6-2022

Implementation and Evaluation of Deep Neural Networks in Commercially Available Processing in Memory Hardware

Prangon Das
pcd3897@rit.edu

Implementation and Evaluation of Deep Neural Networks in Commercially Available Processing in Memory Hardware

Prangon Das

Implementation and Evaluation of Deep Neural Networks in Commercially Available Processing in Memory Hardware

Prangon Das

June 2022

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

RIT | Kate Gleason College of Engineering

Department of Computer Engineering

Implementation and Evaluation of Deep Neural Networks in Commercially Available Processing in Memory Hardware

Prangon Das

Committee Approval:

Amlan Ganguly *Advisor* Date
Department of Computer Engineering

Muhammad Shaaban Date
Department of Computer Engineering

Mark Indovina Date
Department of Electrical and Microelectronic Engineering

Abstract

Deep Neural Networks (DNN), specifically Convolutional Neural Networks (CNNs) are often associated with a large number of data-parallel computations. Therefore, data-centric computing paradigms, such as Processing in Memory (PIM), are being widely explored for CNN acceleration applications. A recent PIM architecture, developed and commercialized by the UPMEM company, has demonstrated impressive performance boost over traditional CPU-based systems for a wide range of data-parallel applications. However, the application domain of CNN acceleration is yet to be explored on this PIM platform. In this work, successful implementations of CNNs on the UPMEM PIM system are presented. Furthermore, multiple operation mapping schemes with different optimization goals are explored. Based on the data achieved from the physical implementation of the CNNs on the UPMEM system, key-takeaways for future implementations and further UPMEM improvements are presented. Finally, to compare UPMEM's performance with other PIMs, a model is proposed that is capable of producing estimated performance results of PIMs given architectural parameters. The creation and usage of the model is covered in this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The physical separation between the processing unit and the memory unit inside a von Neumann computing architecture makes the memory accesses expensive, both in terms of latency and power consumption [5, 6]. This issue is further aggravated when the application is data-centric. Consequently, data-intensive Machine Learning Applications, especially Deep Neural Networks (DNN) are met with inefficiency and bottlenecked performance when implemented in the von Neumann computing devices such as CPUs and GPUs [7].

Processing in Memory (PIM), a novel computing paradigm, addresses the inefficient data-movement issue of the Von Neumann model by placing the processing units inside the memory chip itself [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. This essentially restricts the circulation of the data within the memory chip and thereby dramatically minimizes the power dissipation and latency from the data movements. Moreover, a PIM can further capitalize on the data locality by processing the data in a processing element nearest to the data [21, 22]. This provides the PIM devices with massively parallel processing capability within the memory chip at very low operational latency and power consumption [12, 13, 14].

UPMEM PIM, a commercially available memory-centric processing hardware, integrates small-scale traditional processors within standard DRAM chips. This allows it to offer up to 15 $\times$ better performance than Intel x86-based servers on a wide range

of data-centric applications [1]. Designed primarily for data-parallel applications (*i.e.* graph processing, genomic sequencing, etc.), the domains of machine learning, deep learning and AI acceleration are yet to be explored.

There are several challenges to implementing the mainstream AI algorithms in the UPMEM processing hardware. First, UPMEM only supports fixed-point operations which requires standard CNNs implementations to be quantized accordingly. Second, UPMEM requires programs to be compiled for its RISC based PIM processors using a custom Clang based compiler. Finally, UPMEM requires the size of any data (*i.e* CNN input and weight data) sent from the host to the PIM be divisible by 8.

This work investigates the implementation of CNN algorithms on the UPMEM PIM environment by working around the aforementioned limitations and challenges. First, CNNs algorithms (*i.e.* YOLOv3 [4] and eBNN [23]) are adapted to UPMEM PIM's computational limitations by using quantized versions and restructuring conflicting parts of the applications. Second, to comply with the UPMEM's compiler, the data-centric portions of an application is separated and compiled insteads of an entire CNN application. Third, to comply with the memory size restrictions for the input/output matrices, padding is used and the size of the actual data is communicated to the PIM prior to processing.

Alongside, this work also demonstrates how the applications are mapped onto UPMEM processing elements to capitalize on UPMEM's multiple parallel layers in their architecture. Furthermore, the CNN functions compiled for the PIM are synchronized with the host in order to preserve the integrity and flow of the application. This allows the presentation of a standardized framework for adapting and implementing any CNN application within the UPMEM PIM system.

Apart from the commercially available UPMEM PIM solutions, a number of promising PIM solutions have been proposed for AI acceleration applications [12, 13, 17, 14]. In order to evaluate the merit of UPMEM PIM as an AI accelerator, CNN-

implementations on UPMEM are compared with with several other high-performance PIM architectures. To facilitate the comparison, a model is created that can estimate the latency of an application given a few parameters: number of operations, operand length, scale function, number of processing elements, etc. This model is generated for both computations and memory access in PIM.

Thesis contributions:

1. A verified methodology for supporting CNN acceleration is presented on the UPMEM PIM solution that does not support this application domain by default.

2. A model for estimating the performance of PIMs is proposed. This model is used to perform comparative evaluation of UPMEM PIM's performance along with several other PIM architecture.

The rest of the document is organized in the following manner: Chapter 2 covers some necessary background information regarding general Processing-in-Memory as well as UPMEM specifically. Chapter 3 covers an indepth exploration of general application specific information for programs running on UPMEM's system. Chapter 4 covers, the implementation of CNNs in the UPMEM system and presents the findings. Chapter 5 covers the creation of a model that can be used to estimate the performance of a PIM for comparison with other PIMs. Chapter 6 concludes this work by presenting key points and provides acknowledgements.

Chapter 2

Background

Before getting started with the specific contributions in this work, some necessary background material needs to be covered first. This chapter covers the topic of Processing-in-Memory (PIM) by describing what they are: accelerators. Various designs of PIMs exist that boast higher and higher accelerations and this chapter covers some theoretical designs. Unlike these designs, there now also exists a commercially available PIM by UPMEM and so the architecture of this PIM is also explored in detail as well.

2.1 Processing in Memory

Von-Neumann computer architectures contain a memory hierarchy alongside a Central Processing Unit (CPU). The closest and smallest memory is the register. Registers are located in the CPU die and often take one cycle. Cache is the next biggest memory and is further from the CPU but is still in the die. Main system memory is off-die and, while much larger, suffers from the largest access penalties. For applications that contain a large amount of memory access, cache misses lead to performance degradation.

Processing-in-Memory (PIM) is a concept that alters the Von-Neumann model and places data-centric processing in main memory. These designs, in contrast to the Von-Neumann model, follow the Harvard architecture in regards to memory location

and access. The general idea is the removal of the necessity to go off die/chip for data when dealing with data-centric applications. This reduces the power consumption necessary per memory access and reduces the wait time for processing elements.

2.1.1 Theoretical PIM Architectures

PIM itself is not a novel concept, the idea is first introduced in 1970 by Stone et al.[9]. However, in recent years, a wide range of theoretical designs have been proposed for PIM, mostly in DRAM technology. These designs vary in granularity. For example, there are bit-wise accelerators that operate on memory sub-array rows such as DRISA[12], Ambit [24], and SCOPE [13]. Computations in these bitwise-accelerators are done using traditional logic gates that are implemented by Boolean bitline logic and more complex logic is ran by serially executing multiple Boolean logic gates.

A more recent design is based around using Look-Up-Tables (LUTs) for computation acceleration. LUT designs fundamentally function by breaking down operands into supported input sizes and accessing a register file using these operands to produce a pre-programmed result. Specific implementations such as LAcc [17] and pPIM [14] have shown higher performance ratings than Bitwise implementations.

Finally, there are designs, such as ISAAC [25], XNOR-SRAM [26] and MRAM-DIMA [27], that utilize analog crossbar array architecture to perform program specific computations.

2.1.2 UPMEM PIM Architecture

UPMEM is, to the best of the author's knowledge, the first and the only commercially manufactured PIM hardware solution based on Dynamic random-access memory (DRAM). UPMEM integrates multiple Processing Elements (PEs) within each DRAM chip, referred to as the DRAM Processing Units (DPU). Figure 2.1 shows an overview of the UPMEM DPU architecture. The DPU features a patented pro-

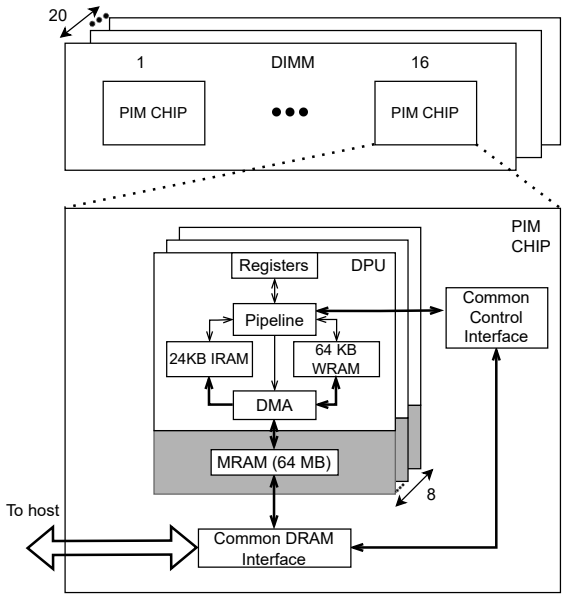

Figure 2.1: Overview of the architecture of the UPMEM DPU. Each DRAM chip contains multiple DPUs, each of which consists of a reduced instruction set (RISC) pipeline, two internal memories (IRAM and WRAM) and an external MRAM [1, 2]

cessing architecture, featuring a pipeline, accompanied by a novel data/instruction memory organization [2]. The architecture and the micro-architecture of the DPU is inpired by the Reduced Instruction Set Computing architecture (RISC). The goal of the DPUs is to support small/large-precision fixed-point computations with high energy-efficiency. The DPU has access to three dedicated physical hierarchies of memory: a) the relatively larger Main RAM (MRAM), b) the internal Instruction RAM (IRAM), and c) Working RAM (WRAM) of the DPUs, with the latter two being located inside the DPUs for a significantly lower access latency compared to MRAM. The capacities of each of these memory hierarchies are listed in Table 2.1. Although the DRAM chips on which the UPMEM PIM system is implemented are in the 25nm technology node, the DPUs themselves are developed in the 65nm technology node. Based on Figure 2.1, it can be observed that the internal DPU follows the Harvard approach where both instructions and data can be accessed through different mediums. However, the separation of the MRAM to outside the DPU with only DMA follows the Von-Neumman model. Thus, UPMEM's DPU system can be seen as a

Table 2.1: UPMEM PIM Attributes

No. of DPUs	2560 (20 DIMM)
No. of DPUs/ DIMM	128
DPU/ Chip	8
Available Memory/ Chip	512 MB
DPU Area	3.75 mm^2
DPU Power Consumption	120 mW
DPU Operating Frequency	350 MHz
DPU Hardware Threads (*i.e* Tasklets)	1-24
DPU Pipeline Stages	11
DPU Registers/ Thread	32
DPU MRAM Size	64 MB
DPU WRAM Size	64 KB
DPU IRAM Size	24 KB

hybrid approach compared to other PIMs that strictly follow the Harvard approach.

Chapter 3

UPMEM PIM Programming

Running any application in a new system requires understanding of the programming infrastructure the system uses. This chapter covers, in more depth, analysis of general programming aspects of the UPMEM PIM system. Topics that are not specific to CNNs but are just as important are covered in this chapter.

3.1 Programming Interface

The programming interface for UPMEM's PIM is developed based on the LLVM compiler infrastructure and therefore only supports a C-based programming environment. Therefore, all CNN algorithms running inside a DPU need to be implemented in the C programming environment. However, it is possible for the host application to be written in Python/C/C++/Java and have the host run C-only DPU programs.

The compiler called, "dpu-clang", has default built-in optimization capabilities of the UPMEM compiling infrastructure (referred to as the "O 0-3" optimization settings). The O0 optimization is no optimization at all. With this optimization parameter the compilation is quickest and most debuggable. The O3 optimization is the highest standard optimization that enables all compiler optimizations disregarding program size and compilation time.

UPMEM's system is interfaced to the host CPU as a memory-centric accelerator. In this configuration, the host CPU delegates the memory intensive compu-

tational workloads to the DPUs. In order to perform this delegation, application profiling is used to identify the specific set of tasks that are to be processed within the DPUs. These operations are characterized by highly data-parallel nature. The created program schedules these tasks to the DPUs, followed by initialization and data/instruction orchestration within the DIMMs to execute the tasks. Meanwhile, the host processor is assigned the remaining tasks as well as the monitoring of the data-communications and operating of the DPUs.

Each DPU in the UPMEM system is capable of processing multiple concurrent threads at a time. These are software-level abstraction of the hardware threads and are referred to as the 'Tasklets'. The execution of each program is further accelerated within the DPU via the concurrent execution of multiple Tasklets in the pipeline. Each tasklet runs the same program and thereby essentially implements a single instruction multiple thread (SIMT) execution environment. Furthermore, multiple DPUs on the system can execute the same program on different parallel data, and therefore implement a single instruction multiple data (SIMD) processing architecture across the DIMMs.

3.2 DPU Memory

The memory shown as MRAM, WRAM and IRAM in figure 2.1 can be accessed via library functions. The memory transfer from the host to the DPUs consists of looping through the input values, weights and output labels and copying the memory to the DPUs MRAM. UPMEM memory transfer functions such as equation 3.1 exist to copy the same data to multiple DPUs at once [1, 2]. However, to transfer different data to different DPUs, a combination of equation 3.2 and equation 3.3 is needed. Using a for each macro, a set of DPUs could have a different memory address given as the start of the buffer using dpu_prepare_xfer, even though it is the same buffer. Then, the dpu_push_xfer would "push" the buffers to the DPUs. The length parameter in

dpu_push_xfer makes sure only the specific buffer size get sent. The symbol_name is the name of the MRAM variable in the DPU that would receive the data. For a WRAM variable the attribute "__host" would be needed as extra. The offset is 0 and the rest are predefined macros [1].

$$\begin{aligned} &dpu\_copy\_to(struct\ dpu\_set\_t\ set,\\ &\quad const\ char^*\ symbol\_name,\\ &\quad uint32\_t\ symbol\_offset,\\ &\quad const\ void^*\ src,\ size\_t\ length) \end{aligned} \tag{3.1}$$

$$\begin{aligned} &dpu\_prepare\_xfer(struct\ dpu\_set\_t\ dpu\_set,\\ &\quad void^*\ buffer) \end{aligned} \tag{3.2}$$

$$\begin{aligned} &dpu\_push\_xfer(struct\ dpu\_set\_t\\ &dpu\_set,\ dpu\_xfer\_t\ xfer,\\ &const\ char^*\ symbol\_name,\\ &uint32\_t\ symbol\_offset,\\ &size\_t\ length,\ dpu\_xfer\_flags\_t\ flags) \end{aligned} \tag{3.3}$$

The DPU's internal program is written to have access to the MRAM data via symbols pointing at the specific buffers for input and output data. The DPU's main function loops through all data and copies the data from MRAM to WRAM and executes the main functionality. Since the DPUs are simpler processing architectures with limited data allocating capabilities, proper data orchestration within the DPU

memory (*i.e.* MRAM, WRAM and IRAM) has to be enforced by the compiler. Memory orchestration between the host and DPUs must ensure that the memory being orchestrated is aligned on 8 bytes and divisible by 8 bytes when being allocated to the MRAM. For implementations where the size of the data is not aligned, padding to the sent/received memory buffers from the DPUs needs to be added. In order to make sure the DPU does not mistakenly include these padded bytes in its computations, the size of the non-padded buffer must be sent from the host to the DPU.

3.2.1 Results

The internal MRAM and WRAM discussed in Sections 2.1.2 and 3.2 contain different access timings. With WRAM located inside the DPU the number of cycles needed for a WRAM access is 1 Cycle. However, with MRAM located outside of the DPU, the DMA shown in 2.1 is activated for every MRAM transfer to the WRAM. The very usage of the DMA incurs a 25 cycle penalty. Furthermore, for every 2 bytes necessary for the transfer, there is a 1 cycle penalty. For example, a transfer of 2048 bytes from MRAM to WRAM takes 1049 cycles as shown in equation 3.4.

$$\begin{aligned} \text{MRAM Access (Cycles)} &= \text{DMA Cycles} + (\text{Total Bytes}/2) \\ &= 25 + (2048/2) \\ &= 1049 \end{aligned} \tag{3.4}$$

3.3 High Precision Computation

While the UPMEM DPU is a 32-bit processor, there does not exist hardware for all possible computation sizes. There is no hardware support for 32-bit fixed point multiplication/division and for any floating point operation. For these high precision com-

```c
#include <stdio.h>
#include <stdint.h>
#include <perfcounter.h>
#include <float.h>

int main() {

        float num = FLT_MAX;
        float num2 = FLT_MAX;
        double result1;

        perfcounter_config(COUNT_CYCLES, true);

        result1 = num * num2;

        perfcounter_t run_cycles = perfcounter_get();
        printf("Cycles: %lu\n", run_cycles);

        return 0;


}
```

Figure 3.1: Sample program used to profile and measure the number of cycles per instruction for a operation. Here two floating point operations are multiplied together. The perfcounter functions measure the amount of cycles from perfcounter_config() to perfcounter_get()

putations, UPMEM's compiler calls subroutines that break down the operations into multiple smaller operations. For example, __mulsi3,__muldf3, __addsf3, and __dddf3 [28] are some routines frequently called in applications. Another aspect of the UPMEM compiler is that 16-bit multiplication operations also use software subroutines under no-optimization but collapse into regular instructions under full optimization.

3.3.1 Results

The aforementioned subroutines carry a large cost in terms of computational cycles. UPMEM does not suggest the usage of any high precision computations in their DPUs due to this cost. To observe the quantifiable effect of using such routines, a program, shown in figure 3.1, is written that prints the number of cycles different instructions take in a single DPU. The resulting data for a program with no compiler optimization operating on the maximum type values for operands is shown in Table 3.1. As seen in Table 3.1, fixed point addition/subtraction takes the same amount of cycles for all precision. Division performs similarly but with a relatively higher

Table 3.1: Number of cycles for different operations in a single DPU

Precision	Addition (Cycles)	Multiplication (Cycles)	Subtraction (Cycles)	Division (Cycles)
8-bit fixed point	272	272	272	368
16-bit fixed point	272	608	272	368
32-bit fixed point	272	800	272	368
32-bit floating point	896	2528	928	12064

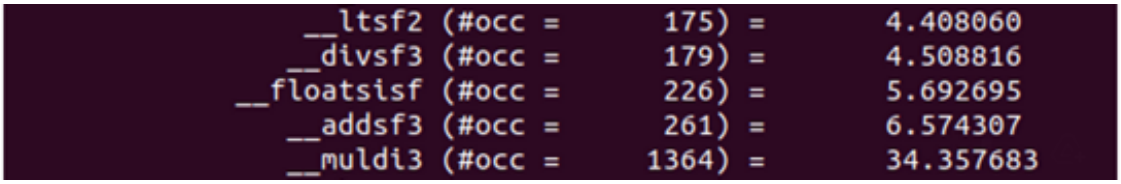

Figure 3.2: Profiling of DPU application with high precision computations. _ltsf2 is a comparison routine, _divsf3 is a division routine, _floatsisf is a conversion routine, _addsf3 is a addition routine and _muldi3 is a multiplication routine, all for floating point operations. The #occ value is the number of times this subroutine is called in the program

count. Compared to 32-bit fixed point addition, 32-bit fixed point multiplication is about ×2.9 slower. Floating point operations performs relatively the poorest inside the DPUs. Compared to 32-bit fixed point addition, 32-bit floating point addition is about ×3.3 slower. Compared to 32-bit fixed point multiplication, 32-bit floating point multiplication is about ×3.2 slower. Compared to 32-bit floating point addition, 32-bit floating point multiplication is about ×2.3 slower. Floating point division has the worst value in comparison to other operations. The actual cycle count includes cycles needed for profiling and thus is better seen in a comparative point of view.

Figure 3.2 shows a sample output of profiling a DPU application that includes high precision computations. The number of occurrences (#occ) shows how many times subroutines used to handle floating point operations are called. Given the large cycle penalty for using high precision computations and the number of times these operations get called, it is suggested for any applications running on the UPMEM system to use low precision computations.

Chapter 4

Analysis of Neural Networks in UPMEM PIM

Convolutions Neural Networks (CNNs) are characterized by a large amount of data-parallel computations such as the Multiply and Accumalate (MAC) operations. Conveniently, the UPMEM PIM system is highly optimized for supporting massively parallel computations. However, the baseline UPMEM system does not offer support for CNN acceleration applications. It's observed that with carefully developed operation mapping and orchestration schemes, it is possible to implement the CNNs in the UPMEM PIM environment with promising performance gains.

In order to achieve the most optimized resource utilization, the CNN acceleration workload is also partially shared by the host. Profiling steps are utilized *i.e.*implementation algorithms are modified in order to separate the data-centric functions, which is usually the convolutional portion, from the rest of the application. This is followed by the deployment of the Convolutional layer/functions to the DPUs while the other layers are executed by the host. The compilation of the non-convolutional layer, and the input/output data orchestration is carried out by the host itself.

Two different mapping schemes for the CNNs in the UPMEM system is presented in this chapter. In the first scheme, multiple images are ran per DPU, for which eBNN [23] is chosen as the target CNN architecture. eBNN is chosen due to its minimalistic architecture consisting of a relatively low number of computations. The second scheme uses multiple DPUs dynamically for operating on a each single image.

For this purpose a relatively larger CNN algorithm of YOLOv3 is chosen. Alongside featuring a relatively deeper network architecture and consequently a significantly larger computational workload, the YOLOv3 implementation also has larger input dimensions. These make it a relatively more challenging computing task compared to the eBNN when it comes to performing multiple images per DPU. The choice of dynamic DPU assignment to this task enables the programming model to use an optimum number of DPUs for processing each layer of the network.

4.1 Implementation of eBNN

4.1.1 eBNN Architecture

Embedded Binary Neural Network (eBNN) [23] presents an advanced form of the Binary Neural Network architecture, introduced in the work BinaryNet[29], which aims to achieve a higher level of optimization and compatibility for embedded processing devices. This architecture focuses on reducing the required memory footprint of temporaries (*i.e.* the intermediate results between layers). This is carried out by using binary quantization of the weights, the temporaries and the inputs, instead of retaining floating-point precision in all the intermediate buffers. To this end, eBNN is chosen since it contains minimal high precision computational workload and therefore allows an easier integration within the DPUs. A custom architecture for eBNN is adopted that consists of one Convolutional-Pooling block, followed by a Softmax layer. One of the key advantages of the BNN architecture is its exclusive utilization of binarized weights that essentially simplify the convolutions to a stream of bitwise computation, followed by accumulations (additions). This significantly reduces the complexity and overhead associated with performing convolutions in the UPMEM DPUs.

4.1.2 Dataset Specification

The data that is used for inference testing is the MNIST Dataset. This is a collection of handwritten digits from 0 to 9. Each digit is represented by an array with 28 rows and columns with each cell represented by a byte. An example is shown in Figure 4.1.

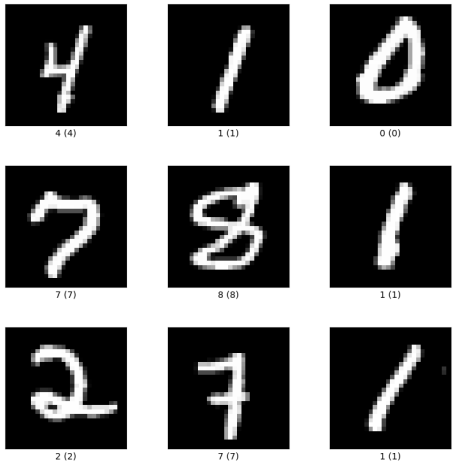

Figure 4.1: MNIST Dataset Example Images [3]

4.1.3 Mapping of Images to DPU

To take advantage of the hardware threading available per DPU, the DPU's convolutional code is adapted to accept multiple input data. This means a single DPU is allowed to work on multiple images concurrently. The images data is copied from the MRAM to WRAM and then delegated to the threads. The number of threads chosen to run in the DPU is 16 because a maximum of 16 images are allowed to be transferred from the MRAM to WRAM. The image limit is due to the size of the image transfer being limited by a maximum of 2048 bytes per transfer.

To further the image throughput, the multiple DPUs available per system is taken advantage of. The input image data buffer for many images is divided by the number

of images per DPU to get the number of DPUs needed. This number of DPUs are run in parallel to finish their batch of images at the max time for one DPU.

After all DPUs complete their convolutional computations for all images, each DPU's temporary result is parsed. This temporary result from a DPU is multiple convolution results for all images sent to the DPU. The host takes this temporary result buffer and serially sends a single image's processed result to the softmax layer for inference. After all temporary results for all images in a single DPU are inferred, the next DPU's result is read.

4.1.4 eBNN Floating-Point Removal

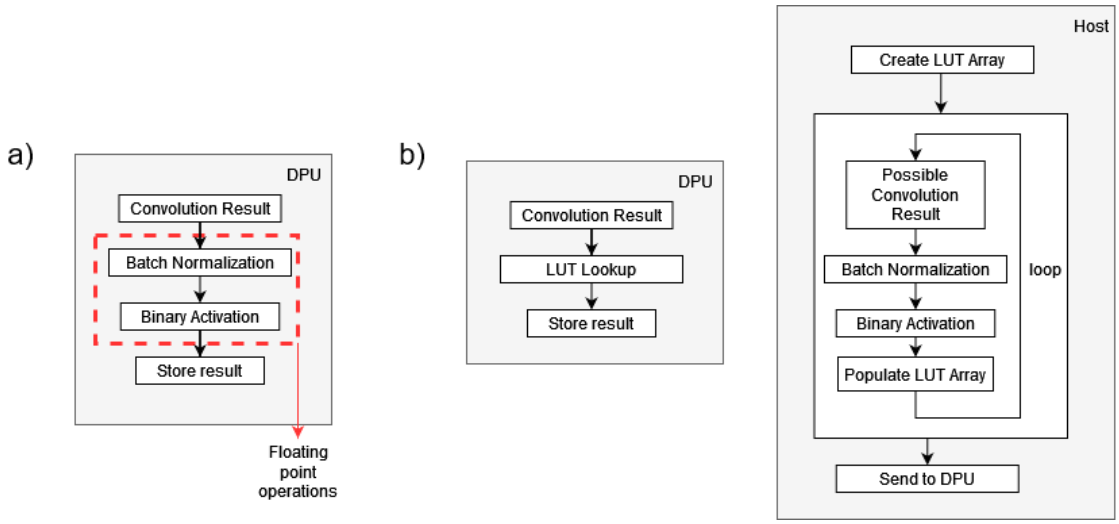

Figure 4.2: Look up table based update to the eBNN architecture to fit DPU usage. (a) shows the default eBNN model that contain floating point operations in the DPU and (b) shows the updated model where the host is responsible for the floating point operations instead

The eBNN implementation, while having quantized inputs, weights and temporaries, still contains floating point operations in its Batch Normalization (BN) and Binary Activation (BinAct) blocks within the Convolutonal-Pool block. As shown in Figre 4.2, in order to replace these floating point operations, a look up table based solution is created that contains the output of the BN-BinAct block as a function of the input of the block.

Figure 4.2(a) is the default flow of the eBNN program if placed inside the DPU. This flow happens for each convolutional-pool. In the eBNN model, the pool result is passed to a Batch Normalization function that uses given floating point values for a specific filter to normalize the value. Then the result is passed through a Binary Activation block to quantize the result. Both functions result in floating point operations for the DPU.

Algorithm 1 LUT Creation

```
1:  procedure LUT Creation
2:      x ← smallest conv. result
3:      y ← largest conv. result
4:      z ← number of filters
5:      LUT ← [ 0.0 ... 0.0 ]
              [  ⋮   ⋱   ⋮  ]  } |y−x|
              [ 0.0 ... 0.0 ]
                   |z|
6:      for i ← x ... y do
7:          for j ← 1 ... z - 1 do
8:              tmp ← i
9:              tmp ← tmp + W0[j]
10:             tmp ← tmp − W1[j]
11:             tmp ← tmp/ W2[j]
12:             tmp ← tmp ∗ W3[j]
13:             tmp ← tmp + W4[j]
14:             if tmp ≥ 0 then
15:                 res ← 1
16:             else
17:                 res ← 0
18:             LUT[(i-x)*z + y] ← res
```

Figure 4.2(b) is the novel methodology to circumvent the DPU's floating point issues. The BN-BinAct block is moved from the DPU to the host. The host runs all possible input values or all possible convolutional-pool results, through the BN-BinAct block and stores them into an 2D array LUT. Psuedocode for this LUT creation is shown in Algorithm 1 where lines 9 through 13 is the BN block and lines 14 through 17 is the BinAct block. The size of the array is dependant on the range

of the input values and the number of filters. The range of the input values are dependant on *only* the filter size. The host only needs access to the weights that the BN block uses, the filter size and the number of filters to perform the LUT creation. The LUT is created by a nested for loop where the outer loop goes through the range of the input values and the internal loop goes through the filter count. The row index of the 2D array is the input value and column index is the filter number. However, since the input value can be negative and the index cannot be, all values are stored in with an offset. This means that the largest negative value is the first index in the array and the largest possible value is the last index in the array. The Host sends this LUT to the DPU and the DPU copies it to from MRAM to WRAM before accessing it instead of running the two floating point blocks. The access of the array inside the DPU is the same as the access in the host.

4.1.4.1 Results

a)

```
__bootstrap (#occ =      54) =      0.137755
    __gesf2 (#occ =      71) =      0.181122
    __subsf3 (#occ =      80) =      0.204082
  __mulXf3__ (#occ =     110) =      0.280612
    __mulsf3 (#occ =     131) =      0.334184
 batch_norm (#occ =     274) =      0.698980
    __ltsf2 (#occ =     759) =      1.936224
    __divsf3 (#occ =     845) =      2.155612
   conv_idx (#occ =     873) =      2.227041
__floatsisf (#occ =     999) =      2.548469
    __addsf3 (#occ =    1129) =      2.880102
    __mulsi3 (#occ =    3325) =      8.482143
       bconv (#occ =    3501) =      8.931122
    __muldi3 (#occ =    5788) =     14.765306
```

b)

```
__bootstrap (#occ =      44) =      0.173283
       bconv (#occ =    1812) =      7.136106
    __mulsi3 (#occ =    2921) =     11.503623
    bdot_3d (#occ =   20613) =     81.179112
```

Figure 4.3: Comparison of the number of floating point subroutines in the same program (a) without and (b) with the LUT based architecture. The number of floating point subroutine calls are reduced from 11+ subroutines, in (a), to 2 subroutines in (b)

The removal of the BN-BinAct blocks results in the reduction of high-precision subroutines, discussed in 3.3, from the eBNN application. Figure 4.3 shows the effect of the removal. The number of floating point subroutine calls are reduced to where there is only the _mulsi3 subroutine is left. This routine could not be removed because it is tied to a dependent part of the program.

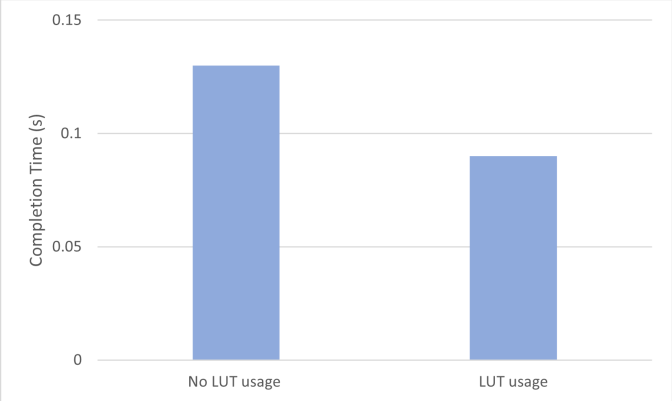

Figure 4.4: Completion time performance comparison of the same eBNN application running 16 images with and without the LUT architecture

Figure 4.4 shows the effect of this new model on the overall timing for 16 images. The new model results in a 1.4x speedup compared to the floating-point inclusive model.

4.2 Implementation of YOLOv3

4.2.1 YOLOv3 Architecture

YOLOv3 [4] is a state of the art object oriented detection CNN with outstanding accuracy and powerful realtime capabilities. YOLOv3 features the Darknet-53 network architecture, characterized by a feature extraction block followed by a feature detector block. As suggested by the name Darknet-53, both of these blocks contain fifty-three convolutional layers. These convolutional layers are heavily data-centric in nature and therefore can benefit from data-centric acceleration in the PIM environment. Since the UPMEM DPUs only supports limited fixed-point/floating-point precision computations, a quantized version of YOLOv3 for this implementation is chosen.

4.2.2 Dataset Specification

The implemented YOLOv3 input dataset is a standard example image with a size of 416x416 pixels. The most common image used in this work is the picture of a dog seen in Figure 4.5.

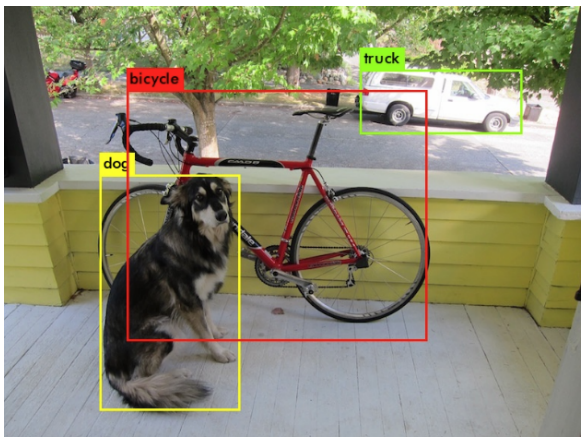

Figure 4.5: Example sample YOLOv3 image, 416x416 [4]. Classification boxes are not in input but are placed as a result of network completion

4.2.3 Mapping of Convolution to DPU

The General Matrix Multiply (GEMM) function is leveraged to implement convolutions within the DPUs. The GEMM function performs convolutions of the input and the weight matrices via a triple nested for-loop. Since quantization/de-quantization is not supported by the DPUs, the GEMM functions are only delegated to the DPUs instead of mapping the entire convolutional layers.

The nested for-loops of the GEMM function, shown in algorithm 2, that forms the convolutional blocks of the YOLOv3 implementation were unrolled. Under this scheme, the number of iterations of the external loop (algorithm line 3), *i.e.* the number of filters for the layer, are spread among a specific amount of DPUs. As demonstrated in Figure 4.6, the mapping is performed such that each DPU receives one row from the weights array (*i.e.* A) as well as the the entire input array (*i.e.* B).

Algorithm 2 GEMM Function

```
1: procedure GEMM Function(M, N, K, ALPHA, array A, array B, array C)
2:     ctmp ← array(4*N)
3:     for i ← 0 ... M-1 do
4:         for k ← 0 ... K-1 do
5:             APART ← ALPHA*A[i*K + k]
6:             for j ← 0 ... N-1 do
7:                 ctmp[j] ← APART*B[k*N + j] + ctmp[j]
8:         for j ← 0 ... N-1 do
9:             C[i*N+j] ← absolutemax(ctmp[j] / 32, 32767)
10:            ctmp[j] ← 0
```

The output are returned to one row of the output array (*i.e.* C). Since, each DPUs works on one row of A and C at a time, the number of DPUs required for executing a YOLOv3 layer is also the number of rows in A and C (M). As shown in Figure 4.6, this is also the number of filters for that layer. Internal parallelization for the DPU

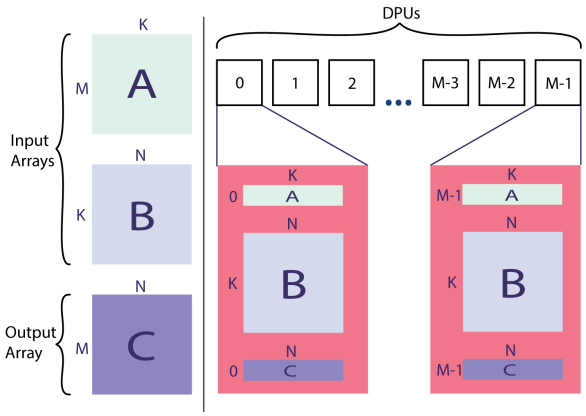

Figure 4.6: Distribution of the YOLOv3 GEMM function matrices A (*i.e* Input), B (*i.e* Weights), C (*i.e* Output) in available DPUs in the UPMEM System. The portion of the matrices that each DPU recieves is one row of A, the entirety of B and one row of C. The row index is dependant on the DPU index in the system.

is done again with tasklets, as introduced previously in section 2.1.2. For the GEMM function, there are dependencies within the second-most outer loop that force the parallelization to be done with the inner-most loop. Each tasklet is responsible for one column index or algorithm variable j of the input and output arrays (*i.e* B and C) and subsequent multiples of that column index of the N sized row.

The host is responsible for dividing up the arrays based on the mapping shown in Figure 4.6 and sending the data to the DPUs. Once the DPUs finished the convolution, the host would read each output row into the memory location of the layer's output.

4.3 Results

This section covers the performance of the UPMEM system running the eBNN and YOLOv3 applications. The section is divided into 4 parts: single DPU performance, multi-dpu performance, implementation takeaways and improvements.

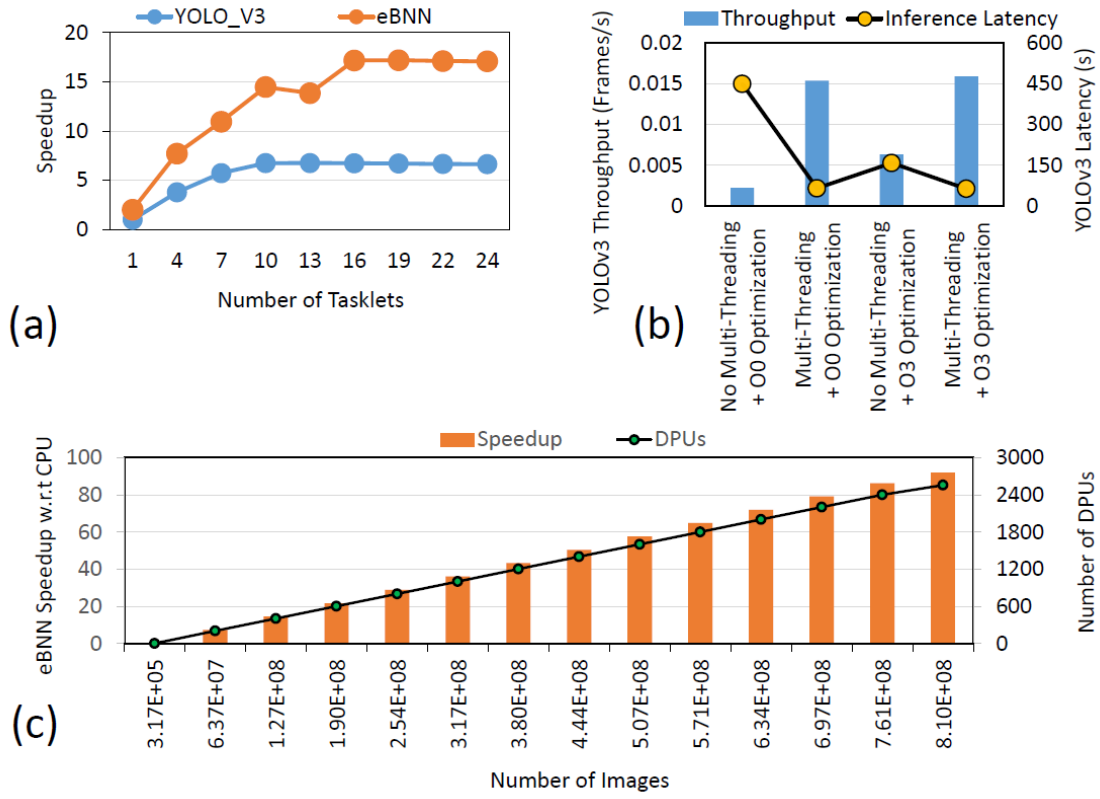

Figure 4.7: Evaluation of the implementation of eBNN and YOLOv3 CNNs on the UPMEM PIM system, including (a) speed-up achieved from multi-threading of execution with respect to no threading in each DPU for both eBNN and YOLOv3, (b) YOLOv3 performance for various combinations of multi-threading and compiler-level optimizations, and (c) speed-up of eBNN inference performance with respect to the CPU (*i.e.* Intel Xeon) for different degrees of operational parallelism.

4.3.1 Single DPU Performance

Each UPMEM DPU is capable of processing multiple concurrent threads or Tasklets on the same processing pipeline. In the context of CNN implementation, each thread

or tasklet may represent one single frame of inference (*i.e.* image) or a portion of a single frame of inference. For eBNN implementation, because each image is so small, multiple images are assigned to each DPU to be processed concurrently. For YOLOv3 a single image is distributed across multiple DPUs because of the difficulty of fitting one image into a DPU. In either case, thread/tasklet concurrency results in higher parallel performance and therefore increased processing throughput. Figure 4.7(a) demonstrates the performance speedup achieved by adopting thread level parallelism compared to the single thread performance for both eBNN and YOLOv3 implementations. It can be observed that the DPU offers increasing speedup for higher number of parallel tasklets, which saturates at around 16 tasklets for eBNN and 11 tasklets for YOLOv3, which aligns with DPUs behaviour observed by the prior works [30]. The reason for saturation at 11 tasklets is because there are 11 stages in the pipeline for a thread to use. At 11 tasklets/threads the pipeline is filled. eBNN does show a drop at 11 tasklets but the speedup increases again because the number of threads match the number of images available to be worked on.

Figure 4.7(b) which depicts the performance of the YOLOv3 for two different optimization criteria. These two criteria allows choices between multi-threading and compiler optimization. The relatively poorest performance is obtained for a combination of no compiler optimization (*i.e.* Optimization Mode O0) and no multi-threading of execution. On the other end of the spectrum, maximum optimization (*i.e.* Optimization Mode O3) is utilized, accompanied by multi-threading of execution, which can be observed to result in the relatively maximum increase in throughput and minimization of execution latency.

With threading and optimization used, the lowest possible single image latency for YOLOv3, using the mapping discussed in section 4.2.3, is found to be 65 seconds. Each layer averages about .9 seconds with a maximum of 6 seconds in one layer. The single image latency for eBNN running on a single DPU is found to be .00148 seconds.

4.3.2 Multi-DPU Performance

Figure 4.7(c) shows the comparison of a fully parallel DPU system and a Intel Xeon CPU running the eBNN application. The parallel nature of the DPUs allows for a linear increase in performance gain for increasingly large workloads. Each DPU is able to run 316800 images with 28×28 dimensions. With a total of 2560 DPUs working in parallel, this essentially means that up to 316800×2560 images can be processed for the latency of a single inference. When compared to a single Intel Xeon CPU, a linear increase speedup of the UPMEM system is observed with respect to the CPU as more DPUs are utilized. The maximum speedup is achieved when the maximum number of DPUs in the UPMEM system is used.

4.3.3 CNN Implementation Takeaways

Based on Figure 4.4 and Section 3.3, the best performance is achieved when the program contains minimal high precision computations. CNN specific alterations like the one done in Section 4.1.4 can be done to mitigate functions needed to be ran inside DPUs. A CNN wide mitigation is to use quantized versions of the CNNs.

Based on Figure 4.7(b), the lowest latency is achieved with multi-threading per DPUs and the highest compiler optimization. The biggest jump is seen when multi-threading is used but using compiler optimization helps as well. Thus, it is recommended that CNN applications use the highest multi-threading and compiler optimization setting possible.

The YOLOv3 implementation suffered in performance because it is difficult to alter the program to use mostly WRAM accesses. This is because there is a large internal buffer which would conflict with having multiple threads and leave almost no space for any other temporary storage. With space difficulties, the program had almost all of its memory accesses go to MRAM. This coupled with the MRAM and WRAM access statistics given in Section 3.2.1, is the main reason why the YOLOv3

implementation did relatively poorly compared to the eBNN implementation that had more WRAM accesses than MRAM accesses. Thus, it is recommended that CNN applications be carefully programmed in the DPU to increase the number of WRAM accesses vs. MRAM ones.

4.3.4 Improvements

After implementation and analysis of running CNNs in the UPMEM system, it is observed that there can be a few improvements to the UPMEM system to help support CNNs better.

Since threading internally in the DPUs is paramount for performance, it can be understood that all CNNs implementations should include DPU threading. As normal, every thread in the DPU system has its own stack. With a WRAM size of 64 KB, the maximum stack size for a DPU program using 11 threads is 5.8 KB. Modern C programmed CNN applications do not contain any internal buffers this small. For the implemented quantized YOLOv3 case, using 11 threads, the internal buffer can reach up to 160 KB. With the UPMEM system following a traditional RISC architecture, supporting modern C based applications, there needs to be an understanding that C based programmed CNNs contain various buffers that are frequently accessed but are larger than 5.8 KB. Thus, an improvement to the system could be to increase the WRAM size to a greater value so as to fit these necessary internal buffers. If this is not doable, then the penalty for accessing MRAM needs to be decreased by a greater extent. This could mean using a different transfer system than DMA.

UPMEM had initially stated in their whitepaper that the DPU frequency would be 600 MHz [1]. Currently, it is at 350 MHz. An increase in DPU frequency would help boost single DPU performance.

Chapter 5

Modeling of Various PIMs

Currently PIM designs exist in varying forms of maturity. Furthermore, these designs vary in granularity in terms of computation unit size. Thus, for any comparison between these PIM designs there must be a general baseline that covers all PIM. This baseline can exist in the form of a performance model. This chapter covers the creation of a model that describes the performance of PIMs based on several architectural parameters such as dataflow and application operand size. This model is then used to estimate several other PIMs in order to allow performative comparison.

5.1 Generic Model

Processing-in-Memory can be observed to fall into some patterns in terms of design and architecture. These patterns exist for PIMs in a range of fundamental design ideas. This spectrum of PIMs, as shown in figure 5.1, is categorized into three categories: bitwise, LUT and pipelined-CPU, where bitwise is fine-grained and pipelined-

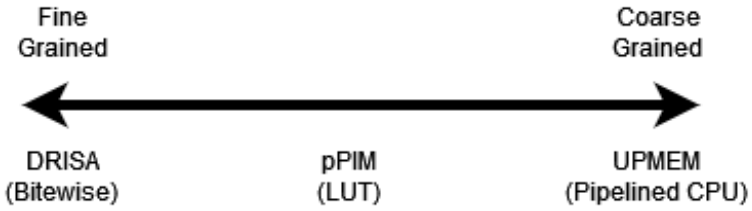

Figure 5.1: PIM chip design granularity spectrum

CPU is coarse-grained.

DRISA can be viewed from a bitwise perspective because it does not create DRAM process logic that incurs large overhead but instead uses simple and serially-computing bitline logic [12]. The bitline logic is mostly Boolean (*i.e* NOR gates) and more complex logic is ran by serially executing multiple Boolean logic gates. These gates are directly connected to the subarray rows and thus reading/writing occurs to and from rows.

pPIM [14] represents the LUT usage case because every processing element or "cluster" in pPIM consists of "cores". These "cores" are just LUTs that, based on 2 4-bit inputs, return a 8-bit output result that is pre-programmed in each latch/register file [16]. Unlike DRISA, pPIM does not use boolean logic gate combination to create outputs but instead chooses a result from a pre-generated list of possible outputs. Thus, these "cores" are of a coarser granularity than DRISA's bitwise logic.

UPMEM [1, 2] is the only PIM at the writing of this work that implements a pipelined RISC based processor as its processing element, as discussed in previous chapters. UPMEM's processing element is coarser in granularity than that of pPIM's LUT implementation because a pipeline is made output multiple stages where each stage could host a "cluster".

All three designs share similarities. Currently, these similarities have not been captured into a model that can describe the performance of these PIMs in a general sense. This chapter focuses on the creation of a model that, given a few architectural parameters, produce an estimated value for the latency of MAC operations in any PIM design. The multiply and accumlate operations are chosen as the default fundamental operation for this work.

The basis for this model is Figure 5.2. The model describes what must occur for an operation, such as a multiplication operation, to execute in the PIMs. There must be time spent on some sort of memory transfer from a location away from the processing

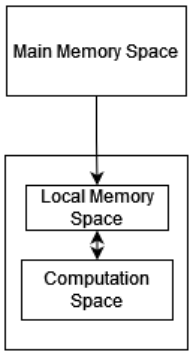

Figure 5.2: Generic Model for PIM Designs

elements in the PIM to a local memory space before computation can occur on the data. Then there must be time spent doing the actual computation. This time spent is represented in equation 5.1 where T_{tot} represents the total time for any number of operations and T_{mem} is the time spent transferring necessary data from a external location to a local location before being processed on and T_{comp} represents the time taken to finish processing computations. This model assumes an unoptomized, worst case PIM solution that does not contain any overlap between memory transfer time and computation time. Furthermore, this model also does not account for direct connections between the main memory space and computation space.

$$T_{tot} = T_{mem} + T_{comp} \tag{5.1}$$

5.2 Computation Model

$$T_{comp} = \frac{C_{comp}}{Freq} \tag{5.2}$$

The computational model breaks down the T_{comp} portion of equation 5.1. T_{comp} is equivalent to the cycles necessary for computation, C_{comp} divided by the frequency of the PIM as depicted in equation 5.2.

$$C_{comp} = C_{op} * \lceil \frac{TOPs}{PEs} \rceil \tag{5.3}$$

C_{comp} is the number of total cycles for all operations running through the system. As shown in equation 5.3, C_{comp} is broken down into two parts: the number of cycles for one operation in the system, C_{op}, and a parallelization factor. This parallelization factor is denoted by the ceil function,$\lceil\ \rceil$, of dividing the total number of operations, $TOPs$, by the number of processing elements available, PEs. The division returns the number of times all PEs must execute serially given each PE working on one operation. This also covers the fact that all PEs work in parallel and take the same number of cycles. The ceil function exists because uneven division requires an extra wave of PEs running serially.

$$C_{op} = f(x) * C_{BB} * D_p \tag{5.4}$$

C_{op}, or the cycles for one operation in a PIM, as shown in equation 5.4, is dependent on 3 things: the number of pipeline stages in the architecture, D_p; the number of cycles for a building block of the architecture to complete its execution, C_{BB}; and a scale function that depending on the input operand size, x, increases the number of cycles necessary for the operation, $f(x)$. D_p exists because if the architecture has multiple stages to complete one operation, then the latency of one operation is multiplied by the number of stages. This model does not account for any optimization done within the pipeline for simplicity. C_{BB} exists because at the most fundamental level, the number of cycles needed for an operation is dependent on the number of cycles for a building block *i.e* logic gate, LUT or ALU to complete. $f(x)$ exists because in all PIMs studied, the number of cycles to complete an operation changes with the size of the operands. This is because a building block in any design can only support a fixed number of input operand length and anything bigger will result in more building blocks used. Thus, $f(x)$ is a parameter that is dependant on the dataflow/architecture of a PIM system.

$$C_{op} = \begin{cases} g(x) * C_{BB} * D_p, x < n \\ f(x) * C_{BB} * D_p, x \geq n \end{cases} \tag{5.5}$$

Variations of equation 5.4 can exist and thus, more generic equations are created to account for them. For example, an aspect of PIMs noted is that sometimes the scale function is not the same for all input operand lengths. To take account for this, equation 5.5 divides the C_{op} calculation into a piecewise function. Until a certain number of bits, n, there is single function but for any bits higher the scale function changes because a different design is used.

$$C_{op} = \begin{cases} \left\{ \sum_{k=0}^{Z} g_k(x) * C_{BB_k} \right\} * D_p, x < n \\ \left\{ \sum_{k=0}^{Z} f_k(x) * C_{BB_k} \right\} * D_p, x \geq n \end{cases} \tag{5.6}$$

Another aspect noted is that there might not always be one building block used to execute an operation. This is reflected in equation 5.6 where the final number of cycles is the result of a combination of different building blocks that each have their own scaling up functions. Here the number of different building blocks is Z with k used to index each building block in the summation. Equation 5.6 collapses into equation 5.5 with only 1 building block and collapses into equation 5.4 with only one scale function.

5.2.1 DRISA

$$C_{\text{mult}} = \begin{cases} g(x) * C_{xnor} * 1, x < 4 \\ f_0(x) * C_{BShift} + f_1(x) * C_{\text{sel}} + f_2(x) * C_{CSA} + \log(x) * C_{FA} * 1, x \geq 4 \end{cases} \tag{5.7}$$

DRISA's architecture best fits equation 5.6 because it is bitwise and thus uses logic

gates as its building block. As shown in equation 5.7, the model requires 2 scaling functions because DRISA uses XNOR gates for operations on bits less than 4. For operands of 4 bits and higher shift, select and adder blocks are used instead. Each block has a different scale function as denoted by $f_{0-2}(x)$. $f_{0-2}(x)$ and $g(x)$ are not known but the function for the full adder, C_{FA}, is known to be $log(x)$. The addition of all blocks fits the DRISA architecture because each block is executed serially with other blocks. D_p is shown to be 1 because there are no pipeline stages in DRISA.

5.2.2 UPMEM

$$C_{\text{mult}} = \begin{cases} g(x) * 1 * 11, x < 16 \\ f(x) * 1 * 11, x \geq 16 \end{cases} \tag{5.8}$$

The UPMEM system fits best with equation 5.5. This is because, as noted in section 3.3, subroutines are called for multiplication operations using operands lengths of 16 bits and greater. Note, compiler optimized programs change the n value from 16 to 32. UPMEM's DPU does contain a pipeline and thus D_p here is 11. C_{BB} is shown to be 1 because it is modeled that a building block is an instruction and a instruction takes 1 cycle per stage. Here $g(x)$ refers to the number of instructions necessary for a multiplication instruction given bit sizes less than 32. Here $f(x)$ refers to the number of instructions given a subroutine call. While specific values for $f(x)$ is not known, it is known that $g(4)$ and $g(8)$ is 4 for a multiplication operation [31].

5.2.3 pPIM

$$C_{mult} = f(x) * 1 * 1 \tag{5.9}$$

pPIM's architecture best fits equation 5.4 because there is only one scale function for the LUT method and it only uses LUT's for any operation. The resulting equation is shown in 5.9. D_p is shown to be 1 because it is known that there are no pipeline

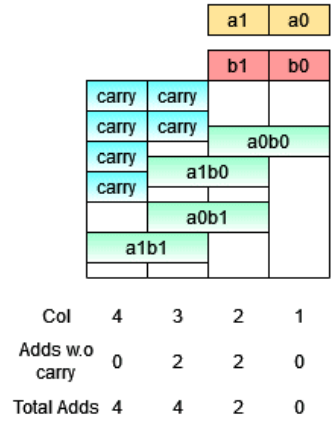

Figure 5.3: Worst case block by block LUT multiplication for 8 bit operands in the pPIM architecture. The orange and red blocks are operands and are divided into 4 bit blocks. Each multiplication of 4 bit blocks result is shown in green blocks. The addition of each column for the green blocks is shown in blue carry blocks

stages in pPIM. C_{BB} is shown to be 1 because it known that a single LUT takes one cycle to execute. $f(x)$ is not known for pPIM because the dataflow for operations is handcrafted and optimized. Thus, how the system performs for large operand sizes is not known. This work provides an estimation of $f(x)$ for a worst case implementation of multiplication where the multiplication and internal carry addition is serially computed.

The basis for the estimation function is shown in figure 5.3 where 2 8 bit operands are being multiplied. The first operand is broken down into a1 and a0 which are 4 bit nibbles and the second operand is broken down into b1 and b0, similarly. Each nibble of a operand multiply with nibbles of the other operand resulting in 4 partial multiplication results. The addition of these partial terms happen with a specific configuration in columns as shown in the figure. Starting from the right each column's partial term carry out is passed as another addition to the column to the left. Thus as shown, the total number of adds increases as you go from the right to the left in a recursive manner and is only dependent on the number of adds without carry.

The number of adds without carry follows a pattern for this sort of multiplication. The pattern is shown in figure 5.4. This pattern increases by 2 until a halfway point

$$8 \text{ bit} \rightarrow 0, 2, 2, 0$$

$$16 \text{ bit} \rightarrow 0, 2, 4, 6, 6, 4, 2, 0$$

$$32 \text{ bit} \rightarrow 0, 2, 4, 6, 8, 10, 12, 14, 14, 12, 10, 8, 6, 4, 2, 0$$

Figure 5.4: Pattern for number of internal adds without carry for pPIM's LUT based multiplication with different operand sizes

where it then falls back down by two.

Algorithm 3 pPIM Multiplication Scale Estimation Function

```
1:  Global total
2:  procedure ESTIMATION FUNCTION(n,k,temp)
3:      if n = 0 then
4:          return 0
5:      else if n > k/2 then
6:          g ← (-2*n)+(k*2)
7:      else if n ≤ k/2 then
8:          g ← (2*n)-2
9:      temp ← temp+g
10:     total ← total+temp
11:     return ESTIMATION FUNCTION(n − 1, k, temp)
```

The pattern in figure 5.4 plus the recursive nature of getting the total additions can be used to generate an function that returns the number of internal additions needed for an worst case-multiplication LUT design. This function is shown in algorithm 3. The pattern is captured in lines 5 to 8, the keeping track of the rolling addition is kept track using lines 9 and 10, and recursively going from the left most column of figure 5.3 to the right most is done using lines 3,4, and 11. The summation of the number of additions returned from this algorithm plus the number of 4-bit multiplications done is the total number of times an LUT has to execute to accomplish a multiplication. Given an LUT takes one cycle for 4-bit operations, then this value is also the number of cycles necessary for multiplication.

Table 5.1: Example usage of computational model for different PIMs. The operand size is 8-bit, the application is AlexNet

		pPIM	**DRISA**	**UPMEM**
1	D_p	1	1	11
2	C_{BB}	1	1	1
3	x	8	8	8
4	Accum.-$f(x)$	2	11	4
5	Mult.-$f(x)$	6	200	4
6	C_{op}	8	211	88
7	PEs	256	32768	2560
8	Freq. (Hz)	1.25E+09	1.19E+08	3.50E+08
9	TOPs (Alexnet)	2.59E+09	2.59E+09	2.59E+09
10	C_{comp} (1 MAC)	8	211	88
11	T_{comp} (1 MAC) (s)	6.40E-09	1.69E-06	2.51E-07
12	C_{comp} (TOPs)	8.0938E+07	1.6678E+07	8.9031E+07
13	T_{comp} (TOPs) (s)	6.48E-02	1.40E-01	2.54E-01
14	Literature Alexnet Latency (s)	6.48E-02	1.40E-01	8.79E-01

5.2.4 Results

The usage of the computational model starting from equations 5.4-5.6 and building up to 5.2 is shown in Table 5.1. Characteristics of all three PIMs are plugged into the equation. The parameters of equation 5.4 are rows 1-5. The result of equation 5.4 is row 6. The parameters of equation 5.3 are rows 7-9. The result of equation 5.3 for one MAC operation is row 10-11. The result of equation 5.3 for AlexNet's MAC operation count is row 12-13. Row 14 is the value gained from using literature provided MAC latency to calculate the AlexNet latency. Since, the fundamental parameters in rows 1-5 and 7-8 are taken from literature as well, the MAC latencies match up and thus the AlexNet latencies match up as well. Data regarding actual AlexNet implementations in these PIMs is not used because data is not available for all PIMs. The UPMEM latency is different because the non-model value uses cycles that include profiling instructions as well.

To use the model described by equation 5.3 for different operand lengths, the number of cycles for a multiplication operation needs to be generated. Table 5.2

Table 5.2: Number of cycles, C_{op} for multiplication given operand size. * denote estimated values

	pPIM	DRISA	UPMEM
4 bit Multiplication	1	110	44
8 bit Multiplication	6	200	44
16 bit Multiplication	124*	380	370*
32 bit Multiplication	1016*	740*	570*

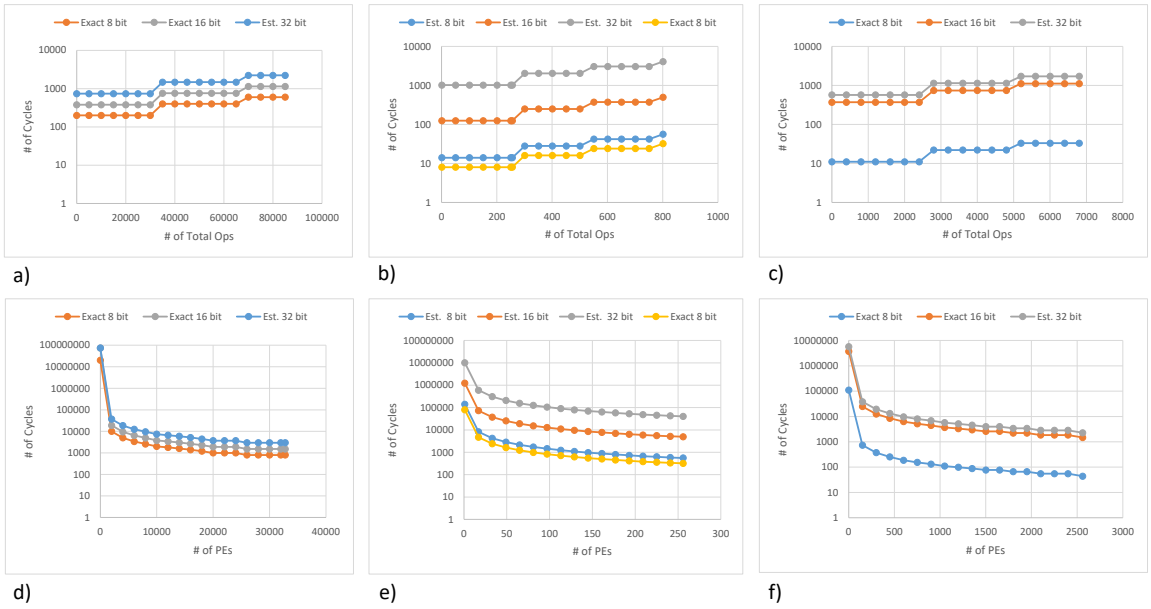

Figure 5.5: Effect of Parameter changes on Equation 5.3. (a) - (c) show cycle effect as total operations increase while number of PEs stays constant, (d) - (f) show cycle effect as number of PEs increase while total operations stays constant. (a) and (d) is DRISA, (b) and (e) is pPIM, (c) and (f) is UPMEM. For (a) - (c) the number of PEs is 32768, 256 and 2560 respectively. For (d) - (f) the number of total operations is 10000, 100000 and 100000 respectively

shows exact and estimated values for the number of cycles or C_{op} for a multiplication operation. Exact values for pPIM and DRISA is taken from their respective literature. The estimated value for DRISA is taken from curve fitting the non-estimated values. Estimated values for pPIM is taken from the result of algorithm 3. The estimated value for UPMEM is taken from estimating the number of cycles a multiplication subroutine takes for the specific operand size[28]. The exact values for UPMEM is taken from counting the number of instructions when observing assembly output of a C-based multiplication program.

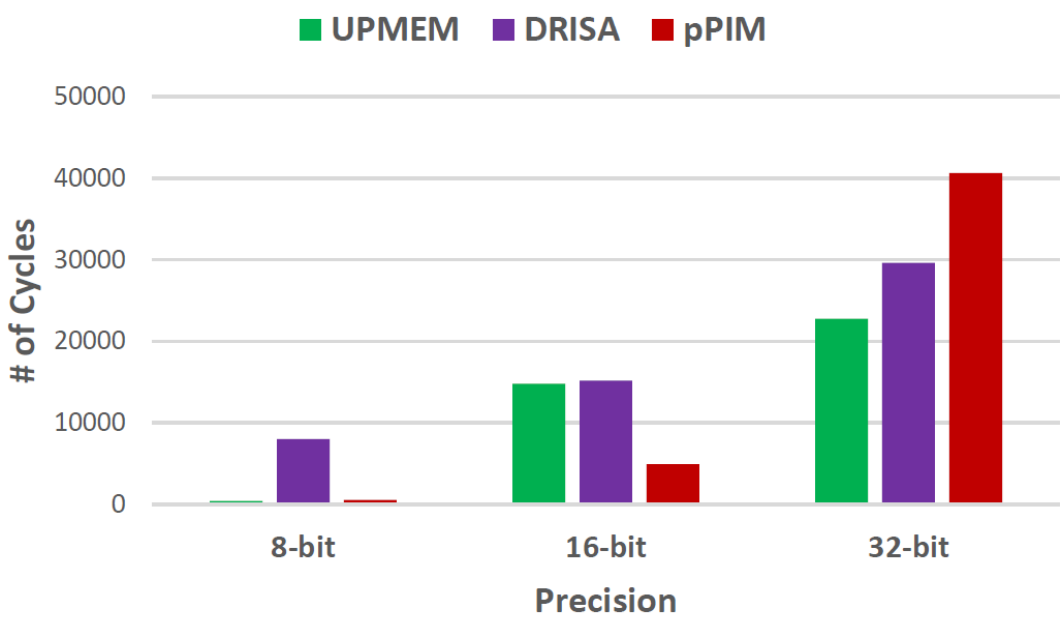

Figure 5.6: Performance comparison of DRISA, pPIM and UPMEM on a multiplication operation using Equation 5.3. The number of PEs is 2560 for all, the TOPs is 100000 for all

Using the C_{op}'s in table 5.2, the other parameters of equation 5.3 are varied for different operand sizes as shown in Figure 5.5. Graphs (a) - (c) show cycle effect as total operations increase while number of PEs stays constant, (d) - (f) shows cycle effect as number of PEs increase while total operations stays constant. (a) and (d) is DRISA, (b) and (e) is pPIM, (c) and (f) is UPMEM. For (a) - (c) the number of PEs is 32768, 256 and 2560 respectively. For (d) - (f) the number of total operations is 10000, 100000 and 100000 respectively. The trend observed for the total operations sweep is a step function, due to the ceil function in equation 5.3. The trend observed for the PE sweep, is a steep drop the moment parallelization is introduced and then a logarithmic negative slope as more parallelization is added. pPIM's graphs include the estimated data using the model described in section 5.2.3 as well as the exact value taken from literature [16]. DRISA's and pPIM have even seperation between precision cycles. UPMEM's does not because of the introduction of subroutines.

Figure 5.6 compares how each PIM architectures does against each other for a

multiplication operation at different operand sizes. The number of PEs and the total number of operations is constant at 2560 and 100000, respectively. This figure, thus, shows how each PIM scales using the model proposed. The number of cycles for each precision is taken from table 5.2. pPIM's cycle numbers are all estimated values while the other are a combination as discussed prior. Figure 5.6 argues that pPIM is best for both 8-bit and 16-bit multiplication but UPMEM does the best for 32-bit.

5.3 Memory Model

$$T_{mem} = T_{transfer} * \lceil \frac{TOPs}{PEs * \left(\frac{size_{buf}}{2 * Len_{op}} \right)} \rceil \tag{5.10}$$

Equation 5.10 shows the memory model for the time spent accessing an external memory space to get data ready for computation. The equation is broken down into two parts: the time for a memory transfer between an external memory space, $T_{transfer}$ and a local space and the number of transfers necessary.

The idea is that the number of times that the memory space is accessed is dependent on the number of total operations necessary to be processed, $TOPs$, divided by the number of operations that can be stored locally and processed on. This requires the processing to start once the maximum number of operations is stored locally. For the entire system, the number of operations that can be stored locally is dependent on number of operations that can be stored in a local buffer times the number of local buffers available. This model assumes that there is only one local buffer per processing element and thus the number of local buffers is the number of PEs. Finally, the number of operations that can be stored locally is dependant on the size of the local buffer in bits, $size_{buf}$ divided by the number of bits per operand, Len_{op} times 2. The times 2 is placed there because each operation requires 2 operands.

The memory transfer time, $T_{transfer}$, is not broken down further because PIMs usually use different DRAM based features to execute the transfer. For example,

DRISA uses the Rowclone [32] methodology for transfering data between subarrays that is not common in other PIMs.

5.3.1 Results

Table 5.3: Memory model analysis with parameters plugged in from equation 5.10

	pPIM	**DRISA**	**UPMEM**
$T_{transfer}$ (s)	6.70E-09	9.00E-08	9.60E-05
TOPs (Alexnet)	2.59E+09	2.59E+09	2.59E+09
PEs	256	32768	2560
$size_{buf}$ (bits)	256	1048576	512000
Len_{op} (bits)	8	8	8
OPs per PE	16	65536	32000
Local Ops	4096	2147483648	81920000
T_{mem} (s)	4.24E-03	1.80E-07	3.07E-03

Table 5.3 captures the analysis of the memory model being used to estimate the time necessary for memory transfer attempts for pPIM, DRISA and UPMEM running an 8-bit AlexNet program. pPIM's $T_{transfer}$ is the tRCD time to copy data from a subarray to a local buffer, DRISA's is the time for a rowclone between subarrays, and UPMEM's is the time for DMA transfes between MRAM and WRAM. The $TOPs$ value is the number of multiply and accumalate instructions for the AlexNet program. DRISA's local buffer is modeled here to be the subarray area that a PE has access to in its design. The size of this buffer $size_{buf}$ is calculated by multiplying the number of rows, columns and lane width in the subarray. UPMEM's local buffer is WRAM, with $size_{buf}$ being 64KB.

Using the T_{mem} taken from table 5.3 and T_{comp} from table 5.1 in equation 5.1 results in the total time for Alexnet. The total time for pPIM is 6.90E-02 s; the total time for DRISA is 1.40E-01 s; and the total time for UPMEM is 2.57E-01 s.

5.4 Model Usage

In this section, the performance of several DRAM-based AI accelerators (*i.e.* PIMs) is compared in terms of CNN (*i.e.* eBNN and YOLOv3) inference performances. The PIMs in discussion feature a wide range of technological maturity. For example, the UPMEM PIM system is fully-developed and a commercially available for use while PIMs such as DRISA [12] and SCOPE [13] are physically prototyped. On the other hand, LACC [17] and pPIM [16, 14] feature synthesized implementations. Therefore, a combination of both in-device implementation and analytical benchmarking is used for comparative evaluations of these devices. The CNN algorithms in discussion are implemented on the UPMEM PIM servers, as discussed previously in Section 4. On the other hand, the other PIM devices are evaluated using the model described in section 5.2 using performance parameters, like total PEs and cycles per MAC operation, of the respective devices reported in the literature [17, 18] on the same algorithms. The fundamental operation measured for all PIM's is the MAC operation.

5.4.1 Results

Table 5.4 shows the performance parameters as well as performance benchmarks of the aforementioned PIM devices for eBNN and YOLOv3 inferences in terms of frames of inferences per second for unit power consumption and frames of inferences per second for unit processing area. With only 120mW power consumption per DPU, UPMEM PIM is the most low-power device (<1W/chip) among the PIMs in comparison. This makes it highly suitable for the adoption in the commercially available DRAM architecture which features a low power rating. At the same time, the UPMEM chips are relatively small and therefore contribute a relatively lower area overhead (*i.e.* 45 %) to the DRAM chip. In comparison, PIMs such as DRISA and SCOPE re-purpose the whole DRAM chip organization for in-memory computing. In fact, SCOPE features

Table 5.4: Hardware Performance Parameters and Performance Benchmarking of various PIM architectures for eBNN and YOLOv3 inferences with 8-bit fixed-point precision

	UPMEM [33]	**pPIM** [14]	**DRISA-3T1C** [12]	**DRISA-1T1C-NOR** [12]	**SCOPE-Vanilla** [13]	**SCOPE-H2d** [13]	**LACC** [17]
Power Consumption/Chip (W)	0.96	3.5	98	98	176.4	176.4	5.3
Area/Chip (mm^2)	30	25.75	65.2	65.2	273	273	54.8
eBNN Latency/Frame (s)	1.48E-03	3.80E-07	8.21E-07	1.96E-06	1.30E-08	4.64E-08	2.14E-07
eBNN Throughput/Power (Frames/s-W)	5.63E+03	7.52E+05	1.24E+04	5.21E+03	4.36E+05	1.22E+05	8.82E+05
eBNN Throughput/Area (Frames/s-mm^s)	1.80E+02	1.02E+05	1.87E+04	7.83E+03	2.82E+05	7.89E+04	8.53E+04
YOLOv3 Latency/Frame (s)	65	0.68	1.47	3.51	0.0233	0.0831	0.384
YOLOv3 Throughput/Power (Frames/s-W)	1.25E-04	4.20E-01	6.94E-03	2.91E-03	2.43E-01	6.82E-02	4.91E-01
YOLOv3 Throughput/Area (Frames/s-mm^s)	1.10E-05	5.71E-02	1.04E-02	4.37E-03	1.57E-01	4.41E-02	4.75E-02

a customized DRAM chip with roughly 4 × larger area than a standard DRAM chip.

Although the per chip area overhead is relatively smaller, the UPMEM DPUs themselves are large processing units. This is because, the DPU features a modified reduced instruction set processing architecture with multiple pipeline stages. Alongside, although the DRAM chips on which the UPMEM PIM system is implemented are in the 25nm technology node, the DPUs themselves are developed in the 65nm technology node. Thus, each chip while smaller can only hold 8 DPUs. As a result, the DPUs have a significantly higher footprint for an equivalent circuit overhead as other PIM architectures such as DRISA, LACC, SCOPE, pPIM which are developed in the range of 22-28nm technology node.

Figure 5.7 shows the graphical representation of this data as well. DRISA's frames/power and frames/area is the poorest out of the analytical models. pPIM and LAcc perform the best in terms of frames/power and SCOPE performs the best in terms of frames/area. The relative performances of all analytical models fit literature results, arguing for the validity of the proposed model. UPMEM's actual performance

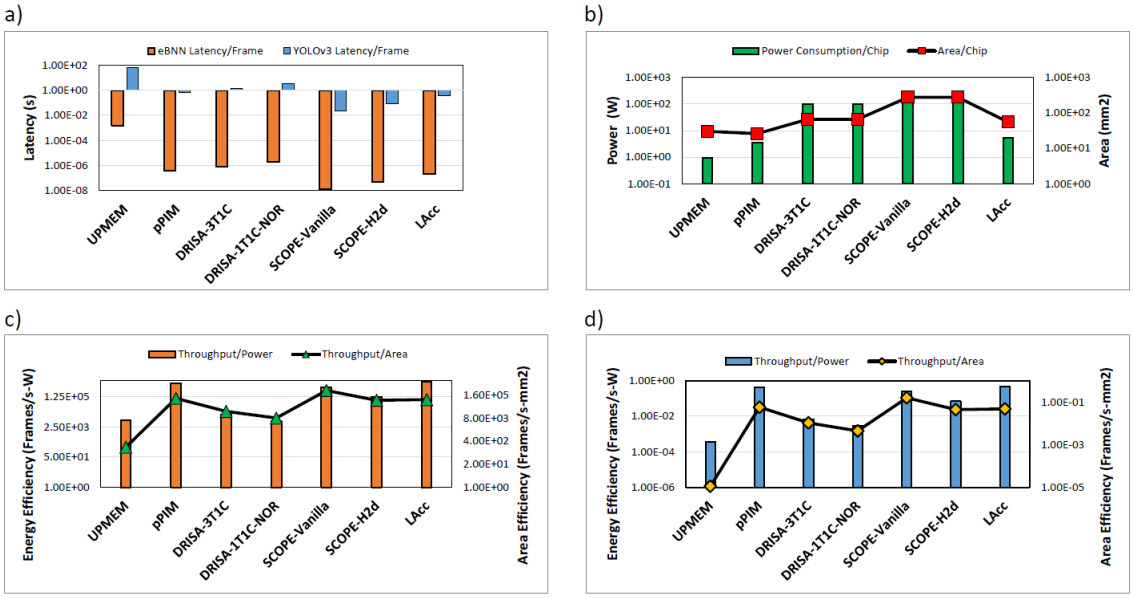

Figure 5.7: Graphical representation of Hardware Performance Parameters and Performance Benchmarking of various PIM architectures for eBNN and YOLOv3 inferences with 8-bit fixed-point precision. (a) shows latency values for both CNNs on PIMs, (b) shows the power/area for PIMs, (c) shows energy throughput and area throughput for PIMs running eBNN, and (d) shows energy throughput and area throughput for PIMs running YOLOv3

in comparison with other analytical PIMs is very poor. With one of the lowest area per chip and the lowest power per chip, the low frames/area or frames/power results stem from the poor latencies of the implementations. Given the most optimal mapping and programming of a CNN application on the UPMEM system, along with the increase in DPU frequency to initially stated values by UPMEM [1], the latencies might decrease enough to allow UPMEM to do better.

Chapter 6

Conclusion

In this work, successful mapping and implementation of Deep Neural Networks (DNN) is presented on the UPMEM PIM system. This work demonstrates the mapping of two Convolutional Neural Networks (CNN) with different sizes, depths, and complexities on the UPMEM PIM using respective mapping and optimization techniques. The results of this implementation show that removal of floating point operations, usage of internal threading and compiler optimization is crucial to getting comparable data. It is observed that its is essential to adopt a proper operation mapping scheme to obtain optimum performance and resource utilization from the UPMEM Processing Elements distributed across many DRAM chips. Furthermore, improvements are presented for the PIM revolving around the memory size and access time inside each DPU to help alleviate programming issues. Nevertheless, it is also observed that the performance speed-up with respect to traditional CPU-based system scales up linearly as more parallel Processing Elements (DPU) are incorporated in the UPMEM system.

This work also presents a model to estimate the performance of bitwise, LUT and pipelined-CPU PIM architectures given several design parameters such as dataflow scaling based on input operand size, number of processing elements (PEs), total operations etc. It is observed that as input precision increases, with number of PEs and total operations staying constant, bitwise and pipelined-CPU designs overtake LUT designs in performance marks. This model is used to generate estimated perfor-

mance marks of several other recently proposed PIM architectures in order to perform comparative evaluations with UPMEM's PIM. It is observed that although UPMEM PIM currently does not offer the highest degree of raw throughput, it is a relatively energy-efficient, low-power PIM solution for CNN acceleration applications.

6.1 Future Work

First, given more time, more work regarding YOLOv3 mapping would be done to find the most optimal implementation methodology. This mapping would try to squeeze as many YOLOv3 image inferences into a single DPU as possible in order to emulate the eBNN implementation multi-image per DPU method. Then the performance of this mapping would be compared to the current mapping to establish which mapping is better. Furthermore, alternative CNNs would be employed and analysed as well. The more CNNs are tested in UPMEM's system the more conclusions could be made in general for CNNs.

Currently, there is a large performance gap between the implementations of eBNN and YOLOv3. This originates from how easy it was to fit eBNN's image convolution within a DPU versus YOLOv3. eBNN's image sizes were so small, there was plenty of memory space within the DPUs. YOLOv3 contained large convolution buffers and internal buffers that made it difficult to the same. However, YOLOv3 is a very complex CNN. Future work can be done to find exact depth or size of a CNN that is best for UPMEM's system. This work can parametrically show when UPMEM's system starts losing performance and for what network size. CNNs from Alexnet to Resnet or choosing a CNN such as eBNN and going from small image sizes to larger sizes can determine how large of an image is supported as well.

It took a large amount of programming trial/error and question and answering between UPMEM to properly insert a CNN within the UPMEM system. Furthermore, the separation of the data-centric portion of the code from the application,

compilation and debugging of the DPU program, and sending of memory between the host and DPUs is all done manually. For an expert of this system, this might not take long but for a scientist or a novice, this may hinder their work. There needs to be a programming standard/methodology or tool that takes care of the programming side of using UPMEM's PIM system or any other future PIMs on the market. For example, the seperation of the data-centric portion of the code requires profiling. Which profiler to use and how to use it is very important to identify memory bound and acceleratable functions. The manual seperation, compilation and memory tansfer can be handled as well with tools. OPENCL, a parallel system standard, tackles heterogenous systems such as PIM-host systems and provides a framework to offload accelerable code. OPENCL can optimize the usage of a parallel system. Using this framework, mapping of images or convolution layers could be delegated autonomously to DPUs. Multiple images could be grouped and sent for accelerated inference in batches. This could be beneficial for systems with multiple ranks of DPUs. Furthermore, more complex synchronization could be done between the host and the DPUs. The usage of OPENCL or any other tool is necessary to be detailed in a future work.

6.2 Acknowledgment

This work was supported in part by the US National Science Foundation (NSF) CAREER Grant CNS-1553264. The author would like to thank UPMEM, a fabless semiconductor company, based in Grenoble, France for providing access to their product/service and technical support.

Bibliography

[1] UPMEM, "Upmem processing in-memory: Ultra-efficient acceleration for data-intensive applications (white paper)," 2020.

[2] R. J. Devaux F., "Memory circuit with integrated processor," 2019, u.S.Patent 10,324,870. [Online]. Available: https://uspto.report/patent/grant/10,324,870

[3] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs*, vol. 2, 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist

[4] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *CoRR*, vol. abs/1804.02767, 2018. [Online]. Available: http://arxiv.org/abs/1804.02767

[5] A. Nowatzyk, Fong Pong, and A. Saulsbury, "Missing the memory wall: The case for processor/memory integration," in *23rd Annual International Symposium on Computer Architecture (ISCA'96)*, May 1996, pp. 90–90.

[6] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally, "Scaling the power wall: A path to exascale," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014.

[7] "Cpu, gpu and mic hardware characteristics over time," April 2021. [Online]. Available: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

[8] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning, "In-memory intelligence," *IEEE Micro*, vol. 37, no. 4, pp. 30–38, 2017.

[9] H. S. Stone, "A logic-in-memory computer," *IEEE Transactions on Computers*, vol. C-19, no. 1, pp. 73–78, 1970.

[10] D. Patterson, K. Asanovic, A. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. Yelick, "Intelligent ram (iram): the industrial setting, applications, and architectures," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, 1997, pp. 2–7.

[11] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 336–348.

[12] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2017, pp. 288–301.

[13] S. Li, A. O. Glova, X. Hu, P. Gu, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Scope: A stochastic computing engine for dram-based in-situ accelerator," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 696–709.

[14] P. R. Sutradhar, M. Connolly, S. Bavikadi, S. M. Pudukotai Dinakarrao, M. A. Indovina, and A. Ganguly, "ppim: A programmable processor-in-memory architecture with precision-scaling for deep learning," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 118–121, 2020.

[15] P. R. Sutradhar, S. Bavikadi, M. Connolly, S. K. Prajapati, M. A. Indovina, S. M. Pudukotaidinakarrao, and A. Ganguly, "Look-up-table based processing-in-memoryarchitecture with programmable precision-scalingfor deep learning applications," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2021.

[16] M. Connolly, P. R. Sutradhar, M. Indovina, and A. Ganguly, "Flexible instruction set architecture for programmable look-up table based processing-in-memory," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, 2021, pp. 66–73.

[17] Q. Deng, Y. Zhang, M. Zhang, and J. Yang, "Lacc: Exploiting lookup table-based fast and accurate vector multiplication in dram-based cnn accelerator," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, June 2019.

[18] S. M. Shivanandamurthy, I. G. Thakkar, and S. A. Salehi, "Atria: A bit-parallel stochastic arithmetic based accelerator for in-dram cnn processing," in *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2021, pp. 200–205.

[19] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast bulk bitwise and and or in dram," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 127–131, July 2015.

[20] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "Computedram: In-memory compute using off-the-shelf drams," ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 100–113. [Online]. Available: https://doi.org/10.1145/3352460.3358260

[21] D. Lavenier, J.-F. Roy, and D. Furodet, "Dna mapping using processor-in-memory architecture," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2016, pp. 1429–1435.

[22] D. Lavenier, R. Cimadomo, and R. Jodin, "Variant calling parallelization on processor-in-memory architecture," in *2020 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2020, pp. 204–207.

[23] B. McDanel, S. Teerapittayanon, and H. T. Kung, "Embedded binarized neural networks," *CoRR*, vol. abs/1709.02260, 2017. [Online]. Available: http://arxiv.org/abs/1709.02260

[24] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2017, pp. 273–287.

[25] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 14–26.

[26] S. Y. et al., "Xnor-sram: In-memory computing sram macro for binary/ternary deep neural networks," *IEEE Journal of Solid-State Circuits*, 2020.

[27] A. D. P. et al., "An mram-based deep in-memory architecture for deep neural networks," in *IEEE International Symposium on Circuits and Systems*, 2019.

[28] "Gnu compiler collection (gcc) internals." [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/

[29] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *ArXiv*, vol. abs/1602.02830, 2016.

[30] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture," *arXiv preprint arXiv:2105.03814*, 2021.

[31] "Compiler explorer." [Online]. Available: https://dpu.dev/

[32] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2013, pp. 185–197.

[33] T. P. Morgan, "Accelerating compute by cramming it into dram memory," Oct 2019. [Online]. Available: https://www.upmem.com/nextplatform-com-2019-10-03-accelerating-compute-by-cramming-it-into-dram/