# Kernel Mechanisms for Efficient GPU Accelerated Deep Neural Network Inference on Embedded Devices

Hemant Nigam

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science and Software Engineering

University of Washington

2018

Reading Committee:

Michael Stiber, Chair

Erika Parsons

Yosinori Watanabe

Kelvin Sung

Program Authorized to Offer Degree:

Computing & Software Systems

University of Washington

## Abstract

Kernel Mechanisms for Efficient GPU Accelerated Deep Neural Network Inference on Embedded Devices

Hemant Nigam

Chair of the Supervisory Committee:
Professor Michael Stiber, Ph.D.
Computing & Software Systems

Embedded platforms with integrated graphics processing units (GPUs) are popular choices, for use-cases, like Autonomous machines, to run the Deep Neural Networks (DNNs) inference workload. However, due to a rapid increase in data volume, DNN inference is becoming even more computationally intensive and memory sensitive, which necessitates a mechanism for improving DNN inference efficiency on existing embedded systems.

This Master's thesis investigates the memory sensitivity of DNN inference – specifically, the impact of off-chip memory (DRAM) contention on DNN inference performance. It demonstrates a prototype *GPU aware memory isolation mechanism*: a locking mechanism in the GPU driver to

reduce DRAM contention caused by multicore CPUs, thus improving DNN inference efficiency. Experiments performed on a Jetson TX2 board running the Linux4Tegra OS shows the benefits of our proposed mechanism, with up to 13.5% speedup of a micro-benchmark and up to 41% and 86% speedup of two object detection benchmarks.

# ACKNOWLEDGEMENTS

First of all, I would like to thank my chair Professor Michael Stiber for his guidance and excellent support during the project. Ever since my very first graduate course on multimedia and signal computing, to my independent study on neural networks, and to this thesis work, Professor Stiber has always been a great teacher and advisor.

Further, I would like to express my appreciation to my thesis committee members Professor Erika Parsons and Professor Kelvin Sung for offering their time and encouragement.

Special thanks to Dr. Yosinori Watanabe for his continued mentorship and support, and encouraging me to pursue Master's program. He has been my source of inspiration and motivation ever since I worked under his leadership at Cadence Design Systems. Working with him was the most fulfilling and humbling experience of my life.

I would also like to thank my internship mentor Allen Martin at Nvidia for his guidance and generous support enabling me to work on the latest Nvidia platform in this thesis.

# DEDICATION

To Neha,

my loving and caring wife,

and my parents,

whose sacrifice, unconditional support and tolerance

made it possible for me to complete this work.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# GLOSSARY

ANN - Artificial neural network.

API - Application programming interface.

AR - Augmented reality.

ASIC - Application specific integrated circuit.

Average IPC - Mean of IPC reported for different test-cases like solo, corun1, corun2, etc.

Average MPS - Mean of MPS reported for different test-cases like solo, corun1, corun2, etc.

BW - Memory bandwidth.

BW_LIMIT - Memory bandwidth limit assigned to each CPU core.

BW_LIMIT_1000 - Label reporting data when CPU cores are assigned memory bandwidth limit of 1000 MB/s.

BW_LIMIT_3000 - Label reporting data when CPU cores are assigned memory bandwidth limit of 3000 MB/s.

BWLOCK - Bandwidth lock. A bandwidth reservation mechanism.

CNN - Convolution neural network.

Corun1 – Label reporting data for test-case when one instance of traffic generator (memory intensive application) is running on core1 along with benchmark running on core0.

Corun2 – Label reporting data for test-case when two instance of traffic generator (memory intensive application) is running on core1 and core2 along with benchmark running on core0.

Corun3 – Label reporting data for test-case when three instance of traffic generator (memory intensive application) is running on core1, core2, and core3 along with benchmark running on core0.

Corun4 – Label reporting data for test-case when four instance of traffic generator (memory intensive application) is running on core1, core2, core3, and core4 along with benchmark running on core0.

Corun5 – Label reporting data for test-case when five instance of traffic generator (memory intensive application) is running on core1, core2, core3, core4, and core5 along with benchmark running on core0.

COTS - Commercial off the shelf products.

CPU - Central processing unit, e.g., ARMv8, Intel x86.

CUDA - Compute unified device architecture developed for Nvidia’s GPU.

DMA - Direct memory access. When peripherals access off-chip memory (DRAM) asynchronously and independently of CPU.

DNN - Deep neural network. A type of artificial neural network with more than one hidden layer.

DRAM - Dynamic random access memory. Off-chip memory used in modern hardware.

Driver - Software that exposes hardware capabilities to the userspace applications and other kernel space drivers.

DSP - Digital signal processor

FC - Fully connected. A type of DNN hidden layer.

FPGA - Field programmable gate arrays.

GB - Gigabytes.

GPU - Graphics processing unit.

HW - Hardware.

HWA - Hardware dedicated for specific function like video encoding and decoding.

IPC - Instruction per cycle. The average number of instructions executed for each clock cycle of a CPU or GPU. IPC may vary depending on the software being run on a given CPU or GPU and overall hardware architecture, specially the memory hierarchy. IPC is often used as a performance metric for a specific algorithm.

ISP - Image signal processor.

KT - DRAM throughput achieved by a CUDA kernel running on GPU.

Kernel space - A region in memory (DRAM) exclusively reserved for running operating system (OS) kernel and device drivers.

LLC - Last level cache in a CPU's memory hierarchy. For example, L2 cache is the LLC in ARM Cortex A57 and Nvidia Denver CPU.

MB - Megabytes.

MLP - Multi-layer perceptron.

MMIO - Memory mapped input-output.

MMU - Memory management unit.

MPS - Megapixels per second. Megapixels of image processed by object detection benchmark in a second.

MSE - Mean square error.

NN - Neural network.

NO_BW_LIMIT - Label reporting data when GPU aware memory isolation mechanism is disabled.

OS - Operating system.

RET - Relative execution time. Ratio of the execution time of CUDA kernel being measured to the execution time of baseline CUDA kernel.

SGD - Stochastic gradient descent. An optimization algorithm used in neural networks.

SMP - Streaming multiprocessor. A GPU block consisting of CUDA cores, registers, shared memory, etc.

Solo – Label reporting data when only benchmark is running on core0.

Speedup - A number representing relative performance of benchmark run in two different scenarios - 1) GPU aware memory isolation mechanism is enabled. 2) GPU aware memory isolation mechanism is disabled.

SRM - Stall reason memory. A metric representing percentage of time GPU streaming multiprocessor (SMP) is stalled due to either memory throttling or due to the outstanding memory request.

STG - Synthetic traffic generator. A program created for generating synthetic DRAM traffic.

SW - Software.

Userspace - A region in memory (DRAM) used by software application.

VR - Virtual reality.

WMB - Winograd micro-benchmark. Benchmark created using Winograd convolution kernel from Nvidia's cuDNN library.

# Chapter 1. Introduction

## 1.1 Motivation

Deep Neural Networks (DNNs) are increasingly deployed in many artificial intelligence applications, like computer vision [37] [57] [69], speech recognition [70], and other prediction tasks [22]. This has been mainly due to an increase in the prediction accuracy of DNNs, which has been made possible by the availability of large data-sets and the substantial computational power in the form of hardware accelerators, like Graphics Processing Units (GPUs). DNN accuracy has either exceeded traditional approaches [4], or it has increased to human level accuracy for tasks, like conversational speech recognition [3].

Given their high accuracy, there is an increasing demand to embed DNN inference in applications running on embedded devices used in domains, like consumer electronics (mobile phones, AR/VR), robotics, autonomous driving and surveillance (drones). These embedded devices are constrained by limited memory, computational capabilities and power budget [1]. As such, they cannot easily cope with the associated computational complexity of a DNN model, like VGGnet [13], which typically consists of more than 100 million parameters and memory footprint ranging between 200-500 MB. Similarly, AlexNet [10], a popular Deep Convolutional Neural Network (CNN) model, requires 1.45 billion operations per input image with 240MB weights [8]. That's why DNN inference efficiency has become the focus of NN researchers and industry experts.

Hardware accelerators (e.g. GPUs [40] [77], Field Programmable Gate Arrays (FPGAs) [14] [80], and Application Specific Integrated Circuits (ASICs) [1] [41]), Quantization techniques [71] - [75], DNN model optimization techniques, like Pruning [76] - [79], and Software accelerators, like Nvidia's TensorRT [19] have been utilized to improve the efficiency of the DNN. Broadly, the aforementioned approaches can be categorized as follows 1) Improving computational efficiency, 2) Improving memory efficiency, or 3) Improving both computational and memory efficiency.

GPUs combined with quantization, pruning and SW accelerators, like TensorRT are the most widely used techniques to improve the inference execution, attributed to GPUs' high throughput and memory bandwidth. Therefore, common off the shelf (COTS) embedded systems with integrated GPUs, like Jetson TX2 [38] are becoming increasingly popular for DNN inference deployment. Such embedded systems are typically designed as heterogeneous systems; a system comprising of more than one kind of compute units, like CPU, GPU, DSP[1], etc. for better energy efficiency, sharing single off-chip memory[2] (DRAM). However, shared hardware resources, like off-chip memory are prone to contention when accessed by multiple tasks running on multi-core CPUs, leading to performance degradation of GPU accelerated DNN inference tasks.

To illustrate the problem stated above, we evaluate two GPU accelerated object detection benchmarks - YOLO and DetectNet, executed on Jetson TX2 embedded platform (6 CPU cores + 256 core GPU). Figure 1 shows the result of an experiment using YOLO benchmark (left side of the image) and DetectNet benchmark (right side of the image) in 6 different test scenarios labeled as solo and corun[1-5] on X-axis. Y-axis shows the performance (recorded as the number of megapixels processed by benchmark in a second (MPS)) relative to performance reported in Solo. Solo reports the performance when the benchmark is run in isolation (without any CPU application (co-runner) running on other CPU cores) on core 0. *Corun1* shows the relative performance when the benchmark is co-scheduled with one memory intensive CPU application running on Core 1. Similarly, *Corun2, Corun3, Corun4* and *Corun5* show relative performance when the benchmark is co-scheduled with two, three, four and five memory intensive CPU applications (co-runners) running on CPU core[1-2], core[1-3], core[1-4] and core[1-5] respectively. From the figure, we can observe that benchmark performance of both YOLO and DetectNet degrades by ~50% when co-scheduled with five other memory intensive CPU applications (Corun5). The primary cause of performance degradation is the single DRAM shared between CPU and integrated GPU on Jetson TX2 platform.

Our study found that the impact of DRAM contention on DNN inference workload running on the embedded platform has been largely overlooked. This context motivates us to investigate the

[1] DSP is an abbreviation for digital signal processor

[2] The terms "memory" and "DRAM" have been used interchangeably in this document

impact of DRAM contention on DNN inference efficiency running on COTS embedded system (Jetson TX2) and design OS kernel mechanism that will improve the inference efficiency on embedded systems.

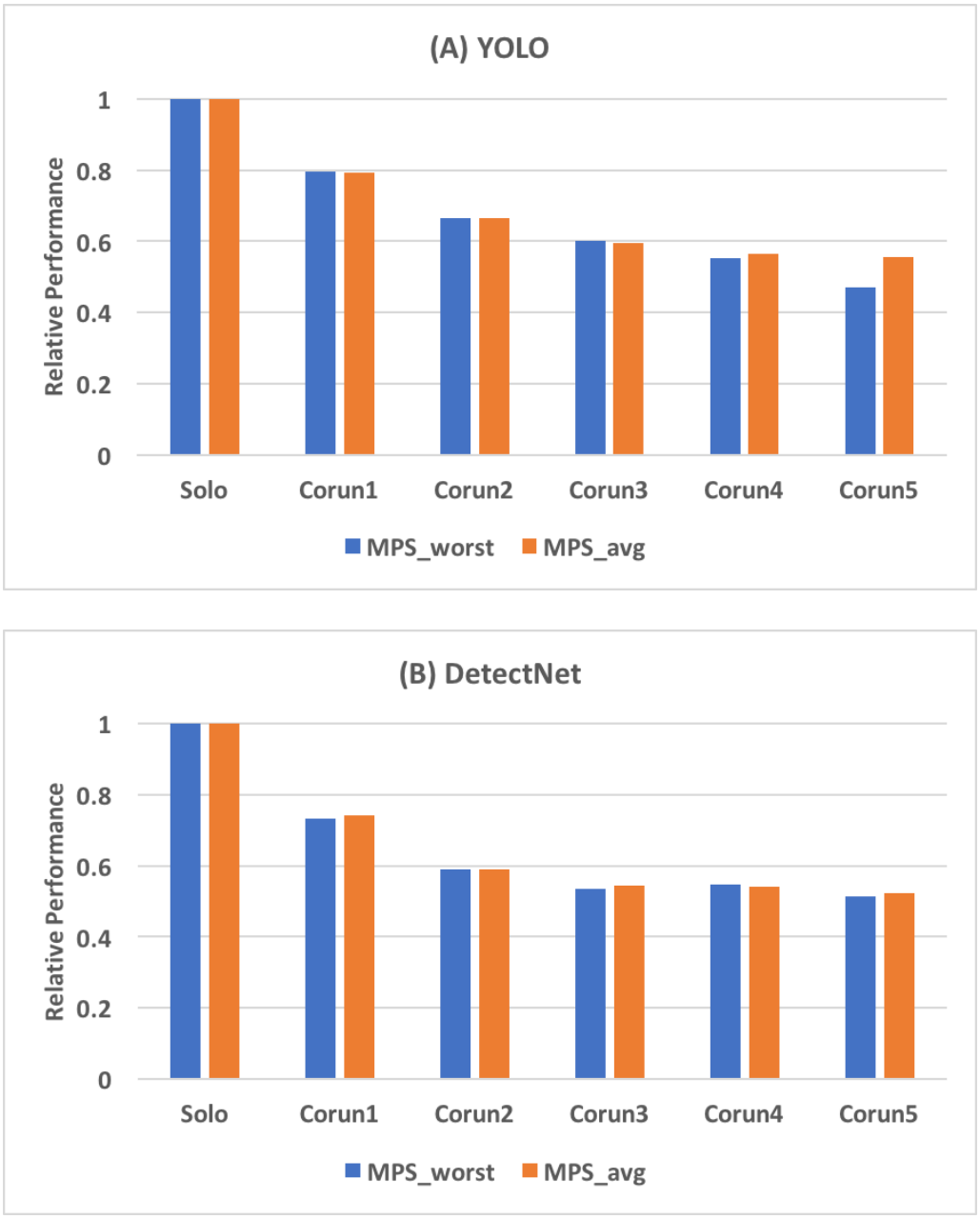


Figure 1. Example performance analysis of YOLO (A) and DetectNet (B) benchmarks. Performance is measured in terms of megapixels of an image processed, abbreviated as MPS, in a second. MPS_worst shows the worst-case performance, MPS_avg shows the average performance.

## 1.2 Research Goal and Contribution

The main objective of this thesis is to investigate how OS kernel-based mechanisms can be used to increase the efficiency of Deep Neural Network (DNN) inference by reducing DRAM contention on multi-core embedded systems with integrated GPU.

This thesis makes following main contributions:

— Investigates the DNN inference benchmarks on Jetson TX2 embedded platform.

— Examines the correlation between DRAM contention and compute-intensive Winograd kernel used in convolution layers of DNNs.

— Extends the memory (DRAM) bandwidth isolation mechanism "MemGuard" [45] to Jetson TX2 (An ARM-based embedded platform with integrated Nvidia GPU)

— Evaluates the impact of DRAM (memory) bandwidth isolation mechanism on DNN inference benchmark executing on Jetson TX2.

— Documents the inference execution flow at the system level (includes both hardware (HW) and software (SW)).

## 1.3 Thesis outline

This thesis has been structured in such a way that the background concepts are discussed first, so the reader has necessary foundation required to understand the methodology and analysis results.

Chapter 2 introduces the concepts used in this thesis - Neural Networks and Embedded Systems running Neural Network inference tasks. First, DNNs are introduced, followed by a discussion about training and inference. Later, Convolutional Neural Networks (CNNs), a variant of DNNs, are presented, followed by a discussion about target embedded system used in this thesis.

Chapter 3 discusses related work by reviewing software mechanisms for increasing efficiency of GPU accelerated tasks, which forms the foundation of our work on improving DNN inference efficiency by reducing DRAM contention.

Chapter 4 introduces two object detection benchmarks, a micro-benchmark, and a synthetic traffic generator, used for evaluation of the proposed mechanism.

Chapter 5 discusses the preliminary investigation done using the benchmarks discussed in Chapter 4.

Chapter 6 presents the design and implementation of a GPU aware bandwidth isolation mechanism to reduce DRAM contention causing DNN inference slowdown.

Chapter 7 provides the detailed testing and evaluation results of the proposed GPU aware bandwidth isolation mechanism. It begins with a summary of experiment setup, followed by performance analysis in terms of reduction in Instruction per cycle (IPC) slowdown.

Chapter 8 summarizes the observation and issues encountered during the thesis followed by the suggestion on future work.

# Chapter 2. Background and Concepts

The discussion below is based upon *Deep Learning* [24] book by Goodfellow, Bengio and Courville, *Neural Network Design* [25] book by Hagan et al. and Stanford's *CSS231n: Convolution Neural Network for Visual Recognition* course material [26]. The aforementioned materials would be a great starting point for anyone interested in DNN.

## 2.1 Artificial Neural Networks

Artificial neural networks, and, deep neural networks, in particular, have been researched for over three decades. In fact, the earliest research in the field of the neural network can be traced back to as early as 1943 when Warren McCulloch, a neurophysiologist, and Walter Pitts, a mathematician wrote a paper on how neurons might work using electrical circuits. Since then several key research developments happened which formed the foundation of DNNs being used today (Figure 2).

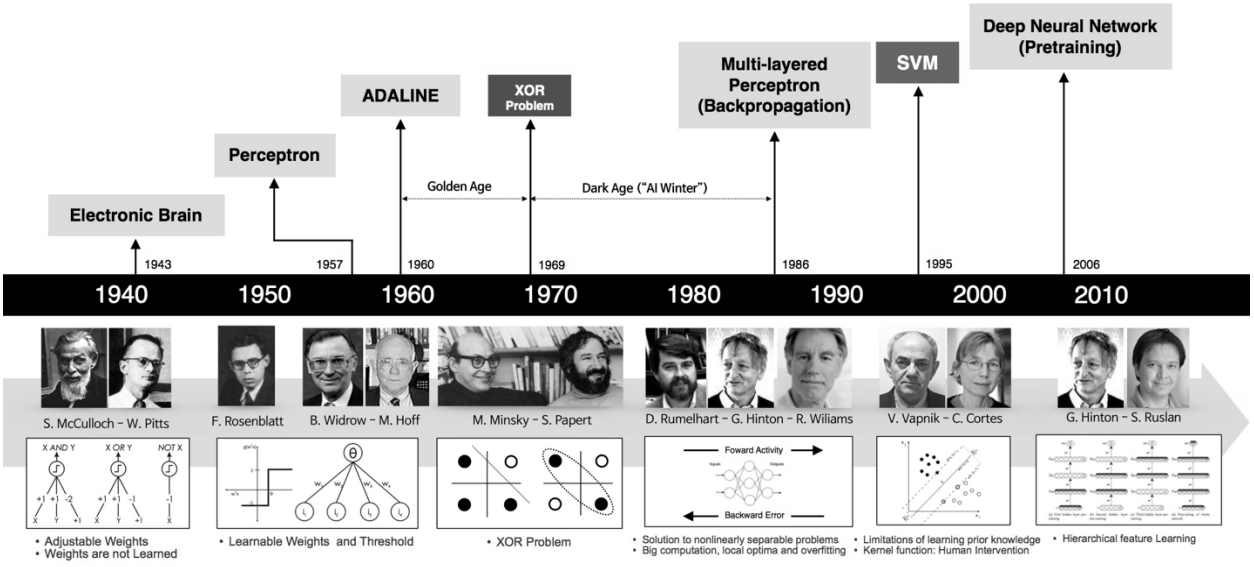


Figure 2. History of Neural Network (Image adapted from [89])

However, it was not until 2012, when DNN went from a just a research topic to being so mainstream. This was made possible primarily because of the availability of massive amount of compute power in the form of GPU computing, required to train and execute these DNNs and the availability of massive labeled datasets required for training.

Before discussing DNNs, it is essential to understand what is a Neural Network, or more precisely an Artificial Neural Network (ANN). ANNs are a complex network of artificial neurons initially inspired by biological functioning and structure of the human brain. The human brain is responsible for utilizing sensory inputs from surrounding environment to make complex day-to-day decisions, to learn and remember an event for a period. The human brain can be thought of as a complex computational system, consisting of approximately 100 billion biological neurons, connected by 10-100 trillion synapses [27]. Each neuron consists of a cell body which receives input signals from its dendrites and produces output signals at its axon. Axons are further connected to dendrites of other neurons via synapses. Synapses are responsible for the transfer of output signal from one neuron to the other by tweaking the signal intensity [27].

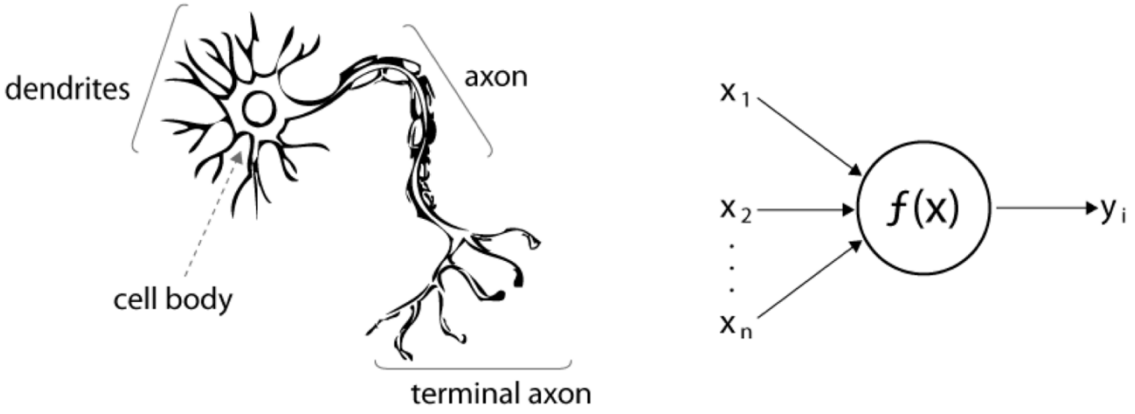


Figure 3. Biological Neuron (Left) vs. Artificial Neuron (right) (Image adapted from [87])

The artificial neuron (shown in Figure 3) is the basic unit of ANNs. It consists of an *activation function* which is analogous to synapses (shown in Figure 4) found in a biological neuron. An artificial neuron receives several input signals $X_i$ from other artificial neurons in the ANN. The input signal could be an image pixel or audio samples based on the domain of problem to which ANN is applied. Each neural input is assigned a configurable weight also known as synaptic weight $W_i$. The neuron sums the product of each neural input and corresponding synaptic weight. The resultant sum is offset with a bias value $b$ to make the ANN model more general. This neuron output is further fed into activation function $f$ to generate the final output $p$.

$$p = f(X_i * W_i + b) \tag{1}$$

The weight $W_i$ can be thought of as tuning parameter that controls the behavior of the neuron when subjected to external stimuli. The weight parameter can be tweaked to approximate desired final output value [26].

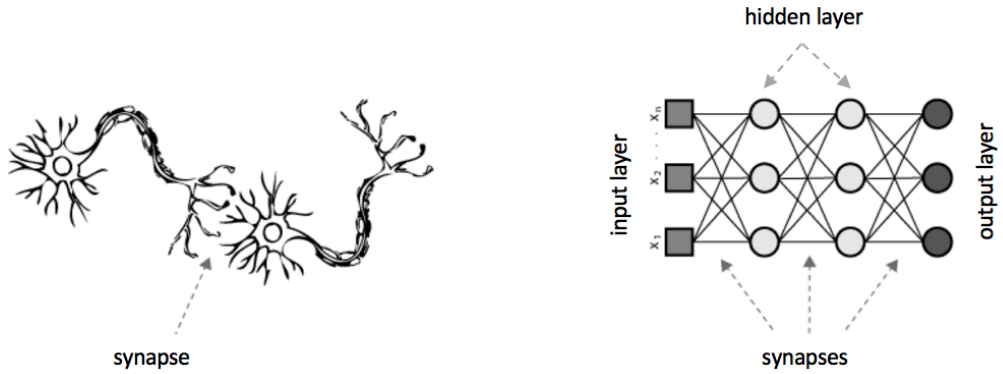


Figure 4. Biological synapse (left) vs. Artificial synapses in ANN (right) (Image adapted from [87])

Activation function *f* could be a sigmoid function (shown in Figure 5) that maps input in range {-∞, +∞} to output in range {0,1}

$$Sigmoid(x) = \frac{e^x}{e^x + 1} \tag{2}$$

or it could be a rectified linear unit (ReLu) (shown in Figure 5) which filters all negative values and passes all positive values as output.

$$ReLu\ (x) = \max(0, x) \tag{3}$$

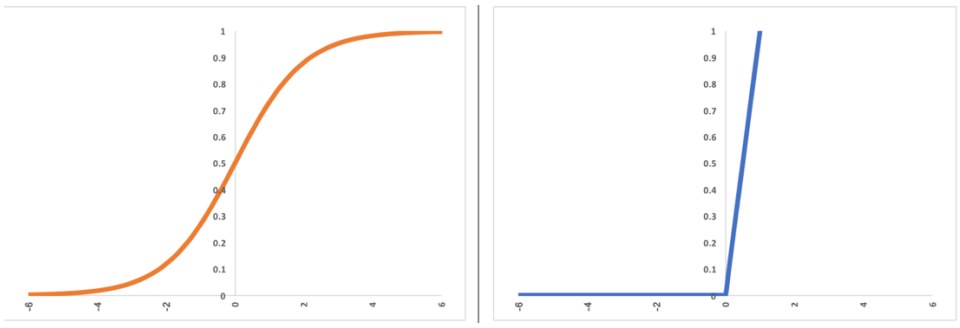

Figure 5. Activation Functions- Sigmoid (Left) vs ReLu (Right)

## 2.2 Deep Neural Networks

The neurons in an ANN are organized in the form of a layer. For instance, Figure 4 (right) shows an ANN with one input layer, one output layer, and two hidden layers. An ANN with one layer can approximate only simple function of the inputs. To model complex neural network, more intermediate layers would be required. These intermediate layers are known as hidden layers. An ANN with multiple hidden layers is popularly known as a DNN. There is no consensus, however, on how many hidden layers are required for ANN to qualify as DNN.

### 2.2.1 DNN Training

DNN parameters, i.e., synaptic weights $W_i$ and bias $B_i$ are not fixed values, but they are learned during DNN training phase. The goal of training phase is to tune the weights and biases so that the training error $E$ of the entire DNN model is minimum. Training error $E$ can be calculated using loss function which varies according to the task being performed. Intuitively, the loss function calculates how much the model output deviates from the desired output. Mean square error (MSE) is one of the popular loss functions. It is calculated by summation of the square of the difference of expected output $Y_i$ and actual model output $X_i$.

$$MSE = \frac{1}{n}\sum_{i=0}^{n-1}[Y_i - X_i]^2 \tag{4}$$

#### 2.2.1.1 Backpropagation with Gradient Descent

Stochastic Gradient Descent (SGD) employed with the back-propagation algorithm is one of the most popular optimization techniques used for DNN training. Before starting the DNN training process, an input image is fed to the first layer of DNN. The training starts by assigning random values to the DNN network parameters (weights and biases). The output is calculated for each neuron layer by layer. This process is the *forward pass* of the back-propagation algorithm. The resulting final output at the last layer is compared against the desired output using loss function, like MSE to calculate training error $E$. This training error is minimized using SGD optimization technique. SGD does so by calculating the $\partial E/\partial W$ which is the effect each weight $W$ has on the computed training error. The gradients are calculated during the backward pass which involves back propagating training error from last or outermost layer to the first or innermost layer.

Equations 5 and 6 show the calculations that happen in a loop where each synaptic weight is updated by a small value $\Delta W$ which is proportional to the gradient of the training error $E$ with respect to weight $W$.

$$\Delta W = \partial E / \partial W \tag{5}$$

$$W_{new} = W_{old} - \alpha * \Delta W \tag{6}$$

$\alpha$ is a constant value also known as learning rate. The higher learning rate can reduce the training time in some circumstances, but can also lead to unstable behavior when network parameters ($W$ and $B$) change too much in each iteration, preventing the learning algorithm from converging on an optimal solution [25] [26].

### 2.2.2 DNN Inference

The inference is the stage where a pre-trained model; a DNN model which is already trained using the training process described earlier in Section 2.1.1.3, is used to infer or predict the unseen test samples. The inference stage includes the same forward pass as the training stage to predict the final values. Unlike training, the inference stage doesn't include a backward pass for computing the training error and updating the weights. As Figure 6 illustrates, DNN training, unlike inference, involves a backward pass which propagates the error calculated after the forward pass through the DNN layers and updates their weights.

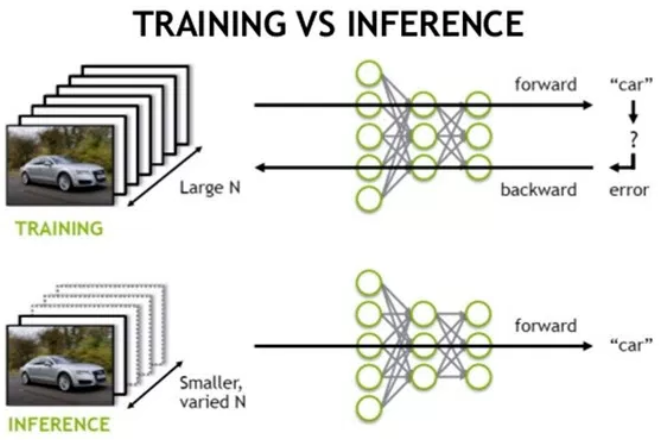


Figure 6. Training vs. Inference (Image adapted from [19])

## 2.3 Convolution Neural Networks

A convolutional neural network (CNN) is inspired by visual cortex, part of the brain which is responsible for visual cognition. The visual cortex appears to have multiple regions devoted to visual processing, which to a coarse approximation can be thought of as a sequence of operations that extract features of increasing complexity from visual input. This was first demonstrated in experiments by David Hubel and Torsten Wiesel in 1962. Their experiment showed that certain neurons in the visual cortex respond to only features of certain orientation [28]. Similarly, CNNs use several neuron layers, and feature extractors called *filters* to extract various attributes, like edges from an image.

CNNs have been widely used for image classification task where an input image is classified as one of the target classes. It can also be implemented to generate probabilities of each target class. For example, a CNN for animal image classification might predict the likelihood of an image as follows: Cat = 0.7, Dog=0.2, and Goat=0.1.

CNN can also be thought of as a specialized form of feedforward neural networks. The main difference is that instead of using a flat array as input, CNN uses the actual 3D (2D if the color channel is excluded) image as an input and performs several operations including convolutions to generate output.

### 2.3.1 CNN Layers

A typical CNN consists of the following layers:

#### 2.3.1.1 Convolution Layer

Convolution layers are one of the most compute-intensive layers in a CNN, and a standard setup can account for 90% of CNN total compute time [30]. A convolution layer accepts raw image pixels as input in the form of a 2D matrix of dimension NxP. During forward propagation, the convolution layer performs element-wise multiplication between the input image and a 2D weight matrix filter (sometimes also referred as *kernel*). The filter is essentially a 2D matrix which is designed based on the type of image features being extracted. For instance, Figure 7 illustrates convolution operation between an input image (I) and Sobel vertical edge detection kernel (K) [97], which extracts vertical edges from the input image. The input image shown in blue color is a

6x6 matrix, and the edge detection kernel shown in green color is a 3x3 matrix. During convolution operation, the kernel is slid across the spatial dimension (width and height) of the input image, and per-element multiplication between entries of the kernel and the corresponding entry of the input image is calculated and added. In Figure 7, orange sub-matrix in the input image shows the current sliding window of the kernel for entry *5* (element $I_{3,3}$ of input image matrix) located at the center of the orange matrix. The output of per-element multiplication of orange sub-matrix and kernel is a 3x3 matrix shown in purple color. The elements of the 3x3 purple matrix are added to generate the convolution output corresponding to element $I_{3,3}$ shown in red color.

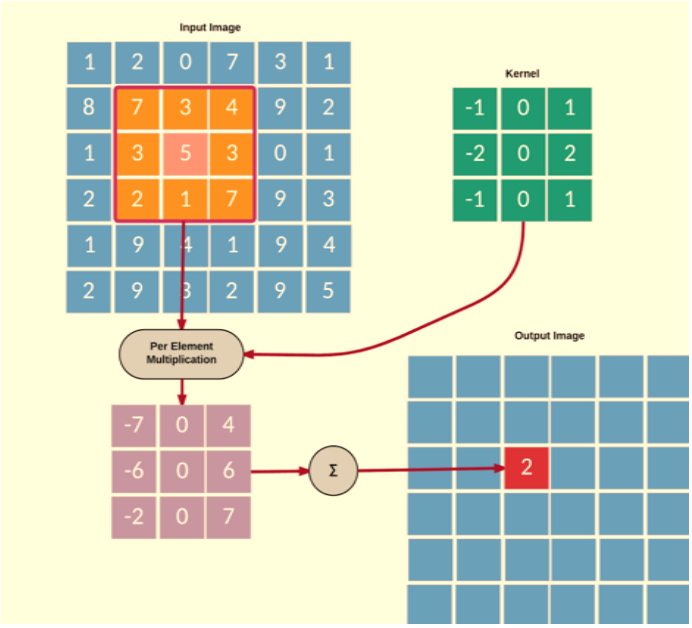


Figure 7. Convolution of input image and kernel (Sobel edge detection)

A convolutional layer can also be configured to capture more than one feature at once by deploying multiple filters. For example, one filter may be used for detecting horizontal edges in an input image; and another filter may be used for detecting vertical edges.

### 2.3.1.2 Max-Pooling Layer

A max-pooling layer is generally inserted between successive convolution layer in a CNN. It is responsible for down-sampling input features, received from preceding convolution layer, along the spatial dimension (width and height) without doing any actual learning, like convolution layers. Down-sampling (reduced number of features) is done for two main reasons – 1) Reduce the computation required in the succeeding convolution layer 2) Reduce probability of overfitting.

During forward propagation, the pooling layer takes MxM region of input feature and generates single value S, the maximum value in the MxM region. So, for input feature of dimension NxN, the pooling output will be of dimension (N/M) x (N/M).

#### 2.3.1.3 Fully Connected Layer

Fully connected (FC) layer is based on traditional Multi-Layer Perceptron (MLP), a class of feedforward neural network with at least one hidden layer, where each neuron in the previous layer is connected to every neuron in the next layer. The FC layer can be thought of as a reasoning layer which takes high-level features generated by previous layers, which could be a convolution, pooling or another FC layer, and tries to learn a non-linear combination of these features to classify the input images into different categories based on training dataset [29]. A typical CNN for image recognition uses alternating convolution and max-pooling layers followed by a couple of fully connected layers.

## 2.4 Winograd Algorithm

As discussed in the Section 2.3 , convolution is the most compute-intensive layer in CNN. Though direct convolution operation is simple; it performs poorly on GPUs [65]. Therefore, researchers have focused on developing alternative methods for convolution operation, and the Winograd minimal filtering algorithm is one of them. It was initially proposed by Copper Smith Winograd in 1980 [32]. The Winograd minimal filtering algorithm has been adapted for DNN convolution operation and is available in popular DNN libraries, like cuDNN. Winograd-based convolution reduces the computational requirement by reducing the number of multiplication operations and compensating it with more addition operations and intermediate products [65]. This leads to an increase in the memory storage requirement of the Winograd based convolution operation, eventually increasing the pressure on off-chip DRAM [66]. To understand the memory sensitivity of Winograd based convolution kernel running on the embedded platform, we created a micro-benchmark which is discussed in Section 4.2 .

A Winograd convolution, represented by $C(m \times n, r \times s)$, using Winograd's minimal filtering algorithm, requires $(m+r-1) * (n+s-1)$ multiplication operations for computing convolution of a $(m \times n)$ image with a $(r \times s)$ filter [103]. Whereas, a direct convolution, represented by $M(m \times n, r \times s)$, using matrix multiplication method requires $(m*n*r*s)$ multiplication operations for

computing the same convolution [103]. So, *C(6 x 6, 3 x 3)* uses (6+3-1)*(6+3-1) = 64 multiplication operations, whereas *M(6 x 6, 3 x 3)* uses (6*6*3*3) = 324 multiplication operations. This is an algorithm complexity reduction of ~5 (324/64). More discussion about algorithmic complexity of Winograd's minimal filtering based algorithm can be found in [103].

## 2.5 DNN on Embedded Systems

### 2.5.1 Embedded SoC Architecture with integrated GPU vs. discrete GPU

The NVIDIA Jetson TX2 [68] is a heterogeneous SoC featured in the NVIDIA Jetson Development board series. It is the first embedded processor to have same advanced features and architecture as a modern desktop GPU while still using the low power draw of a mobile chip[3].

#### 2.5.1.1 Specification

As shown in Figure 8, the Jetson TX2 employs a heterogeneous SoC ("Parker") comprising of a CPU subsystem with a quad-core 2.0-GHz 64-bit ARMv8 A57 processor; a dual-core 2.0-GHz 7-way superscalar[4] ARMv8 Denver processor configured in a heterogeneous multiprocessor[5] (HMP) configuration, and an integrated Pascal GPU with 256 CUDA cores. Memory subsystem consists of two 2-MB L2 caches, one shared by the four A57 cores and one shared by the two Denver cores. Both L2 caches support coherency and connect to off-chip DRAM memory through a proprietary coherent interconnect fabric (CPU switch fabric in Figure 8). The GPU has two streaming multiprocessors (SMs), each providing 128 1.3-GHz CUDA cores that share a 512-KB L2 cache. The six CPU cores and integrated GPU share 8 GB of 1.866-GHz LPDDR4 DRAM memory. Apart from the CPU subsystem, the memory subsystem, and the GPU, the Jetson TX2 SoC comprises of few other cores and hardware accelerator (HWA). For instance, there are ARM Cortex R5[6]-based cluster for providing always-on and advanced security functionality. Then there are HWAs, like Camera ISP, Video encodes (nvenc), Video decodes (nvdec), 2D Graphics and

[3] Jetson TX2 achieves 1 TFlops single precision peak performance at < 11 W) [38]

[4] Superscalar processor can execute more than 1 instruction per clock cycle

[5] Processor with heterogeneous CPU cores

[6] Cortex R5 is a real-time microprocessor from ARM

Display controllers, all sharing the same memory bus (Memory fabric and arbitration in Figure 8) for access to off-chip DRAM.

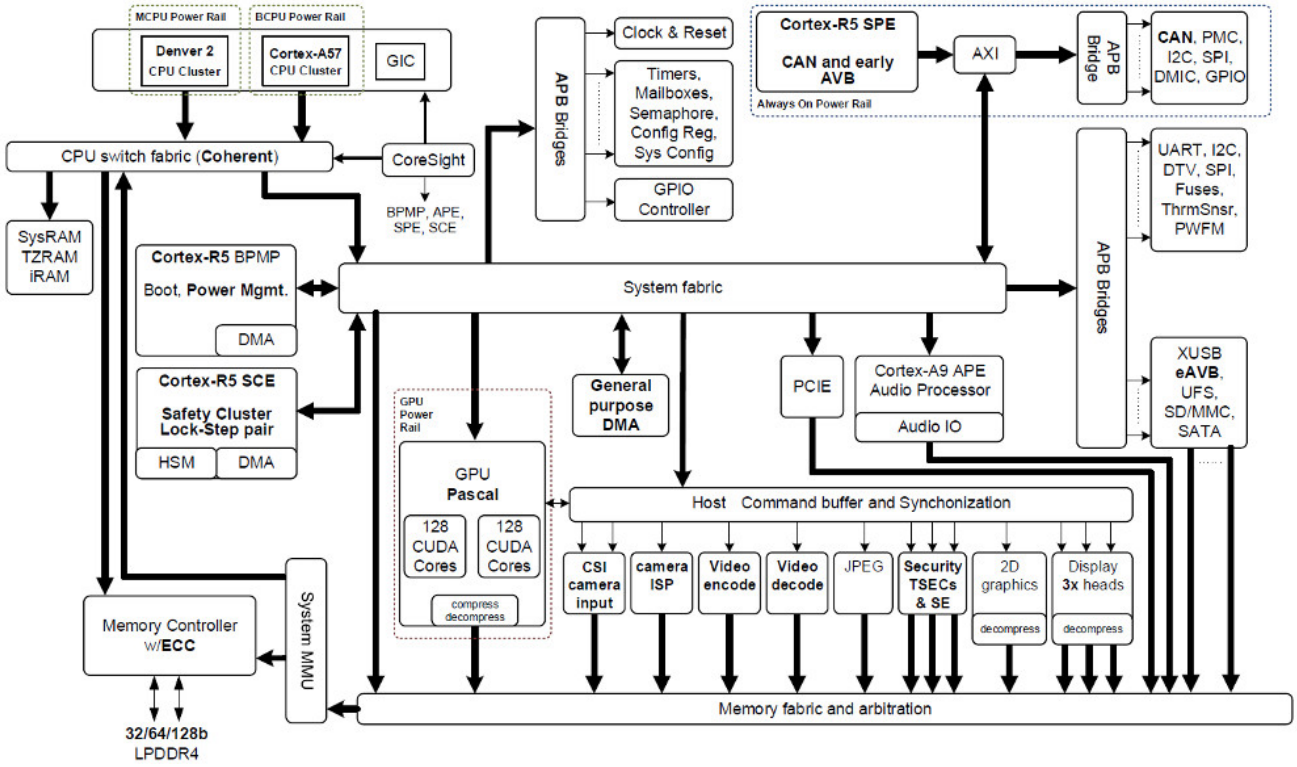


Figure 8. Jetson tx2 (Parker SoC) Architecture (Image taken from [68])

### 2.5.2 Challenges in Embedded SoC

Modern embedded SoCs integrate numerous computational cores and hardware accelerators (HWA) on a single chip to increase the energy efficiency of the overall system [84] [85], unlike desktop systems that have discrete accelerators or engine communicating over the PCI-E bus. In case of Jetson TX2, the GPU is integrated onto the same chip with CPU clusters and shares the same off-chip DRAM. This avoids the overhead of moving data between system DRAM and dedicated GPU DRAM, like in the case of a system with discrete GPU [64]. However, integrated GPUs may cause other overheads arising due to shared DRAM. For instance, Jetson TX2 has a single DRAM shared between 2 CPU clusters – quad-core A57 cluster and dual-core Denver2 cluster with their own Last level cache (LLC), Pascal GPU and HWAs, like nvenc, nvdec, etc. Each of these DRAM clients can cause contention at DRAM.

Figure 9 shows a simplified view of the Jetson TX2 architecture with notable contention points indicated using circles. Contention point (1) is caused by CPU and GPU instructions triggering shared L2 cache misses or memory transactions initiated by HWAs, like nvenc using system DMA. Point (2) is another probable contention point caused at the memory fabric shared between integrated GPU and HWAs. Both these contention points can cause significant delays during execution of GPU accelerated tasks, like DNN inference that triggers multiple DRAM accesses in parallel. Unlike SoCs with integrated GPU shown in Figure 9, a system with discrete GPU, like the one shown in Figure 10 does not have this issue as discrete GPUs have their own dedicated DRAM. However, they may still experience contention at point 1 and 2 (shown in Figure 10) during initial movement of data from system DRAM to dedicated DRAM.

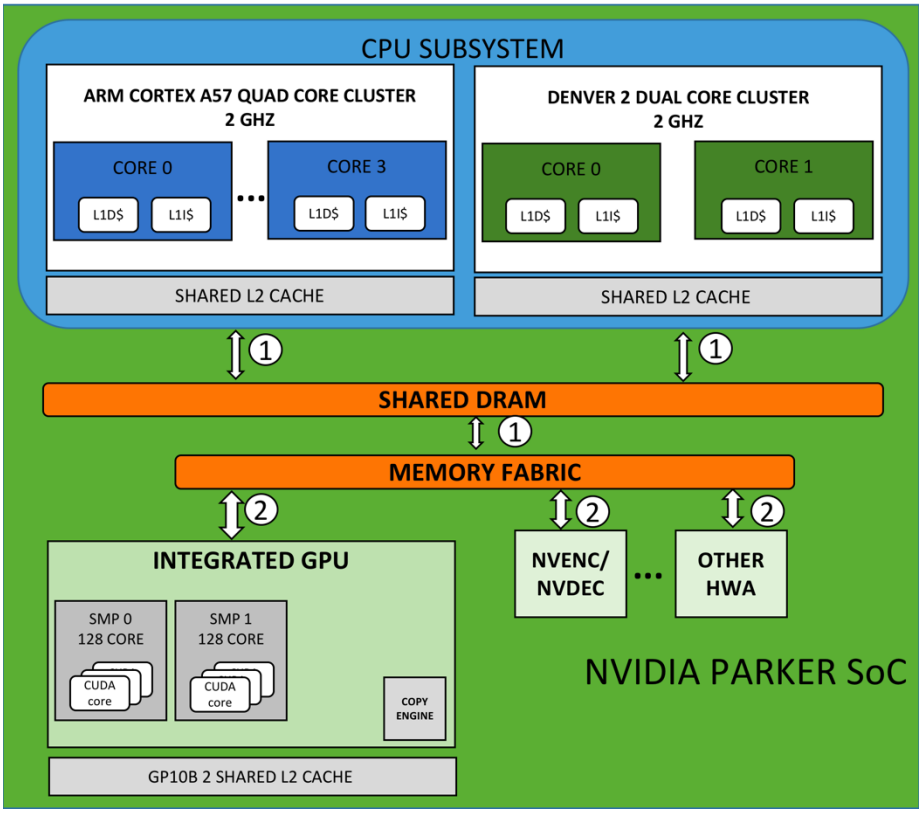

Figure 9. Jetson TX2 Embedded SoC with integrated GPU

### 2.5.3 DNN Pipeline at System Level

This section presents an overview of DNN inference pipeline at system level depicting HW as well as SW stack. The Nvidia GPU software stack consists of the GPU device driver (nvgpu), the memory manager (nvmap), the CUDA driver, the CUDA runtime, and userspace libraries, like cuDNN, cuBLAS, etc.

Figure 11 illustrates the system level view of a typical DNN inference execution on Jetson TX2 platform using an object detection application as an example. Object detection application is responsible for detecting objects in input image fed from video file or camera sensor. Internally it utilizes a pre-trained[7] DNN model whose parameters (weights and biases) are already optimized for object detection task. In the first step, object detection app invokes GPU accelerated CUDA kernels corresponding to different layers of DNN, like convolution or pooling layer using API[8] calls. These CUDA kernels can be hand-coded, or they can be part commercial off the shelf (COTS) libraries, like Nvidia cuDNN [40] [58], a userspace library of highly optimized primitives for DNN operations, like convolution, pooling, normalization, etc. One of the benefits of using cuDNN routine is that it works with arbitrary input dimension, i.e., based on input dimension and other configurations appropriate kernels are launched which maximizes occupancy, throughput, etc. This way the cuDNN library allows researchers to focus on tuning neural network architecture instead of low-level GPU performance tuning [58]. All benchmarks used in this thesis use the cuDNN primitives.

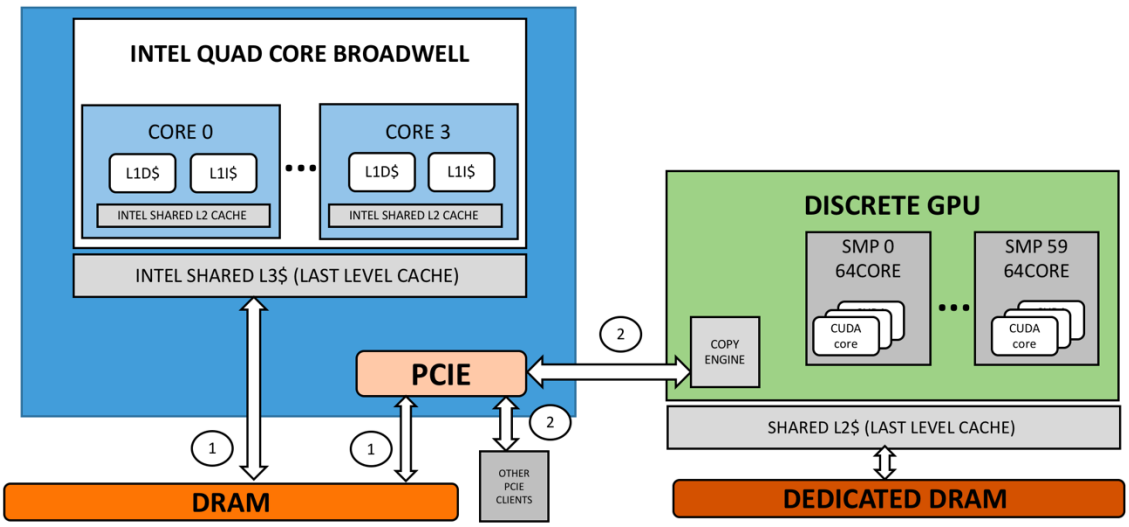


Figure 10 Reference Intel CPU with discrete GPU

[7] Pre-trained model is a DNN model which is already trained (i.e., network parameters, like weights and bias are already optimized for a particular application), and it can be deployed in the form of binary on target machines, like mobile.

[8] API refers to application programming interface

In the second step, CUDA kernels internally call CUDA runtime[9] (using APIs) to submit GPU specific commands to the GPU compute core (also referred as CUDA core). CUDA runtime acts as an interface between userspace program and underlying OS kernel.

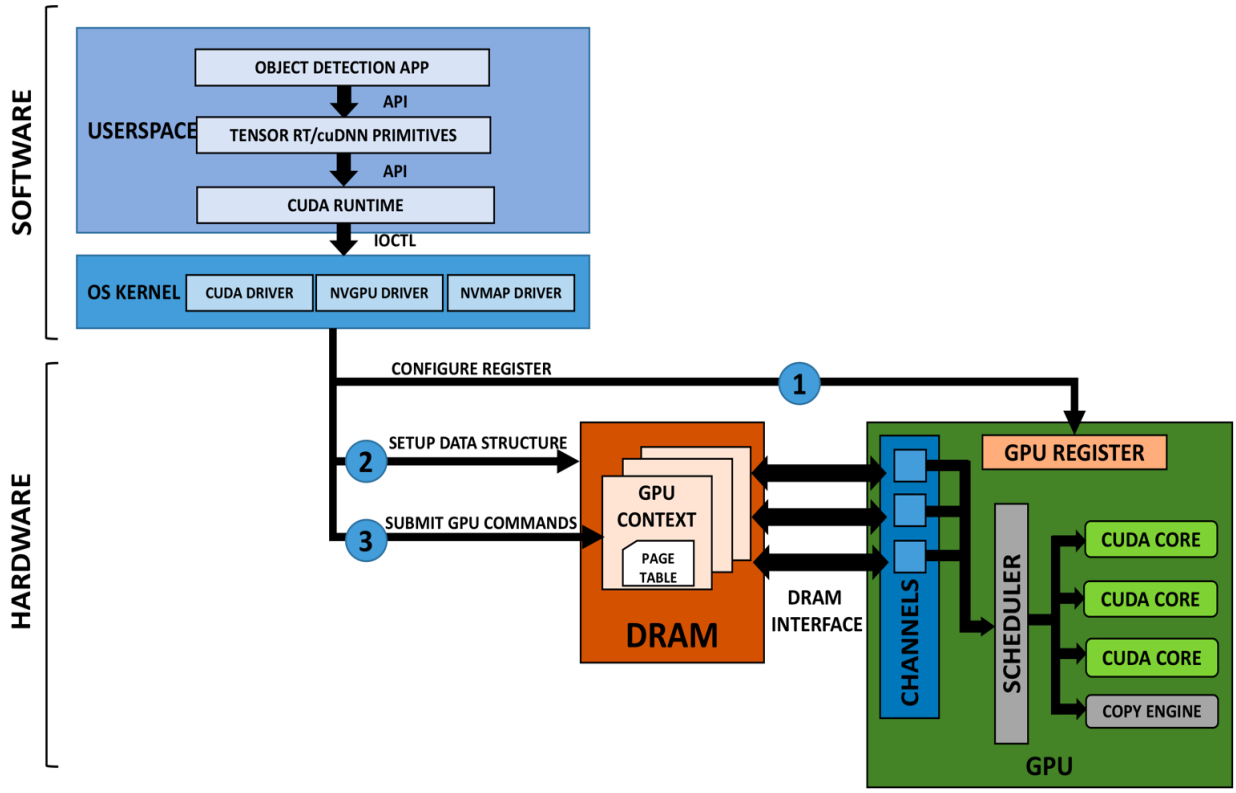


Figure 11. DNN Pipeline at System Level

At the OS kernel level, the GPU software stack provides a set of GPU commands (OS level representation of CUDA kernels found in userspace application) and IOCTL system calls to enable interaction between device driver (CUDA, nvgpu, nvmap) and the CUDA runtime engine to manage memory allocation, data movement, and the GPU command execution on the CUDA core. The GPU device driver is responsible for three main high level hardware operations – the GPU register configuration, data structure setup in the DRAM, and the GPU command submission. The

[9] CUDA runtime engine is a set of software libraries enabling general purpose computing (GPGPU) on CUDA supported Nvidia GPUs. It allows programmers to execute highly parallel algorithms, like convolution on thousands of compute cores on supported GPUs It provides APIs for context management, thread management, and memory management.

GPU register configuration is done using MMIO[10] for enabling or disabling a feature on the GPU hardware. The GPU device driver creates one or more GPU contexts, a data structure used by the device driver to maintain GPU state in DRAM. The GPU state contains information, like page table mapped to the GPU's virtual memory space in the OS kernel and the GPU fifo which is mapped to GPU channels in hardware. The GPU channels are directly attached to scheduler unit inside the GPU. The GPU scheduler unit is responsible for dispatching the GPU commands to the appropriate unit which could be a CUDA compute core or a GPU copy engine. In case of CUDA kernel launch, corresponding GPU command at OS driver level is submitted to one of the GPU channels associated with the corresponding GPU context. The GPU channel submits the GPU command to the scheduler which dispatches command for execution on one of the CUDA core.

### 2.5.4 Memory Management in Jetson TX2 using NVMAP

Nvidia provides "nvmap": a memory management driver for its Jetson platform running L4T OS. Nvmap coordinates with the existing Linux memory subsystem for virtual memory management including cache management, MMU handling, etc. It also supports features, like GPU page faulting and demand paging introduced in Nvidia's pascal-based GPUs (the one in Jetson TX2). These features make GPU programming a lot easier by eliminating the need for a developer to manage memory programmatically.

A GPU program typically starts by allocating memory in user-space virtual memory which is mapped to kernel space virtual memory using the Linux mmap[11] system call. Users can use this mapped memory to copy data buffer to the OS kernel efficiently. The driver also provides virtual memory for each GPU channel to support multi-tasking, and address of this device virtual memory is visible to user-space so that they can handle where to send data for computation.

[10] MMIO refers to memory mapped input/output (IO) operation which is used by OS drivers to access hardware registers.

[11] Mmap is a POSIX compliant system call for mapping file or device to memory

# Chapter 3. Related Work

Recently, a lot of research has been focused on using SW mechanisms to improve the performance of CPU or GPU accelerated tasks by improving memory efficiency of multi-core SoCs augmented with hardware accelerators, like GPU. These SW mechanisms can be broadly divided into three categories.

The first category is focused on cache aware partitioning technique [63] to reduce the impact of interference from co-running CPU applications by providing performance isolation of caches. In general, cache partitioning technique consists of following two steps. First, measurement of task or CPU core performance in terms of Instruction Per Cycle (IPC): average number of CPU instructions executed for each CPU clock cycle, or cache miss rate. Performance data is used to determine the cache set quota that needs to be allotted for a given optimal performance. Measurement is done via profiling [91] or monitoring [92]. Second, the optimal cache partition determined in the first step is enforced via cache ways partitioning [93] or cache level quota enforcement [94]. Cache Way partitioning works by allocating units of cache resource in terms of cache ways. For instance, a 16 ways set-associative cache can be divided into four equal partitions of 4-ways and allotted to each core of a quad-core CPU resulting in 25% bandwidth reservation for each core. The above mechanism can be enforced by modifying the cache replacement policy to make sure that a particular core or tasks (depends on partitioning granularity level) don't exceed the allotted quota. Similar to cache partitioning, DRAM partitioning technique [62] [95] works by allocating exclusive access of DRAM bank to a particular core or task.

The second category is focused on arbitration mechanism for providing deterministic memory access. For instance, Forsberg et al. presented GPUGuard [43], a memory arbitration mechanism to provide deterministic memory access to CPUs and integrated GPUs. GPUGuard is based on the PREM [44] execution model in which program execution is divided into compute and memory phases. These phases are scheduled at different time intervals to avoid any memory interference caused by the co-scheduled process. This approach requires an additional mechanism for global scheduling of compute and memory phases resulting in significant overhead. This approach also requires significant restructuring of existing application code.

The third category is focused on using HW performance counters for coarse-grained control over a task's memory bandwidth consumption. For example, Yun et al. proposed a memory performance isolation technique called MemGuard [45]. MemGuard is implemented as a Linux kernel module (LKM). It works by monitoring the per-core memory usage calculated by using HW performance counters. Once a CPU core exceeds its allotted memory bandwidth budget, an interrupt is generated causing suspension of all tasks running on that core. The tasks remain suspended until the interrupted core is allowed to use more memory bandwidth.

MemGuard incurred significant overhead due to coarse-grained control over task's bandwidth consumption. The overhead issue was addressed in BWLOCK [46] mechanism, an extension of MemGuard technique, which provides user-assisted fine-grained control over memory bandwidth. BWLOCK works by explicitly specifying a memory performance critical section in each application by using a lock API. On encountering the locks, the non-real-time (NRT) tasks are throttled based on pre-defined bandwidth allotted to them.

Even though MemGuard combined with BWLOCK reduces the overhead involved in bandwidth control, their practical applicability is limited because it has been verified on a server grade platform with Intel CPU and no integrated GPU. All the analysis was focused on the impact of co-running non-real-time (NRT) CPU threads on the target real-time (RT) CPU threads.

This thesis aims to apply MemGuard to high performance embedded SoC with ARM cores and integrated GPU, like Jetson TX2 and evaluates its impact on GPU accelerated bandwidth sensitive micro-benchmark (discussed in Section 4.2 ) and Neural Network inference benchmarks (discussed in Section 4.1 ).

## 3.1 MemGuard Overview

MemGuard is an OS kernel-level mechanism to isolate DRAM bandwidth available to a CPU core on a multicore platform. It does so by simulating a system that is comprised of N smaller uni-core subsystems, each core having its own dedicated memory subsystem. With the notion of the dedicated memory subsystem, each core is guaranteed a portion of overall system bandwidth based on user input.

MemGuard achieves bandwidth isolation by monitoring the aggregated memory request made by each CPU core. Memory requested by each core is counted by using the HW performance counter "PERF_COUNT_HW_CACHE_MISSES" available through Linux perf_event [67] interface. Yun. et al. demonstrated that memory bandwidth can be guaranteed by ensuring that the aggregate of memory requests from all core is less than the maximum achievable DRAM bandwidth.

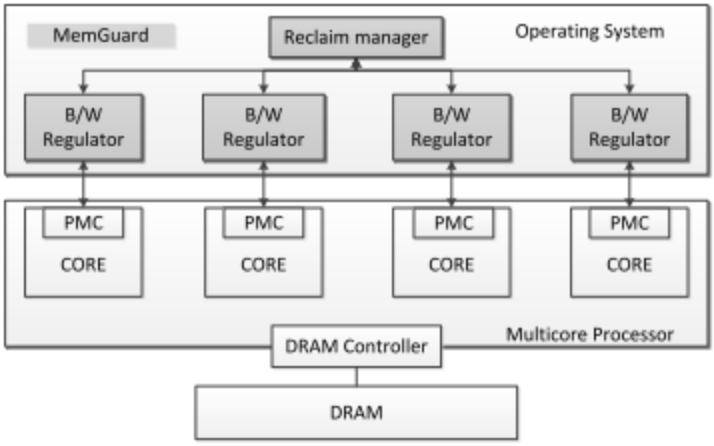


Figure 12. MemGuard Architecture (Image adapted from [45])

Figure 12 illustrates the original architecture of MemGuard. MemGuard consists of two main components - Bandwidth Regulator and Reclaim manager.

Bandwidth[12] regulator (labeled B/W Regulator in Figure 12) is responsible for monitoring system wide and per core memory bandwidth usage. It makes sure that system-wide memory usage does not exceed the DRAM bandwidth budget $B_i$ reserved for each core $C_i$. Budget $B_i$ is a system dependent parameter, which can be set by the user based on empirical data collected beforehand using a micro-benchmark discussed in [45]. Per-core bandwidth regulator is called at every scheduler tick (1ms). It adjusts the memory budget for each core and registers a performance counter (labeled PMC in Figure 12) based overflow interrupt handler called whenever a core $C_i$ has exhausted its assigned memory budget. Interrupt handler de-queues all the running tasks from run-queue of core $C_i$. The de-queued task can be rescheduled during the next tick when fresh

[12] Bandwidth refers to memory bandwidth, the rate at which data can be read from or stored into off-chip memory (DRAM).

memory budget is assigned to the core $C_i$. The overhead of programming PMC and de-queuing/en-queuing task is around 0.2% (2 microsecond (overhead) / 1 millisecond (scheduler period)).

Reclaim manager is responsible for dynamically adjusting the memory budget of each core based on a statistical method called Exponentially Weighted Moving Average. More discussion about the reclaim manager can be found in the paper [45].

## 3.2 Other Related Work

Below is a quick survey of literature on improving DNN inference efficiency by using techniques, like quantization, network pruning, and memory access acceleration. In this thesis, we explore the option of augmenting these techniques to improve DNN inference efficiency in the presence of memory intensive CPU applications.

- Quantization – a collection of techniques focused on using lower precision data formats, like FP16 for data storage and computation on embedded platforms without having a significant impact on DNN inference accuracy [15] [71] - [75].
- DNN network pruning – this technique is focused on removing redundant weights in the network resulting in reduced memory footprint [15] [76] - [79].
- Memory access acceleration – a collection of hardware optimization technique focused on the reduction of memory access time by bringing computation closer to the storage [41] [42] [56].

# Chapter 4. Methodology

This thesis uses a micro-benchmark (WMB) based on Winograd kernel, two DNN based object detection benchmarks: DetectNet [33] and YOLO [34], and a synthetic traffic generator (STG) for evaluation of the proposed GPU aware memory isolation mechanism.

## 4.1 Benchmark Models

The two object detection benchmarks are representative of different memory access patterns and resource requirements of real-world DNN application deployed on embedded platforms, which makes them good candidates for demonstrating the wide applicability of proposed mechanism.

On the one hand, DetectNet is a small CNN model based on GoogleNet [37] and trained using multiped-500 dataset [33] to detect only two object categories – pedestrian and luggage in an image. It is compatible with Nvidia's TensorRT inference acceleration engine (TensorRT) and supports 16-bit precision for both computation and storage of network parameters. On the other hand, YOLO is a more sophisticated CNN model based on a fully convolution network[13] (FCN [88]) and trained using the COCO object detection dataset [35] to detect more than 80 object categories. It is not compatible with TensorRT and supports only 32-bit precision for computation and storage of network parameters. Both DetectNet and YOLO supports Nvidia's cuDNN library and uses Winograd based kernel in the convolution layer.

Both DetectNet and YOLO benchmark model comes with demo application source code and pre-trained weights available online at [52] and [102] respectively. In case of DetectNet, the demo (detectnet-camera) application's source code was modified to take video stream from a file as input instead of the onboard camera installed on Jetson TX2. The modified version required installing OpenCV version 3.0. OpenCV was configured as per instruction on [53] and built natively on Jetson TX2[14]. In case of YOLO, demo application (Darknet) works with a video file by default.

[13] Fully convolutional network is a regular CNN with last fully connected layer replaced by another convolutional layer capturing the global context of the scene in the image. For instance, FCN can also provide a rough idea of the location of the objects in the image along with their labels.

[14] Note that Nvidia provides pre-built opencv4tegra, but it is based on OpenCV 2.4 which doesn't work with the gstreamer pipeline used in the modified detectnet-camera application.

Darknet is compiled with OpenCV, GPU and cuDNN options enabled. The cuDNN option is required to enable convolution using Winograd kernel instead of SGEMM kernel in the convolution layer. Darknet provides a command line option to set the detection threshold value. Threshold value allows the user to control which detected object is displayed based on their confidence score (or probability). An object detected with a confidence score c=0.25 will be displayed only if threshold value t <= 0.25. In this thesis, all the experiments were conducted with a threshold value of 0.5.

### 4.1.1 Test Data

The DetectNet benchmark uses a video[15] of passengers with their luggage at an airport terminal (as shown in Figure 13) as test data for object detection. And YOLO uses a video capture[16] of objects, like table, chair, utensil, etc. in a house (as shown in Figure 13) as test data for object detection.

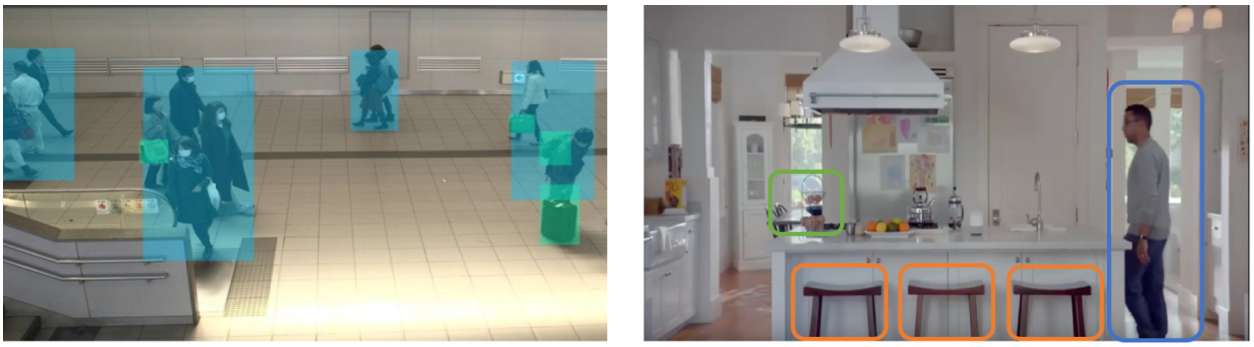

Figure 13. Reference input frame from test video file [96] [97]. DetectNet (Left) and YOLO (Right)

## 4.2 Micro-Benchmark

This thesis uses a synthetic micro-benchmark called WMB for detailed analysis of Winograd kernel available in Nvidia's cuDNN library. WMB is designed in such a way that it mimics the behavior of compute-intensive and time-consuming DNN convolution layer and provides

[15] The video resolution is 340x480 and consist of total 375 frames.

[16] The video resolution is 340x480 and has 112 frames.

performance isolation by avoiding any memory contention occurring due to child threads spawned during execution of DNN benchmarks (discussed in Section 4.1 ). Child threads often perform tasks, like input video decoding and video rendering using the HWA and therefore resulting in additional memory pressure over the DRAM bus. WMB enables us to study the impact of DRAM bandwidth contention on the performance of Winograd kernel.

The main loop of WMB program is shown in Algorithm 1.

**Algorithm 1** Winograd Micro-benchmark (WMB)

```
1: procedure WMB (input; output, kernel, size, count) /*The convolution of
   input and kernel */
2:     r ← 0
3:     while r < count do
4:         output ← WinogradConvolution(input, kernel, size)
5:         r ← r + 1
6:     end while
7:     return output  /* The convolved image is output */
8: end procedure
```

## 4.3 Traffic Generator

A synthetic traffic generator inspired by Bandwidth Bandit [17] [18] is created to quantitatively analyze the performance degradation that maybe caused by shared DRAM contention in the benchmarks discussed in Section 4.1 and 4.2.

### 4.3.1 Bandwidth Bandit

Bandwidth bandit [17] [18] is a profiling tool that works by executing the benchmark application whose performance analysis is required and a Bandit application which steals memory bandwidth from the benchmark application, simultaneously. The amount of bandwidth stolen can be varied (by changing the number of instances of traffic generator) to analyze the correlation between bandwidth and benchmark slow-down. The slowdown is measured in terms of IPC: average number of instructions executed in a clock cycle. A graph is plotted for IPC as a function of its available bandwidth. Figure 14 below shows the graph from the original paper [18], where the author [18] has demonstrated the performance degradation of application *milc* running on Intel Xeon E5520 core. The x-axis shows the Bandit bandwidth, the y-axis (left) shows milc's bandwidth and total bandwidth (milc + Bandit) and Y-axis (right) shows milc's IPC. When the Bandit does not steal any bandwidth, milc's bandwidth is around 2.7 GB/s and IPC around 0.7.

However, as the Bandit starts stealing the bandwidth, milc's bandwidth and IPC starts decreasing gradually. This means that the milc application is sensitive to the available memory bandwidth.

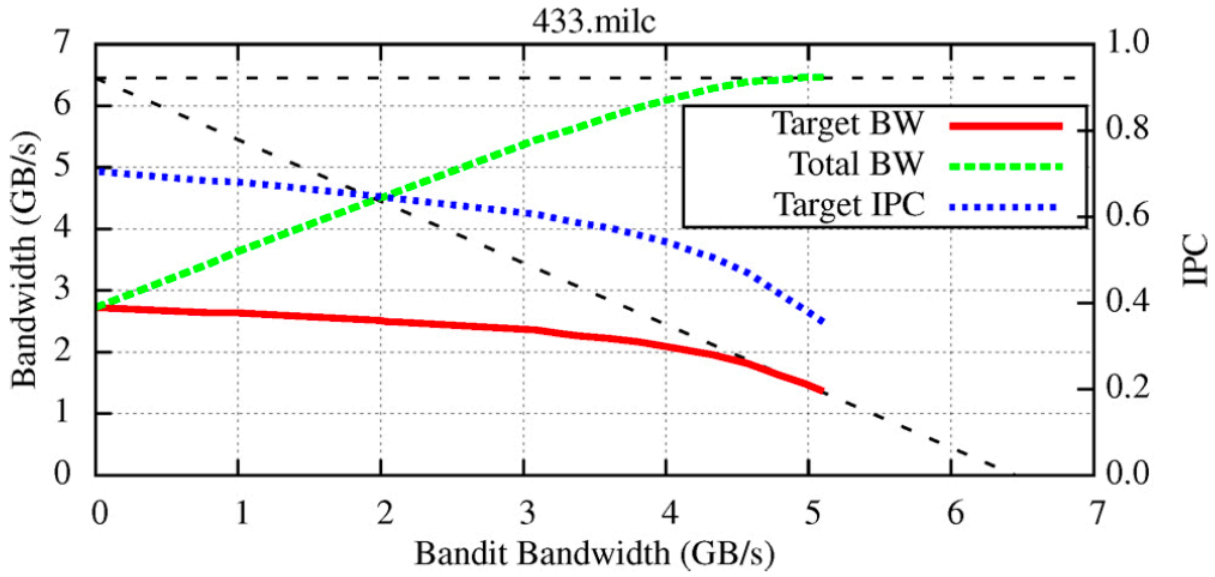


Figure 14. Bandwidth Bandit observation from original paper [18]

### 4.3.2 Traffic Generator Design

Eklov et al. demonstrated in [17] [18] that DRAM bandwidth consumed by a program is directly proportional to the number of LLC misses generated by the same program. In other words, DRAM contention can be created by having a program executing in a way that it triggers LLC misses. Traffic generator is designed on the same concept. As shown in Algorithm 2, it uses an integer array of size 16MB and iterates over the array in a loop. In each iteration, an index from the successive cache line is read/written causing an LLC miss. Since ARM v8 cores (Cortex A57 and Denver) in Jetson TX2 supports out-of-order processing, multiple outstanding memory requests can be generated in parallel. In fact, Denver 2 core in Jetson TX2 can handle up to 64 outstanding reads and 30 outstanding write requests at a time [68]. Traffic generator exploits this feature and results (in Figure 15) show that it can generate up to 11 GB/s bandwidth. The result is verified by counting the number of LLC misses caused by the traffic generator. LLC misses are counted using Linux perf tool compiled manually from source code on Jetson TX2 platform.

**Algorithm 2** Traffic Generator Main Loop

```
1:  #define CACHE_LINE_SIZE 64
2:  #define MEMORY_SIZE 16384 /*16MB*/
3:  int mem_size = MEMORY_SIZE * 1024;
4:  int *ptr = 0;
5:  int loop()
6:  {
7:          register int i;
8:          for( i = 0; i < mem_size/sizeof(int); i+=(CACHE_LINE_SIZE/sizeof(int)))
9:          {
10:                 ptr[i] = i;
11:         }
12:         data_written += mem_size;
13:         return 1;
14:}
```

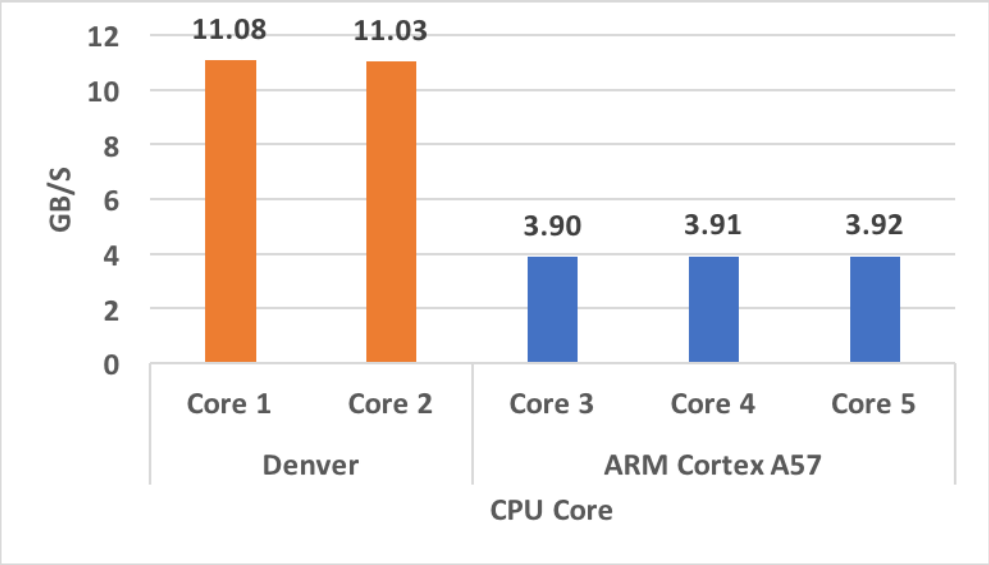


Figure 15. Memory bandwidth consumed by traffic generator when running in isolation on each core [1-5]

## 4.4 Metrics

This thesis uses following metrics for evaluation of proposed GPU aware memory isolation mechanism.

- Execution time – This metric represents the total execution time of the GPU kernel as reported by nvprof profiling tool.

$$RET = \frac{Execution\ Time\ of\ Reference\ Kernel}{Execution\ Time\ of\ Baseline\ Kernel} \quad (7)$$

- Frames per second (FPS) - This metric shows how well an object detection application is performing in a given scenario. FPS is calculated in the main loop of object detection pipeline by using below equation.

$$FPS = \frac{1}{Time\ elapsed\ in\ seconds} \quad (8)$$

- MegaPixels Per Second (MPS) - Like FPS, this metric is also used for evaluating the performance of object detection application. It shows the number of megapixels of the image that can be processed by object detection application in a second. MPS is calculated using equation (9).

$$MPS = \frac{Image\ size\ in\ Megapixels}{Time\ elapsed\ in\ seconds} \quad (9)$$

- Instruction Per Cycle (IPC) - IPC refers to the number of instruction executed by GPU in a clock cycle. It is a simple metric that helps evaluate the throughput of GPU achieved while executing a kernel. For a fixed clock frequency, a higher IPC means GPU is performing better. A drop in IPC value means GPU is under-performing due to some resource constraint.

- Average DRAM Throughput (ADT) - This metric is used to understand the throughput performance of GPU kernels. Since Pascal GPU on Jetson TX2 platform does not provide metric for LLC miss which is required for calculation, therefore, this thesis uses performance events (listed in Table 1) recorded using nvprof tool to calculate the throughput achieved by a GPU kernel. DRAM throughput is calculated using Eq. 10-12.

- Stall Reason Memory (SRM) - This metric shows the percentage of time GPU streaming multiprocessor (SMP) is stalled due to either memory throttling or due to the outstanding memory request. This metric is the sum of two GPU metric *stall_memory_throttle* and *stall_memory_dependency* collected using Nvidia profiler nvprof.

Table 1 GPU Performance Events

| GPU Event Name | Keyword | Description |
|---|---|---|
| l2_subp0_write_sector_misses | L2WM0 | Cache store miss |
| l2_subp1_write_sector_misses | L2WM1 | Cache store miss |
| l2_subp0_read_sector_misses | L2RM0 | Cache load miss |
| l2_subp1_read_sector_misses | L2RM1 | Cache load miss |
| elapsed_cycles_sm | CYCLE | Aggregate clock cycles of all streaming multiprocessor (SM) when a single warp was active |

$$Throughput = \frac{Data\ Transferred\ (in\ GB)}{Time\ elapsed} \tag{10}$$

$$Cache\ miss = L2WM0 + L2WM1 + L2RM0 + L2RM1 \tag{11}$$

$$Data\ Transferred = \frac{Cache\ miss * cache\ line\ size}{1024 * 1024 * 1024} \tag{12}$$

$$Time\ Elapsed\ (s) = \frac{\frac{CYCLE}{Number\ of\ SM}}{GPU\ Clock\ Frequency(in\ HZ)} \tag{13}$$

# Chapter 5. Preliminary Investigation

Although research on the topic of faster convolution is active, it has been focused on high-end systems used in desktops or datacenters. There isn't enough performance data available for faster convolution algorithms, like Winograd executed on embedded systems which is the target platform of this thesis. To fill this gap, we conducted a preliminary investigation of benchmarks and micro-benchmark with the aim to answer the following questions:

- What is the impact of bandwidth contention on object detection benchmarks?
- What is the impact of bandwidth contention on Winograd kernel based micro-benchmark?

The analysis data also serve as a base line for further evaluation of the proposed GPU aware bandwidth isolation mechanism.

## 5.1 Experiment Setup

The embedded platform used in the evaluation is Nvidia Jetson TX2 development board [38] [68]. The Jetson board is connected over LAN/WLAN to the Intel-based host machine running Ubuntu 16.04 which serves as a primary development machine. Jetson board is flashed with Nvidia Jetpack SDK 3.0 which includes L4T: a Linux based OS for Nvidia's Tegra processor-based boards and compatible driver packages. It also consists of all developer libraries required for AI and computer vision-based application development. In this thesis, following libraries were required to be installed – Nvidia Tensor RT, cuDNN, CUDA and Jetson Multimedia API.

Apart from other libraries, Nvidia visual profiler (nvvp) is installed on the host machine for remotely profiling the benchmarks and other test programs running on target Jetson platform. Nvvp provides insight into various low-level performance metrics which is collected from GPU performance counters.

### 5.1.1 Power Mode Setting

Jetson TX2 supports multiple preset performance modes, like Max Q, Max P, and Max N through a binary utility called “nvpmodel”. However, this utility was not used because it does not disable the frequency regulators and this leads to inconsistent results over a period of time.

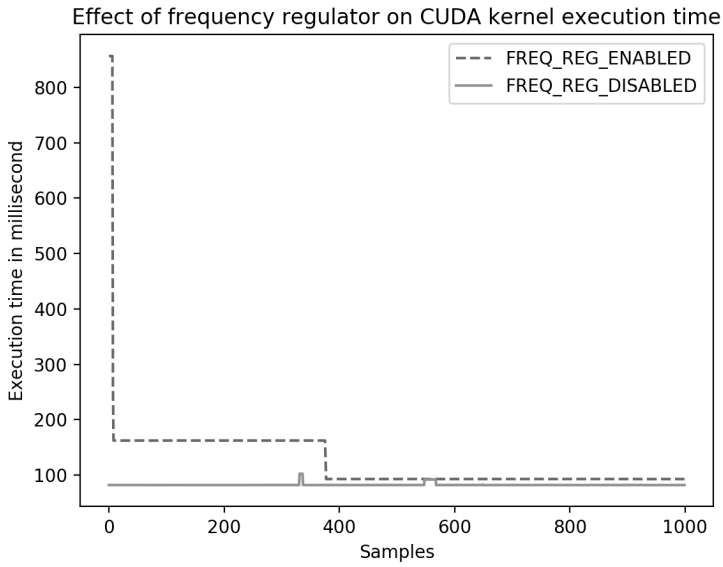


Figure 16. Effect of Frequency Regulator

Figure 16 shows the effect of frequency regulator on CUDA kernel (Winograd) execution time. When frequency regulator is enabled, the runtime initially ranges above ~800 ms for the first few iteration, then dropping to ~160 ms for next 400 iterations and finally dropping to ~90ms for remaining iterations. This sharp change is due to the frequency regulator adjusting the clock speed of CPU, GPU, and other power domains dynamically, based on task load. To avoid any inconsistency in the results, all the experiments were run in maximum performance mode by turning off the frequency regulators for CPU and GPU. The script used for setting performance mode is listed in Appendix A.

## 5.2 Traffic Generator Data

Traffic generator is used to characterize the memory sensitivity of the benchmarks and micro-benchmark discussed in Section 4.2 . To facilitate this characterization, we calculated the maximum bandwidth that can be consumed while executing multiple instances of the traffic generator, each instance running on a specific core. Since the benchmark and micro-benchmarks used in this thesis are always pinned[17] to CPU core0 of the Jetson TX2 platform, traffic generator instance is executed only on the remaining CPU core [1-5], and corresponding bandwidth data is

[17] Programs are pinned to a specific CPU core using the sched_setaffinity() Linux API [90]

calculated.

Figure 17 shows the bandwidth data generated. The x-axis shows the number of traffic generator instances running on Jetson TX2 while the y-axis shows the bandwidth consumed in GB/s.

- *corun1* shows the bandwidth consumed by one instance of the traffic generator running on core1
- *corun2* shows the total bandwidth consumed by two instances of the traffic generator running on core1 and core2
- *corun3* shows the total bandwidth consumed by three instances of the traffic generator running on core1, core2, and core3
- *corun4* shows the total bandwidth consumed by four instances of the traffic generator running on core1, core2, core3, and core4
- *corun5* shows the total bandwidth consumed by five instances of the traffic generator running on core1, core2, core3, core4, and core5

We can notice that corun1 consume maximum bandwidth of ~11 GB/s while corun2 consumes maximum bandwidth of ~13.62 GB/s. We can also notice that bandwidth consumption has plateaued around ~14.4 GB/s in case of corun3, corun4 and corun5 suggesting that the saturation bandwidth limit has been reached. Saturation bandwidth is the maximum bandwidth that can be consumed by running five instances of traffic generator.

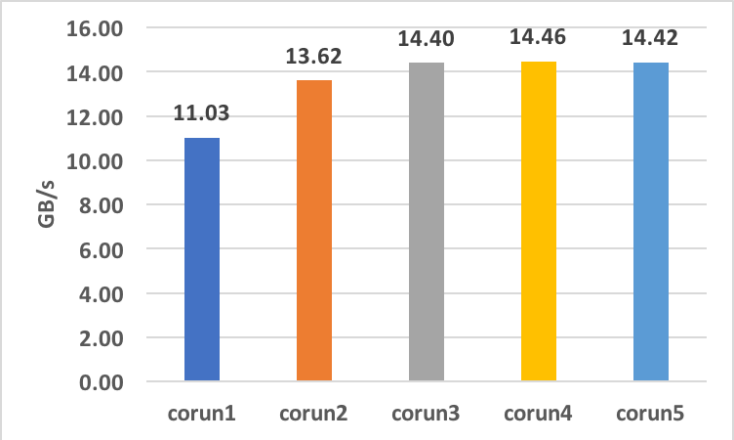


Figure 17 Aggregate memory bandwidth of traffic generator

## 5.3 Benchmark Analysis

Figure 18 and Figure 19 show the relative execution time (RET): the ratio of the execution time of the target kernel to the execution time of the baseline kernel (the least time-consuming CUDA

kernel), of top 4 time-consuming CUDA kernels invoked during execution of DetectNet and YOLO benchmarks. In case of DetectNet, *trtwell_scudnn_128x128_relu_interior_nn* is the baseline CUDA kernel, and *add_bias_kernel* is the baseline CUDA kernel in case of YOLO.

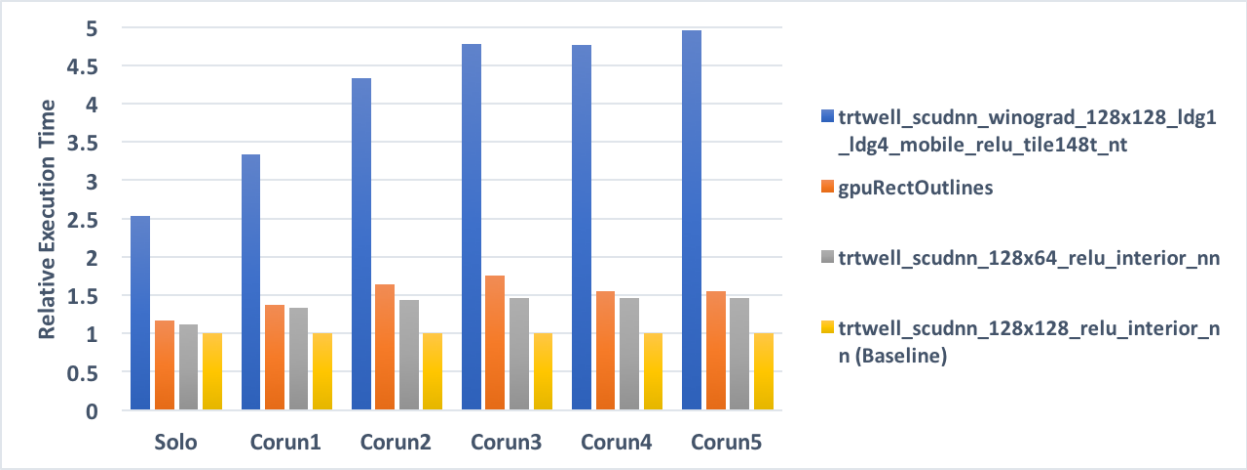


Figure 18. DetectNet: Relative execution time comparison of Top 4 CUDA kernels normalized to *trtwell_scudnn_128x128_relu_interior_nn* (baseline) kernel with different CPU co-runners

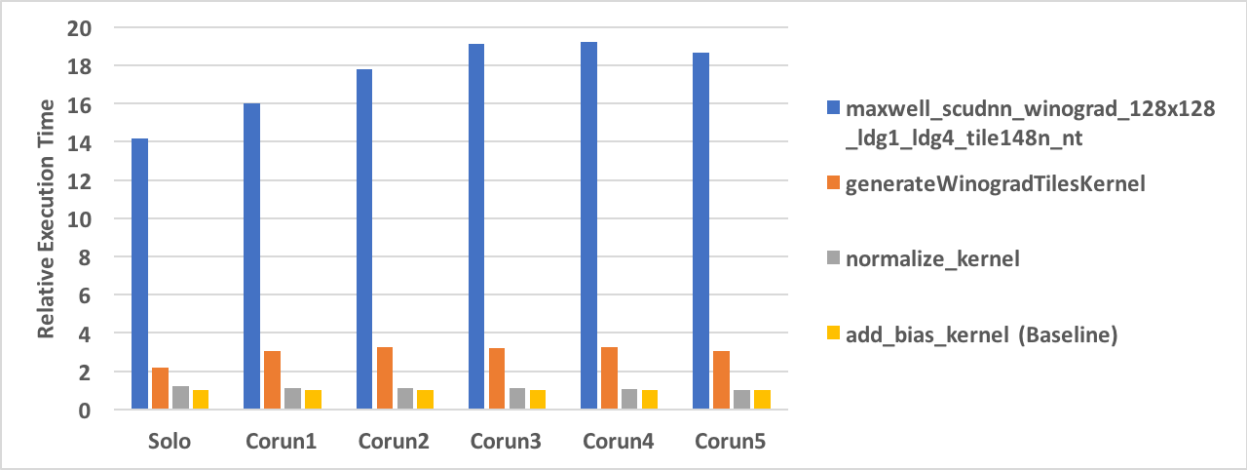


Figure 19. YOLO: Relative execution time comparison of Top 4 CUDA kernels normalized to *add_bias_kernel* (baseline) kernel with different CPU co-runners

The x-axis label *solo* shows the RET when the benchmark is executed in isolation on core0. *corun1* shows the RET when the benchmark is co-scheduled with a synthetic traffic generator (also referred as *co-runner* in this document), described in Section 4.3.2 , running on core1. *corun2* shows the RET when the benchmark is co-scheduled with two co-runners on core1 and core2. Similarly, *corun3, corun4* and *corun5* show the RET when the benchmark is co-scheduled with three, four and five co-runners, executing on core [1-3], core [1-4], core [1-5] respectively.

As shown in Figure 18 and Figure 19, Winograd based CUDA kernels[18]: *trtwell_scudnn_winograd_128x128_ldg1_ldg4_mobile_relu_tile148t_nt* (referred as K1 in rest of the document) in DetectNet and *maxwell_scudnn_winograd_128x128_ldg1_ldg4_tile148n_nt* (referred as K2 in rest of the document) in YOLO are the most time-consuming kernels in all the scenarios. K1 and K2 take up to 5x and 19x more time than the baseline kernel in DetectNet and YOLO benchmark respectively. It can also be noticed that RET of K1 and K2 along with other kernels varies with different CPU co-runners. For instance, RET of K1 increases from 2.5 in solo to ~5 in corun5. Similarly, RET of K2 increases from 14 in solo to ~18 in corun4. The variation in RET can be attributed to the degree of memory bandwidth contention caused by different co-runners running on core [1-5].

### 5.3.1 Performance Degradation

Figure 18 and Figure 19 demonstrated that RET changes as we increase the number of memory intensive co-runners causing memory bandwidth contention. However, it was not clear which kernel was impacted most by the increase in memory bandwidth contention. Therefore, we plotted another graph (Figure 20) to show the impact of memory intensive co-runners on the performance of different CUDA kernels. Relative performance is the ratio of the execution time of the target kernel when co-scheduled with co-runners (represented by label *corun [1-5]* on X-axis) to the execution time of the target kernel when running in isolation (represented by label *solo*). It is clear from Figure 20 that Winograd based CUDA kernels K1 and K2 are one of the most impacted kernels. In case of DetectNet benchmark, K1 performance degrades by almost 60 % (compared to baseline *solo*) when executed with five co-runners, while the next most impacted kernel slows down by only 40%. Similarly, in case of YOLO benchmark, K2 performance degrades by 45% approximately (compared to baseline *solo*) when co-scheduled with 5 CPU co-runners, next only

[18] Winograd based CUDA kernels are provided by Nvidia's cuDNN and TensorRT library.

to *generateWinogradTilesKernel* kernel whose performance degrades by 50% approximately.

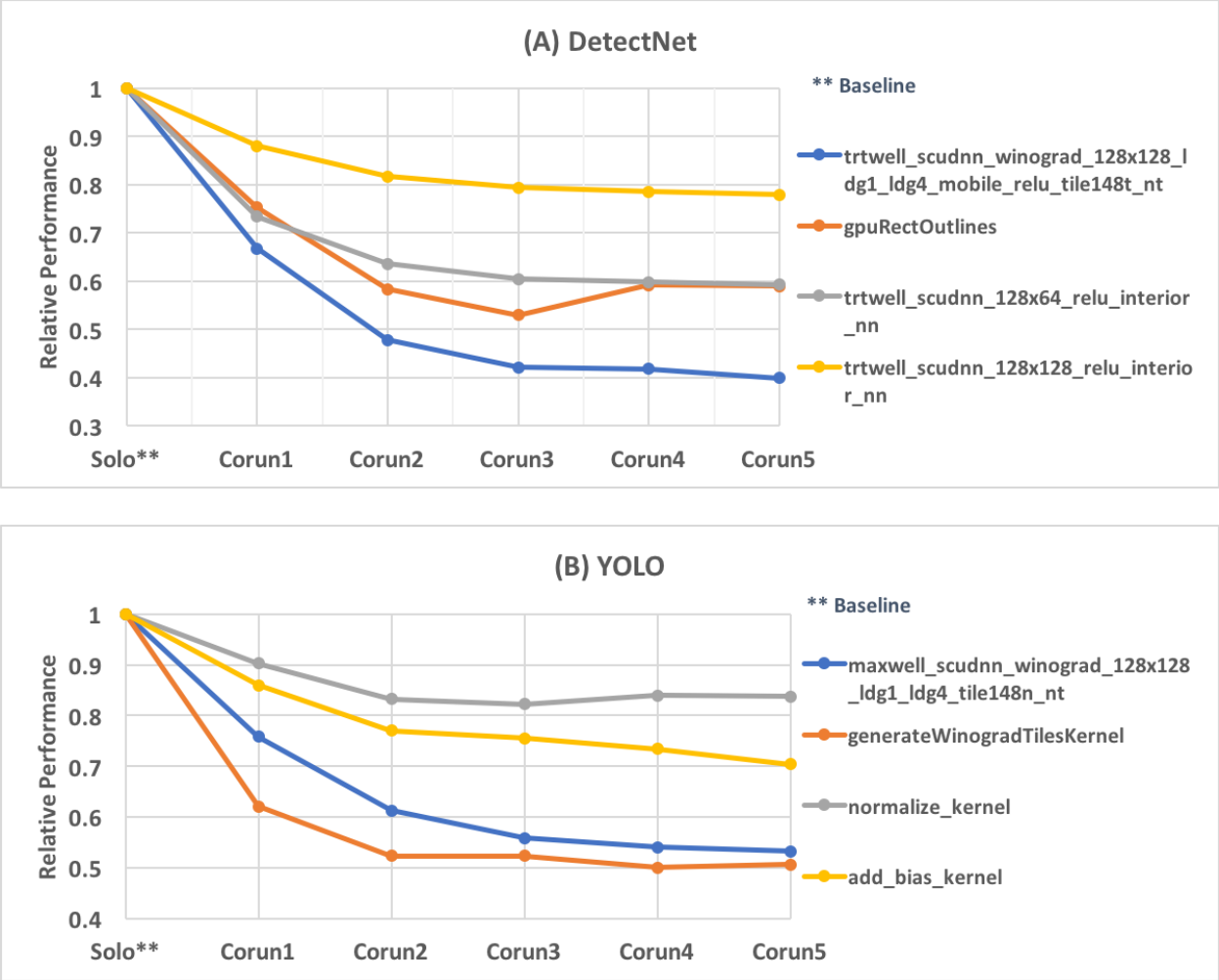


Figure 20. Performance degradation of CUDA kernels due to CPU co-runners (normalized to baseline solo)
DetectNet (A) and YOLO (B)

## 5.4 Micro-benchmark Analysis

During benchmark analysis, we found that Winograd based kernels are the most time-consuming and the most impacted due to memory intensive co-runners. To analyze the memory sensitivity of Winograd kernel, we ran a micro-benchmark (discussed in Section 4.2 ) with input images of resolution 256p, 512p, 1024p and 2048p.

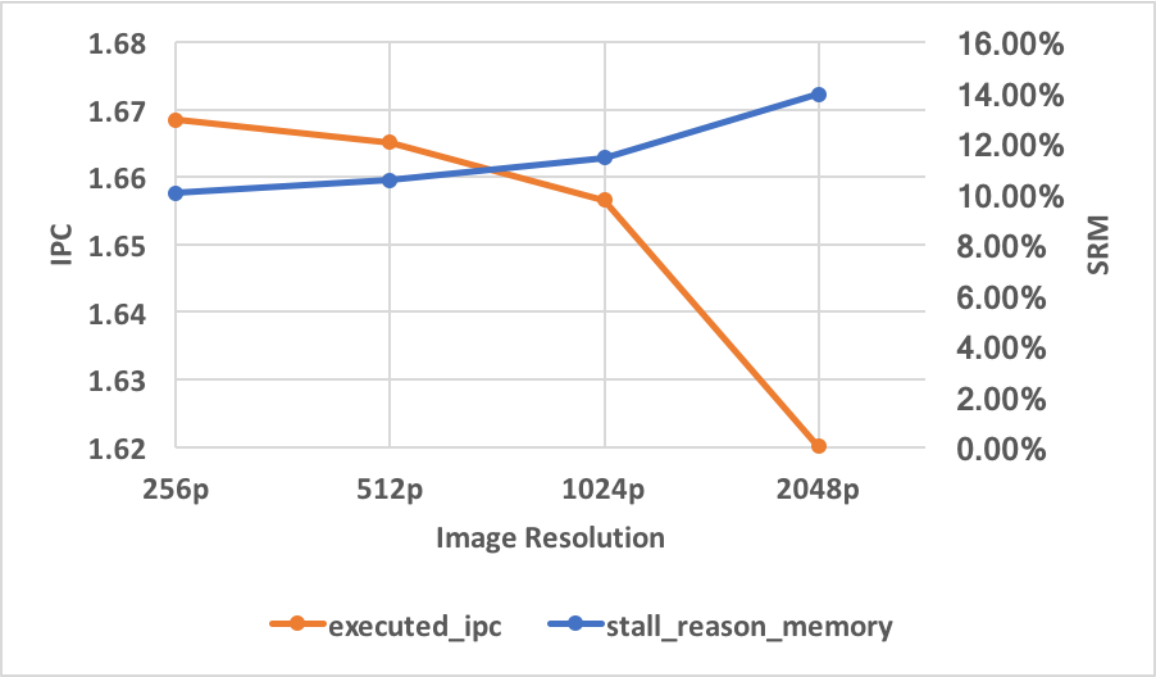


Figure 21. Correlation between IPC and SRM for four different image resolution - 256p, 512p, 1024p, 2048p (Average of 1000 iteration)

Figure 21 shows the performance (IPC) of Winograd kernel (based on 1000 iterations for each experiment) for input image sizes ranging between 256p – 2048p and how it relates to GPU SRM: stall reason memory (a GPU metric that shows the fraction of total instances when a GPU warp[19] was stalled due to memory throttling, or other memory dependencies). We can see that the IPC of the Winograd kernel degrades and SRM increases with increasing input size, indicating a negative correlation between IPC and SRM. In other words, the Winograd kernel is sensitive to memory bandwidth. To further validate our hypothesis, we ran micro-benchmark along with co-runners for each image resolution mentioned earlier.

Figure 22 shows the raw data collected using Bandwidth Bandit method. The graph shows Winograd kernel's IPC (the right y-axis), Throughput achieved in GB/s (the left y-axis) as a function of bandwidth (in GB/s) stolen by co-runners for four different image resolution 256p-2048p. Result for each image resolution is shown using the label in format IPC_#resolution and KT_#resolution. For example, IPC_256p and KT_256p shows Winograd kernel's IPC and throughput achieved in GB/s respectively for image resolution 256p.

[19] A GPU warp is the most basic unit of scheduling on Nvidia GPUs

When the co-runner does not consume any DRAM bandwidth, the throughput achieved by Winograd kernel ranges between 5 GB/s to 6.9 GB/s, and baseline IPC varies between 1.62 to 1.67 for different image size. However, when co-runner (single instance running on core1) steals ~11 GB/s bandwidth, both IPC and throughput achieved by Winograd kernel drops sharply to ~1.56 and ~4.8 GB/s respectively, in case of image size 512p. The IPC and throughput achieved by Winograd kernel continue to decrease as the memory bandwidth stolen by co-runner increases. Similar observation can be made for other image sizes 256p, 1024p and 2048p.

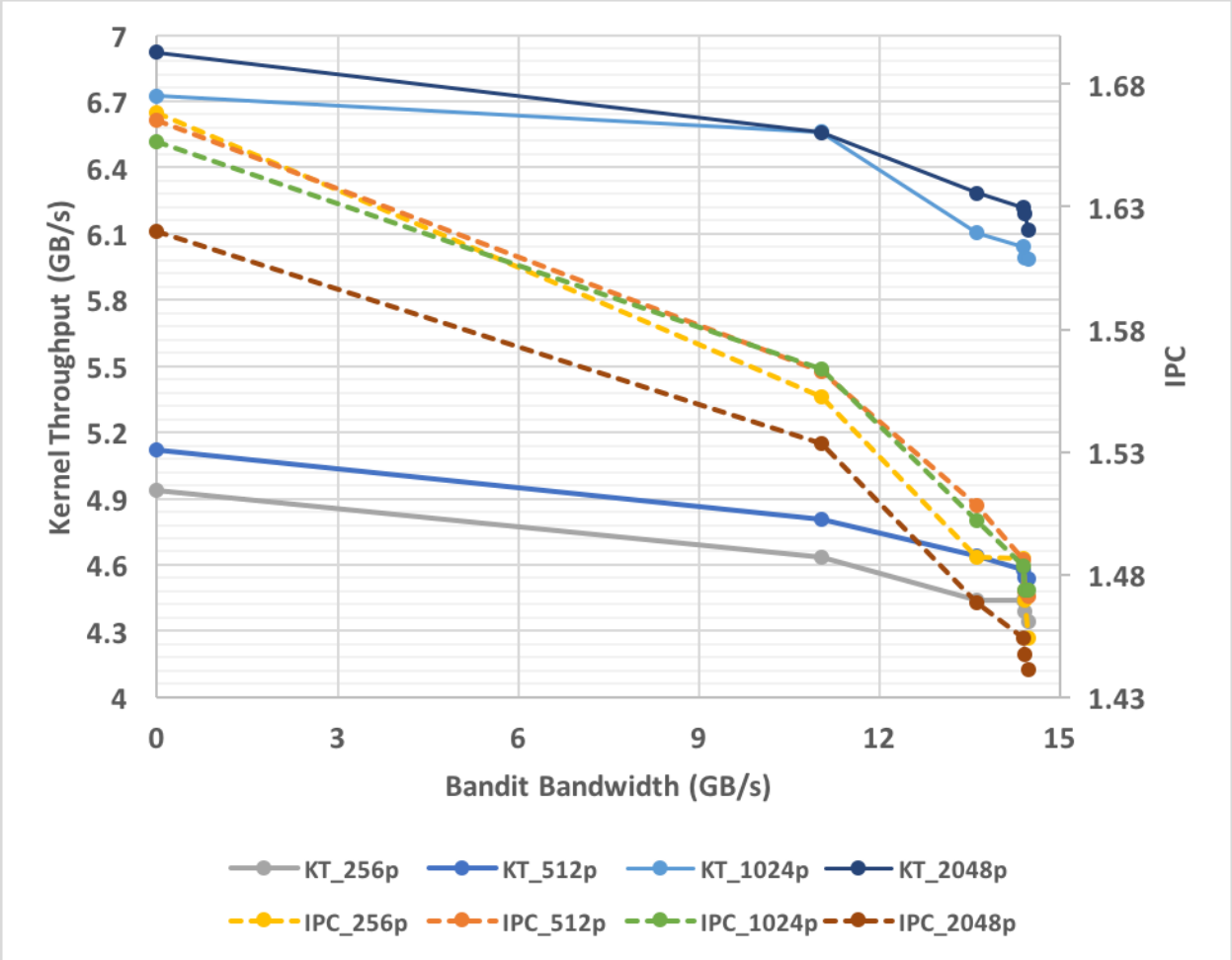


Figure 22. Kernel Throughput (Left Y-axis), IPC (Right – Y-axis) as a function of Bandit Bandwidth

Based on above data, we can conclude that Winograd kernel is sensitive to off-chip DRAM bandwidth. As the available DRAM bandwidth reduces due to DRAM contention caused by CPU co-runners (synthetic traffic generator discussed in SectionChapter 1. 4.3 ), the IPC and throughput achieved by Winograd kernel are reduced.

# Chapter 6. Design, Implementation and Verification of GPU Aware Memory Bandwidth Isolation Method

The experiment described in Chapter 4 demonstrated that Winograd kernel running on GPU is sensitive to off-chip DRAM bandwidth[20] contention caused by interference of co-running CPU applications even though they are running on different CPU cores. The main cause of bandwidth contention can be attributed to the lack of dedicated DRAM memory for GPU in Jetson TX2 platform. In Jetson TX2, there is only one DRAM which is shared among multiple masters, like CPU subsystem, GPU and other HWAs, like nvenc, nvdec, etc. All these masters contend for the DRAM bandwidth which prevents GPU from utilizing maximum DRAM bandwidth available on the platform, thus affecting the performance of DRAM bandwidth sensitive CUDA workload, in our case Winograd kernel, executing on GPU. Since Winograd kernel has the most impact on DetectNet and YOLO inference benchmarks, their overall performance is impacted due to the Winograd kernel slowdown.

So, the DNN inference slowdown can be reduced, or in other words, the DNN inference efficiency can be improved by reducing DRAM contention. One way to reduce DRAM contention is to isolate memory bandwidth available to the DNN inference task. The memory isolation technique used in this thesis is based on MemGuard [45] and BWLOCK [46] by Yun et al. MemGuard achieves bandwidth isolation by monitoring aggregated memory request made by each CPU core. Memory requested by each CPU core is counted by using HW performance counter "PERF_COUNT_HW_CACHE_MISSES" available through Linux perf_event [67] interface. BWLOCK extends MemGuard by providing a mechanism for regulating memory bandwidth from a userspace application. The detailed architecture of MemGuard and BWLOCK is discussed in Chapter 3.

## 6.1 Limitations of MemGuard and BWLOCK

Both MemGuard and BWLOCK suffer from few limitations. First, both mechanisms have been implemented and verified for a server grade Intel-based multicore platform with no integrated GPU. In other words, they have not been implemented keeping GPU workload in mind. Second,

[20] The rate at which data can be read from or stored into DRAM (off-chip memory)

BWLOCK suffers from system level slowdown caused by the userspace application level bandwidth locking approach which incurs performance overhead due to communication between userspace component and bandwidth regulator (MemGuard component). Third, BWLOCK requires instrumentation of user application code. This thesis proposes a GPU aware memory isolation mechanism to address the limitations.

## 6.2 GPU Aware Memory Isolation Mechanism

The proposed GPU aware memory isolation mechanism addresses the limitations discussed in Section 6.1 issues by extending MemGuard and BWLOCK to embedded SoCs with integrated GPUs, like Jetson TX2. Unlike BWLOCK, it uses OS kernel level locking mechanism to isolate bandwidth for GPU tasks. As a result, there is no performance overhead caused by userspace component, and all GPU tasks are automatically guaranteed memory bandwidth. Also, it doesn't require any modification in the user application code.

### 6.2.1 Design and Implementation

Figure 23 illustrates the architecture of proposed GPU aware memory isolation mechanism. The proposed mechanism is based on MemGuard. However, it does not use the Reclaim manager module. As shown in Figure 23, the proposed mechanism includes two main components:

1) Per-CPU[21] GPU aware memory bandwidth regulator (referred as MEMORY BW REGULATOR in Figure 23) kernel module and

2) Modified nvgpu driver module with memory locks

Both modules are discussed in detail in Section 6.2.1.1 and 6.2.1.2 . PMU shown in Figure 23 refers to performance monitoring unit, a hardware module present in ARM-based cores (A57 and Denver) responsible for monitoring hardware performance events, like cache miss events. PMU was referred as PMC in the original paper [45].

The execution flow of GPU accelerated inference application with GPU aware memory isolation mechanism shown in Figure 23 is summarized below.

[21] Per-CPU means one instance of program or function running on each core

When GPU accelerated inference application, like object detection is executed in userspace, it calls the optimized CUDA kernels, from the cuDNN and TensorRT libraries, corresponding to DNN layers, like convolution. The CUDA kernel, in turn, calls the CUDA runtime which interacts with the OS kernel space components, like NVGPU driver, CUDA driver and NVMAP driver for submitting the CUDA kernel to the GPU hardware. NVGPU driver informs per-cpu GPU aware memory bandwidth regulator to throttle CPU cores, not executing the GPU accelerated task, so that GPU encounters less memory contention. The GPU aware bandwidth regulator throttles the CPU cores, with the help of PMU and CPU scheduler driver, by limiting the memory bandwidth available to them.

#### 6.2.1.1 GPU Aware Memory Bandwidth Regulator

GPU aware memory bandwidth regulator is based on MemGuard's bandwidth regulator kernel module. However, the core algorithm has been simplified for GPU. Algorithm 3 lists the pseudocode of key methods used in the GPU aware bandwidth regulator module. Function *scheduler_tick_handler*() is called at every scheduler tick to check for following two conditions:

1) If bandwidth lock is acquired (using bw_lock API discussed in Section 6.2.1.2 ) by a GPU task (line 4)

2) If GPU is busy (using gpu_busy() API discussed in Section 6.2.1.2 )) in executing a task (line 12)

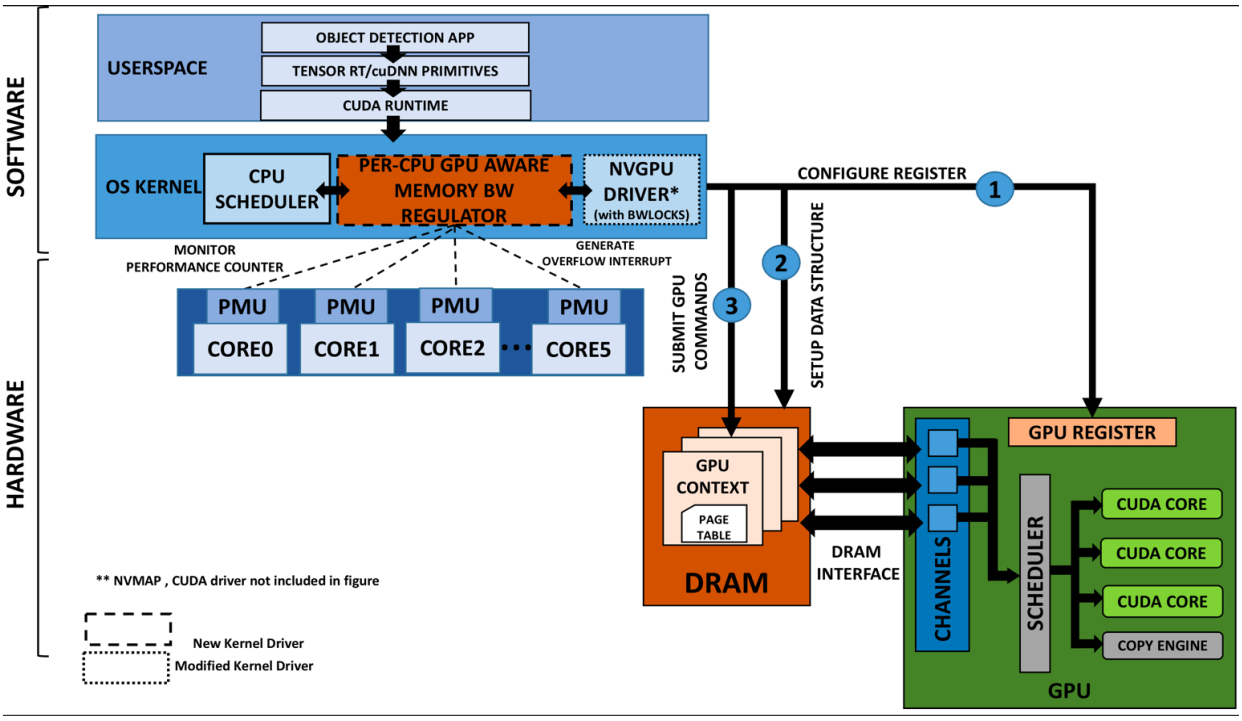


Figure 23. GPU Aware Memory Isolation Mechanism Architecture

If either of the above two conditions is true, then a memory bandwidth budget *bw_limit* is assigned to all the cores except the one assigned that is GPU task, which is assigned unlimited budget (line 7-15). Unlike MemGuard, where *bw_limit* is calculated using statistical method EWMA, the bandwidth budget *bw_limit* is static and is user configurable using a *sysfs node*[22] called param_bw_limit[23], exposed by bandwidth regulator kernel module.

*Memory_budget* (in MB/s) is converted to *cme_limi*t (line 17), the number of cache miss events (an estimate of memory bandwidth) that a CPU core can exhaust before it is being throttled. Memory bandwidth budget consumed by any CPU core is converted to the number of cache miss events by using equation 14.

[22] Sysfs node is a pseudo file system which exposes kernel module's configuration parameter to user space [98]

[23] param_bw_limit is available at /sys/kernel/debug/gpubwlock/ after loading bandwidth regulator kernel module

$$\text{cme\_limit} = \frac{\text{Bandwidth (in MB per second)}}{\text{Cache\_line\_size}} * \text{scheduler\_tick\_period} \quad (14)$$

Once *cme_limit* is calculated, an overflow interrupt handler *cme_interrupt_handler* is registered using *register_interrupt*_handler (Internally, it calls Linux kernel's perf_event interface API "*perf_event_create_kernel_counter*" [67]) (line 17). On registering the interrupt handler, the Linux perf_event interface starts monitoring a raw[24] performance counter *L1D_CACHE_REFILL* using PMU (Performance monitoring unit is shown in Figure 23). PMU generates a cache miss event overflow interrupt once *cme_limit* is reached. On receiving the overflow interrupt, the overflow interrupt handler *cme_interrupt_handler* is called where interrupting core is determined, and Linux kernel scheduler is called to de-queue all tasks from interrupted CPU core's running queue. The de-queued tasks are rescheduled in next scheduler tick (line 22-24).

**Algorithm 3** GPU Aware Memory Bandwidth Regulator

```
1:  function scheduler_tick_handler
2:  {
3:        task = current_task;
4:        lock_acquired = check_bandwidth_lock();
5:        if (lock_acquired) then
6:        {
7:              if (task->bw_locked)
8:                    memory_budget = unlimited_budget;
9:              else
10:                   memory_budget = bw_limit;
11:       }
12:       else if (gpu_status_busy())
13:             memory_budget = bw_limit;
14:       else
15:             memory_budget = unlimited_budget;
16:
17:       cme_limit = convert_budget_to_cme(memory_budget)
18:       register_interrupt_handler(cme_interrupt_handler,cme_limit);
19:}
20:function cme_interrupt_handler()
21:{
```

[24] RAW performance counters are micro-architecture specific counters that can be read using raw event IDs in the Linux perf_event interface.

```
22:       core = find_cpu_core();
23:       task_list = dequeue_all_cpu_task(core);
24:       schedule_task_for_next_scheduler_tick(task_list);
25:}
```

#### 6.2.1.2 Memory Locks in NVGPU driver

GPU aware memory bandwidth regulator module discussed in the previous Section 6.2.1.1 depends on memory locks, a simple API which tells memory bandwidth regulator module to trigger bandwidth throttling of CPU cores. Unlike BWLOCK, which reserves memory bandwidth for CPU tasks, the proposed implementation uses bandwidth regulator for isolating memory bandwidth for GPU accelerated tasks only. Therefore, it is crucial to introduce locks in the memory critical section of CUDA kernel (compute unit of GPU accelerated application) execution pipeline.

The memory locks can be introduced in the CUDA kernel execution pipeline either at userspace level or OS kernel level (in the GPU driver). The former approach would require inserting locks in userspace application code just before the CUDA kernel execution launch. However, this would incur additional communication overhead between userspace code and kernel space memory bandwidth regulator module. Also, each GPU application must be instrumented in order to take benefit of bandwidth regulator. GPU aware memory isolation mechanism avoids above issues by introducing locks in the memory critical section of the GPU driver.

Figure 23 illustrates three high-level GPU operations (numbered as 1,2,3 in the Figure) performed (at the driver level) during execution of CUDA kernel on GPU and Figure 24 lists the low-level GPU driver function calls associated with each GPU operation. The low-level driver execution flow can be summarized as below:

During the first operation, the GPU device is opened and configured using the *gk20a_ctrl_dev_ioctl* and *gk20a_ctrl_dev_open* methods. These methods are called as soon as GPU application is scheduled by the OS. During the second operation, GPU's virtual memory is initialized using the *gk20a_as_dev_ioctl* and *gk20a_init_vm* methods, and appropriate data structures, like GPU context, etc. are created in memory (DRAM). During the third operation, the *gk20a_channel_ioctl* method is used to create and manage GPU channels and use these channels to submit CUDA kernels launched from user-space application to the CUDA cores.

Based on above information, memory bandwidth regulator module can benefit from following two kinds of event scenarios to decide when to throttle CPU cores – First event indicating memory bandwidth regulator that GPU is busy (but not necessarily executing memory critical section) during the first operation discussed before. This would provide coarse-grained control over memory bandwidth. Second event indicating memory bandwidth regulator that GPU is executing memory critical section i.e., CUDA kernel execution in the third operation discussed before. This would provide more fine-grained control over memory bandwidth compared to the first event.

We created two sets of APIs *gpu_busy/gpu_not_busy* and *bw_lock()/bw_unlock()* for the two event scenarios discussed above to indicate bandwidth regulator module when to throttle bandwidth limit of CPU cores. First set of APIs, *gpu_busy / gpu_not_busy* is called during first operation while the second set of APIs, *bw_lock / bw_unlock*, is called during the third operation shown in Figure 24.

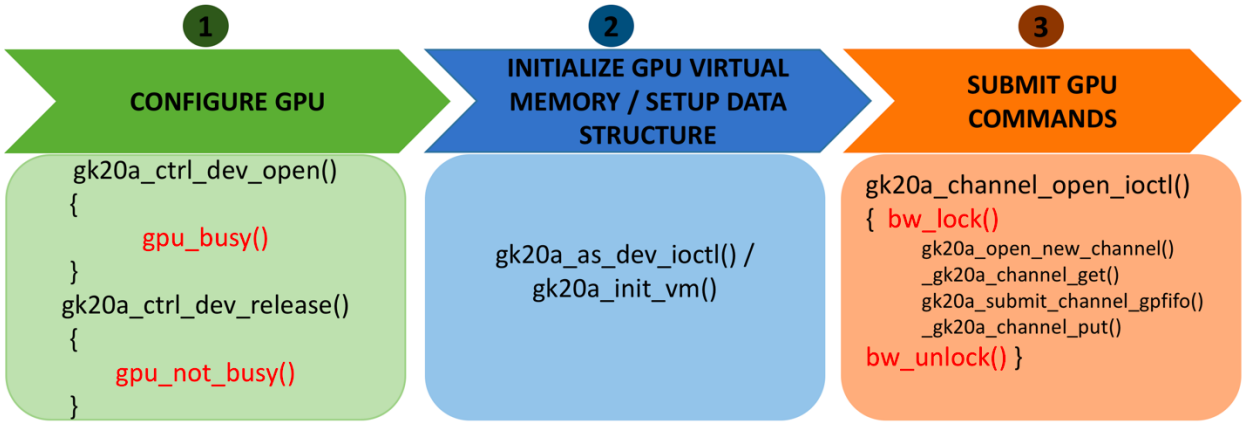


Figure 24 GPU driver function calls associated with GPU operation

### 6.2.1.3 Implementation Issue

While implementing GPU aware memory bandwidth regulator, it was found that hardware event used in MemGuard [45] is mapped to L1 cache instead of L2 cache, which is the last level cache in Jetson TX2, required for estimating DRAM bandwidth. Also, the alternative raw performance event *L2D_CACHE_REFILL*[25] is currently not supported by Linux perf interface. Therefore, we

[25] Performance event reflecting L2 cache miss

evaluated next available raw performance event *L1D_CACHE_REFILL*[26], supported by both ARM Cortex A57 and Nvidia Denver core.

We designed a simple experiment with the assumption that no hardware performance optimization techniques, like cache pre-fetcher[27] is available on Cortex A57 and Denver core. In the light of this assumption, traffic generator program, designed to generate DRAM traffic by causing L2 cache misses should cause the same number of L1 data cache misses because cache line size is same (64 bytes) for both L1 and L2 cache.

As part of the experiment, the number of L1D_CACHE_REFILL events were counted for Cortex A57 and Denver cores while executing the traffic generator program discussed in 4.3 in a given time period. The average bandwidth ($B_E$) of each core number was estimated from L1D_CACHE_REFILL count using equation 15. Estimated average bandwidth ($B_E$) was compared against the average memory bandwidth ($B_R$) reported by traffic generator. Table 2 reports data captured during the experiment. The first column shows the CPU core on which experiment was done. The second column shows the bandwidth reported ($B_R$) by traffic generator based on total data written in given time (10s in this case). The third column shows the number of L1D cache miss reported by Linux perf tool. The fourth column shows the average bandwidth estimated ($B_E$) based on L1D cache miss using equation 15. The fifth column shows the estimation accuracy calculated using equation 15.

$$\text{Bandwidth (MB/s)} = \frac{\frac{\text{L1DCacheMiss} * \text{L1 Cache Line Size}}{1024 * 1024}}{\text{TimeInSecond}} \quad (15)$$

where L1 cache line size, in this case, is 64 and TimeInSecond is 10. As we can see in Table 2, estimation accuracy for all A57 cores is almost 100%, while estimation accuracy in case of Denver core is ~87% and ~93%. L1D cache miss based bandwidth estimation method underestimates the actual bandwidth reported by traffic generator in case of Denver core. One of the plausible reasons for the bandwidth underestimation could be hardware performance optimization mechanisms, like

[26] Performance event reflecting L1 cache miss

[27] L2 cache prefetcher is a hardware technique for improving the performance of a system by prefetching cache lines

cache prefetching on the Denver Core which is causing fewer L1 cache misses. This contends the assumption made earlier that no such mechanism is available for the Denver Core.

Since bandwidth estimation was accurate up to 93% in Denver and 100% in ARM core, we decided to go ahead and use L1D cache miss event (mapped to L1D_CACHE_REFILL performance event) in our proposed GPU aware memory bandwidth regulator module.

Table 2. Comparison between bandwidth ($B_R$) reported by traffic generator and bandwidth estimated ($B_E$) by using L1D cache misses reported by Linux perf tool (All data accurate to single decimal place)

| CPU Core | Bandwidth ($B_R$) reported by traffic generator (MB/s) | L1D Cache Miss | Bandwidth estimated ($B_E$) using L1D cache miss (MB/s) | Estimation Accuracy |
|---|---|---|---|---|
| Core1 (Denver) | 11263.3 | 1617706056 | 9873.7 | 87.7% |
| Core2 (Denver) | 11331.5 | 1725391784 | 10530.9 | 93% |
| Core3 (A57) | 4000.7 | 657298199 | 4011.8 | 100.2% |
| Core4 (A57) | 4012.0 | 659625351 | 4026.0 | 100.3% |
| Core5 (A57) | 4000.7 | 656640215 | 4007.8 | 100% |

#### 6.2.1.4 Verification

For verification, we enabled GPU aware memory isolation mechanism on Jetson TX2 and ran GPU accelerated micro-benchmark (discussed in 4.2 ) within infinite loop on core0 and executed traffic generator (discussed in 4.3 ) in parallel, on the remaining core[1-5] one at a time. Figure 25 illustrates data reported during verification. The y-axis shows the data reported by the traffic generator in megabytes per second (MB/s). The x-axis shows the CPU core on which the traffic generator was executed. Label “NO_BW_LIMIT” reports data when GPU aware memory isolation mechanism was disabled and no micro-benchmark was running on core0. Label “BW_LIMIT_3000” and “BW_LIMIT_1000” reports data when GPU aware memory isolation mechanism was enabled, and bandwidth limit was set to 3000 MB/s and 1000 MB/s respectively.

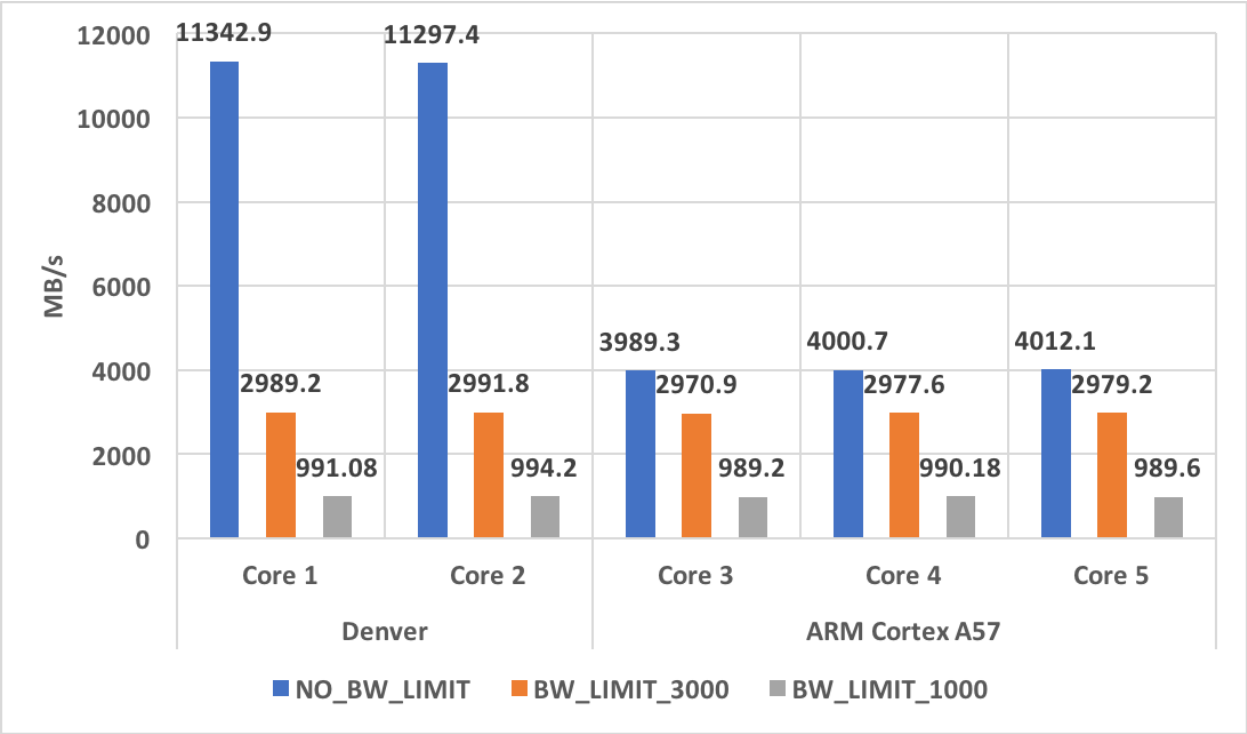


Figure 25 Memory bandwidth reported by the traffic generator

# Chapter 7. Evaluation and Results

This chapter investigates the performance impact of GPU aware bandwidth isolation mechanism on various benchmarks (discussed in Section 4.1 4.2 ).

## 7.1 Experiment Setup

The environment setup used here is same as the setup described in Section 5.1 . The only difference is that Linux kernel running on Jetson platform is patched with modified NVGPU driver. The steps used for patching Linux kernel source, cross-compiling patched Linux kernel on the host machine and flashing the patched kernel on Jetson TX2 can be found at [81]. Once the modified kernel is up and running on Jetson TX2, GPU aware bandwidth regulator kernel module is loaded from userspace using Linux insmod[28] command. Once loaded, the kernel module exposes the kernel module configuration parameter param_bw_limit via a sysfs node to the userspace.

## 7.2 Effect of Memory (DRAM) Bandwidth Limit

The objective of this experiment is to quantitatively analyze the impact of throttling CPU cores (not running GPU tasks) by varying the memory bandwidth budget (also referred as budget) assigned to each core on the performance of proposed GPU aware memory bandwidth isolation mechanism. The performance will be measured in terms of improvement in benchmark performance after bandwidth isolation mechanism is applied.

Figure 26 reports the performance improvement experienced by micro-benchmark (discussed in Section 4.2 ) when GPU aware memory isolation mechanism is enabled. Data are reported for following test-cases –

1) Solo – micro-benchmark is run in isolation
2) Corun [1-5]) – micro-benchmark is run with 1-5 CPU co-runners (traffic generator discussed in Section 4.3 )

[28] insmod is a Linux utility for loading kernel modules from userspace

Results for image resolution 256p, 512p is shown in graph A, while results for image resolution 1024p and 2048p is shown in graph B. The result of each test case and image resolution is labeled using "#test-case_#image_resolution" format. For instance, Solo_256p reports result for Solo test case and image resolution 256p. The y-axis shows *IPC speedup*, change in IPC when micro-benchmark is run with memory isolation mechanism enabled with respect to IPC when micro-benchmark is run with memory isolation mechanism disabled, of the Winograd kernel. The x-axis shows memory bandwidth limit assigned to CPU cores running co-runners (i.e., core[1-5]) in MB/s. Label *inf* on the extreme right of the x-axis represents baseline performance when the IPC is measured with memory isolation mechanism disabled; hence it is always 0 (no speedup). Label *10* shows the IPC speedup when memory bandwidth limit is set to 10 MB/s. Similarly, labels 100, 512, 1000, 10000, 28000 and 56000 shows the IPC speedup when bandwidth limit is set to 100, 512, 1000, 10000, 28000, 56000 MB/s respectively.

Following observations can be made based on data reported in Figure 26. First, the bandwidth isolation mechanism performs best when bandwidth limit is set to 100MB/s. The performance improvement can be seen ranging up to 13.5% in case of corun5 for image size 256p and bandwidth limit set to 100MB/s. Similar performance improvement can be seen when bandwidth limit is set to 10 MB/s in all test cases except for corun5 when improvement is ~7%, 5%, 9% and 3%for image size 256p-2048p respectively which is less compared to performance improvement noticed when bandwidth limit is set to 100MB/s. As the bandwidth limit is increased above 100MB/s, the performance improvement seen earlier starts diminishing. This performance degradation is due to increase in memory contention caused by increasing the bandwidth limit of the CPU cores executing CPU co-runners. For instance, when the bandwidth limit is set to 100MB/s, the approximate aggregate memory bandwidth generated by in test-case corun5 (running five co-runners) is 500 MB/s. However, when the bandwidth limit is increased from 100 MB/s to 512 MB/s and 1000MB/s, the aggregate bandwidth stolen in case of corun5 increases to 2560MB/s (2.5GB/s) and 5000 MB/s (5 GB/s) respectively, causing significant memory contention, therefore, reducing the performance of bandwidth isolation mechanism when compared to bandwidth limit of 100MB/s. As we increase the memory bandwidth limit from 1000 MB/s to 10000MB/s, the performance improvement due to bandwidth isolation mechanism is almost reduced to zero. This is because the aggregated bandwidth stolen in case of corun5 reaches to the maximum possible value of ~14.45GB/s (based on data reported in Figure 15).

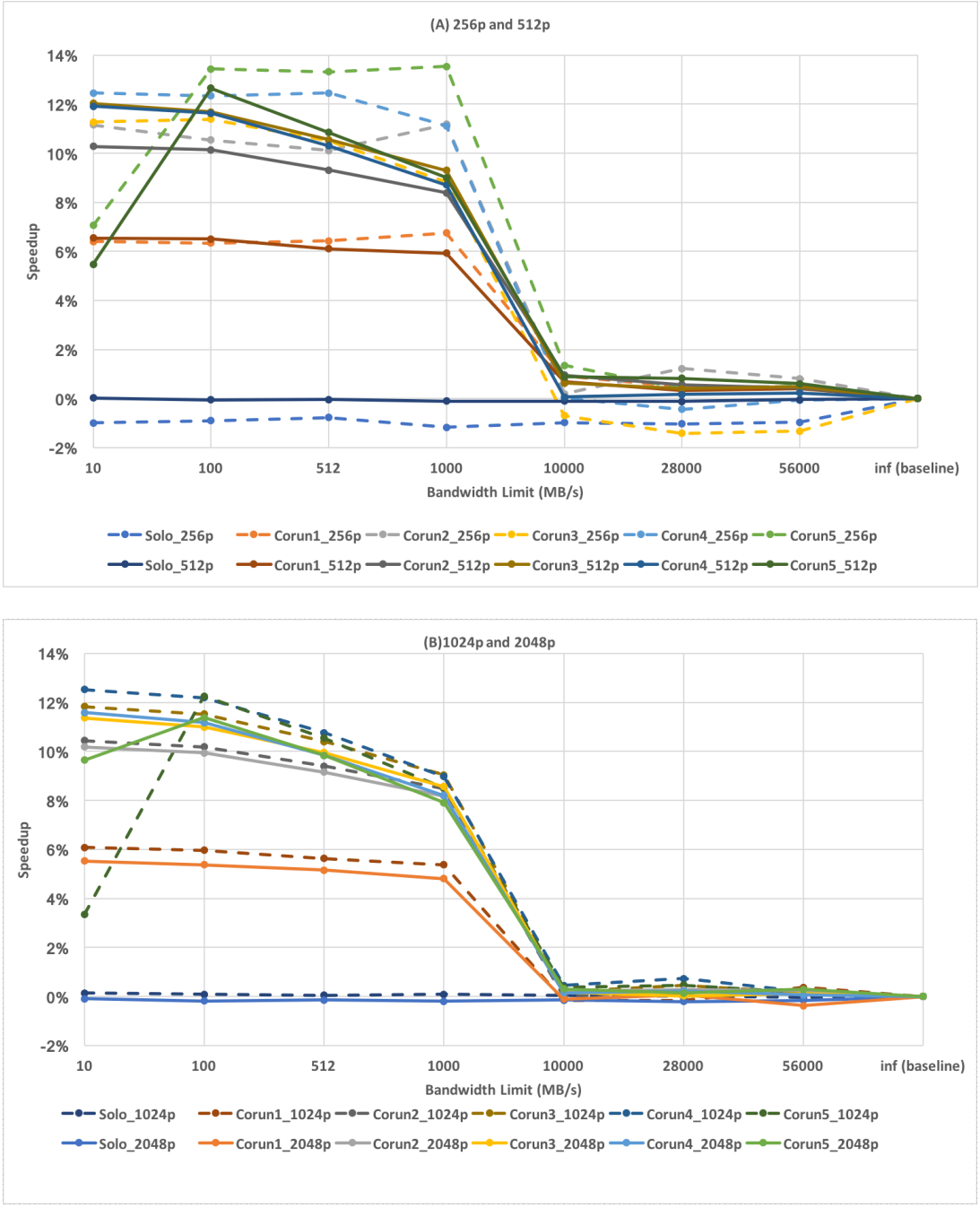


Figure 26. Effect of varying memory bandwidth limit on performance (Speedup) of Winograd kernel based micro-benchmark.
The performance is normalized to IPC measured while bandwidth isolation mechanism is disabled. Top graph (A) shows the results for input image resolution 256p and 512p. Bottom graph (B) shows the results for input image resolution 1024p and 2048p.

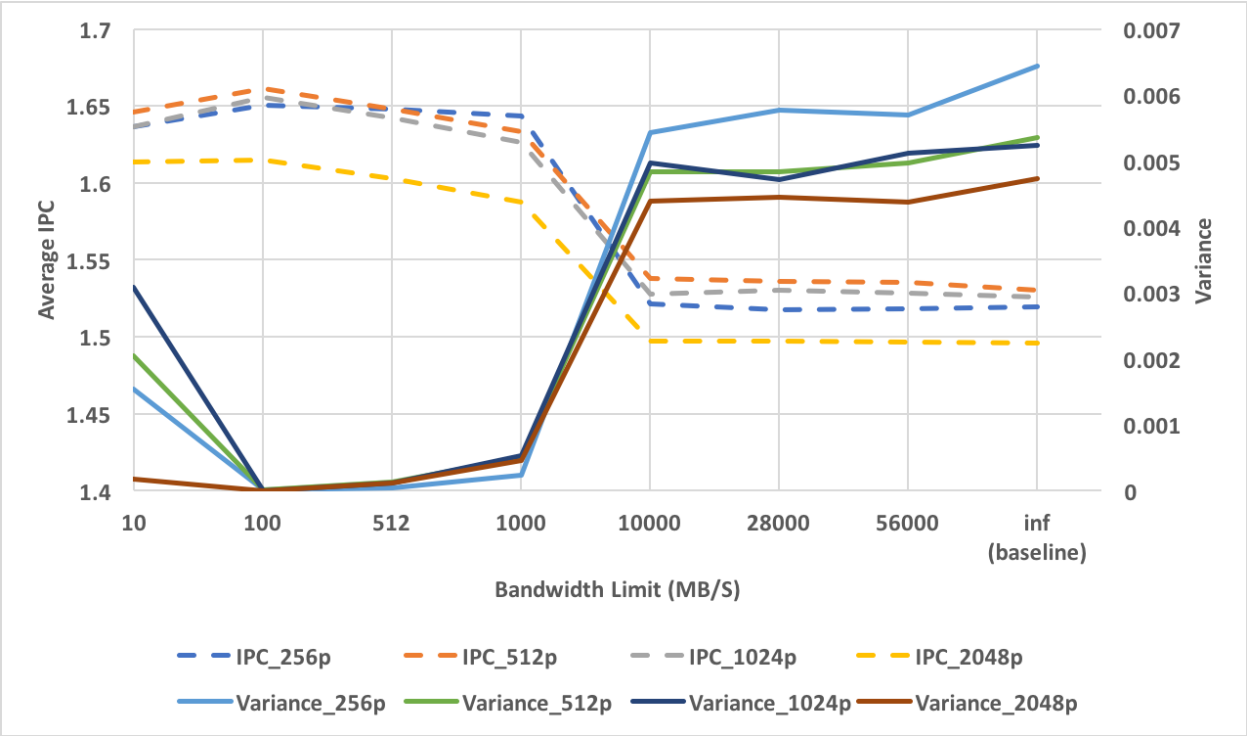


Figure 27. Average IPC (left Y-axis) and Variance (right Y-axis) as a function of memory bandwidth limit

To further validate our observation, that bandwidth isolation mechanism performs best when bandwidth limit is set to 100MB/S, we analyzed the average IPC (mean of IPC achieved during solo, corun1, corun2, corun3, corun4 and corun5), and variance of IPC (same as statistical variance) for each bandwidth limit. Low variance signifies that there is low variation in performance when micro-benchmark is run either in isolation or it is co-scheduled with other co-runners (corun [1-5]). Similarly, high variance means there is higher variation in the performance when micro-benchmark is run in isolation, or it is co-scheduled with other co-runners (corun [1-5]). Average IPC signifies overall performance of micro-benchmark in various scenarios.

Figure 27 illustrates the average IPC (shown on the left y-axis) and variance (shown on the right Y-axis) for image resolutions 256p-2048p. The X-axis labels are same as Figure 26. IPC and variance for different image resolutions are labeled using format “IPC_#resolution” and “Variance_#resolution”. For instance, IPC_256p shows average IPC for image resolution 256p. Similarly, Variance_256p shows variance for image resolution 256p. We can notice in Figure 27 that average IPC is maximum when bandwidth limit is set to 100MB/S for all image sizes 256p-2048p. We can also notice that variance is always very small; therefore changes in variance with

respect to bandwidth limit are insignificant. Based on average IPC and IPC speedup, we can conclude that 100MB/s is the optimal bandwidth limit for the micro-benchmark.

Like Figure 26, Figure 28 reports the impact of varying bandwidth limit on the performance of bandwidth isolation mechanism in case of DetectNet and YOLO benchmarks. The y-axis shows the performance (MPS: megapixels of image processed by benchmark in a second, discussed in more detail in 4.4 ) speedup, the change in performance (MPS) when bandwidth isolation mechanism is enabled with respect to performance when bandwidth isolation mechanism is disabled. The x-axis labels are same as Figure 26 which shows the bandwidth limit applied to CPU cores executing a non-GPU task. Label *inf* on the extreme right of the x-axis shows the baseline performance when MPS is measured with the memory isolation mechanism is disabled; hence it is always 0 (no speedup).

The following observation can be made based on data reported in Figure 28. First, the bandwidth isolation mechanism performs best when bandwidth limit is set to 100MB/s in case of DetectNet and 512MB/s in case of YOLO. The performance is improved by approximately 45% in case of YOLO. While in case of DetectNet, performance is improved by 86% in case of Corun5.

The second observation that can be made from Figure 28 is that the bandwidth isolation mechanism is effective when memory bandwidth limit is in the range 512-1000MB/s for YOLO and 100-1000MB/s for DetectNet.

A third observation we can make from Figure 28 is that the performance of both DetectNet and Yolo benchmark reduces by approximately 70% and 90% respectively when memory bandwidth limit is set to 10MB/s. This significant performance degradation can be explained by the fact that a portion of the DNN inference pipeline in the benchmark application, like fetching and rendering video frames, still runs on the CPU. When the bandwidth limit is set 10MB/s, the aggregate bandwidth available to all CPU cores combined is 60MB/s. Out of this, CPU co-runners executing on core [1-5] consumes 50MB/s. The remaining 10MB/s is the actual bandwidth available to CPU portion of benchmark applications, like fetching, rendering, etc. which may be too low, hence causing the slowdown of the CPU portion of the benchmark.

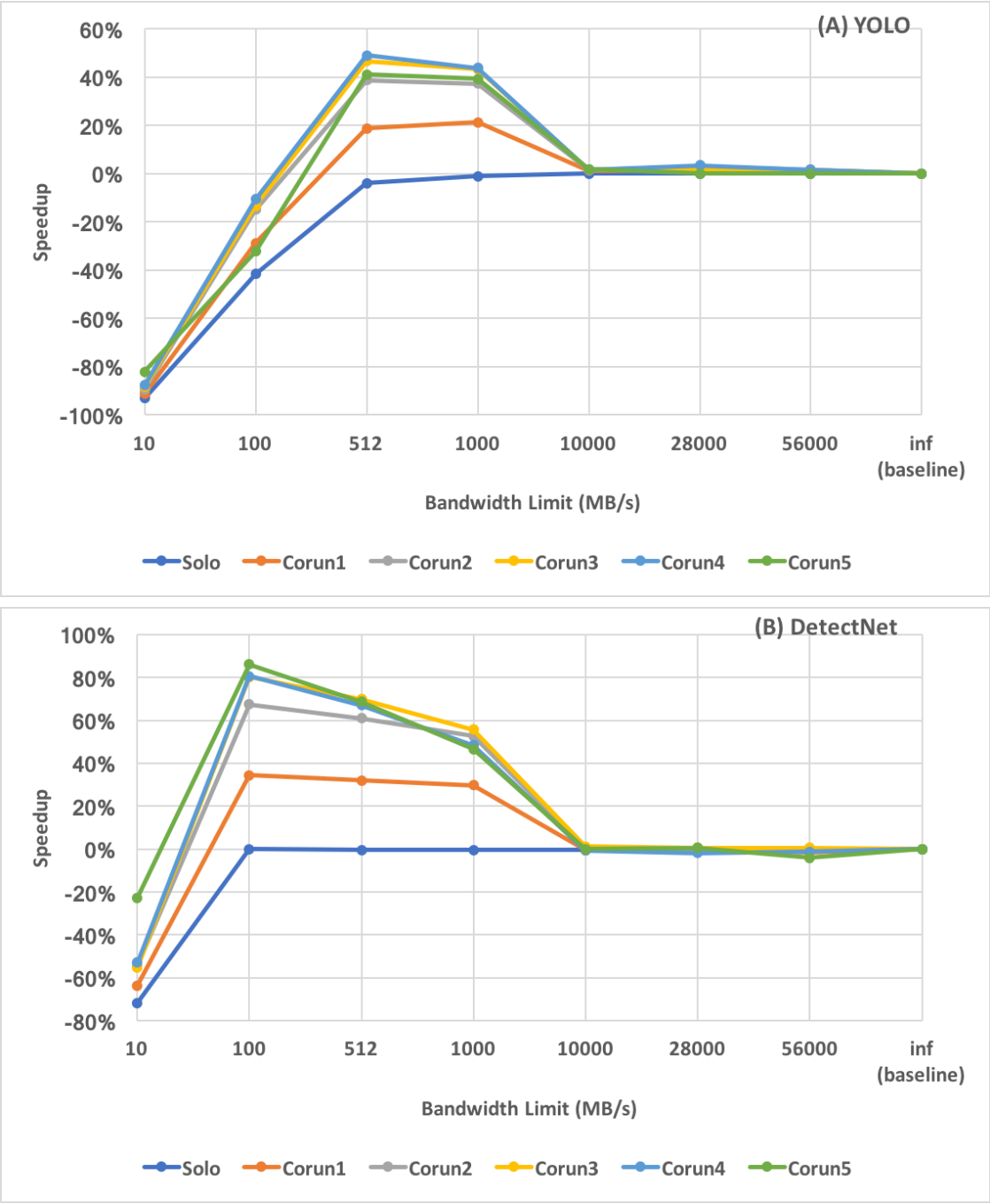


Figure 28. Effect of varying memory bandwidth limit on the performance (speedup) of YOLO (A) and DetectNet (B) benchmark.
The performance is normalized to MPS measured while bandwidth isolation mechanism is disabled.

Since, metric MPS reflects the performance of complete DNN inference pipeline, which includes both GPU accelerated detection task and non-GPU accelerated tasks running on CPU, the slowdown in CPU portion of DNN inference pipeline reduces the overall performance of

benchmark application. To further investigate the slowdown, we analyzed and compared the execution time of individual GPU accelerated kernels with the baseline kernel execution time and found that there is no significant slowdown. Thus, we can say that slowdown is not caused by the GPU accelerated portion of the benchmark application. However, more investigation would be required to validate our hypothesis that performance slowdown of benchmark application is caused by CPU portion of the DNN inference pipeline.

Like micro-benchmark, we analyzed the average performance (MPS) and variance to validate our observation that optimal bandwidth limit for bandwidth isolation mechanism in case of YOLO and DetectNet benchmark is 512MB/S and 100MB/S respectively. Based on data reported in Figure 29, we can make following observation.

First, in case of YOLO, changing bandwidth limit does not affect the variance much. However, the average MPS varies significantly with a maximum of ~0.9 MPS when bandwidth limit is set to 512MB/s. In case of DetectNet, the variance is minimum, and average MPS is maximum when bandwidth limit is set to 100MB/s.

The second observation we can make from Figure 29 is that variance in case of DetectNet is much higher, almost factor of 10 when compared to YOLO. Some of the larger variances can be attributed to the fact that MPS of DetectNet is ~3x compared to YOLO, so one could reasonably expect the variance to be 3x higher. Another possible reason of high variance could be Nvidia's TensorRT inference optimization engine used in DetectNet, which essentially performs graph optimization to make efficient use of available resources, like GPU compute resource and system's memory bandwidth, etc. It is possible that this graph optimization is introducing high variance in the inference task.

Based on average performance (MPS), variance and slowdown we can conclude that optimal bandwidth limit in case of Detectnet and YOLO is 100MB/s and 512MB/s respectively.

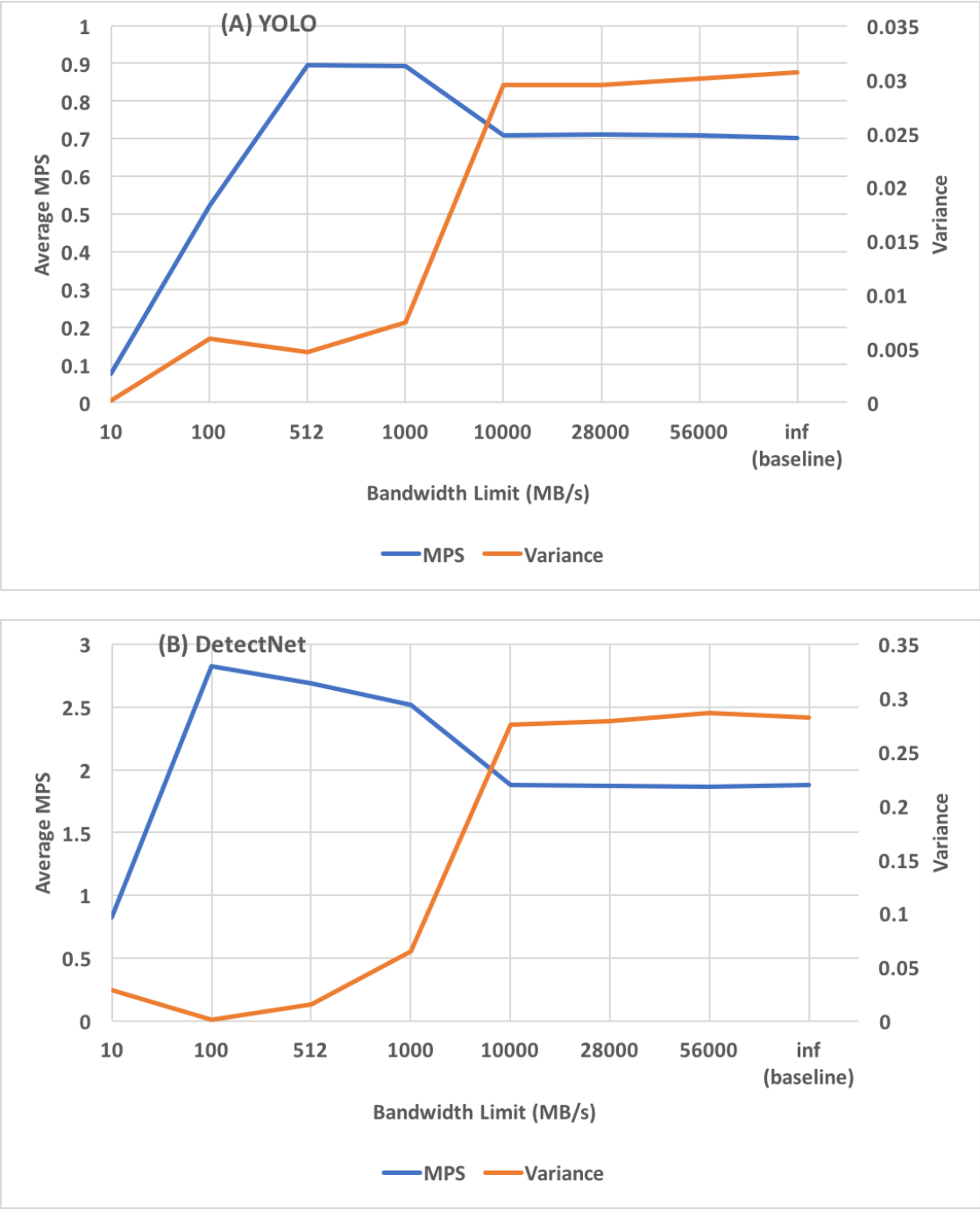


Figure 29. Average MPS (left Y-axis) and Variance (right Y-axis) as a function of bandwidth limit

## 7.3 Timing Analysis of GPU Aware Memory Isolation Mechanism using Micro-Benchmark

As DNN inference performance on embedded system is improving, there has been growing interest in applying it to real-time systems. However, DNN inference is often characterized by unpredictable performance, making it unsuitable to the performance guarantees (in terms of reliability and predictability) of real-time systems. Performance of real-time system is evaluated using timing analysis [99] [100] [101] in terms of WCET; the longest execution time observed when a program is run on target hardware, average execution time (same as statistical mean), variance and range (same as statistical variance and range). WCET helps in determining the upper bound of tasks' execution time [99] on a real-time system. Similarly, variance and range are found useful in understanding the reliability by determining the variance in the system [101].

In this experiment, we measure the impact of GPU aware memory isolation mechanism using timing analysis of micro-benchmark (discussed in Section 4.2 ) when running in isolation. We collect data for 100,000 iterations for two cases – 1) when memory isolation mechanism is disabled and 2) when memory isolation mechanism is enabled with memory bandwidth limit (of CPU cores not running GPU task) is set to optimal value, i.e., 100MB/s.

Figure 30 illustrates the result of this experiment. The x-axis shows sample number, and the y-axis shows execution time in microseconds recorded for each sample. We can notice that the average execution time (shown in the top right of the Figure) of Winograd kernel (used in micro-benchmark) does not change much when memory isolation mechanism is enabled. However, we can notice the reduction in spikes which shows the WCET of Winograd kernel. WCET of Winograd kernel when memory isolation mechanism is disabled (referred as $WCET_D$ in Figure 30) is around ~1775 microsecond whereas WCET when memory isolation mechanism is enabled (referred as $WCET_E$ in Figure 30) reduces by ~20% to ~1410 microsecond. Not only this, the variance and range (of execution time) in the case when memory isolation mechanism is disabled is around 7x and 4x respectively when compared to the case when memory isolation mechanism is disabled.

Above data indicates that GPU aware memory isolation mechanism can improve the predictability of real-time systems running GPU accelerated inference tasks by reducing WCET and variance.

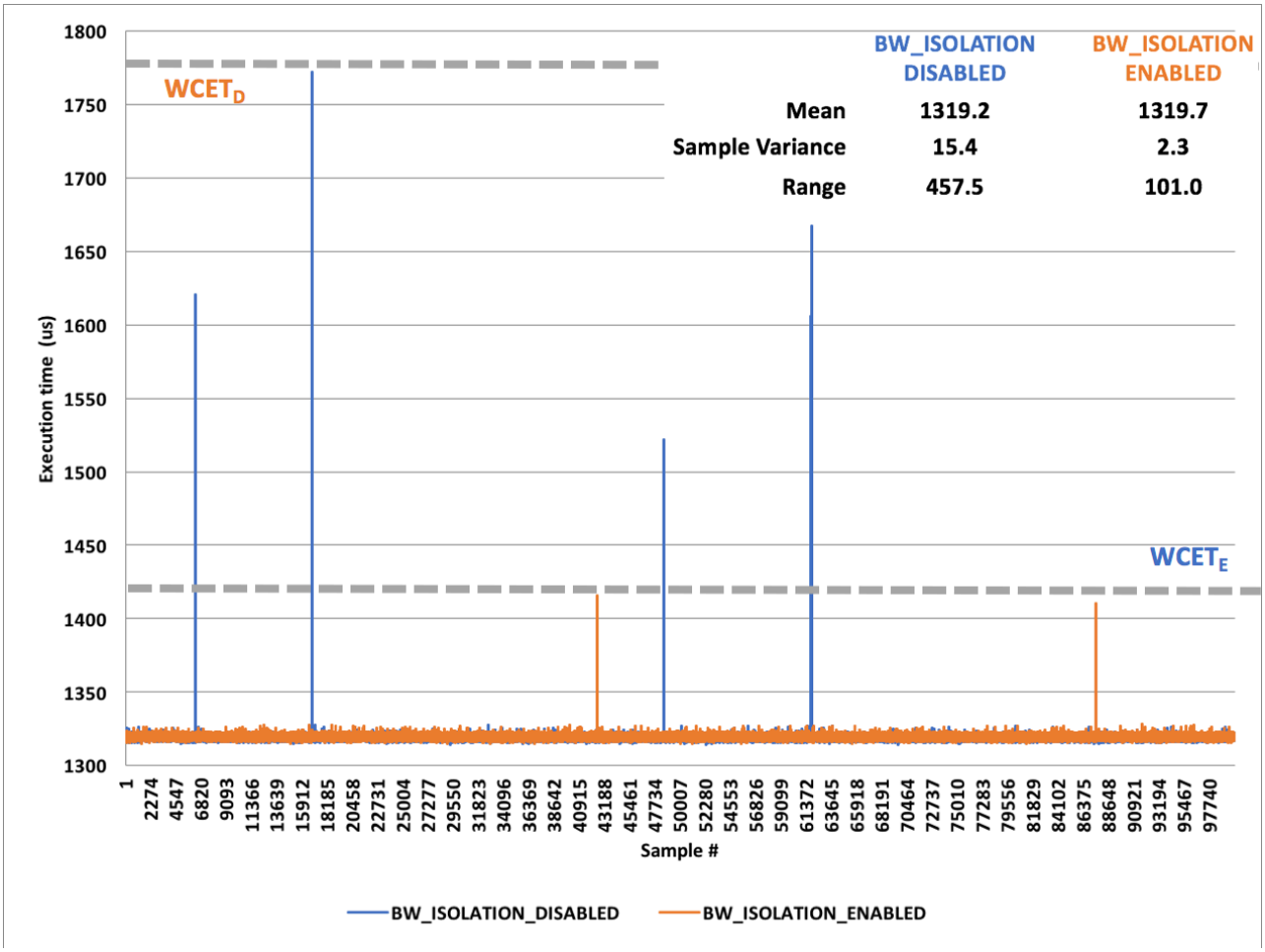


Figure 30. Effect of memory isolation mechanism on worst-case performance of Winograd kernel

(used in micro-benchmark)

## 7.4 Other Experiments

In another experiment, we measured the impact of including L1D cache miss events caused by Linux kernel (by default it is excluded by the Linux perf interface) on the performance of GPU aware memory isolation mechanism and found that it has almost no impact on the proposed mechanism. Detailed results of this experiment can be found in Appendix B.

# Chapter 8. Conclusion

## 8.1 Summary

In this master thesis, we investigated the performance of DNN inference task running on ARM-based Jetson TX2 embedded platform with integrated Nvidia Pascal GPU. Our experiments demonstrated that memory intensive tasks running on CPU cores can significantly slow down the DNN inference task. In our experiment, DetectNet inference benchmark experienced a performance slowdown (reduction in inference speed) by 45%, while YOLO inference benchmark's performance was reduced by 55% in the presence of five CPU applications running on remaining cores.

We designed a micro-benchmark using the most time-consuming CUDA kernel "Winograd convolution" observed during YOLO and DetectNet inference performance analysis, to further analyze the impact of running memory intensive CPU task on CUDA kernel execution. Using this micro-benchmark, we successfully demonstrated that the IPC: average number of GPU instructions executed per GPU clock cycle, of the Winograd based kernel reduces with increase in memory bandwidth consumption from CPU co-runners indicating that the Winograd based convolution kernel is sensitive to DRAM (memory) bandwidth contention.

We implemented a GPU aware memory isolation mechanism (based on "MemGuard" [45], a bandwidth isolation mechanism for the Intel-based platform) to reduce the impact of bandwidth contention on DNN inference performance. The memory isolation mechanism uses a locking mechanism in low-level GPU driver to trigger CPU core throttling and as a result, reduces DRAM contention caused by CPU applications. Using this memory isolation mechanism, we improved the performance (IPC) of the micro-benchmark by 13.5% in case of five co-running CPU applications, bringing the performance on par with baseline (when micro-benchmark is run in isolation). Similarly, the performance of YOLO and DetectNet benchmark was improved by 41% and 86% respectively. However, we also identified a limitation in the proposed approach. It omits the bandwidth requirement of the CPU portion of the DNN inference pipeline, which negates the performance improvement of GPU accelerated stages in some cases, thus reducing the overall DNN inference benchmark performance. Therefore, enough memory bandwidth needs to be allocated to the CPUs to support the computational requirement of the CPU portion of the DNN

inference pipeline.

## 8.2 Future Work

As for future work, we plan to focus on improving GPU aware memory isolation algorithm using following. First, we will extend current mechanism by incorporating a statistical approach to estimate the GPU load and corresponding bandwidth requirement in real-time by continuously monitoring the GPU hardware performance counter. The statistical approach will allow to dynamically set the memory bandwidth limit based on real-time bandwidth requirement. Second, we plan to add the capability of estimating memory bandwidth requirement of CPU portion of the DNN inference application and adjust the bandwidth limit accordingly to improve overall application performance. Finally, we plan to validate GPU aware memory isolation mechanism on other embedded platforms with integrated GPU.

# BIBLIOGRAPHY

[1] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, William J. Dally “EIE: Efficient Inference Engine on Compressed Deep Neural Network”

[2] B. Paul, W. Adolf, S.Rama, H. Lee, S. Lee, J́. Hernández-Lobato, G. Wei, D. Brooks, “Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators”

[3] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, and G. Zweig, “Achieving human parity in conversational speech recognition,” arXiv preprint arXiv:1610.05256, 2016.

[4] Y. LeCun, Y. Bengio, and G. Hinton. Deep Learning. Nature, 521(7553):436–444, 2015.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 265–283, 2016.

[6] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". arXiv preprint arXiv:1512.03385, 2015.

[7] A. Krizhevsk, I. Sutskever, G. Hinton, "Imagenet classification with deep convolutional neural networks". In Advances in neural information processing systems, pp. 1097–1105, 2012.

[8] Suda, Naveen, Chandra, Vikas, Dasika, Ganesh, Mohanty, Abinash, Ma, Yufei, Vrudhula, Sarma, Seo, Jae-sun, and Cao, Yu. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 16–25. ACM, 2016.

[9] Joseph Redmon, Ali Farhadi, “YOLO9000: Better, Faster, Stronger”

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Con- volutional Neural Networks. In NIPS, 2012

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In CVPR, 2009.

[12] Everingham, M. and Van~Gool, L. and Williams, C. K. I. and Winn, J. and Zisserman, A." The PASCAL Visual Object Classes Challenge: A Retrospective

[13] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv preprint arXiv:1409.1556, 2014.

[14] Qiu, Jiantao, Wang, Jie, Yao, Song, Guo, Kaiyuan, Li, Boxun, Zhou, Erjin, Yu, Jincheng, Tang, Tianqi, Xu, Ningyi, Song, Sen, et al. Going deeper with embedded fpga platform for convolutional neural network. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 26–35. ACM, 2016.

[15] Han, Song, Mao, Huizi, and Dally, William J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149, 2015.

[16] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In 43rd Annual International Symposium on Computer Architecture (ISCA), pages 380–392, 2016.

[17] D. Eklov et al., Bandwidth Bandit: Understanding Memory Contention

[18] D. Eklov et al., Design and Evaluation of the Bandwidth Bandit

[19] Nvidia. NVIDIA TensorRT: Programmable Inference Accelerator. [Online]. Available: https://developer.nvidia.com/tensorrt, 21 June. 2017

[20] MEC: Memory-efficient Convolution for Deep Neural Network Minsik

[21] Abuzaid, Firas, Hadjis, Stefan, Zhang, Ce, and Re, Christopher. Caffe control: Shallow ideas to speed up deep learning. CoRR, abs/1504.04343, 2015.

[22] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. Nature, 521(7553):436–444, 2015.

[23] Chung, Jaeyong. INsight: A Neuromorphic Computing System for Evaluation of Large Neural Networks. 2015.

[24] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press. (http://www.deeplearningbook .org)

[25] Martin T. Hagan, Howard B. Demuth, Mark H. Beale, Orlando De Jes. Neural Network Design.

[26] A. Karpathy. (2017). Stanford University CS231n: Convolutional Neural Networks for Visual Recognition, [Online]. Available: http://cs231n.github.io, 21 June. 2017

[27] S. Herculano-Houzel: The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost Proc. Natl. Acad. Sci. USA, 109 (Supp 1) (2012), pp. 10661-10668

[28] Robert H Wurtz. Recounting the impact of Hubel and Wiesel. The Journal of Physiology. 2009 Jun 15; 587(Pt 12): 2817–2823.

[29] Adit Deshpande. A Beginner's Guide To Understanding Convolutional Neural Networks, [Online]. Available: https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/. 21 June. 2017

[30] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In International Conference on Articial Neural Networks, pages 281–290. Springer, 2014.

[31] K.Abdelouahab, C.Bourrasset, M.Pelcat, F.Berry, J.C.Quinton, J.Serot. A Holistic Approach for Optimizing DSP Block Utilization of a CNN implementation on FPGA.

[32] Coppersmith, Don; Winograd, Shmuel (1990), "Matrix multiplication via arithmetic progressions",Journal of Symbolic Computation, (3): 251

[33] DetectNet: Deep Neural Network for Object Detection in DIGITS [Online]. Available: https://devblogs.nvidia.com/detectnet-deep-neural-network-object-detection-digits

[34] J. Redmon, S. Divvalla, R. Girshick, A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection.

[35] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, Piotr Dollár. Microsoft COCO: Common Objects in Context.

[36] NvCaffe [Online]. Available: http://docs.nvidia.com/deeplearning/dgx/caffe-user-guide/index.html. 15 September.2017

[37] Christian Szegedy, Wei Liu , Yangqing Jia , Pierre Sermanet , Scott Reed , Dragomir Anguelov , Dumitru Erhan, Vincent Vanhoucke , Andrew Rabinovich. Going Deeper With Colvolutions.

[38] Dustin Franklin. NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge [Online]. Available: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge. 15 Septmeber. 2017

[39] Mark S. Papamarcos, Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. ISCA '84 Proceedings of the 11th annual international symposium on Computer architecture. Pages 348-354

[40] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning.
[41] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks,” in ISCA, 2016.

[42] D. Pala ; G. Causapruno ; M. Vacca ; F. Riente ; G. Turvani ; M. Graziano ; M. Zamboni Logic-in-Memory architecture made real.preprint arXiv:1704.04861, 2017.

[43] B. Forsberg et al., “Gpuguard: Towards supporting a predictable execution model for heterogeneous soc,” in Design, Automation and Test in Europe (DATE), 2017

[44] Forsberg, Björn, Palossi, Daniele Marongiu, Andrea Benini, Luca. GPU-Accelerated Real-Time Path Planning and the Predictable Execution Model

[45] Caccamo, Marco, Pellizzoni, Rodolfo, Sha, Lui, Yao, Gang, Yun, Heechul. Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms

[46] H. Yun, S. Gondi, et. al, Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms

[47] Chellapilla, Kumar, Puri, Sidd, and Simard, Patrice. High Performance Convolutional Neural Networks for Docu- ment Processing. In Tenth International Workshop on Frontiers in Handwriting Recognition, October 2006.

[48] Denton, Emily, Zaremba, Wojciech, Bruna, Joan, LeCun, Yann, and Fergus, Rob. Exploiting linear structure within convolutional networks for efficient evaluation. CoRR, abs/1404.0736, 2014

[49] Jaderberg, Max, Vedaldi, Andrea, and Zisserman, Andrew. Speeding up convolutional neural networks with low rank expansions. CoRR, abs/1405.3866, 2014.

[50] Jia, Yangqing. Learning Semantic Image Representations at a Large Scale. PhD thesis, EECS Department, University of California, Berkeley, May 2014.

[51] Park, Hyunsun, Kim, Dongyoung, Ahn, Junwhan, andYoo, Sungjoo. Zero and data reuse-aware fast convolution for deep neural networks on gpu. In Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES '16, 2016a.

[52] Jetson Inference: Guide to deploying deep-learning inference networks and deep vision primitives with TensorRT and Jetson TX1/TX2 [Online]. Available: https://github.com/dusty-nv/jetson-inference

[53] Jetson/Installing OpenCV [Online]. Available: https://elinux.org/Jetson/Installing_OpenCV

[54] Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A.: The PASCAL Visual Object Classes (VOC) Challenge. IJCV (2010)

[55] Paul Henderson, Vittorio Ferrari.End-to-end training of object class detectors for mean average precision

[56] Vivienne Sze, Energy - Efficient Hardware for Embedded Vision and Deep Convolutional Neural Network

[57] Alex Krizhevesky, Ilya Sutskever, Geoffrey E. Hinton ImageNet Classification with Deep Convolutional Neural Networks

[58] https://developer.nvidia.com/cudnn

[59] SQUASH: Simple QoS-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators

[60] High-Performance Real-time Architectures for Low-Power Embedded Systems

[61] Y. Tanabe, M. Sumiyoshi, M. Nishiyama, I. Yamazaki, S. Fujii, K. Kimura, T. Aoyama, M. Banno, H. Hayashi, and T. Miyamori. A 464GOPS 620GOPS/W heterogeneous multi-core SoC for image-recognition applications. In ISSCC, 2012.

[62] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni, "PALLOC: DRAM bank- aware memory allocator for performance isolation on multicore platforms," in Proc. Real-Time Embedded Technol. Appl. Symp., 2014, pp. 155– 166.

[63] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank and cache coloring for temporal protection of memory accesses," in Proc. Parallel Archit. Compilation Techn., 2012, pp. 367–376

[64] J. Hestness, S.W. Keckler, D.A. Wood, "GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors"

[65] Andrew Lavin. "Fast algorithms for convolutional neural networks". CoRR, abs/1509.09308, 2015.

[66] M. Cho, D. Brand "MEC: Memory-efficient Convolution for Deep Neural Network"

[67] https://elixir.free-electrons.com/linux/v4.0/source/kernel/events/core.c#L7770

[68] PARKER | TRM | DP-07281-001_v1.0p

[69] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv:1409.1556, 2014.

[70] T. Mikolov, M. Karafi´at, L. Burget, J. Cernock`y, and S. Khudanpur, "Recurrent neural network based language model." in INTER- SPEECH, September 26-30, 2010, 2010, pp. 1045–1048.

[71] S. Zhou, Y. Wang, H. Wen, Q. He, Y. Zou, "Balanced Quantization: An Effective and Efficient Approach to Quantized Neural Networks"

[72] Y. Umuroglu, J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference"

[73] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave gaussian quantization," in CVPR, 2017.

[74] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental Network Quantization: Towards Lossless CNNs with Low- precision Weights," in ICLR, 2017.

[75] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Ben- gio, "Quantized neural networks: Training neural networks with low precision weights and activations," arXiv preprint arXiv:1609.07061, 2016.

[76] V. Sze, Y. Chen, T. Yang, J. Emer, "DSD: Dense-Sparse-Dense Training for Deep Neural Networks"

[77] S. Han, H. Mao, and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” in ICLR, 2016.

[78] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, “Scalpel: Customizing dnn pruning to the underlying hardware parallelism,” in ISCA, 2017.

[79] T.-J. Yang, Y.-H. Chen, and V. Sze, “Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning,” in CVPR, 2017.

[80] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, “Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks”

[81] Building Jetson TX2 kernel [Online]. Availble: https://github.com/jetsonhacks/buildJetsonTX2Kernel

[82] K. He, X. Zhang, S. Ren, and J. Sun. “Deep residual learning for image recognition”

[83] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, et al. “Deep speech 2: End-to-end speech recognition in english and mandarin”

[84] A. Peter Greenhalgh, “Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7,” 2011.

[85] nVidia, “Variable SMP A Multi-Core CPU Architecture for Low Power and High Performance,” 2011.

[86] S. Kato, S. Brandt. “Operating Systems Challenges for GPU Resource Management”

[87] V. G. Maltarollo, K. M. Honório, and A. B. F. da Silva. “Applications of artificial neural networks in chemical problems”, 2013.

[88] J. Long, E. Shelhamer, T. Darrell. “Fully Convolutional Networks for Semantic Segmentation”

[89] A. Ruiz. “Deep Learning with Data Science Experience” [Online]. Available: https://medium.com/ibm-data-science-experience/deep-learning-with-data-science-experience-8478cc0f81ac. 19 November. 2017.

[90] http://man7.org/linux/man-pages/man2/sched_setaffinity.2.html

[91] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT-13), 2004

[92] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-aware Scheduling and Partitioning. In Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA-8), 2002.

[93] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. Journal of Supercomputing, 28(1):7–26, 2004.

[94] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural Support for Operating System-Driven CMP Cache Management. In Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques (PACT-15), 2006.

[95] N. Suzuki, H. Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein, and R. R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In IEEE International Conference on Embedded Software and Systems (ICESS), 2013.

[96] https://www.youtube.com/watch?v=OsXedJq1aWE

[97] I. Sobel, G. Feldman, "A 3x3 Isotropic Gradient Operator for Image Processing"

[98] https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt

[99] R. Pellizzoni, Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems

[100] S. Chattopadhyay, S. Narayan, Worst Case Execution Time Analysis of Automotive Software

[101] R. Wilhelm, Static Timing Analysis for Hard Real-Time Systems

[102] https://github.com/pjreddie/darknet

[103] A. Lavin, S. Gray, Fast Algorithms for Convolutional Neural Networks

# APPENDIX A.

# SCRIPT USED FOR SETTING FREQUENCY REGULATOR ON JETSON TX2

```
#!/bin/bash
if [[ $(id -u) -ne 0 ]]; then
    echo "must be root"
    exit 1
fi
echo userspace >
/sys/devices/17000000.gp10b/devfreq/17000000.gp10b/governor
cat /sys/devices/17000000.gp10b/devfreq/17000000.gp10b/max_freq >
/sys/devices/17000000.gp10b/devfreq/17000000.gp10b/userspace/set_freq
echo 0 > /sys/module/qos/parameters/enable
echo 0 > /sys/kernel/debug/tegra_cpufreq/M_CLUSTER/cc3/enable
echo 0 > /sys/kernel/debug/tegra_cpufreq/B_CLUSTER/cc3/enable
echo 2000 > /sys/kernel/debug/tegra_cpufreq/freq_compute_delay
echo "userspace" | tee
/sys/devices/system/cpu/cpu*/cpufreq/scaling_governor > /dev/null
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq >
/sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
cat /sys/devices/system/cpu/cpu1/cpufreq/scaling_max_freq >
/sys/devices/system/cpu/cpu1/cpufreq/scaling_setspeed
cat /sys/devices/system/cpu/cpu2/cpufreq/scaling_max_freq >
/sys/devices/system/cpu/cpu2/cpufreq/scaling_setspeed
cat /sys/devices/system/cpu/cpu3/cpufreq/scaling_max_freq >
/sys/devices/system/cpu/cpu3/cpufreq/scaling_setspeed
cat /sys/devices/system/cpu/cpu4/cpufreq/scaling_max_freq >
/sys/devices/system/cpu/cpu4/cpufreq/scaling_setspeed
cat /sys/devices/system/cpu/cpu5/cpufreq/scaling_max_freq >
/sys/devices/system/cpu/cpu5/cpufreq/scaling_setspeed
cat /sys/kernel/debug/bpmp/debug/regulator/vdd_core/max_uv >
/sys/kernel/debug/bpmp/debug/regulator/vdd_core/override
cat /sys/kernel/debug/bpmp/debug/regulator/vdd_sram/max_uv >
/sys/kernel/debug/bpmp/debug/regulator/vdd_sram/override
cat /sys/kernel/debug/bpmp/debug/regulator/vdd_cpu/max_uv >
/sys/kernel/debug/bpmp/debug/regulator/vdd_cpu/override
cat /sys/kernel/debug/bpmp/debug/regulator/vdd_aon/max_uv >
/sys/kernel/debug/bpmp/debug/regulator/vdd_aon/override
cat /sys/kernel/debug/bpmp/debug/regulator/vdd_gpu/max_uv >
/sys/kernel/debug/bpmp/debug/regulator/vdd_gpu/override
if [[ -e /sys/kernel/debug/tegra_cpufreq/auto_cc3 ]] ; then
    echo 0 > /sys/kernel/debug/tegra_cpufreq/auto_cc3
fi
```

```
echo 1 | tee /sys/kernel/debug/bpmp/debug/clk/*/mrq_rate_locked >
/dev/null
#GPU power/clock gating controls
echo 0 | tee /sys/devices/gpu.0/aelpg_enable > /dev/null
echo 0 | tee /sys/devices/gpu.0/elpg_enable > /dev/null
echo 0 | tee /sys/devices/gpu.0/blcg_enable > /dev/null
echo 0 | tee /sys/devices/gpu.0/slcg_enable > /dev/null
echo 0 | tee /sys/devices/gpu.0/elcg_enable > /dev/null
```

# APPENDIX B.

## a) Effect of OS kernel memory access

Linux perf interface provides the option to include or exclude HW events caused by OS kernel. By default, GPU aware bandwidth isolation mechanism excludes L1D cache miss event caused by OS kernel memory access. In this experiment, we measure the effect of including L1D cache miss event caused by OS kernel memory access on the performance of GPU aware Memory Bandwidth Isolation Mechanism. The performance will be measured in terms of relative IPC normalized to IPC recorded when minimum bandwidth is set to its optimal value (discussed on page) for corresponding benchmark.

As per Figure 31, relative IPC lies in the range ~{0.997, 1.002}, ~{0.999, 1.005}, ~{0.9984, 0.9995} and ~{1.0002, 1.0008} for image 256p, 512p, 1024p and 2048p respectively. Similarly, when minimum bandwidth is set to 512MB/S, the relative IPC lies in the range ~{0.985,1}, ~{0.9995,1.0005}, ~{0.9982,0.9989} and ~{0.9993,1.0012}for image 256p, 512p, 1024p and 2048p respectively. We can notice that relative IPC change is minimal. We can conclude that effect of kernel memory access is minimal on the performance of bandwidth isolation mechanism.

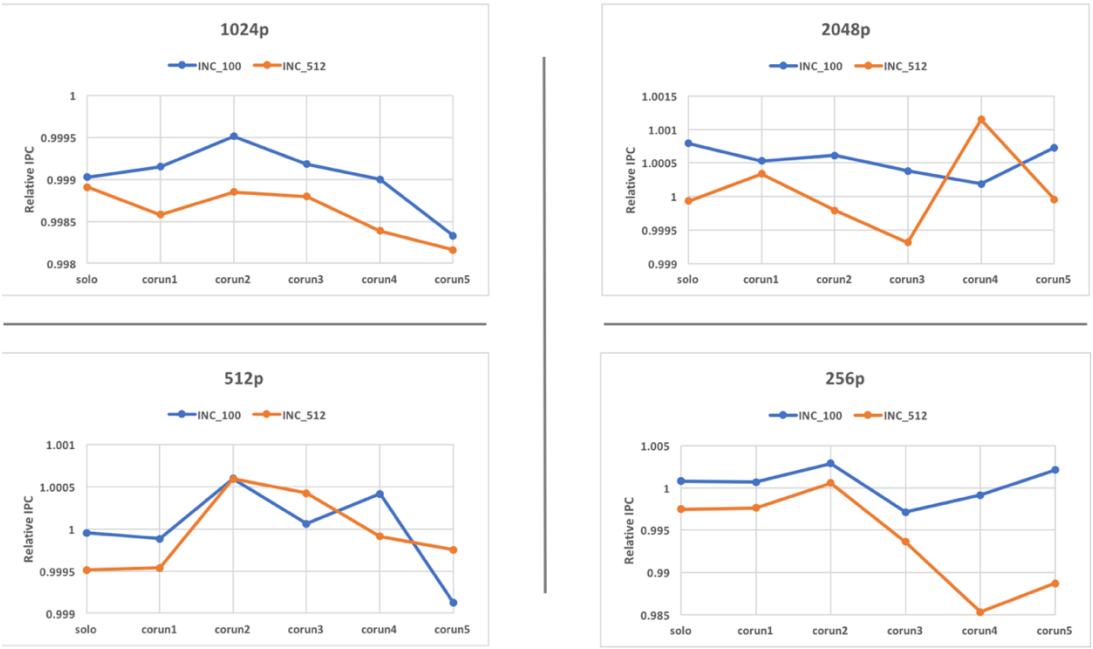


Figure 31. Effect of kernel memory access on the performance of bandwidth isolation mechanism.

INC_512 shows result when minimum bandwidth is set to 512MB/s. INC_100 shows results when minimum bandwidth is set to 100MB/s.