MBR-Sim: A Speed-of-Light Model to Explore Machine Learning Accelerator Architectures

Mehul Goel
San Jose, USA
mehulgoel873@gmail.com

Ben Maydan
Urbana Champaign, Illinois, USA
bmaydan2@illinois.edu

Raj Parihar*
San Jose, USA
prince.parihar@gmail.com

***Abstract*—Modern computing systems deploy a plethora of machine learning accelerators (MLA) to solve some of the most complex and challenging problems in computer vision (CV), natural language processing (NLP), recommendation systems and many other domains. Depending upon the use cases, these MLA based systems can be part of anything from simple handheld mobile devices to large scale, complex systems that are used in the clouds and supercomputers. Their adoption is mainly fueled by unprecedented success in various traditional and non-traditional prediction and generative tasks. Designing MLA based systems in an efficient manner to achieve the best performance and energy efficiency requires a careful design space exploration and detailed modeling to understand the workload behaviors in order to make near-optimal architectural decisions.**

One of the most predominantly used framework to study the impacts of various architectural choices is Speed-of-Light (SOL) model that provides early insight into the system and enables designers to co-optimize for various desired metrics. In this paper, we present an SOL model – named as MBR-sim – that can be used to study various CNN and NLP workloads in the context of various MLA systems to make better design decisions. With a set of sensitivity studies, we also demonstrate the use case where this tool can be used to help understand various design trade-offs and aid software development process. Finally, we open-sourced MBR-sim for the wider research community to explore, validate and reproduce their ideas to conceive, architect and design more efficient and greener systems.

***Index Terms*—Machine Learning Accelerator, Speed-of-Light Models, Roofline Analysis, Pipeline parallelism, ResNet50, BERT**

I. Introduction

In the last decade we have witnessed unprecedented growth of user data across various application segments and a lot of it is from social media interactions. To understand various trends, which are essential for many businesses, we not only have to process this huge data but also in an efficient manner to minimize capital expenditure (capex) and operating expenses (opex). To handle such requirements posed by big-data, traditional CPU and GPU based systems are increasingly displaced by application specific chips/systems and one broad category is MLA based systems.

MLAs are being rapidly adapted as the preferred option for carrying out various computations these days. A few well known examples are Google's TPU [1], Microsoft's Brainwave [2], and many other similar systems that are running large scale training and inference workloads for various applications. Today most big companies that are dealing with large data, deploy custom hardware to accelerate these applications. Examples include large scale machine learning models with trillions of parameters. For example, e-commerce giants like Amazon, Alibaba and social media companies deploy a whole range of models to understand user behaviors to recommend products, content based on their past interactions and insights.

Designing MLA based systems to achieve best performance and energy efficiency requires a careful design space exploration and detailed modeling to understand the workload behaviors in order to make near-optimal architectural decisions. A detailed performance modeling and cycle-level simulation is often too costly in terms of schedule (takes few months, if not years) and developers time (10s of person-years). Designing these systems is not only difficult but it requires complex simulation framework and performance modeling infrastructure, which is often only applicable to a set of architectures and doesn't extend well to other accelerator families.

Early in the exploration phase, one of the proven approaches across the industry is to understand the high level behavior of the system and various bottlenecks arising from insufficient compute, memory or communication resources. In this paper, we propose a generic performance model framework, known as MBR-sim, to address the need of analyzing various kind of general and domain-specific MLAs. Our framework treats a system as a collection of "rate engines" which work together to achieve a common goal. The proposed framework takes a system configuration as a set of tune-able parameters and runs the simulated accelerator against any workload.

The rest of the paper is organized as follows: Section II presents a brief overview of related work. Section III provides additional detail about the motivation. Section IV presents our detailed methodology and how ML models are mapped to various hardware resources. We present experimental setup and analysis in Section V and potential future work in Section VI before concluding our findings in Section VII.

II. Related Works

This section presents a brief overview of workload parallelism and various other ML Accelerator Simulators.

*This work was done when this Author was not affiliated to any of his current or previous organizations or employers.

A. ML Parallelism Overview

1) Pipeline Parallelism: Pipeline parallelism (PP), also known as streaming, is based on assembly line production. This mechanism often achieves best utilization of resources, minimizes idle time at the cost of end-to-end latency. As shown in Figure 1 (A), we divide a workload graph into N equal stages or sub-graphs and then map them into N hardware resources capable of executing them efficiently such that each unit finishes the assigned jobs in roughly equal time [3]. PP is effectively used in various inference applications where achieving best throughput and highest utilization of resources is more important than inference latency.

2) Model Parallelism: In modern ML systems, we deploy ML models that are quite large. These are few trillion parameter models and are hard to store in a single device. In order to achieve best performance, these models are broken into multiple sub-graphs which are executed across multiple devices. This kind of parallelism is known as a naive model parallelism (MP). A more sophisticated model parallelism micro-batches the batched model in such a way as is shown in Figure 1 (C), the deep learning model being partitioned across multiple machines. True model parallelism allows the partitioned sections to be run concurrently for different inputs similar to a software pipeline approach [3].

3) Data Parallelism: Data parallel (DP) and batched mode are often used interchangeably where the processing steps across a set of compute resource is exactly same. However, the input to these resources are different as in from different sources or users. As shown in Figure 1 (B) we split the data (consist of large number of inputs) into multiple nodes, where each node operates or does same compute on these data in parallel [3]. DP is quite popular in inference application where the network are rather simpler and there is a lot of data to process across a large number of compute blocks to achieve high throughput.

4) Tensor Parallelism: Finally, large models also process huge tensors and in many cases the processing elements level memory may not be enough to hold these large tensors. In that case, tensors (either activation or weights or both) divided into multiple chunks and streamed to processing elements one after another. This kind of parallelism is known as Tensor parallelism (TP). As shown in Figure 1 (D), we are breaking a weight into multiple chunks and assigning to a set of compute elements to reduce the processing time. However, depending upon the way we partition these tensors, it might results into increased communication and might hurt overall utilization and performance.

B. Putting It Together

In MBR-sim, we primarily explore PP due to inference model use case and relatively smaller workloads. Some large layers are assigned to multiple processing elements which follows TP. Going forward, we do plan to support DP and MP as part of our future exploration of large models.

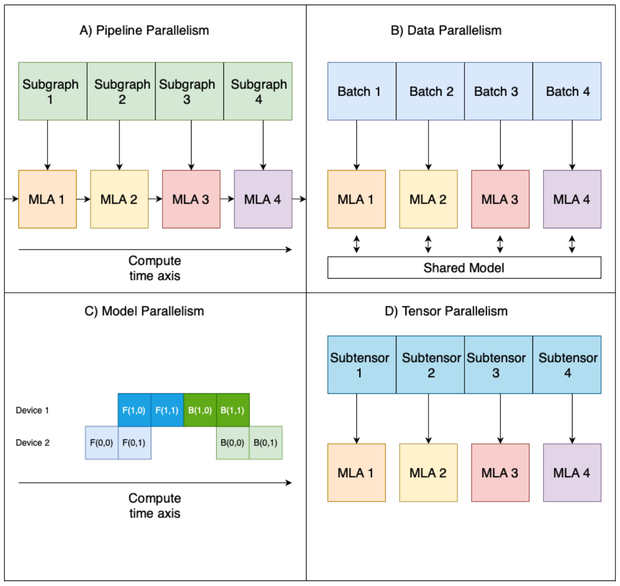

Fig. 1. Types of ML Workload Parallelism

III. Motivation

A. ML Accelerator Simulators

There are a wide variety of ML Accelerator Simulators. Some simulators are built to test chips that the project created, like PUMA, while others try to simulate a wide vareity of silicon. Simulators like timeloop, STONNE, and MAESTRO focused on Deep Neural Networks, while Sparseloop was unique in focusing on sparse tensor accelerators. The salient features of each simulator are listed in Table I.

TABLE I
Other ML Accelerator Simulators

Name	Main Features
STONNE [4]	- Cycle-level microarchitectural simulator - Evaluation of both dense and sparse real, unmodified DNN models
MAESTRO [6]	- Showcases execution time and energy efficiency of a data flow for a DNN model
Sparseloop [8]	- Models sparse tensor accelerators - Comprehends a large set of architecture specifications, including various data flow and acceleration features
Timeloop [10]	- Includes a mapper that has the best way to schedule operations which allows accurate projections - Dataflow and memory hierarchy co-design allow for optimizing energy efficiency shown through model
PUMAsim [9]	- Simulator specifically created as part of overall PUMA work. - Incorporates functionality, timing, and power models for the architecture

Table I lists various open-source ML accelerator simulators and clearly shows the need for a generic performance model which can simulate a number of workloads from a simpler description for a set of architectures. A high-level analytical model like SOL has many use cases in the whole exploration process. Early in the design phase, it can help understand the workload behavior. For example, we can understand which layers are compute vs memory vs communication bound based on arithmetic intensity.

Secondly, it can be used as cost-model for MLtools and to help understand for example which quantization and sparsity level optimizations bring more benefits while maintaining the desired accuracy.

Thirdly, SOL can also help ML compiler to understand the efficacy of various mapping and scheduling decisions and provide guidance about various heuristics to improve the overall performance.

Finally, an end-to-end SOL model can provide early projections of key workloads and help enable the comparison with other systems and competitive analysis to help justify the choice of architecture. It can also point out to various bottlenecks which can be addressed via hardware-software co-optimization to build better systems.

IV. Our Methodology

This section presents our detailed modeling and high-level description of each phase. To simulate a workload, we need a fairly high-level description of the workload which can be either layer or task level with the right parameters such as tensor sizes, data types, etc. We also need various system components described as the knobs to the framework.

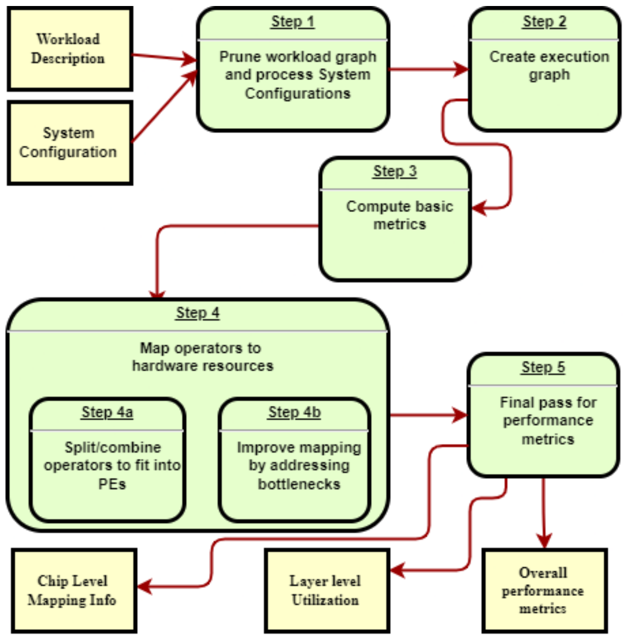
Workload Description
System Configuration
Step 1
Prune workload graph and process System Configurations
Step 2
Create execution graph
Step 3
Compute basic metrics
Step 4
Map operators to hardware resources
Step 4a
Split/combine operators to fit into PEs
Step 4b
Improve mapping by addressing bottlenecks
Step 5
Final pass for performance metrics
Chip Level Mapping Info
Layer level Utilization
Overall performance metrics

Fig. 2. SOL methodology and various steps

A. Workloads Under Consideration

There are 6 different workloads represented in this study. The main two are ResNet50 (CNN) and BERT-Small (NLP). The SOL model can accept any other similar workloads, covering a wide variety of ML models. In this paper, MobileNet-Base, MobileNet-Large, BERT-Base, and BERT-Lare were also tested. Some of the key metrics of the two primary workloads are listed in Table II.

TABLE II
Workload Information

Workload	MACS(Bil.)	Layers	Wgts(Mil.)
ResNet50v1.5 (MLPerf)	4.089	122	25.5
BERT-Small (Encoder)	5.35	15	13.5
BERT-Small (Workload)	130	15	340

B. Workload Description

A workload for MBR-Sim requires specific information to be accurately represented. For each layer, we need its name and the type that the layer might be (MatMul, SoftMax, Convolution, etc.) These two pieces of information are the most crucial in determining the effective utilization and certain characteristics of the layer. The model also needs the dimensions for the input and output tensors, and the weight tensors if this is a linear layer. This information is used to determine the sizes of each tensor, and it also helps estimate the overall cycles.

The SOL model assumes that all tensors have 3 dimensions for Input and Output, with 4 dimensions for Weight. For any layer with fewer dimensions, the trailing dimensions are assigned a value of 1 to keep our model simplistic. The last piece of information required by our simulation is the datatype for each different tensor for each layer. The simulation is based on the Int8 datatype, however, conversions between that and Fp16, Int16, and more data types are automatically handled within the simulation.

Testing additional workloads is not a challenge, however, the simulation may need additional configuration information. Performance for SIMD Operations that are not listed will need to be added, and any Linear Operations beyond MatMul and Convolution will require additional support.

C. Cycles Estimations

There are a few assumptions as a part of the model. One of the assumptions is that inter-chip communication will not be a limiting factor for most cases and chip-to-chip bandwidth is adequate. This means that as data is transferred between chip resources, these bandwidth limitations will not have a significant impact on the overall performance.

Another assumption made is that most workloads are mapped in the processes defined below. Alternate mapping solutions will impact the overall performance in a way that is unpredictable, so the current mapping algorithm is not specialized based on the workload. Another assumption is the general performance of SIMD operations (Add, Multiply, SoftMax, Eltwise Add, etc.). The performance of said operations is dependent on the actual silicon being modeled, but for base testing, a relative performance is assumed.

To calculate chip-level information, there are a couple equations in place. For the overall throughput, if we assume a pipeline parallelism configuration is in use, then the worst-case stage cycles are used to base most calculations. If, however, the configuration is set to tensor parallelism, then the throughput

Algorithm 1 Metric Equations for Pipeline Parallelism

```
$MACUtilization \leftarrow (totalMACS/((MACBW) * tiles))/maxLayerCycles$
$IPS/Chip \leftarrow systemFreq/maxLayerCycles$
$Latency \leftarrow 1/(IPS/Chip)$
```

is calculated by the total cycles in the layers divided by the number of tiles.

D. Mapping

The current mapping algorithm has two stages. The first stage consists of one pass through all of the layers that need to be mapped. If the number of layers is greater than the number of tiles available, the simulator's priority is to combine the smallest layers together (as long as the combined layers weight does not exceed the capacity). It will only combine two layers that are adjacent to each other within the workload. To start this selection, the simulator finds the smallest layer and then finds its smallest adjacent layer, using the layer cycles to sort and calculate. However, if the number of layers is less than the number of tiles available, the the simulator will focus on splitting the largest tiles in half and decreasing the sizes of the bottleneck layers. The largest tiles (compared using layer cycles) are split in half through each tensor (Input, Output, Weight).

Algorithm 2 Mapping Stage 1 Algorithm

```
$nodes \leftarrow []$    ▷ Nodes is a sorted array by layer cycles
$targetNumTiles \leftarrow 64$
while $nodes.length \neq targetNumTiles$ do
    if $nodes.length < targetNumTiles$ then
        $largestNode \leftarrow$ maxLayer($nodes$)
        $splitNodes \leftarrow$ splitNode($largestNode$)    ▷ Function defined in appendix
        $nodes$.remove($largestNode$)
        $nodes$.add($splitNodes$)
    else if $nodes.length > targetNumTiles$ then
        $sNode \leftarrow$ minLayer($nodes$)
        $pairedNodes \leftarrow$ adjacentLayers($sNode$)    ▷ Function defined in appendix
        $pNode \leftarrow$ minLayer($pairedNodes$)
        $cNode \leftarrow$ combine($sNode$, $pNode$)    ▷ Function defined in appendix
        $nodes$.remove($sNode$, $pNode$)
        $nodes$.add($cNode$)
    end if
end while
```

The second stage is to manage the bottlenecks and optimize the tiles efficiently. For this, the simulator iterates in a loop where it seeks out the 2 smallest adjacent layers and the largest layer. If the combined cycle count for the smallest layers is less than the current largest layer's cycles, it makes sense to combine the smallest layers and break the largest layer.

This makes sense since the system minimizes the bottlenecks by shrinking the number of outliers on the smaller and larger layers. To keep layer splitting simple, all layers are split in half, bringing each split layer closer to the average layer cycles across the silicon. This system of splitting and combining was inspired by a Binary Tree.

By combining the smallest layers the cycles are added together, while the cycles from the largest layer are halved since the layer is now mapped to two separate tiles. This decreases the max cycles on the chip as the current bottleneck layer was split in half. Splitting and combining layers will keep the number of tiles utilized consistent, while also decreasing the overall number of bottlenecks. During the combination and splitting of layers, the model utilizes the algorithms from the prior pass. This continues to occur until the case that the smallest layers when combined would have larger cycles than the current largest node, thus increasing the bottleneck layer.

Algorithm 3 Mapping Stage 2 Algorithm

```
$nodes \leftarrow [ListofNodes]$
$excludeNodes \leftarrow []$
while $nodes.length \neq 0$ do
    $sNode \leftarrow$ minLayer($nodes$)
    $pairedNodes \leftarrow$ adjacentLayers($sNode$)    ▷ Function defined in appendix
    $pNode \leftarrow$ minLayer($pairedNodes$)
    $lNode \leftarrow$ maxLayer($nodes$)
    if $sNode.cycle + pNode.cycle > lNode.cycle$ then
        $nodes$.remove($sNode$)
        $excludeNodes$.add($sNode$)
    else
        $splitNodes \leftarrow$ splitNode($lNode$)    ▷ Function defined in appendix
        $nodes$.remove($largestNode$).add($splitNodes$)
        $cNode \leftarrow$ combine($sNode$, $pNode$)    ▷ Function defined in appendix
        $nodes$.remove($sNode$, $pNode$).add($cNode$)
    end if
end while
$nodes \leftarrow excludeNodes$
```

V. Experimental Analysis

This section presents our experimental setup and a few configurations to help understand the key results, insights, and various other interesting features. To test the current model, experiments utilized a sample base hardware configuration. The key characteristics of this configuration are listed below:

A. System Configurations

To experiment and utilize our model, we needed to create a sample system. For example, we needed to determine the Elements/cycle for SIMD Performance. We determined the values below by analyzing kernel performance for our sample hardware, along with estimating the relative performance of each layer compared to each other

Alongside SIMD Performance, there were additional configurations. These included the relative conversion factors between different datatypes and some simulation parameters. These are the knobs that can easily be modified to see

TABLE III
SIMD PERFORMANCE INT8/FP16* (ELEMENTS/CYCLE)

SIMD Layer Type	SCALE	RELU	ADD/MUL	TRANSPOSE	GELU*	MAXPOOL	AVGPOOL	SFMAX*	LYRNORM*
Performance (Elem/cy)	16	16	16	8	8	4	2	2	2

performance differences between different, but very similar chips. Using these knobs, we can easily conduct sensitivity analyses and find the relative performances of different chips.

TABLE IV
BASELINE ML ACCELERATOR HARDWARE CONFIGURATION

Simulation params	**Value**
Number of Tiles (N)	64
GEMM Throughput (MACs/cy)	1024
NoC Bandwidth (B/cy)	32
SIMD Vector Width (bit)	512
Tile Weight Capacity (KB)	2048
Clock Freq (MHz)	1000

B. Primary Results

The primary focus was on MAC Utilization and IPS/Chip projections. In Figure 3, the IPS/Chip varies between each workload. However, MAC utilization is very consistent for ResNet50 and BERT workloads, suggesting that they are bottlenecked by GEMM operation. The later sensitivity analyses performed will help us decipher these bottlenecks and see the relative performance of different designs.

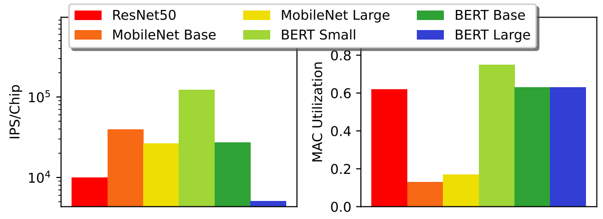

Fig. 3. Primary results for MAC utilization and IPS/Chip. BERT-Large is a single encoder result

One thing to note is that all BERT results are for a single encoder running on a single chip. This means that the full BERT workloads will require one chip/encoder layer with a batch size equal to the of encoder layers to achieve this IPS.

C. Mapping Information at Tile Level

Figures 4 and 5 show the mapping information for ResNet50 and BERT-Small. Due to the mapping algorithms 1 and 2, the actual tiles have relatively even loads. The largest tile has layer cycles less than 2x those of the smallest tile, thus fitting within our requirements for a “balanced” distribution. The Layer IDs correspond to specific linear layers within the workload.

D. MAC Sensitivity

To understand the scaling due to higher MAC resources, we present the MAC sensitivity study in Figure 6. The baseline system is 1024 MAC bandwidth. Clearly, all workloads, excluding MobileNet, are limited by MAC Bandwidth. Each

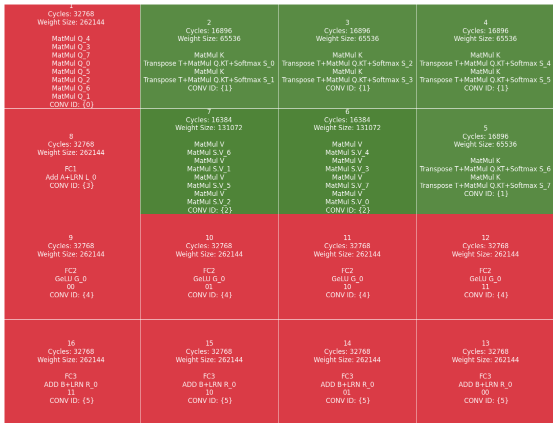

Fig. 4. Mapping information BERT-Small (1 Encoder) mapped in a 16 tile system with a 2048 KB Tile Weight Capacity. Even though there are some bottleneck tiles, it is hard to reduce cycles because other tiles’ cycles are in the same order as bottleneck layer tiles.

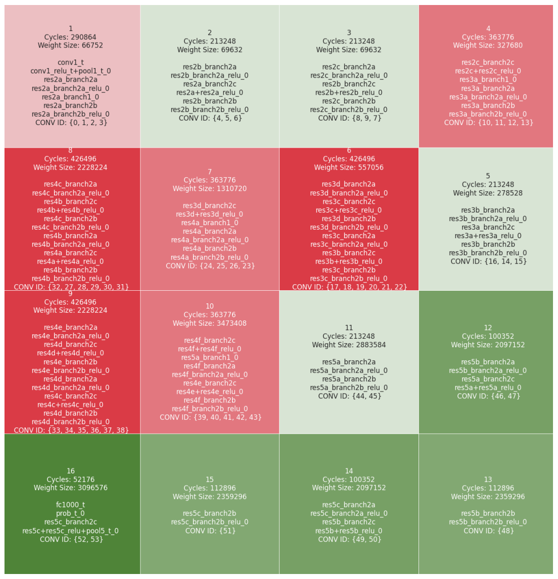

Fig. 5. Mapping information ResNet50 mapped in a 16 tile system with a 4096 KB Tile Weight Capacity.

time the MAC BW doubles, the throughput also doubles. With this information, it is easy to understand that these workloads in the current system are bottlenecked by GEMM hardware.

E. SIMD Sensitivity

Another sensitivity analysis could be of the SIMD vector width. The model can look at a variety of different options to showcase estimated performance. In Figure 7, the SIMD Vector Width for the same hardware configuration is showcased.

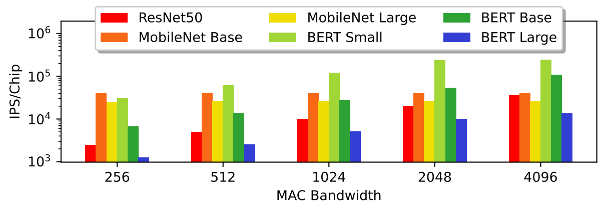

Fig. 6. GEMM Throughput Sensitivity Analysis

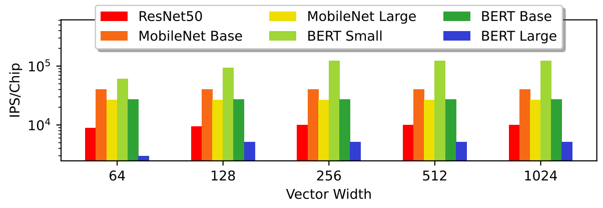

Fig. 7. The SIMD Vector width sensitivity analysis shows that only at 128-bit wide do bottlenecks appear.

F. NoC Sensitivity

Similarly, a NoC sensitivity study can be performed on the workloads. In Figure 8, the results are showcased.

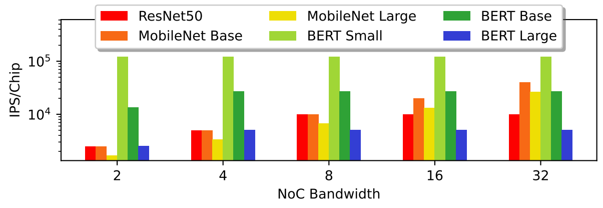

Fig. 8. The NoC bandwidth sensitivity analysis shows the performance of 5 different bandwidths. Performance degradation is only seen in the lowest bandwidth of 4 B/cy.

G. Tile Weight Capacity

Another sensitivity analysis included in the model is testing specific Tile Weight Capacities. For example, in Figure 9, we see that our Baseline Configuration would see a nearly 50% improvement in performance by doubling the tile weight capacity for BERT. However, for ResNet50, the performance would roughly remain unchanged.

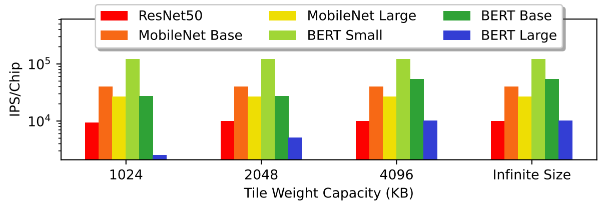

Fig. 9. The tile weight sensitivity analysis for the sample hardware configuration

H. Source Code

Our SOL project is open-sourced through GitHub at https://github.com/MBR-sim/MBR-sim. While it is free for academic purposes, for commercial use cases explicit written approval is required from the authors of this paper.

VI. Future Work

In the current version of the SOL model, we primarily use PP which is efficient for inference applications to achieve high throughput and good resource utilization. However, PP also results in relatively higher latency and for training workloads and we may have to use other types of parallelism.

In the next version of the simulator, we plan to support various other kinds of parallelism along with more workloads. The current version supports CNN and NLP BERT-like workloads. Going forward, our plan is to enable support for auto-regression models like GPT-3 and recommendation system which often rely on large embedding tables and sparseNN compute.

VII. Conclusion

Today we deal with an unprecedented scale of data which is generated from various social media interactions and other similar sources. In order to process and gain insights from such huge data, we have successfully deployed a set of machine learning algorithms. These algorithms are incredibly complex, and ultimately require specialized hardware in order to minimize inference time and power consumption. Many special use case tools have been developed to analyze the performance of these models to see how to improve them, but nobody has focused on creating such tools to analyze models on ANY hardware. In this paper, we proposed an SOL performance modeling tool which abstracts away any unnecessary hardware complexities while providing the needed granularity to effectively tune any model which you so desire.

References

[1] N. Jouppi et. al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”, Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA), June 2017 Pages 1–12
[2] J. Fowers et. al., “A Configurable Cloud-Scale DNN Processor for Real-Time AI”, Proceedings of the 45th International Symposium on Computer Architecture (ISCA), June 2018
[3] T. Ben-Nun and T. Hoefler, “Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis,” arXiv, Sep. 2018, Pages 7-18.
[4] F. Muñoz-Martínez et. al.“STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators”, IEEE Computer Architecture Letters, 1 July-Dec. 2921, vol. 20, no. 2, Pages 122-125
[5] N. Binkert et. al., “The gem5 simulator”, AGM SIGARCH Computer Architecture News, May 2011, vol. 39, no. 2, Pages 1-27
[6] H. Kwon et. al. “Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach”, MICRO ’52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, October 2019 Pages 754-768
[7] V. J. Reddi et. al., “MLPerf Inference Benchmark,” 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, Pages 446-459
[8] Y. Wu et. al. “Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling,” 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO), 2022 Pages 1377-1395.
[9] A. Ankit et. al. “PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference,” ASPLOS ’19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April 2019 Pages 715-731.
[10] A. Parashar et al., ”Timeloop: A Systematic Approach to DNN Accelerator Evaluation,” 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2019, Pages 304-315

Appendix

Pseudo-code function definitions

Algorithm 4 Definition of combineNode()

```
$fNode$                                   ▷ This is first node to combined
$sNode$                                   ▷ This is second node to combined
$cNode \leftarrow fNode$.copy()
$cNode.outputTensor \leftarrow sNode.outputTensor$
$cNode.macs \leftarrow fNode.macs + sNode.macs$
$cNode.simd \leftarrow fNode.simd + sNode.simd$
```

Algorithm 5 Definition of splitNode()

```
$oNode$                                   ▷ This is the original node to be split
$sNodes \leftarrow []$                    ▷ Array of split Nodes
$i \leftarrow 0$
while $i < 2$ do
    $sNode \leftarrow oNode$.copy()
    $sNode.outputTensor \leftarrow oNode.outputTensor/2$
    $sNode.inputTensor \leftarrow oNode.inputTensor/2$
    $sNode.weightTensor \leftarrow oNode.weightTensor/2$
    $sNode.macs \leftarrow oNode.macs/2$
    $sNode.simd \leftarrow oNode.simd/2$
    $sNodes$.add($sNode$)
    $i \leftarrow i + 1$
end while
```

In Algorithm 6, each node has a unique ID. This unique ID corresponds to the nodes location within the workload. A lower ID means that the node occurred earlier in the workload.

Algorithm 6 Definition of adjacentLayers()

```
$oNode$                                   ▷ This is the original node
$nodes \leftarrow []$                     ▷ Array of all Nodes
$aNodes \leftarrow []$                    ▷ All adjacent nodes, currently empty
while $nodes.length \neq 0$ do
    $node \leftarrow nodes$.remove(0)
    if $node.ID = oNode.ID - 1$ then
        $aNodes$.add($node$)
    else if $node.ID = oNode.ID$ + 1 then
        $aNodes$.add($node$)
    end if
end while
$adjacentNode \leftarrow$ minLayer($aNodes$)
```