understanding and analysis

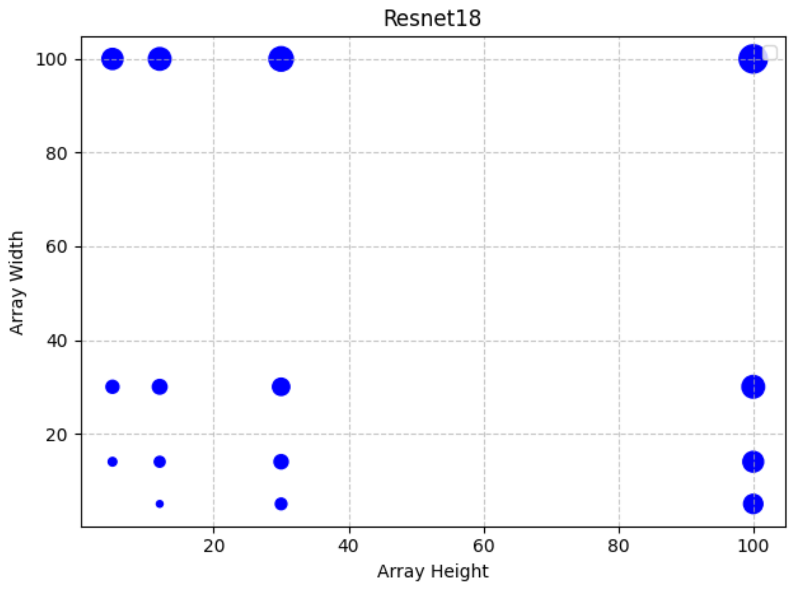
- Bandwidth
    - Am taking the average of all the SRAM and DRAM usage and calling it simply as Avg Bandwidth.
- There are simply too many values, so am sticking to only plots.

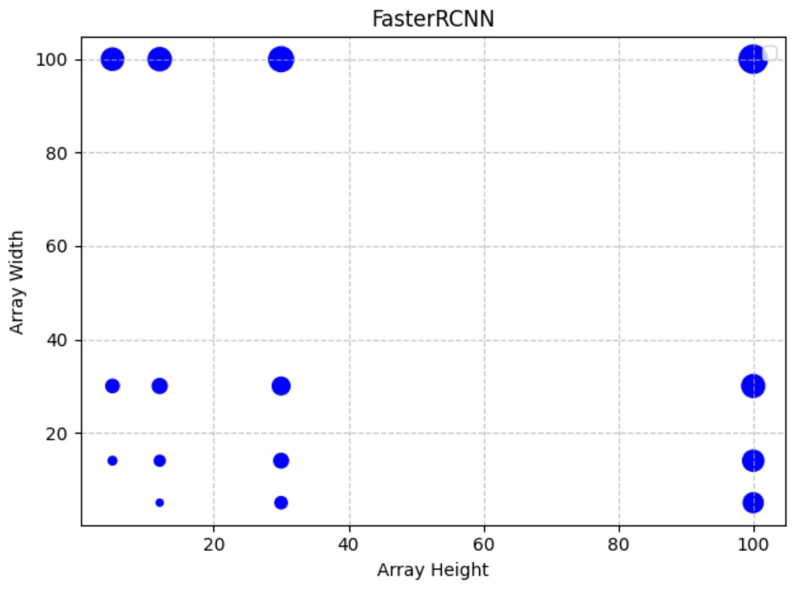
Bandwidth

Plots

- Y-axis: array width value
- X-axis: array height value
- Size of the marker: Normalized Avg bandwidth





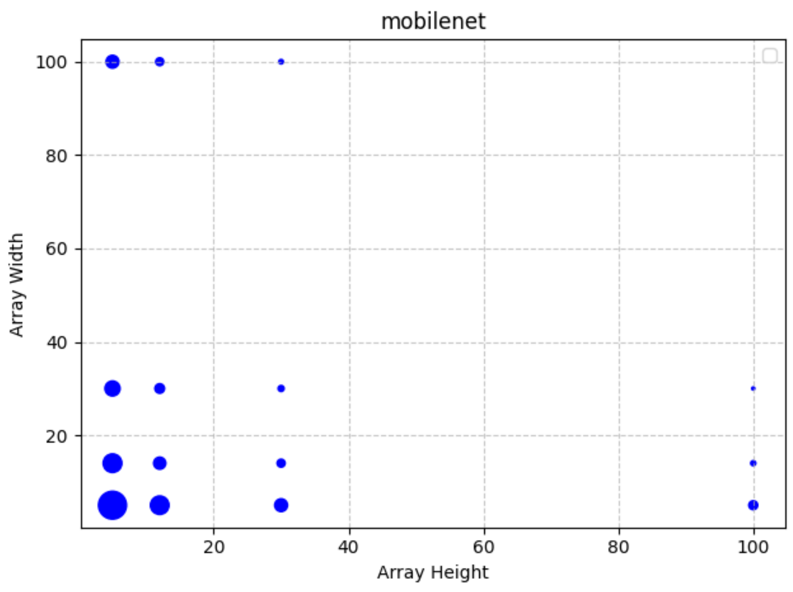


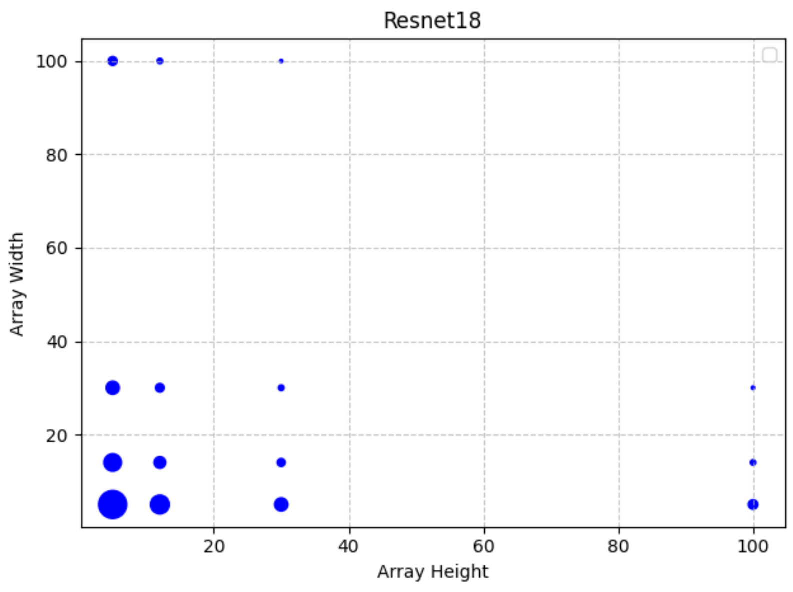
Observations

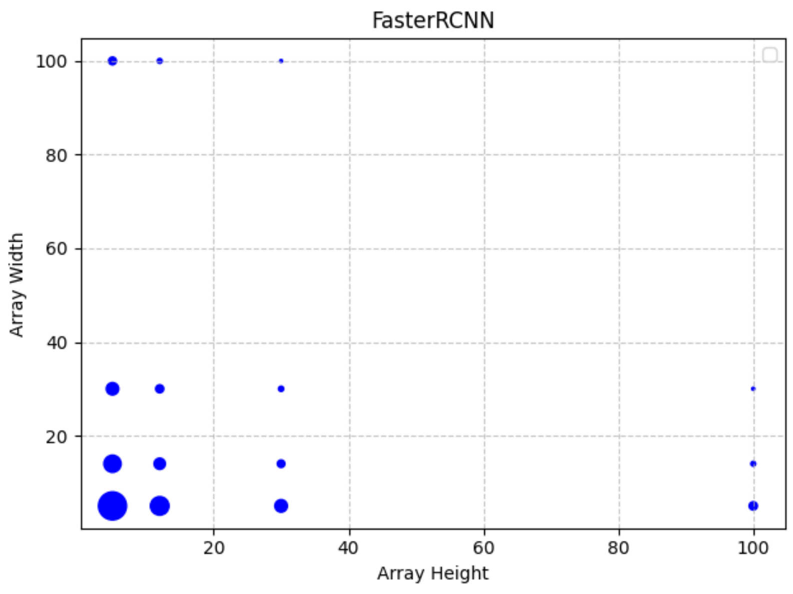
- Across all the models, using [100, 100] results in the maximum bandwidth used.
- Across all the models, using [5,14] or [12, 5] both seems to providing the least bandwidth used. As of now, we don't know how this choice would impact the runtime. My current intuition is that the lowest PE size would result in the maximum runtime. Lets look at that next.

Runtime as cycles

Plots







Observations

- As we predicted earlier, the largest array yields the shortest runtime while the smallest array yields the longest runtime.
- Because of this using the earlier choice of [5,14] or [12, 5] is not a wise one as it would result in longer runtimes.

- A middle ground would be to use [30,30]. It results in considerably shortest runtime while using considerably smaller bandwidth. We can also go with [100,100] but it might exponential increase the cost and reduce the energy efficiency.

We have successfully used SCALE-Sim to gain insights into different design parameters, how it can affect the outcome of the parameters. We have also tried to make an informed decision based on the metrics as to which parameters might be cost effective to make a production version.