In-Memory Computing based Acceleration: Large-Scale to Edge Computing

By

Ahmet Alper Goksoy

A preliminary report for the degree of

Doctor of Philosophy

(Department of Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2023

Preliminary Examination Date: 06/01/2023

Preliminary Exam Committee:

Umit Y. Ogras, Associate Professor, Electrical and Computer Engineering, University of Wisconsin-Madison

Yu Hen Hu, Professor, Electrical and Computer Engineering, University of Wisconsin-Madison

Younghyun Kim, Assistant Professor, Computer Sciences, University of Wisconsin-Madison

Chaitali Chakrabarti, Professor, Electrical, Computer and Energy Engineering, Arizona State University

© Copyright by Ahmet Alper Goksoy 2023
All Rights Reserved

Dedicated to my son, my parents Ayşe Gülnihal and Ismail Hakkı,
my brother Osman Gökalp, and my wife Betül.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In-memory computing (IMC) based architectures enable energy-efficient inference and training of machine learning (ML) algorithms. As DNN complexity increases, the need for larger architectures is inevitable. Monolithic IMC architectures face yield and fabrication cost challenges because of significant area overhead. Hence, 2.5D/3D architectures are proposed for large-scale DNN accelerators using small chips (chiplets). Therefore, there is a critical need for research on designing optimized architectures for chiplets. In addition, there is an increasing demand to implement neural networks on mobile edge devices for both on-chip training and inference following recent developments in Internet of Things (IoT). Nonetheless, due to the significant computational resources required for training, deploying neural networks on edge devices with limited resources poses a challenge. The energy efficiency of IMC accelerators enables system designers to consider this architecture for edge devices. Therefore, on-chip training and inference using IMC architectures can enable energy-efficient edge computing. We perform research in the following areas to address these problems: (i) Big Little Chiplets: We develop a heterogeneous big-little chiplet-based IMC architecture that utilizes big and little IMC-based chiplets coupled with an optimal NoP configuration, (ii) On-Chip Training: We develop a ReRAM-based in-memory computing accelerator for on-chip training and inference of millimeter Wave (mmWave) CNN and inference of RGB CNN models for personalized home-based rehabilitation systems.

1 INTRODUCTION

State-of-the-art deep neural networks (DNNs) have become more complex with deeper, wider, and more branched structures to cater to demanding applications [4, 5]. The growing complexity reduces hardware inference performance due to increased memory accesses and computations [4]. To boost the performance and energy efficiency, in-memory computing (IMC)-based architectures embed the matrix-vector-multiplications within the memory arrays [3, 6, 7, 8, 9]. However, IMC architectures with stationary weights stored on the chip result in significant area overhead and fabrication cost [2, 3]. Hence, 2.5D/3D architectures are adopted to design large-scale DNN accelerators using an array of small chips (i.e. chiplets) connected by a network-on-package (NoP) [10, 11].

Prior studies have demonstrated chiplet-based architectures based on both IMC and conventional multiply-and-accumulate (MAC) engines for DNN acceleration [2, 12, 1, 10, 13, 11, 14, 15, 16, 17, 18, 19, 20]. However, existing schemes do not consider the non-uniform distribution of weights and activations within DNNs while designing the chiplet-based architecture. Figure 1.1(a) and Figure 1.1(b) show the distribution of activations and weights (normalized) across all layers of ResNet-50 on ImageNet and VGG-19 on CIFAR-100. The initial layers have more activations between layers but have fewer weights. A larger number of activations lead to more on-chip data movement, while fewer weights imply reduced computations. In contrast, the latter layers have more weights and fewer activations, resulting in increased computations and reduced data movement. Hence, the chiplet-based IMC architectures should be optimized to match the non-uniform algorithm struc-

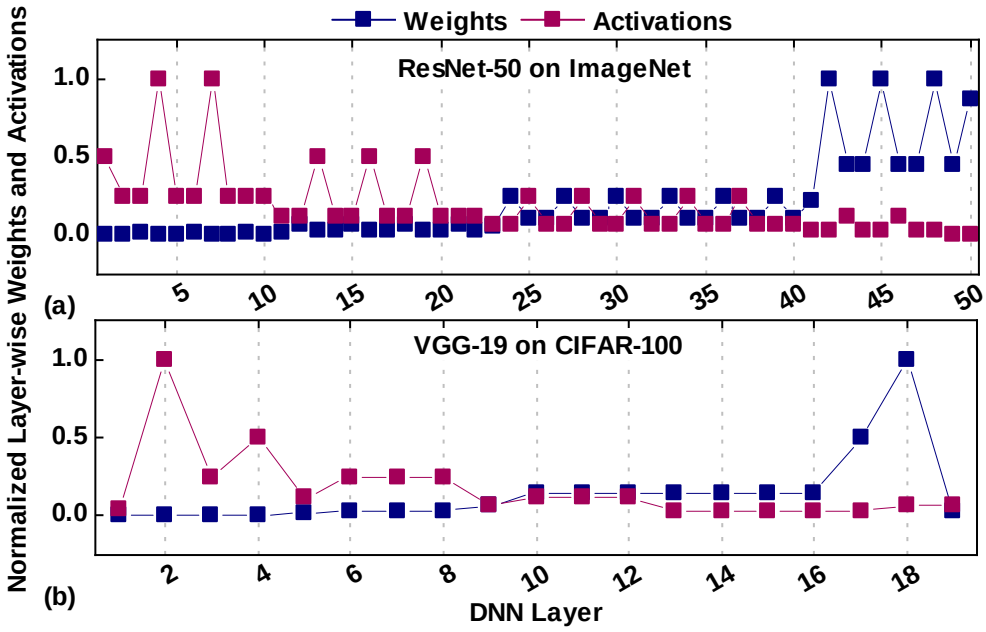

Figure 1.1: Normalized layer-wise activation/weight distribution for (a) ResNet-50 (ImageNet) and (b) VGG-19 (CIFAR-100). Initial/latter layers are activation/weight dominated.

ture and maximize the efficiency of computation and data movement across the DNN layers.

Figure 1.2(a) shows the IMC utilization of four different DNNs using a homogeneous chiplet-based RRAM IMC architecture. The architecture utilizes chiplets with 16 tiles, where each tile consists of an array of 16 IMC crossbar arrays of size 256×256 [2]. The chiplets are interconnected by a 32-bit wide NoP operating at 250MHz, having the signaling scheme in [21]. Smaller DNNs like DenseNet-40 on CIFAR-10 have 29% IMC utilization, while larger DNNs like VGG-19 on CIFAR-100 achieve 40% IMC utilization. A lower IMC utilization leads to increased IMC array arrays and in turn, higher energy and latency. Furthermore, a single NoP structure

results in significant area overhead due to the large NoP driver and interconnect cost. Figure 1.2(b) shows that for the homogeneous structure, the NoP accounts for 90% and 50% of the total area for VGG-19 on CIFAR-100 and DenseNet-40 on CIFAR-10, respectively. In addition, the increased NoP bus width leads to higher NoP energy with up to 53.75× higher cost relative to an 8-bit multiply-and-accumulate (MAC) operation in 16nm technology node [12].

Optimizing the architecture and the NoP will lead to efficient execution of DNN models. Therefore, this work addresses the inefficiency of homogeneous chiplet-based IMC architectures that fail to exploit the underlying distribution of weights and activations within DNNs. To this end, we propose a heterogeneous chiplet-based IMC architecture that integrates big and little-chiplet banks, as illustrated in Figure 2.1. Specifically, we develop an algorithm to determine the optimal configuration of the big-little IMC chiplet architecture. The little-chiplet bank consists of little chiplets interconnected by an interposer-based NoP (chiplets are placed closed to each other) [21]. Similarly, the big-chiplet bank consists of big chiplets interconnected by a bridge-based NoP [22]. Little chiplets consist of fewer/smaller IMC crossbars or processing element (PE) arrays, while the big chiplets have more/larger IMC crossbars or PE arrays. In addition, each chiplet (big/little) utilizes a local DRAM to store the weights of the DNN.

In addition to the hardware architecture, we also propose a new technique to map DNNs onto the big-little chiplet-based IMC architecture. Taking a cue from the non-uniform distribution of the weights and activations within the DNN, we propose to map the early layers within a DNN onto the little chiplet bank and the

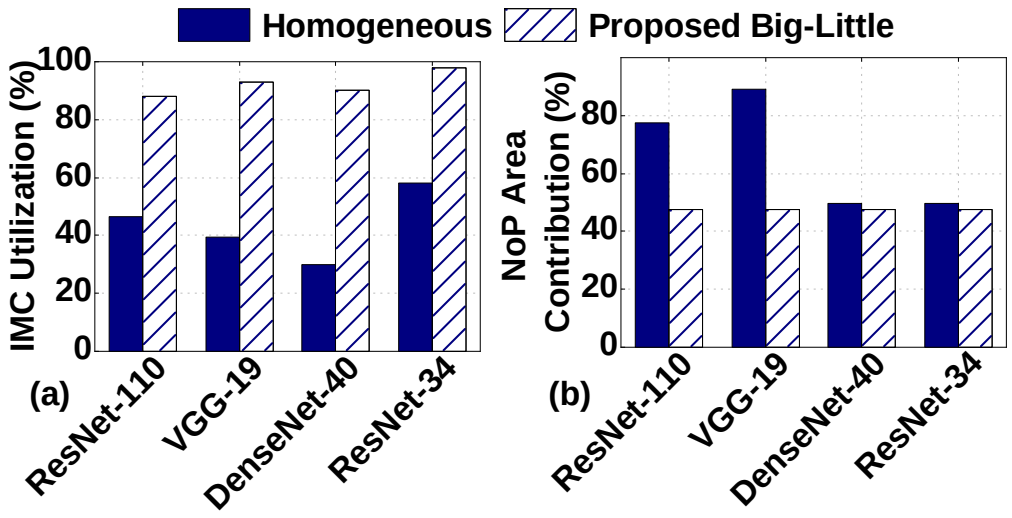

Figure 1.2: IMC utilization for different DNNs using a homogeneous chiplet RRAM IMC architecture [2] and the proposed heterogeneous big-little chiplet architecture. The heterogeneous big-little architecture improves the IMC utilization.

subsequent layers onto the big chiplet bank. The smaller structure of the weights in the early layers results in higher utilization within the little chiplet bank, while the larger layers towards the end of the DNN achieve high utilization on the big-chiplet bank. To achieve this, we develop a custom mapping algorithm that performs the mapping of the DNN on to the big-little architecture. We note that, the algorithm is universal and applies to the case when the resource in a given big-little chiplet is not enough to store all DNN weights.

We exploit the activation distribution by utilizing an interposer-based NoP with high bandwidth within the little chiplet bank, which houses the early layers with higher on-chip data movement. Simultaneously, the subsequent layers with lower on-chip data movement (fewer activations) utilize the bridge-based NoP with lower bandwidth within the big chiplet bank. Experimental evaluation of

the proposed big-little chiplet-based RRAM IMC architecture on ResNet-50 on ImageNet shows up to 259×, 139×, and 48× improvement in energy-efficiency with lower area compared to Nvidia V100 GPU, Nvidia T4 GPU, and SIMBA [1] architecture, respectively.

In addition, there is an increasing demand to implement neural networks on mobile edge devices for both on-chip training and inference following recent developments in the internet of things (IoT). Nonetheless, due to the significant computational resources required for training, deploying neural networks on edge devices with limited resources poses a challenge. One such case is home-based rehabilitation systems. Home-based rehabilitation using video cameras and wearable sensors has attracted significant attention due to its potential to help millions of people [23, 24, 25, 26]. For example, a recent study shows more than a two-fold increase in the number of amputations during the COVID-19 pandemic [27]. Remote monitoring and rehabilitation can complement infrequent and prolonged in-person visits to enable early diagnosis and intervention.

Prevalent use of home-based rehabilitation systems requires addressing three critical challenges. *First*, these systems must be sufficiently accurate to detect abnormal behavior and produce actionable data for health professionals. This requirement leads to the *second* challenge: sophisticated algorithms, such as machine learning (ML) and artificial intelligence (AI) techniques. Offloading these algorithms to the cloud is not a desirable solution since sending raw sensor data incurs high communication energy and latency while threatening user privacy. Hence, the *third* challenge is accomplishing home-based rehabilitation by running algorithms

locally, at the edge.

Recent techniques enable home-based rehabilitation using RGB cameras [23, 24], wearable inertial measurement units (IMU) [26], and millimeter-wave (mmWave) radar sensors [25]. These techniques collect sensor data as the patients perform rehabilitation movements. Then, they process the sensor data, typically using a convolutional neural network (CNN), to produce human joint coordinates. While these approaches show strong potential, they have one fundamental shortcoming. All prior techniques train their inference models with the user data available at design time. Then, they assume that future users, whose number is likely to be much larger than the training set, will use the produced model for inference. Even if the CNNs inference models generalize to arbitrary users, there is no guarantee that their accuracy will remain accurate. Hence, this limitation jeopardizes the first requirement: high accuracy in estimating the joint coordinates. As a result, there is a strong need for approaches that customize the deep learning models to specific users through on-device training in the home environment.

RGB cameras are the most common sources since they offer true-color real-world information. However, *always-on cameras at home* can raise serious privacy concerns such as facial information leakage. In contrast, mmWave radar, an emerging wireless sensing device, can accurately measure objects' moving trends while retaining privacy. In this work, we focus on systems that use mmWave radar inputs since they also have significantly lower processing requirements than RGB camera inputs and do not require users to wear any special sensors. We assume that an inference model, such as CNN, is trained offline to convert mmWave signals to human joint

positions. Then, it is acquired by a new patient for home-based rehabilitation. Since the accuracy of this model is limited by the offline data, the proposed system aims to customize the initial model to the new user, as illustrated in Fig. 4.1. The camera is *activated only during this customization process* to produce the human joint coordinates using the video frames. Then, these joint positions are used as a reference to supervise the incremental training of the inference model that uses the mmWave signals. After the customization, *only the mmWave signals and corresponding inference model* are used during the device lifetime, achieving over 13-fold inference time and 131-fold power consumption savings.

Current processors used for inference at the edge (e.g., at home) have limited processing capability due to their cost and energy constraint. For example, the Nvidia Jetson Xavier NX board can perform inference using mmWave inputs in 149.7 μs per input frame. However, training using RGB camera input reference takes 620 ms per frame on the same device. It can barely achieve a 1.6 frame per second (FPS) operation, which is impractical considering realistic 30 FPS or higher video frame rates. Storing the video frames and performing inference later is also not practical due to excessive memory requirements. Hence, practical solutions require novel AI hardware and methodologies to perform on-device training at the edge. To address this need, we propose an energy-efficient on-device training approach that enables personalized home-based rehabilitation, PHR. The proposed approach first generates the ground truth 3D joint coordinates data using RGB cameras. These coordinates are used to supervise on-device training. Then, we customize a baseline mmWave human pose estimation model using energy-efficient

on-device training. After the customization, our framework uses mmWave radar signals and the customized home-based rehabilitation model.

In summary, this preliminary report makes the following contributions:

Big-Little Chiplets [28]:

- We propose a heterogeneous big-little chiplet-based IMC architecture that utilizes a big and little IMC-based chiplet compute structure coupled with an optimal NoP configuration (interposer and bridge).
- We present a custom mapping strategy of DNNs onto the big-little chiplet IMC architecture that exploits the non-uniform distribution of weights and activations,
- Our experiments of the proposed big-little chiplet-based RRAM IMC architecture on ResNet-50 on ImageNet achieve up to 259×, 139×, and 48× improvement in energy-efficiency and lower area compared to Nvidia V100 GPU, Nvidia T4 GPU, and SIMBA [1] architecture, respectively.

On-Chip Training [29]:

- An energy-efficient on-chip training framework that customizes mmWave-based human pose estimation model for higher accuracy.
- A Resistive RAM-based in-memory computing accelerator for on-chip training and inference of mmWave and inference of RGB models.

- Experimental results that demonstrate the practical real-life use of our framework, with a 28.01% lower error, 611.1× lower inference energy, and 14.0× faster training than a baseline model on Nvidia Jetson Xavier NX [30].

The rest of the report is organized as follows. The literature survey is discussed in Chapter 2. Chapter 3 presents the first completed preliminary work, Big-Little Chiplets for IMC of DNNs, and discusses its role in large-scale computing. The second completed preliminary work, energy-efficient on-chip training approach for personalized home-based rehabilitation systems is presented in Chapter 4. The first proposed work on sparse neural network optimization is discussed in Chapter 5. Chapter 6 presents the second proposed work on carbon footprint optimization. Other work that I have completed during my research is discussed in Chapter 7. Finally, Chapter 8 concludes this report.

2 LITERATURE REVIEW

2.1 Chiplet-based Architectures

Chiplet-based architectures are well explored for high-performance computing applications [14, 15, 10, 31, 16, 13, 32]. A co-design flow considering architecture, chip, and package for a chiplet-based system is proposed in [14]. A detailed design space exploration with the proposed co-design flow shows significant improvement in power consumption and area with respect to a monolithic design. Vivet et al. [15] proposed a chiplet-based system with 96 computing cores and a 3D memory are distributed over 6 chiplets. Another recent work proposed a 2,048 chiplet (14,336 cores) wafer-scale processor that utilizes a bridge-based integration [10]. The authors discuss the challenges of designing a wafer-scale processor and provide insights into power delivery, clock routing, and testing.

Chiplet-based architectures have proven to be both more energy-efficient and cost-effective than monolithic architectures for complex DNNs. Several prior studies proposed chiplet-based architectures for DNN acceleration [1, 33, 12]. The authors of [1] proposed a fine-grained 36-chiplet architecture for DNN inference acceleration. Each chiplet utilizes a homogeneous structure with 16 PEs that operate using a weight stationary dataflow. The chiplets are connected by a 6×6 NoP mesh that utilize the ground-referenced signaling technique [21]. The authors of [12] proposed a hierarchical and analytical framework, NN-Baton, to analyze DNN mapping and communication overheads in a chiplet-based DNN accelerator. NN-Baton supports different mapping schemes of DNNs onto the chiplets. Furthermore, an

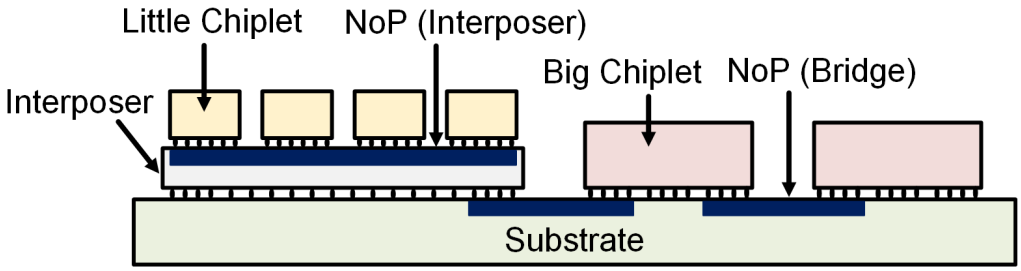

Figure 2.1: Cross-sectional view of the big-little chiplet-based IMC architecture. The architecture consists of a little chiplet bank with little chiplets (connected by an NoP within the interposer and a big chiplet bank with big chiplets connected by a bridge NoP. NoP properties: 1.5–8mm length, 2–4.5µm pitch, and 0.5–2µm width.

analytical model to quantify the communication overhead for NoP is also presented in NN-Baton. A family of chiplet topologies are proposed in [33]. The authors explored different chiplet topologies and compared their performance through an analytical metric which estimates latency. A chiplet-based IMC benchmarking tool for design space exploration, SIAM, is proposed in [2]. SIAM supports different chiplet architectures, IMC crossbar tile structures, NoP, NoC, and DRAM estimation. However, all prior studies assume a homogeneous chiplet structure across all chiplets interconnected by a single NoP. Furthermore, none of the prior works considered the non-uniform distribution of weights and activations in the DNN during the mapping process. Hence, many chiplets remain under-utilized while the large NoP leads to an increased area and energy overhead.

In contrast to prior works, we propose a heterogeneous big-little chiplet-based IMC architecture that combines big chiplet bank with a bridge-based NoP and a little IMC chiplet bank with an interposer-based NoP to enhance the IMC utilization and improve energy efficiency. Furthermore, we propose a customized methodology

that exploits the non-uniform distribution of DNN weights and activations in mapping the DNNs onto the big-little chiplet IMC architecture. To the best of our knowledge, this is the first heterogeneous chiplet-based IMC architecture that leverages different IMC structures collectively with a heterogeneous NoP coupled with a customized DNN mapping.

2.2 Home-based Rehabilitation Systems

Home-based rehabilitation systems draw significant attention, especially during the pandemic era, since they facilitate patient access to rehabilitation exercises at home and reduce in-person physical therapy sessions. To this end, researchers established the relationship between human joint location and rehabilitation movements [23, 24]. Authors in [23] proposed an approach that can get human joint information and face videos to relate the pain to the patient's movement. UI-PRMD [24] dataset provides exercise data using Kinect and motion capture system. To check whether the exercises conform to standards, rehabilitation systems require accurate human pose estimation, usually obtained from RGB images [34, 35]. OpenPose [34] and HRNet [35] are the most representative approaches that achieve fast and accurate human pose estimation from RGB sources.

mmWave radar-based pose estimation addresses privacy concerns and enhances robustness to the environment compared to RGB-based approaches, thus being an emerging solution for rehabilitation systems [25, 36, 37]. These approaches map 3D mmWave point cloud to ground truth human joints using smaller CNNs

than those processing RGB video ones since the mmWave frames are significantly smaller than their RGB counterparts. However, existing techniques focus on offline learning and algorithm design. Since they assume that offline-design CNNs will generalize to arbitrary users, they only consider inference during rehabilitation exercises. Hence, they do not deal with on-device training after a new patient starts using the system. In strong contrast, our proposed framework achieves end-to-end real-time mmWave-based human pose estimation, including training and inference.

The acceleration of both training and inference is critical for the real-time execution of applications. To this end, we utilize a Resistive RAM (ReRAM)-based in-memory computing (IMC) AI accelerator for our framework. IMC-based hardware accelerators perform computation inside memory units to reduce off-chip data communication. ReRAM-based approaches achieve high density and low energy consumption. Therefore, they are widely used for machine learning acceleration [3, 1, 38, 6, 28]. The training of CNN models is vulnerable to gradient precision and the write endurance and nonlinear properties of ReRAM architectures can cause an accuracy loss during training [39, 40]. Authors in [40] and [41] proposed methods to mitigate these problems. By utilizing these methods, on-chip training on ReRAM-based IMC accelerators is possible without seeing a significant accuracy drop.

In contrast to prior work, we propose PHR, a ReRAM-based IMC accelerator with energy-efficient on-chip training capacity for home-based rehabilitation systems. To the best of our knowledge, it is the first system that enables real-time processing of RGB image data, fast on-chip training of mmWave radar-based human pose

estimation, and energy-efficient model inference for continuous patient usage.

2.3 Task Scheduling Techniques for Heterogeneous Architectures

Schedulers have evolved significantly to adapt to different requirements and optimization objectives. Static [42, 43] and dynamic [44] task scheduling algorithms have been proposed in the literature. Completely Fair Scheduler (CFS) [44] is a dynamic approach that is widely used in Linux-based OS and aims to provide resource fairness to all processes while the static approaches presented in [42, 43] optimize the makespan of applications. CFS [44] was initially developed for homogeneous platforms, but it can also handle heterogeneous architectures (e.g., Arm big.LITTLE). While CFS may be effective for client and small-server systems, high-performance computing (HPC) and high-throughput computing (HTC) necessitate different scheduling policies. These policies, such as Slurm and HTCondor, are specifically designed to manage a large number of parallel jobs and meet high-throughput requirements [45, 46]. On the other hand, DSSoCs demand a novel suite of efficient schedulers that execute at nanosecond-scale overheads since they deal with scheduling tasks that can execute in the order of nanoseconds.

The scheduling overhead problem and scheduler complexities are discussed in [47, 48, 49]. The authors in [47] propose two dynamic schedulers, named as CATS and CPATH, where CATS detect the longest and CPATH detects the critical paths in the application. CPATH algorithm shows inefficiency in terms of its higher

scheduling overhead. Motivated by high scheduling overheads, [50] propose a new scheduler that approximates an expensive heuristic algorithm using imitation learning with low overhead. An imitation learning-based scheduler approximates an expensive heuristic with a low overhead [50]. However, the scheduling overhead is still approximately 1.1 μs, making it inapplicable for DSSoCs with nanosecond-scale task execution. Energy-aware schedulers for heterogeneous SoCs have limited applicability to DSSoCs because of their complexity and large overheads [51, 52].

Several scheduling algorithms that demonstrate the benefits of using multiple schedulers are proposed in [53, 54, 55]. Specifically, the authors in [53] propose a technique that switches between three schedulers dynamically to adapt to varying job characteristics. However, the overheads of switching between policies are not considered as part of the scheduling overhead. The approach in [55] integrates static and dynamic schedulers to exploit both design-time and runtime characteristics for homogeneous processors. The hybrid scheduler in [55] uses a heuristic list-based schedule as a starting point and then improves it using genetic algorithms. However, it does not consider the scheduling overhead of the individual schedulers. The authors in [48] discuss the performance comparison of a simple round-robin scheduler and a complex earliest deadline first (EDF) scheduler and their applicability under different system load scenarios.

Using insights from literature, we propose a novel scheduler that combines the benefits of the low scheduling overhead of a simple scheduler and the decision quality of a sophisticated scheduler (described in Section 7.1.3) based on the system workload intensity. To the best of our knowledge, this is the first approach that uses

a novel runtime preselection classifier to choose between simple and sophisticated schedulers at runtime to enable scheduling with low energy and nanosecond scale overheads in DSSoCs.

3 BIG-LITTLE CHIPLETS FOR IN-MEMORY ACCELERATION OF DNNS: A SCALABLE HETEROGENEOUS ARCHITECTURE

3.1 Overall Architecture

Figure 3.1(a) shows the top-level block diagram of the heterogeneous big-little chiplet IMC architecture. The architecture consists of two banks of IMC chiplets, a little bank (shown in yellow color) and a big bank (shown in light red color). The little IMC chiplet bank consists of chiplets with smaller and fewer IMC crossbar arrays compared to the big chiplets. It is placed on an interposer that houses the NoP. The NoP provides high bandwidth and a compact structure for on-package communication within the little chiplet bank. At the same time, the increased size and count within the big chiplet bank allow for higher computation capability. The big chiplets are directly connected to the substrate using micro-bumps. A bridge-based NoP is utilized within the big chiplet bank for on-package communication. Long wires of the bridge NoP allow easy integration of the big chiplets. We utilize the Y–X routing methodology for the NoP. Each chiplet (big and little) consists of a local DRAM (DDR4 in this work) that stores the weights required for the IMC crossbar arrays.

Figure 3.1(b) shows the structure of a IMC chiplet. Each chiplet utilizes a hierarchical structure that consists of an array of big (bottom of Figure 3.1(c)) or little IMC tiles (top of Figure 3.1(c)) and each tile consists of an array of IMC crossbars or PEs. In addition, the chiplet contains a pooling unit, non-linear activation unit,

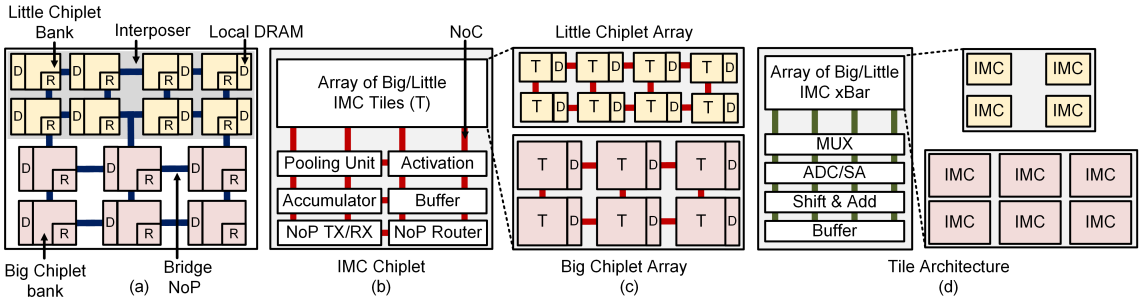

Figure 3.1: (a) Overview of the big-little chiplet IMC architecture. The little chiplet bank utilizes smaller chiplets connected by a interposer-based NoP while the big chiplet bank utilizes bigger chiplets connected by a bridge-based NoP. Each chiplet utilizes a local DRAM, (b) IMC chiplet architecture (big and little). Each chiplet consists of an array of IMC tiles and a dedicated NoP transceiver and router, (c) The little chiplet bank consists of fewer and smaller tiles while the big chiplet bank consists of more bigger tiles. Both chiplet structures utilize a mesh-based NoC for on-chip communication, and (d) Structure of each tile within the big and little chiplet. It consists of an array of IMC crossbar arrays and associated peripheral circuits with an interconnect similar to that in [3]. The little chiplet consists of fewer and smaller IMC crossbars while the big chiplet has larger and more IMC crossbar arrays.

accumulator, and buffer. The accumulator is used for the partial sum accumulation across different tiles within the chiplet. Furthermore, the buffers allow for efficient data movement in and out of the chiplet. Each IMC chiplet consists of a dedicated NoP transceiver used for the transmission and reception of packets across the NoP. In this work, we adopt the NoP transceiver from [21]. Each transceiver consists of a local PLL circuit that provides the clock for the transceiver. A five-port router is utilized for routing of the data across the NoP.

Each IMC chiplet utilizes a local DRAM to store the weights. The local DRAM allows for external memory access, thus making our proposed big-little architecture a generic platform. If a DNN does not fit on the entire chip, the DRAM stores all the

weights necessary for each chiplet. First, the DRAM loads the necessary weights into the IMC crossbar arrays. Next, while the computation is performed, the DRAM loads the next set of weights of the DNN. The buffer is designed to support a ping-pong operation [56]. The weights from the DRAM are loaded into the first buffer stage (ping) and then moved to the second buffer stage (pong). Therefore, the big-little IMC chiplet architecture masks the DRAM latency with the computation latency, achieving high throughput.

Finally, Figure 3.1(d) shows the structure of an IMC tile. Each array in the crossbar consists of PEs that perform the computations. In this work, we focus on a resistive random-access-memory (RRAM) based IMC crossbar array due to its superior energy-efficiency [3]. The computations are performed in the analog domain by turning on all wordlines (WL) together and performing accumulation along the bitline (BL). The inputs are given through the WL while the weights are stored within the RRAM cells. Each IMC array consists of specialized peripheral circuitry that assists the computation. The peripheral circuitry includes a column multiplexer (mux), an analog-to-digital converter (ADC), a shift and add circuit, and a buffer. The column mux is used to share the ADC across columns of the IMC array. The ADC converts the MAC output in the analog domain across each column into the digital domain. The big-little IMC architecture does not utilize a digital-to-analog converter (DAC) by employing bit-serial computing. The shift and add circuit handles the positional value of each bit within the multi-bit input activations that are computed using the IMC arrays. The buffers within the tile are utilized for storing the partial sums and the input activations.

The following two sections present the implementation details and experimental evaluations, respectively.

3.2 Parameters of the Big-Little Architecture and Mapping

This section describes the implementation and mapping details of the Big-Little chiplet architecture.

The underlying non-uniform distribution of weights and activations within a DNN results in an increased number of activations in the early layers and larger number of weights in the subsequent layers (Figure 1.1). This non-uniform weight distribution leads to under-utilization of chiplets in the early layers, thus a lower overall IMC utilization. To improve the IMC utilization, crossbar arrays with smaller size (e.g. 32×32 instead of 128×128) can be used everywhere. However, using smaller crossbar arrays also leads to increasing number of chiplets in the system. In turn, larger number of chiplets in the system increases the area as well as energy consumption (due to higher relative area and energy of the peripheral circuits) masking the benefit of using chiplet-based system. Therefore, a balance between crossbar array size and number of chiplets in the system is necessary. To this end, we propose a technique to optimize the big-little chiplet configuration as discussed next.

3.2.1 Configuration of the big-little chiplets

We first determine the configuration of big-little chiplets by computing the tile utilization with different big-little chiplet configurations for a given DNN. Algorithm 1 shows our proposed technique to find the utilization. The inputs to the algorithm are

1. the set of crossbar sizes for the little chiplets ($\mathcal{X}_{\mathcal{L}}$) and the big chiplets ($\mathcal{X}_{\mathcal{B}}$),
2. set of number of tiles in the little chiplets ($\mathcal{T}_{\mathcal{L}}$) and the big chiplets ($\mathcal{T}_{\mathcal{B}}$),
3. number of little chiplets ($\mathcal{N}_{\mathcal{L}}$) and big chiplets ($\mathcal{N}_{\mathcal{B}}$),
4. the DNN structure,
5. the total number of chiplets in the system.

We note that the initial layers of the DNN are mapped on to little chiplets since there are fewer weights in the initial layers. A DNN layer is mapped on to a chiplet when number of tiles required for that layer is less than the number of remaining tiles in the chiplet, i.e., the available resource on the chiplet is sufficient for the layer (as shown in line 13–17 of Algorithm 1). Once a layer (layer-j) is mapped on to a chiplet, the tile utilization is computed as:

$$\begin{aligned} IMC_j &= \left\lceil \frac{K_j^x \times k_j^y \times N_j^{if}}{x} \right\rceil \times \left\lceil \frac{N_j^{of} \times Q}{x} \right\rceil \\ u_j &= 100 \times \frac{K_j^x \times k_j^y \times N_j^{if} \times N_j^{of} \times Q}{IMC_j \times x \times x} \end{aligned} \tag{3.1}$$

where K_j^x and K_j^y are the kernel sizes of layer-j, N_j^{if} and N_j^{of} are the number of i/p and o/p features for layer-j, Q is the quantization precision, IMC_j is the number of IMC crossbars required for layer-j and x is the IMC crossbar size ($x \times x$). Once the resources of a chiplet are exhausted, the next chiplet is considered for mapping. This process continues until no chiplet (little/big) is available.

In the proposed method, for each chiplet configuration, we obtain the average utilization for a particular DNN after each layer is mapped(line 36 of Algorithm 1). Then we sort (in descending order) the configurations based on the utilization and save the top K configurations. The above procedure is repeated for M different DNNs and the configuration with highest utilization which is common for all DNNs is considered as the final configuration for the big-little chiplet system. We note that K and M are user-defined parameters and our proposed technique is independent of these parameters.

3.2.2 Configuration of the big-little NoP

The heterogeneous chiplet configuration (discussed in Section 3.2.1) improves the overall chiplet utilization by using smaller chiplets that match well to the early layers with fewer weights. However, the initial DNN layers produce higher number of activations compared to later layers. Therefore, the volume of traffic between little chiplets (used for initial DNN layers) is higher than the traffic volume between big chiplets (used for later DNN layers). Hence, the network-on-package (NoP) configuration between little chiplets needs to be different than that of the big chiplets. To this end, we propose a technique to determine optimal NoP configuration for a

system with big-little chiplet targeted for a particular DNN. Algorithm 2 shows the technique to determine NoP configuration for a particular DNN. The inputs to the algorithm are:

1. big-little chiplet configuration obtained from Algorithm 1,
2. set of NoP bus width for the little chiplets ($\mathcal{W}_{\mathcal{L}}$) and the big chiplets ($\mathcal{W}_{\mathcal{B}}$),
3. set of NoP frequency for the little chiplets ($\mathcal{F}_{\mathcal{L}}$) and the big chiplets ($\mathcal{F}_{\mathcal{B}}$),
4. the DNN structure.

We evaluate the energy-delay product of communication for each NoP configuration in the set of configurations. An analytical expression based evaluation is incorporated to perform fast exploration in the NoP configuration space. First, we evaluate communication volume of each NoP configuration given a particular DNN. The communication volume is equivalent to the number of packets transferred between two chiplets, and the number of packets (P) is expressed as $P = \frac{b}{w}$, where b is the number of bits to be communicated and w is the NoP bus width. We divide the number of packets by NoP frequency (f) to obtain an approximation of NoP latency $d = \frac{P}{f} = \frac{b}{w \times f}$. Next, we compute NoP power consumption by assuming that it is proportional to cube of NoP frequency [57]; $p = f^3$. Then the approximate energy consumption (e) is computed by multiplying communication latency and communication power; $e = d \times p$. Finally, communication EDP between each pair of chiplet (edp) is computed as:

$$edp = e \times d = d \times p \times d = d^2 \times f^3 = \left(\frac{b}{w \times f}\right)^2 \times f^3 = \frac{b^2 \times f}{w^2} \tag{3.2}$$

The total communication EDP for each NoP configuration for a particular DNN is obtained by adding the communication EDP between each pair of chiplets. A total of K NoP configurations with lower EDP are saved and the above procedure is repeated for M different DNNs. The configuration with lowest cost which is common for all DNNs is considered as the final NoP configuration for the big-little chiplet system. Similar to the technique of selecting big-little chiplet configuration (described in Section 3.2.1), K and M are the user defined parameter and our proposed technique is independent of these parameters.

3.2.3 Mapping a Previously Unseen DNN to a System on big-little Chiplets

So far, we described our proposed technique to determine the optimal configuration of big-little chiplet and the NoP. The optimal configuration is determined by performing design space exploration with several DNNs. However, an unknown DNN (not seen before) may be encountered at runtime. Moreover, there is no guarantee that all the weights of a given DNN will fit in the on-chiplet resources since the number of DNN parameters seem to be continuously growing. In these cases, we need to divide the entire DNN into multiple parts and load the weights of each part from DRAM before executing. Algorithm 3 shows the DNN partitioning as well as the mapping technique. The input to the algorithm is the DNN structure, big-little chiplet configuration and big-little NoP configuration. First, we compute the number of in-memory computing bits available on the system (S_B). Specifically, for each type (little/big) of chiplets, we multiply the number of available chiplets

(n_l/n_b), the number of tiles in each chiplet (t_l/t_b), the number of crossbar array in each tile (16), and the size of IMC crossbar array for big and little chiplets (x_l/x_b). Then we add the product for big and little chiplets to obtain the total number of in-memory computing bits available on the system (S_B):

$$S_B = (n_l \times t_l \times 16 \times x_l \times x_l) + (n_b \times t_b \times 16 \times x_b \times x_b) \tag{3.3}$$

Next, we compute the number of bits required to store all the weights of the DNN (D_B). Assuming average utilization of $u(0 < u \leqslant 1)$, the total number of partitions (Pr) required for the DNN is computed by taking the ceiling of the quotient obtained by dividing the required number of bits to store all weights (D_B) by the available number of in-memory bits on the system (S_B):

$$Pr = \left\lceil \frac{D_B}{S_B \times u} \right\rceil \tag{3.4}$$

For each partition, first, we compute the utilization of i^{th} layer on a big chiplet (U_B^i) as well as on a little chiplet (U_L^i). We compute U_B^i and U_L^i using Equation 3.1. If the big chiplet utilization $((U_B^i))$ is less than the little chiplet utilization (U_L^i) and the little chiplet bank is not exhausted, then the layer is mapped onto a little chiplet, as shown in lines 7–12 of Algorithm 3. Otherwise, we compute the number of big chiplets required (a_B) to map the rest of the layers. If a_B is less than or equal to the number of available big chiplets (A_B), then we map the rest of the layers to the big chiplet bank, else the algorithm throws an error since the resource requirement exceeds the available capacity (shown in line 18–23 of Algorithm 3). Thus, we

ensure that the initial layers with fewer weights are mapped into little chiplets and the latter layers with higher number of weights are mapped onto big chiplets with more computation resources. Therefore, our proposed custom mapping of the DNN onto the big-little chiplet architecture ensures high IMC utilization.

3.3 Experimental Evaluation

3.3.1 Experimental Setup

Evaluation platform: To evaluate the proposed heterogeneous big-little IMC chiplet architecture, we use a customized version of the open-sourced tool SIAM [2]. The customization includes the addition of the custom mapping scheme detailed in Section **??**. In addition, we handle the big-little chiplet IMC architecture by adding the number of each type (big/little) of chiplets, the number of tiles inside big and little chiplets, and the big-little IMC structure. Furthermore, we also assume that each type of chiplet can use different NoP width. The simulator performs the mapping of a given DNN onto the big-little IMC chiplet architecture. The outputs include area, energy, latency, throughput, energy efficiency, and IMC utilization (for all individual components in the architecture). Finally, we add support for intermediate DRAM access (DDR4 [58]) for each chiplet to handle the case where all weights do not fit on the system at once. We plan to open-source the tool and optimization methodology upon acceptance of the paper.

DNN algorithms and architectural parameters: We evaluate the proposed heterogeneous chiplet architecture with DenseNet-40 (0.26M) on CIFAR-10, ResNet-

Table 3.1: Set of configurations considered to determine big-little chiplet and NoP structure.

Chiplet Configuration		NoP Configuration	
Parameter	Values in the Set	Parameter	Values in the Set
$\mathcal{X}_{\mathcal{L}}$	{32, 64}	$\mathcal{W}_{\mathcal{L}}$	{16, 32, 64}
$\mathcal{X}_{\mathcal{B}}$	{128, 256, 512}	$\mathcal{W}_{\mathcal{B}}$	{4, 8, 12, 16, 20, 24}
$\mathcal{T}_{\mathcal{L}}$	{9, 16, 25}	$\mathcal{F}_{\mathcal{L}}$	{600, 1000, 1400, 1800} MHz
$\mathcal{T}_{\mathcal{B}}$	{36, 49}	$\mathcal{F}_{\mathcal{B}}$	{600, 800, 1000} MHz

110 (1.7M) on CIFAR-10, VGG-19 (45.6M) on CIFAR-100, ResNet-34 (21.5M) and ResNet-50 (23M) on ImageNet. We utilize an RRAM-based IMC structure for DNN inference with the following parameters: one bit per RRAM cell, a R_{off}/R_{on} ratio of 100, ADC resolution of 4-bits with 8 columns multiplexed, operating frequency of 1GHz [59, 3], and a parallel read-out method. We use 8-bit quantization for the weights and activations, and a 32nm CMOS technology node. The chiplets are placed to achieve the least Manhattan distance. The NoP parameters include E_{bit} of 0.54pJ/bit [21], interconnect parameters width, length, and pitch for the interposer-based NoP from [21] and for bridge-based NoP from [60] (Figure 2.1), per lane NoP TX/RX area of 5,304 μm^2, and NoP clocking circuit area of 10,609 μm^2 [61]. In addition, we also model the μbump for both the interposer [62] and bridge-based [63] NoP by utilizing the PTM models [64].

Table 3.2: Performance comparison of each component of a homogeneous (Little only, Big only) chiplet architecture and the heterogeneous Big-Little IMC chiplet architecture for VGG-19 on CIFAR-100.

Configuration	Area					Energy					Latency				
	IMC (%)	NoP (%)	NoC (%)	Total (mm^2)	Normalized to big-little (×)	IMC (%)	NoP (%)	NoC (%)	Total (mJ)	Normalized to big-little (×)	IMC (%)	NoP (%)	NoC (%)	Total (ms)	Normalized to big-little (×)
Little only	11.9	88.0	0.1	952.1	10.9	99.7	0.2	0.1	1.3	4.1	99.7	0.1	0.2	1.6	1.3
Big only	44.0	55.5	0.5	597.2	6.8	78.6	11.0	10.4	0.43	1.3	99.6	0.1	0.3	3.2	2.7
Big-Little (this work)	**52.4**	**47.4**	**0.2**	**87.4**	**1.0**	**99.8**	**0.1**	**0.1**	**0.32**	**1.0**	**99.2**	**0.3**	**0.5**	**1.2**	**1.0**

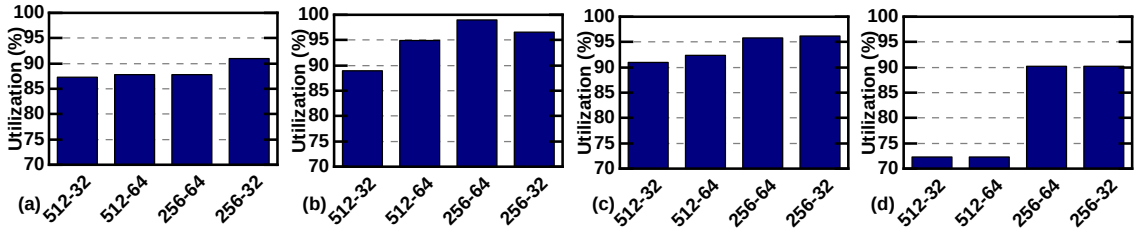

Figure 3.2: IMC utilizations for different DNNs across different big-little chiplet-based RRAM IMC configurations for (a) ResNet-110, (b) ResNet-34, (c) VGG-19, (d) DenseNet-40. Based on the utilization, we choose crossbar size of big chiplet as 256×256 and crossbar size of little chiplet as 64×64 (256–64).

3.3.2 Big-Little IMC Structure and NoP

This section demonstrates the parameters related to big-little IMC structure and big-little NoP. Specifically, we consider four DNNs (mentioned in Section 3.3.1) and execute Algorithm 1 to determine the top 10 (K=10) configurations with highest utilization for each DNN. We consider a system with 36 chiplets to limit the total area and power consumption of the system. Table 3.1 shows the input parameters ($\mathcal{X}_\mathcal{L}, \mathcal{X}_\mathcal{B}, \mathcal{T}_\mathcal{L}, \mathcal{T}_\mathcal{B}$) to the algorithm. We vary the number of little chiplets from 1 to 35 while maintaining the total number of chiplets to be 36. Then, we choose the best configuration which is common for all four DNNs. We observe that a system with *25 little chiplets* with a *64×64 IMC crossbar* and *25 tiles per chiplet*, and *11 big chiplets*

with a *256×256* IMC crossbar and *36 tiles per chiplet* provides best utilization across all four DNNs. Figure 3.2 shows the utilization for a system with 25 little and 11 big chiplets with varying size of crossbars (both for big and little chiplet) for all four DNNs. In this case, we also fixed the number of tiles per chiplet to 25 for the little chiplets and 36 for the big chiplets. Figure 3.2 reveals that the configuration where the crossbar size of the big chiplets is 256×256 and the crossbar size of the little chiplets is 64×64 (256–64, 256 denotes crossbar size of big chiplets and 64 denotes crossbar size of little chiplets) shows higher utilization than other configurations for three out of four DNNs. Only in the case of ResNet-110, the configuration 256–32 shows higher utilization than 256–64. However, we choose 256–64 over 256–32 since it provides more on-chip resources, lower area and energy efficiency for the IMC crossbar array (due to peripheral circuits).

Similarly, we execute Algorithm 2 for four DNNs to obtain the NoP configuration. Table 3.1 shows the set of different NoP parameters ($\mathcal{W}_{\mathcal{L}}, \mathcal{W}_{\mathcal{B}}, \mathcal{F}_{\mathcal{L}}, \mathcal{F}_{\mathcal{B}}$ used as inputs to Algorithm 2). The parameters are adopted from [65]. EDP for NoP is obtained for all NoP configurations for the four DNNs. Then, the NoP configuration having the lowest EDP for all four DNNs is chosen. Based on the EDP results, the big NoP frequency and the little NoP frequency is set to 600 MHz and 1 GHz, respectively; the big NoP bus width and the little NoP bus width is set to 24 and 32, respectively. Figure 3.3 shows the normalized NoP EDP for different combination of bus width for big and little chiplets. For illustration purpose, we show VGG-19 and ResNet-34 since these two DNNs utilize more than 34 out of 36 chiplets. From Figure 3.3, it is observed that the configuration with big NoP bus width of 24 and little NoP bus

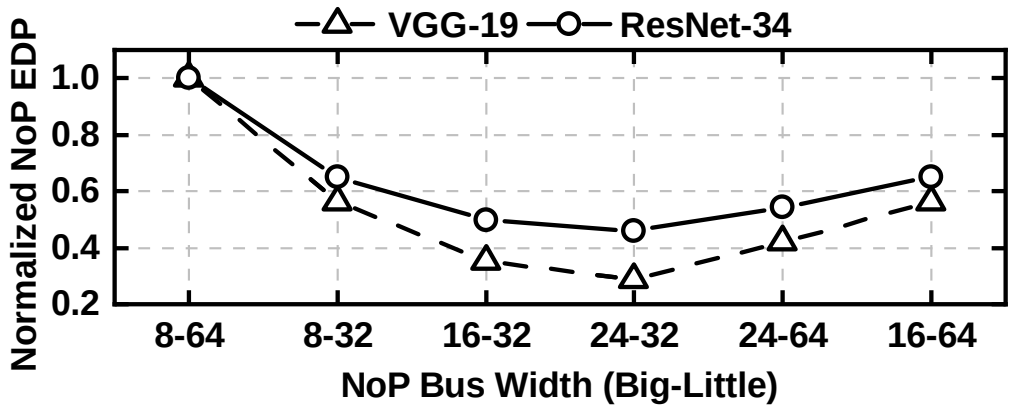

Figure 3.3: Normalized NoP EDP for different bus-widths for VGG-19 and ResNet-34. The NoP with bus width of 24 for big and 32 for little chiplets (24–32) shows lowest EDP.

Table 3.3: Performance comparison of a homogeneous (Little only, Big only) chiplet architecture and the heterogeneous Big-Little IMC chiplet architecture for different DNNs.

Configuration	Utilization (%)				Area (mm^2)				Energy (mJ)				Latency (ms)			
	Res-110	VGG-19	Dense-40	Res-34	Res-110	VGG-19	Dense-40	Res-34	Res-110	VGG-19	Dense-40	Res-34	Res-110	VGG-19	Dense-40	Res-34
Little only	69	92	58	93	171.7	952.1	71.5	657.8	1.4	1.3	0.22	41.1	23.0	1.6	1.6	13.1
Big only	44	59	32	82	220.0	597.2	220.2	595.9	0.28	0.43	0.11	3.7	1.1	3.2	0.02	20.2
Big-Little (this work)	**88**	**93**	**90**	**98**	**87.4**	**87.4**	**87.4**	**87.4**	**0.18**	**0.32**	**0.06**	**8.2**	**1.1**	**1.2**	**0.03**	**48.6**

width of 32 shows the lowest EDP. Since little chiplets produce higher number of activations than the big chiplets, it is intuitive that little NoP are wider (larger bus width) than the big NoP.

3.3.3 Comparison with Baseline Architectures with Homogeneous Chiplets

We compare the performance of our proposed big-little chiplet architecture with respect to two baseline architectures with homogeneous chiplets [2]. **1) Little**

only: In this configuration, we consider a system where the configuration of all chiplets as well as the NoP is same as that of the little chiplets. **2) Big only:** In this configuration, we consider a system where the configuration of all chiplets as well as the NoP is same as that of the big chiplets. We note that, the total number of chiplets with 'Little only' and 'Big only' configurations vary for different DNNs.

Table 3.2 shows the performance comparison for 'Little only', 'Big only' and the proposed big-little architectures for VGG-19 on CIFAR-100. In this table, the performance of each component of the architecture, i.e. IMC, NoP and NoC is shown. Our proposed big-little chiplet architecture results in a balanced distribution of the area among the circuit and NoP components, while the NoC accounts for a minimal portion (0.2%) of the total area. In 'Little-only' architecture, NoP becomes the bottleneck for area since the chiplets have smaller size, hence more number of chiplets are required which increases the NoP. In 'Big only' architecture, NoP consumes more energy due to higher volume of data movement between each pair of chiplets. In contrast, the proposed big-little architecture with its high IMC utilization and reduced on-chip communication as well as on-package data movement results in less total energy consumption and less inference latency. Overall, the proposed heterogeneous big-little architecture achieves up to 10.9× lower area, 4.1× lower energy, and 2.7× lower latency than 'Little only' and 'Big only' architectures.

Next, we compare the IMC utilization and the performance (area, energy and latency) for ResNet-110, VGG-19, DenseNet-40, and ResNet-34 against 'little only' and 'big only' architecture. For VGG-19, our proposed big-little architecture achieves the highest IMC utilization of 93% compared to 92% and 59% for 'Little only' and

'Big only', respectively. Similarly, the big-little architecture achieves 88%, 90%, and 98% IMC utilization for ResNet-110, DenseNet-40, and ResNet-34, respectively, up to 2.8× greater than 'Little only' and 'Big only' architectures. We observe that the big-little architecture provides *up to 7.8× improvement in energy and up to 21× improvement* in inference latency with respect to baseline homogeneous architectures. 'Big only' architecture consumes less energy and less latency than big-little architecture for ResNet-34, but in this case, the area of 'Big only' is 6.8× higher than big-little architecture. To better analyze the performance comparison, we plot the energy-delay-area product (EDAP) for all DNNs, as shown in Figure 3.4. The big-little chiplet architecture provides up to 329× lower EDAP than the 'Little only' and 'Big only' architectures across all four DNNs. Although 'Big only' architecture shows improvement in energy consumption and inference latency with respect to big-little for ResNet-34, the EDAP with 'Big only' is 1.3× higher than big-little architecture in this case. Hence, the proposed big-little IMC architecture achieves optimal performance through reduced EDAP at higher IMC utilization across different DNNs.

3.3.4 Results with DRAM (DDR4)

In this section, we show the performance results when the resource on a big-little chiplet-based system is not sufficient to store all the weights of a given DNN. In that case, the DNN is divided into multiple partitions. One partition is mapped on to the big-little chiplets at a time. While the computations of a partition of the DNN are performed, the weights corresponding to the next partition are loaded

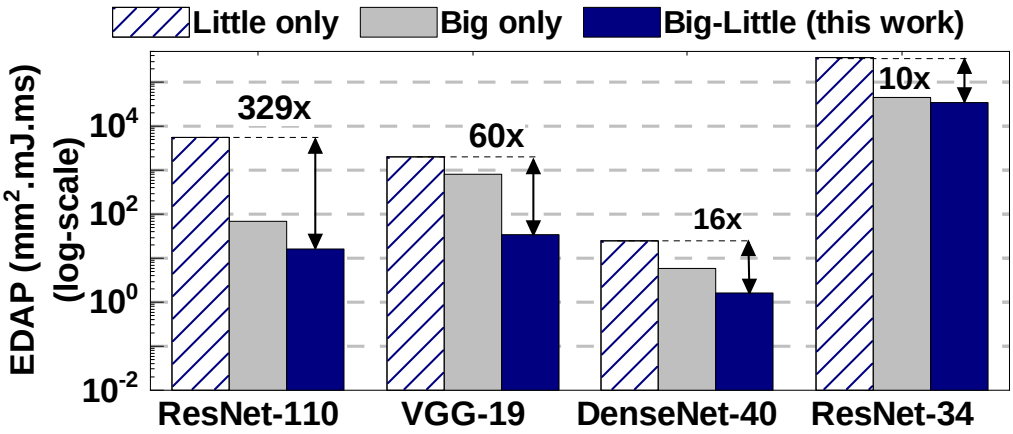

Figure 3.4: EDAP comparison (log-scale) of the big-little chiplet-based RRAM IMC architecture to 'Little only' and 'Big only' chiplet-based RRAM IMC architectures. The big-little architecture achieves up to 329× improvement compared to 'Little only' architecture.

from the DRAM into the ping-pong buffer. The additional DRAM accesses result in increased energy. At the same time, the impact on latency is reduced through the ping-pong buffers [56]. Table 3.4 shows the ratio between DRAM energy and compute energy for VGG-16 and VGG-19 with systems having different number of chiplets. We observe that, the ratio of DRAM energy to computation energy increases with reduction in the system sizes for both the DNNs. With decreasing system size, more weights need to be stored and loaded from DRAM, thereby increasing DRAM energy.

3.3.5 Comparison with State-of-the-art Work

Table 3.5 shows the comparison of the proposed heterogeneous big-little RRAM IMC chiplet architecture with an Nvidia T4 and V100 GPU, and SIMBA [1]. The

Table 3.4: Ratio between DRAM energy and compute energy for VGG-16 and VGG-19 with systems having different number of chiplets (**All weights of VGG-19 fit on chip with this configuration, significantly reducing the DRAM energy).

# Chiplets	VGG-16		VGG-19	
	#partitions	Ratio	#partitions	Ratio
36	2	1.1	1	0.08**
25	2	2.1	2	131
16	3	3.6	2	161

Table 3.5: Comparison with other platforms for ResNet-50 on ImageNet (*reported in [1]).

Platform	Area (mm^2)	Energy Efficiency (Images/s/W)
Nvidia V100 GPU*	815	8.3
Nvidia T4 GPU*	525	15.5
SIMBA [1]	215	45
Big-Little (this work)	**85**	**827**

big-little chiplet architecture achieves a lower area for the architecture due to the custom RRAM-based IMC and the optimized NoP structure. Compared to the Nvidia V100, Nvidia T4, and SIMBA architecture, the big-little IMC architecture achieves 9.6×, 6.2×, and 2.5× area improvement and 99.6×, 53.4×, and 18.4× energy-efficiency improvement, respectively. The improved energy efficiency is attributed to the higher IMC utilization, analog computation within the RRAM-based IMC, reduced NoP data movement and bus width, and the absence of intermediate DRAM transactions for weights and partial sums.

Algorithm 1 Determining Big-Little Chiplet Configuration

Input: DNN structure, number of chiplets (N_C), set of crossbars sizes for the little chiplets ($\mathcal{X}_\mathcal{L}$) and the big chiplets ($\mathcal{X}_\mathcal{B}$); set of number of tiles in the little chiplets ($\mathcal{T}_\mathcal{L}$) and the big chiplets ($\mathcal{T}_\mathcal{B}$); number of little chiplets ($\mathcal{N}_\mathcal{L}$) and number of big chiplets ($\mathcal{N}_\mathcal{B}$)
Output: Tile utilization for each configuration i (U_i)
$N_{cfg} \leftarrow$ number of configurations in the set containing all possible combinations of the elements in $\mathcal{X}_\mathcal{B}$, $\mathcal{X}_\mathcal{L}$, $\mathcal{T}_\mathcal{L}$, $\mathcal{T}_\mathcal{B}$, $\mathcal{N}_\mathcal{L}$, $\mathcal{N}_\mathcal{B}$
$L \leftarrow$ number of DNN layers
for $i = 1 : N_{cfg}$ **do**
    n_l = Number of little chiplets in Config-i
    n_b = Number of little chiplets in Config-i
    $j \leftarrow 0$ // Number of layers already mapped
    $U \leftarrow 0$ // Sum of utilization
    $n_l^u \leftarrow 0$ // Number of little chiplets used
    while $n_l^u \leqslant n_l$ and $j < L$ **do**
        $j_t \leftarrow$ Number of tiles required for layer-j
        $r_t^l \leftarrow$ Number of remaining tiles in the little chiplet
        if $j_t < r_t^l$ **then**
            Map layer-j to the little chiplet
            $u_j \leftarrow$ Tiles utilization for layer-j (Eq. 3.1)
            $U = U + u_j$;
            $j = j + 1$
        else
            $n_l^u = n_l^u + 1$
        end if
    end while
    $n_b^u \leftarrow 0$ // Number of big chiplets used
    while $n_b^u \leqslant n_b$ and $j < L$ **do**
        $j_t \leftarrow$ Number of tiles required for layer-j
        $r_t^b \leftarrow$ Number of remaining tiles in the big chiplet
        if $j_t < r_t^b$ **then**
            Map layer-j to the big chiplet
            $u_j \leftarrow$ Tiles utilization for layer-j (Eq. 3.1)
            $U = U + u_j$;
            $j = j + 1$
        else
            $n_b^u = n_b^u + 1$
        end if
    end while
    $U_i = \frac{U}{L}$
end for

Algorithm 2 Determining Big-Little NoP Configuration

Input: DNN structure, number of chiplets (N_C), set of NoP bus widths for the little chiplets ($\mathcal{W}_{\mathcal{L}}$) and the big chiplets ($\mathcal{W}_{\mathcal{B}}$); set of NoP frequency for the little chiplets ($\mathcal{F}_{\mathcal{L}}$) and the big chiplets ($\mathcal{F}_{\mathcal{B}}$), mapping of layers to the big-little chiplet ($\mathcal{L} \rightarrow \mathcal{C}$)
Output: NoP EDP for each configuration-i (E_i)
$N_{cfg} \leftarrow$ number of configurations in the set containing all possible combinations of the elements in $\mathcal{W}_{\mathcal{L}}$, $\mathcal{W}_{\mathcal{B}}$, $\mathcal{F}_{\mathcal{L}}$, $\mathcal{F}_{\mathcal{B}}$
$L \leftarrow$ number of DNN layers
n_l = Number of little chiplets
n_b = Number of big chiplets
for $i = 1 : N_{cfg}$ **do**
    w_l = Bus-width of little chiplets in Config-i
    w_b = Bus-width of big chiplets in Config-i
    f_l = NoP frequency of little chiplets in Config-i
    f_b = NoP frequency of big chiplets in Config-i
    $E_i \leftarrow 0$ // Initializing EDP of Config-i
    for $j = 1 : n_l$ **do**
        Compute edp_j by from Equation 3.2
        $E_i = E_i + edp_j$ // Communication EDP
    end for
    for $k = 1 : (n_b - 1)$ **do**
        Compute edp_k from Equation 3.2
        $E_i = E_i + edp_k$ // Communication EDP
    end for
end for

Algorithm 3 Mapping DNN Layers to Big-Little Chiplets

Input: DNN layers ($\mathcal{L}$), IMC crossbar size in Big chiplet (x_b), IMC crossbar size in little chiplet (x_l), number of tiles in big chiplets (t_b), number of tiles in little chiplets (t_l), number of available big chiplets (n_b), number of available little chiplets (n_l)
Output: Mapping of layers to of big-little chiplet ($\mathcal{L} \rightarrow \mathcal{C}$)
Compute S_B by following Equation 3.3
Compute Pr by following Equation 3.4
for $j = 1 : Pr$ **do**
    $\mathcal{L}_j \rightarrow$ DNN layers for partition-j; $\mathcal{L}_j \in \mathcal{L}$
    for $i = 1 : |\mathcal{L}_j|$ **do**
        $a_L \rightarrow 1$ // Number of little chiplets used
        Compute utilization of i^{th} (U_B^i) layer on a big chiplet using x_b, t_b
        Compute utilization of i^{th} (U_L^i) layer on a little chiplet using x_l, t_l
        if $((U_B^i < U_L^i) \& (a_L \leqslant A_L))$ **then**
            Map i^{th} layer to little chiplet.
            if Resource in a_L is exhausted **then**
                $a_L \rightarrow a_L + 1$
            end if
        else
            Compute # of big chiplets (a_B) required to map layer-i – layer-$|\mathcal{L}_j|$
            assert($(a_B \leqslant A_B)$, 'Error')
            for $k = i : |\mathcal{L}_j|$ **do**
                Map k^{th} layer to big chiplet.
            end for
            break
        end if
    end for
end for

4 ENERGY-EFFICIENT ON-CHIP TRAINING FOR CUSTOMIZED HOME-BASED REHABILITATION SYSTEMS

4.1 Home-Based Rehabilitation System

This section first overviews the proposed system. Then, Section 4.1.2 presents the proposed hardware platform used for the on-chip training and inference. Section 4.1.3 discusses the on-chip training and ground truth generation. Finally, Section 4.1.4 presents hardware implementation.

4.1.1 Overview of the Proposed PHR System

Initial Offline Training: The initial inference model, a CNN in this work, is trained offline using a limited set of users. Two video cameras and a mmWave radar detect the patients' movements during training. An *RGB-CNN* [35] model transforms the video frames to 2D joint coordinates. Then, we find the 3D joint coordinates by triangulating 2D joints from two cameras. Finally, these coordinates are used as references for supervised training of a *mmWave-CNN* [25] model that can produce joint coordinates using only the mmWave radar, as shown in Fig. 4.1.

Customization at Home: The initial *mmWave-CNN* can have a poor performance when a new patient starts using it at home, as demonstrated in Section 4.2. Therefore, we personalize it to the new user before continuous execution in three steps. First, we activate RGB cameras for a short duration and generate the 2D joint coordinates by employing the *RGB-CNN* model used in the initial training. Then,

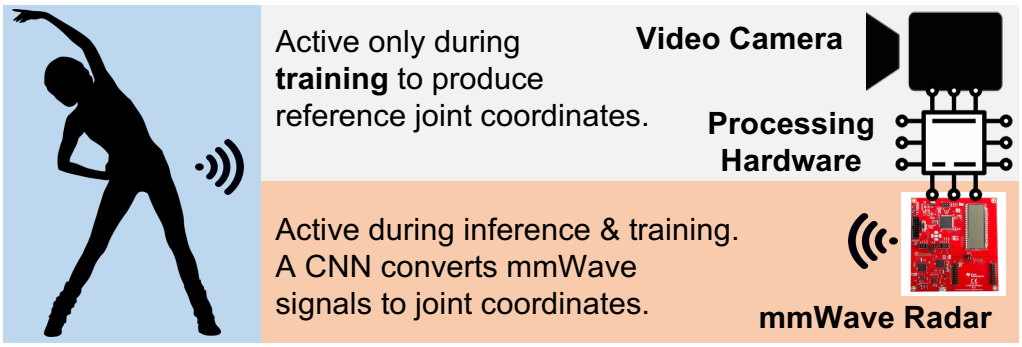

Figure 4.1: Illustration of the target rehabilitation system. The RGB camera is used *only during training* to generate the reference joint coordinates when the initial model is customized to the target user. Once the model that uses mmWave signals is trained, only the mmWave radar is used for inference.

the 3D joint coordinates are found in the same way as the initial training. Finally, we incrementally train the *mmWave-CNN* using these reference coordinates and mmWave signals as inputs. After the customization, the new user uses the product only with mmWave signals and the inference of *mmWave-CNN* model for rehabilitation feedback. Running *RGB-CNN* for RGB image inference and *mmWave-CNN* for both on-chip training and inference are not feasible on a conventional SoC, as demonstrated in Section 4.2. Therefore, we propose an IMC-based hardware accelerator to perform these tasks.

In summary, the proposed PHR pipeline consists of (i) taking RGB images with two cameras, (ii) inferring 2D human key points from the images using *RGB-CNN*, (iii) obtaining ground truth 3D human joints by triangulating two 2D human joints, and (iv) training and inferring the human joints with mmWave signals using *mmWave-CNN*.

4.1.2 In-Memory Computing-based Hardware Acceleration

We employ an in-memory computing (IMC)-based hardware accelerator because it combines highly-dense storage and computation into the same hardware unit. It also alleviates the latency and energy consumption of memory accesses. Energy-efficient DNN accelerators utilizing IMC technology [3, 1, 6] have been proposed in the literature in the last few years. IMC-based architectures integrate multiple processing units, called tiles, into the system, as shown in Fig. 4.2. These tiles are connected via a network-on-chip (NoC) for the data communication of activations, errors, and gradients. Each tile has processing elements along with input/output buffers and an accumulator. Processing elements are composed of crossbar arrays that store the neural network model weights and perform computations and peripheral circuits such as ADCs, adder trees, and buffers. This work uses the mapping methodology described in [41] to map the weights onto the crossbar arrays. We used a ReRAM-based IMC architecture for inference and training. We also include an SRAM-based hardware block for training to avoid excessive writes to the ReRAM crossbar arrays because of the write endurance issues of ReRAM. The IMC architecture also includes activation units, buffers, and accumulators. More detailed discussions about the on-chip training on IMC accelerators and hardware configurations are provided in Section 4.2.

4.1.3 On-Chip Training for Personalization of *mmWave-CNN*

To customize the inference model, *mmWave-CNN*, to a new user, we first need the ground truth joint coordinates. We use two RGB cameras on the edge device to get

precise 3D joint coordinates, as described in Section 4.1.1 and Fig. 4.1. The reason for utilizing two RGB cameras is that we can generate the ground truth with minimum error with two RGB images. Two RGB images are then processed by the *RGB-CNN* model. For the *RGB-CNN* model, we followed the structure given in [35], which has 28.5M parameters with a backbone pre-trained on the ImageNet dataset. Then, we generate 3D human joint coordinates by triangulating a pair of 2D human joints. The mmWave radar sensor provides inputs to *mmWave-CNN* model. The *mmWave-CNN* [25] is composed of 3.2M parameters with two convolutional layers followed by two fully connected layers. The radar sensor generates a point cloud which is composed of points reflected from an object in sight. For each point, the radar sensor generates five features which will be used as inputs for the *mmWave-CNN* model. These are x, y, and z coordinates, the Doppler velocity, and the reflection intensity. Then, using the radar data, we customize the baseline *mmWave-CNN* model to the new user's body pose with the on-chip training on the IMC hardware accelerator. The training consists of four parts: feedforward, error calculation, weight gradient calculation, and weight update. Each step requires DRAM access to store and load the necessary data, as there are thousands of input data. The accuracy and on-chip training results are discussed in Section 4.2.3 and 4.2.5. After the on-chip training, the *mmWave-CNN* is customized for the new user. Finally, the last step in the framework is the customized *mmWave-CNN* model inference that is performed on the IMC accelerator using the tiles of crossbar arrays.

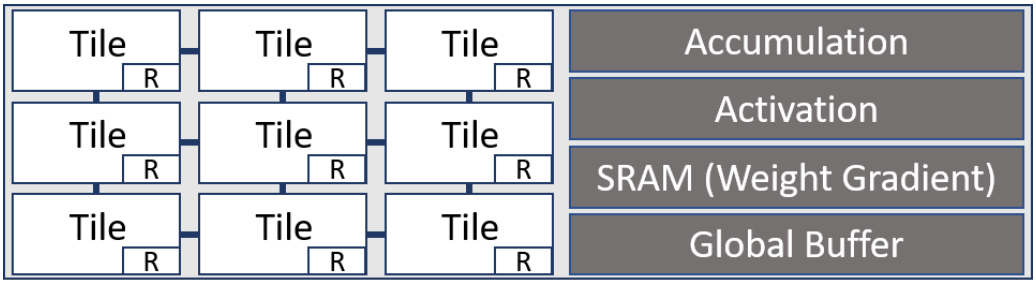

Figure 4.2: The architecture of IMC-based hardware accelerator. Feedforward, error calculation, and weight update stages are performed in the accelerator tiles whereas the weight gradient calculation is executed in the weight gradient block. Tiles are connected via NoC. (R: NoC Router)

4.1.4 Hardware Implementation and Exploration

The crossbar array sizes in each tile affect the structure and performance of the IMC hardware accelerator. Smaller crossbar arrays increase the number of tiles, resulting in traffic congestion in the NoC because of high data movement. In contrast, larger array sizes reduce the utilization of the IMC accelerator. Thus, it causes an area overhead compared to small crossbar sizes. Therefore, we explore the hardware parameters to achieve optimal energy and area efficiency, as discussed in Section 4.2.5.

ReRAM-based architectures are vulnerable to the write endurance problem and nonlinear properties. These properties include device-to-device (D2D) and cycle-to-cycle (C2C) variations, long-term potentiation (LTP), and long-term depression (LTD) [39]. Therefore, we analyzed the nonlinear properties of ReRAM-based IMC hardware accelerator design. Detailed explorations of nonlinear properties, variations, and their effects on the accuracy are given in Section 4.2.5. The on-chip training for the customization is an incremental training approach that requires

less number of epochs, thus the lower number of writes on the crossbar arrays. Therefore, our framework can achieve better accuracy without assuming limitless writes.

We use the following parameters in our IMC-based accelerator design: an ADC precision of 4 bits, 8-bit quantization for weights and activations, one bit per ReRAM cell, an R_{off}/R_{on} ratio of 100, and 1 GHz operating frequency, and 32 nm technology node [3]. We employ NeuroSim V2.1 [41] to evaluate the performance and area of the proposed ReRAM-based IMC hardware accelerator and data communication between the main memory and the accelerator. NeuroSim V2.1 supports both inference and training on-chip. Only the weight gradient calculation part requires extra hardware (SRAM) for the on-chip training because this stage needs heavy writing of errors into the arrays along the batch. Feedforward, error calculation, and weight update stages are implemented on the ReRAM tiles. Data movement within the accelerator between tiles via an NoC is evaluated using a customized version of BookSim [2]. In this version, we use a traffic trace-based cycle-accurate execution using a mesh NoC architecture.

4.2 Experimental Results

4.2.1 Experimental setup

We conduct our experiments with the mRI open-source mmWave human pose estimation dataset [26]. mRI offers over 160K synchronized 3D point cloud from mmWave radar (TI IWR1443) [66], and ground truth 2D and 3D human joints

from two Kinect V2 cameras [67]. It evaluates ten clinical-suggested rehabilitation movements covering the main parts of the human body, performed by twenty diverse subjects. The dataset benchmarks mmWave-based human pose estimation using HRNet-W32 [35] (images to 2D key points) and MARS [25] (mmWave point cloud to 3D human joints). HRNet-W32 and MARS generate 17 human joint points in 2D and 3D space, respectively. We implemented these CNN-based networks as the *RGB-CNN* and *mmWave-CNN* models using PyTorch. For the initial training, we used an SGD optimizer with momentum ($\beta = 0.9$), and an initial learning rate of 0.001 for 100 epochs with a batch size of 128.

4.2.2 Baseline Accuracy before Customization

We trained the baseline MARS model using half of the user subjects in the dataset. To produce accurate results, we generated three random sets given in Table 4.1. Then, the trained baseline model is tested against the remaining ten subjects for each set. We used *Mean Per Joint Point Error* (MPJPE) and *Procrustes Analysis MPJPE* (PA-MPJPE) metrics, which are widely used for human joint estimation studies [68]. MPJPE calculates the mean Euclidean distance between the reference

Table 4.1: Random set configurations for experimental evaluations and training results of the baseline *mmWave-CNN* model.

Sets	Training Subjects	Test Subjects	**MPJPE (mm)**	**PA-MPJPE (mm)**
1	1-3,7,9,14,16,17,19,20	4-6,8,10-13,15,18	130.7 ± 3.4	76.4 ± 1.4
2	1,2,5-7,9,13,17-19	3,4,8,10-12,14-16,20	133.5 ± 2.5	78.4 ± 1.9
3	3,5,6,8,10-12,14,18,20	1,2,4,7,9,13,15-17,19	134.5 ± 1.2	78.5 ± 1.1

and the prediction joints. PA-MPJPE uses a similarity transformation for a further rigid alignment.

The *mmWave-CNN* model achieves 130.7 mm MPJPE and 76.4 mm PA-MPJPE for the test subjects in *Set-1*. In contrast, the corresponding errors for the subjects in the training set are 97.8 mm and 57.1 mm, which is about 25% lower. The results are similar for other sets with slightly over 130.7 mm MPJPE and 76.4 mm PA-MPJPE for the test subjects, as shown in Table 4.1. These results show that the initial model can achieve very high accuracy for the known subjects, but the accuracy degrades for new users. Even if one can improve the test accuracy by adding users to the training set, it is impractical to add all potential users who will buy the rehabilitation system. Therefore, *enabling incremental online training at the edge is imperative to adapt the initial model to new users*.

4.2.3 Test Accuracy after Customizing to New Users

On-device training for customization comprises three steps:

1. The video camera is activated for a short period (2.5 minutes, i.e., 4500 frames),
2. The *RGB-CNN* model inference generates the ground truth joint coordinates used for supervision,
3. The baseline *mmWave-CNN* model is incrementally trained using the point cloud data from the mmWave radar sensor as input and the ground truth joint coordinates from step 2 as labels.

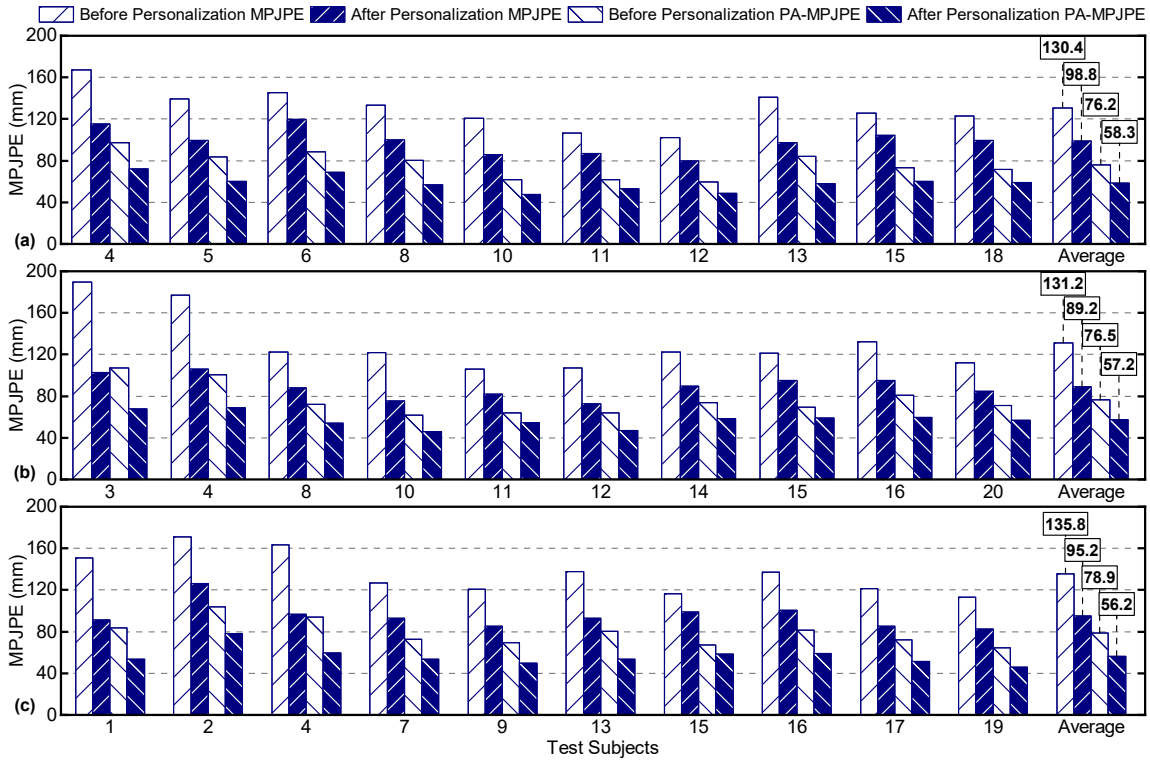

Figure 4.3: MPJPE and PA-MPJPE comparisons for all three random sets. Results show MPJPE and PA MPJPE before customization using 10 subjects for training and after customization which is customized for each test subject separately. Parts (a), (b), and (c) represent *Set-1*, *Set-2*, and *Set-3* results, respectively. As they are randomly split, each plot shows the results for different subjects.

We employ the three random sets used for baseline accuracy analysis (Table 4.1) to evaluate the proposed on-device learning technique. Specifically, we customize the baseline model for each subject in the test subjects one-by-one. Fig. 4.3 compares the accuracy of the personalized model to the baseline model. Both MPJPE and PA-MPJPE results improve significantly for all users in *Set-1*, as shown in Fig. 4.3(a). On average, the customization improves the MPJPE and PA-MPJPE by 23.89% and 22.94%, respectively. Most importantly, the average MPJPE and PA-MPJPE reduce

Table 4.2: Hardware results for *mmWave-CNN* model inference and training on Jetson Xavier NX with 2 configurations and our framework with 2 configurations and the speedup comparisons. 128×128 and 256×256 represent the crossbar array sizes. (P: PHR, J: Jetson)

	Configurations	Jetson-1 (2 CPUs)	Jetson-2 (6 CPUs)	PHR-1 (128x128)	Improvement (×) P-1 vs J-1	Improvement (×) P-1 vs J-2	PHR-2 (256x256)	Improvement (×) P-2 vs J-1	Improvement (×) P-2 vs J-2
Inference	Power (mW)	3379.0	8724.0	12.1	**277.8**	**717.3**	25.7	131.4	339.2
	Time per frame (µs)	162.2	149.7	10.8	15.0	13.8	4.2	**38.5**	**35.5**
	Energy per frame (µJ)	548.0	1305.9	1.1	488.8	1162.2	0.9	**611.1**	**1452.7**
Training	Power (mW)	9761.4	9986.0	2954.5	**3.3**	**3.4**	4223.3	2.3	2.4
	Time per frame (µs)	306.9	299.4	32.3	9.5	9.2	21.8	**14.0**	**13.7**
	Energy per frame (µJ)	2995.4	2989.6	95.6	31.3	31.2	92.3	**32.4**	**32.3**

to 98.8 mm and 58.3 mm, within only 2 mm of their training accuracy. Fig. 4.3(b) and Fig. 4.3(c) show that these improvements are observed for *Set-2* and *Set-3* as well. The MPJPE improvement ranges from 17.3 mm (14.92%) to 87.1 mm (45.82%), while PA-MPJPE improvement is between 9.0 mm (13.22%) to 39.5 mm (36.83%). In conclusion, the customized model performs consistently and significantly better than the baseline model for all subjects in each random set, enabling accurate feedback necessary for home-based rehabilitation.

4.2.4 Energy and Performance Results

Home-based rehabilitation systems should run on a compact edge device for the practicality of the approach. Therefore, we implemented the baseline model on an Nvidia Jetson Xavier NX [30] board for hardware performance comparisons. The Jetson board has 6 Carmel Arm CPU cores, an Nvidia GPU with 384 CUDA cores, and 48 tensor units. We compared the hardware results of our framework against the Jetson board for both training and inference.

Model Inference: We used two different hardware configurations for performance

comparisons. *Jetson-1* and *Jetson-2* shown in Table 4.2 have 2 and 6 active CPU cores (1.9 GHz) and GPU (1.1 GHz) on the board for processing, respectively. The Jetson board's power consumption is on the Watts scale (3.38 and 8.7 W). The latency per frame varies between 149.7 μs and 162.2 μs. The lower energy consumption per frame comes from *Jetson-1* with 548 μJ as it utilizes 2 CPU cores.

Similarly, we evaluated the proposed IMC accelerator using two configurations. The crossbar sizes are selected as 128 $\times$ 128 and 256 $\times$ 256 for *PHR-1*, and *PHR-2*, respectively. The detailed hardware exploration is discussed in Section 4.2.5. Our accelerator's power consumption is on the scale of milliwatts (12.1 to 25.7 mW) which results in improvements up to 717.3$\times$ compared to the Jetson board. The latency per frame are 10.8 μs and 4.2 μs for *PHR-1* and *PHR-2*, respectively. When we compare against Jetson configurations, *PHR-2* shows a higher speedup with 38.5$\times$ and 35.5$\times$ against *Jetson-1* and *Jetson-2*, respectively. We see considerable improvements in energy consumption as IMC accelerators are highly energy-efficient. The energy consumption per frame of PHR is 1.1 and 0.9 μJ for *PHR-1* and *PHR-2*, respectively. *PHR-2* has a greater improvement in energy consumption with 611.1$\times$ and 1452.7$\times$ compared against *Jetson-1* and *Jetson-2*, respectively, as it has less number of tiles and less NoC energy.

Model Training: Table 2 also shows the model training results of our proposed framework and Jetson configurations. We include two sets of results, one per epoch and one per frame. The latency per epoch for *Jetson-1* and *Jetson-2* are measured as 1.37 and 1.34 seconds, whereas *PHR-1* and *PHR-2* achieve 0.14 and 0.09 seconds, respectively. *PHR-2* shows a higher speedup of 14.0$\times$ and 13.7$\times$

against *Jetson-1* and *Jetson-2* configurations. The energy consumption of *Jetson-1* and *Jetson-2* configurations are both 13.4 J. *PHR-1* and *PHR-2* achieve 0.43 J and 0.41 J, respectively, which results in an improvement of 31.3× and 31.2× for *PHR-1* and 32.4× and 32.3× against *Jetson-1* and *Jetson-2*, respectively. The energy consumption reduction of training is lower than that of inference. The reason is the high DRAM access required in the training and the gradient calculation performed in the IMC accelerator's SRAM block. SRAM IMC consumes higher energy than the ReRAM crossbar array but because of the write endurance problem seen in ReRAM crossbar arrays, writing too much data on ReRAM is not practical. *PHR-1* achieves an improvement of 3.3× and 3.4× against *Jetson-1* and *Jetson-2* configurations, respectively while *PHR-2* achieves an improvement of 32.3× and 2.4× against *Jetson-1* and *Jetson-2* configurations, respectively.

Our framework also utilizes HRNet-W32 as the *RGB-CNN* model for the ground truth joint coordinate generation. Table 4.3 compares the results on the Nvidia Jetson board to the proposed framework. The energy consumption per frame is 9.1 J with a latency of 622.2 ms for Jetson, while they are 0.05 J and 9.7 ms for PHR. The energy consumption and latency of *RGB-CNN* inference are higher than *mmWave-CNN* model because HRNet-W32 is a deeper network with densely connected layers. It requires more DRAM accesses for data movement as all the network weights do not fit in the accelerator. Despite this, it can provide sufficient performance to process at runtime with over 30 FPS.

Table 4.3: Hardware results for *RGB-CNN* inference on Jetson Xavier NX with 6 CPU cores and our framework with 256 × 256 crossbars.

	Jetson	**PHR**	**Improvement (×)**
Power (W)	14.5	0.4	**34.8**
Time per frame (ms)	622.2	9.7	**64.1**
Energy per frame (J)	9.1	0.05	**782**

4.2.5 Hardware Architecture Exploration

Crossbar Array Size: We next analyze the effect of the crossbar sizes in each tile. Previous sections considered only 128 × 128 and 256 × 256 array sizes. When we reduce the size to 64 × 64, the inference latency (356.7 µs) becomes higher than the Jetson board latency, making the configuration impractical. At the same time, an array size larger than 256 × 256 reduces the utilization to less than 50% and also increases the area overhead. Therefore, we decided to use only *PHR-1* and

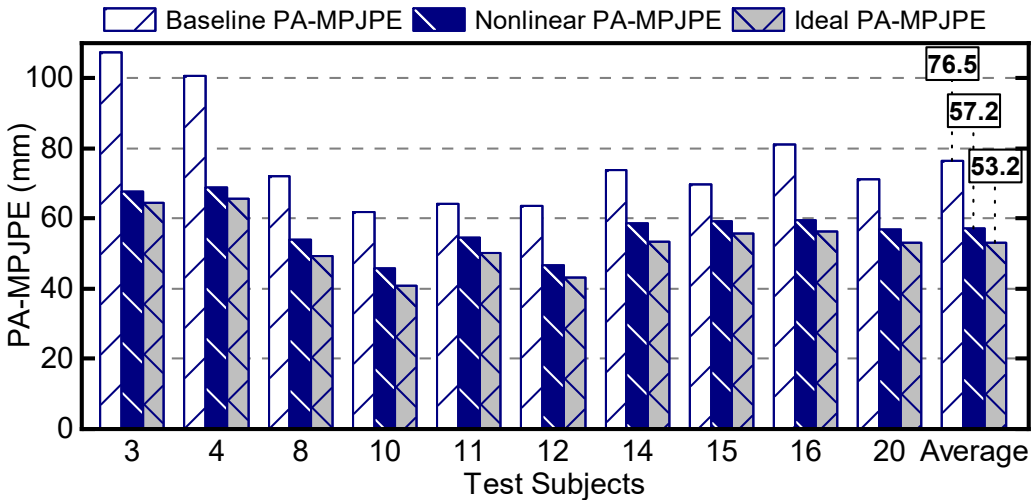

Figure 4.4: PA-MPJPE comparisons for the baseline model (*Baseline*), a customized model with nonlinear properties (*Nonlinear*), and a customized model without nonlinear properties (*Ideal*) for 10 test subjects from *Set-2*.

PHR-2 configurations which have 128 $\times$ 128 and 256 $\times$ 256 crossbar array sizes in our evaluations.

Nonlinear Properties of ReRAM: Our analysis assumes a C2C variation of 2%, D2D variation of 0.1%, and nonlinearity of 0.5/-0.5 following [38]. An ideal case would have no variations and nonlinearities. Fig. 4.4 compares the PA-MPJPE with the selected nonlinear properties and the ideal case for user *Set-2*. On average, we see an improvement of 24.3% and 29.8% against the baseline model for *Nonlinear* and *Ideal* cases, respectively. The difference between the cases varies between 3.1% and 8.1%, with an average of 5.5% for ten subjects. Note that even with the nonlinear properties of ReRAM, we can achieve significant improvements compared to the baseline model after the customization.

5 PROPOSED WORK – 1: COMMUNICATION-AWARE SPARSE NEURAL NETWORK OPTIMIZATION

Deep neural networks (DNNs) exhibit a high degree of redundancy due to dense interconnections between successive layers. Besides posing overfitting risks, redundant connections increase the communication cost and implementation overhead, thus leading to lower performance and energy efficiency when implemented in hardware. Indeed, many pruning techniques aim at removing DNN connections with minimal impact on their accuracy [69, 70, 71]. Sparse neural networks are preferred since they can enable minimal communication and implementation overhead, thus significantly reducing the computation and memory requirements.

Sparse inter-layer connections enable significantly faster and more energy-efficient DNNs. However, sparsity alone is *not* sufficient since good algorithmic performance *does not* necessarily translate into real performance on hardware. For instance, some inter-layer connections can lead to long paths when mapped on hardware. Consequently, they can undermine the overall hardware performance due to high communication latency and energy costs. For example, the sparse evolutionary training (SET) approach [72] drastically decreases the training time using sparse graphs instead of pruning a trained network; this makes the training scalable, while improving the test accuracy on a wide range of datasets, including multi-layer perceptron (MLP) and convolutional neural networks (CNNs) for unsupervised and supervised learning.

Although it can achieve higher accuracies, the networks remain oblivious to

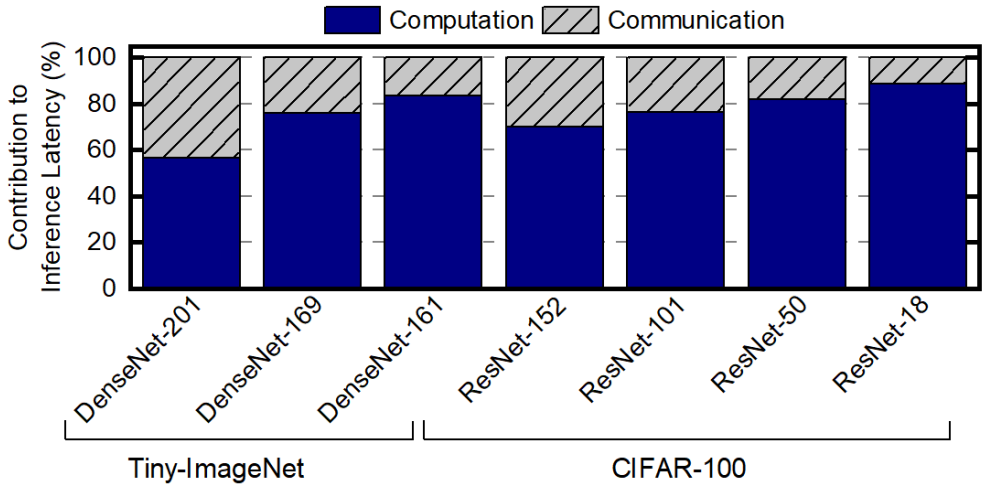

Figure 5.1: Percentage contribution to inference latency for various networks on two datasets. The communication latency can take up to 43% of the total inference latency.

the real hardware. The performance of DNNs on real hardware is critical since it determines the inference latency and power consumption. For example, a DNN targeting real-time applications, such as autonomous driving, may become impractical if the inference latency violates the timing constraints. To analyze the inference latency, we perform experiments using an in-memory computing (IMC)-based DNN accelerator where the inter-layer communication for activation data movement is implemented via a network-on-chip (NoC). We use a state-of-the-art reinforcement learning based mapping algorithm [73] with unpruned networks using two different datasets. Our evaluations show that the communication between the DNN layers alone can take up to 43% of the total inference latency for a wide range of DNNs, as depicted in Figure 5.1.

Although sparse training can achieve a higher accuracy than a network with no

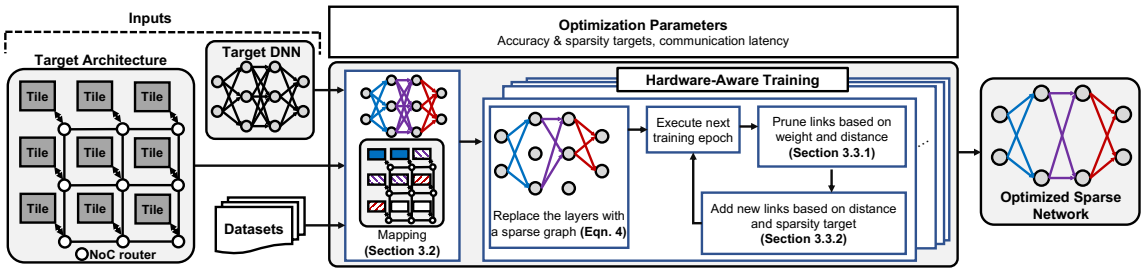

Figure 5.2: Overview of the proposed approach. It consists of mapping the target DNN onto the target architecture using latency-aware mapping and hardware-aware dynamic sparse training. The training process first replaces the DNN layers with sparse graphs; then, at the end of each epoch, employs hardware-aware pruning and link addition. Each circle in the target DNN represents the feature map of DNNs; each link in the target DNN represents the weights of DNNs. The weights are mapped onto the in-memory computing (IMC) tiles with the same color as the corresponding links. The circles and the rectangles in the target architecture denote the NoC routers and IMC tiles, respectively.

pruned links [74], if the network remains oblivious to the target hardware while pruning and adding links, then, this can lead to unacceptable latency and power overheads. Therefore, there is a strong need for *hardware and communication-aware sparse training* methodologies that can lead to shorter communication distances when the DNN layers are mapped onto hardware resources.

Starting from these observations, we propose a novel communication-aware sparse neural network optimization technique applicable to both fully connected and convolutional layers. The first step of the proposed technique maps the DNN layers on the target hardware resources, e.g., to the processing tiles of an NoC. Our proposed mapping technique minimizes the distance the packets between two consecutive DNN layers need to travel in the NoC which helps reducing the overall communication latency. The second step performs hardware-aware dynamic

sparse training. Suppose two nodes in the DNN are connected by links with non-zero weights. If these DNN nodes are mapped onto different tiles on the NoC, the activations generated between these nodes will incur communication costs during inference. The proposed technique prunes the p−percent of the weights based on the significance of weight columns towards any inference decision per unit communication cost. Finally, we maintain the target sparsity throughout the training process by choosing an equal number of weight columns (p−percent) with the smallest communication cost and adding them back to the network at the end of each epoch.

6 PROPOSED WORK – 2: CARBON FOOTPRINT OPTIMIZATION

The artificial intelligence (AI) era is rising at a pace not seen before. The machine learning model sizes and required computations increased five orders of magnitude from 2018 to 2022, and we expect to see this trend continue. The increase in computation resulted in high energy consumption and resource usage, which led to an unsustainable environment and a high carbon footprint. Considering the trend in AI computing, National Science Foundation started a program that targets a sustainable computing program in 2022. Sustainable AI is a promising solution that targets environmentally sustainable models by measuring and reducing the carbon footprint of the development and production of AI models.

For sustainable AI, developing tools is crucial to detect, model, and alleviate carbon footprint activity. Existing carbon footprint tools [75, 76, 77, 78, 79, 80, 81] monitor central processing units (CPU), graphical processing units (GPU), and memory consumptions. However, in recent years, custom hardware designs like machine learning accelerators have been commonly used for AI models' training and inference processes because of their superior performance and energy efficiency. These accelerators have been used in various systems, from servers to edge devices. For example, on-device learning using accelerators gets interest as it improves data privacy [82]. On the other hand, using federated learning for a small ML model can have a comparable carbon footprint with orders of magnitude bigger transformer models [83]. Therefore, it is also essential to reduce the carbon footprint at the edge. Nevertheless, the lack of carbon footprint tools for hardware accelerators

prevents researchers from analyzing the behavior of the hardware and AI models in the development and deployment processes.

Current tools use performance counters to track the CPU, GPU, and memory activity. These counters measure the number of memory accesses, energy consumption, and execution time. Measurements are executed in real-time and have a negligible overhead on the runtime of the process and energy consumption. However, they are not compatible with simulation tools widely used for designing custom hardware. Simulation tools enable accurate and fast design space exploration before realizing the hardware design. Tracking carbon footprint in this stage is significant because, after the tape-out of the design, it is challenging to change the behavior of the hardware to adjust to the carbon footprint activity. Therefore, tracking custom hardware's carbon footprint using simulators is critical for the environment and society.

In addition to tracking carbon footprints, it is essential to implement strategies for reducing them. Data centers, for example, strive to offset their operational carbon footprint by incorporating renewable energy sources. On the other hand, in the realm of ML models, several techniques have been employed to minimize resource and operational consumption. One such approach is model layer freezing, which aims to decrease memory usage and training time during ML model training by selectively freezing specific layers and excluding them from the backpropagation process. This technique helps conserve resources and optimize the training process. However, its impact on the carbon footprint is overlooked.

We propose integrating a carbon footprint tracker with an in-memory comput-

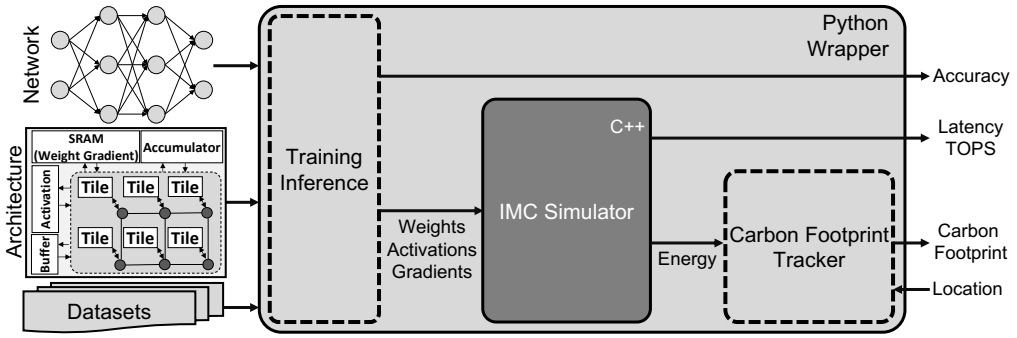

Figure 6.1: Overview of the proposed framework. Inputs to the framework are the target network, target architecture, dataset, and location. First, the training or inference part is performed in the Python wrapper using the PyTorch library. This wrapper outputs the accuracy based on the quantization. Then, quantized weights, activations, and gradients are sent to the IMC simulator. This simulator outputs latency and TOPS. Then, energy consumption and location information are used by the carbon footprint tracker for each epoch, outputting the carbon footprint.

ing (IMC) based machine learning accelerator simulator. This simulator tracks the custom hardware's carbon footprint and electricity consumption for training and inference of machine learning models. Using this tool, the hardware designers can analyze the carbon footprint performance of the custom hardware in the early design stage by varying the hardware configuration. On top of carbon footprint, our tool also provides other hardware metrics like latency, energy consumption, TOPS, and area that helps the design of the hardware. Furthermore, we integrate various AI models for sustainability analyses of different aspects of machine learning algorithms. Machine learning model developers can analyze the training and inference behavior of the AI models they develop and try to reduce the design's carbon footprint for deployment. We evaluate the platform for popular CNN models regarding carbon footprint with various metrics like quantization, hardware parameters, and layer freezing to show that designing sustainable and robust AI

models with energy-efficient machine-learning accelerators is possible. Moreover, we propose a carbon footprint optimization algorithm that utilizes a characterization procedure that shows the network behavior for various freezing scenarios. This characterization procedure constitutes the algorithm's backbone and guides the algorithm's search. The operation of the optimization framework can be summarized in the following steps. *First*, we perform trial training runs where we freeze particular layers at specific epochs, considering the vast design space. *Second*, we fit curves to interpolate the freezing data for epochs where information is missing, completing the overall understanding. *Third*, we provide the data obtained from the characterization process to the algorithm, which explores the design space to identify the optimal freezing epochs that minimize carbon footprint while maintaining negligible accuracy loss. Furthermore, we discussed the potential ways to minimize the carbon footprint of custom hardware.

7 OTHER WORK: DAS: DYNAMIC ADAPTIVE SCHEDULING FOR ENERGY-EFFICIENT HETEROGENEOUS SOCS

7.1 Dynamic Adaptive Scheduling Framework

7.1.1 Overview and Preliminaries

This work considers streaming applications that can be mod- eled by a data flow graph (DFG). Consecutive data frames are pipelined through the tasks in the graph. Unlike the current practice, which is limited to a single scheduler, DAS allows the OS to choose one scheduling policy $\pi \in \Pi_S = \{F, S\}$, where F and S refer to the *fast* and *slow* schedulers, respectively. Once the predecessors of a task are completed, the OS can call either a fast ($\pi = F$) or a slow scheduler ($\pi = S$) as a function of the system state and workload. The OS collects a set of performance counters during the workload execution to enable two aspects for the DAS framework: (1) precise assessment of the system state, (2) desirable features for the classifier to dynamically switch between the *fast* and *slow* schedulers.

Table 7.1: Type of performance counters used by DAS framework

Type	Features
Task	Task ID, Execution time, Power consumption, Depth of task in DFG, Application ID, Predecessor task ID and cluster IDs, Application type
Processing Element (PE)	Earliest time when PE is ready to execute, Earliest availability time of each cluster, PE utilization, Communication cost
System	Input data rate

Table 7.1 presents the performance counters collected by DAS. For a DSSoC with 19 PEs, it uses 62 counters. The goal of the fast scheduler F is to approach the theoretically minimum (i.e., zero) scheduling overhead by making decisions in a few cycles with a minimum number of operations. In contrast, the slow scheduler S aims to handle more complex scenarios when the task wait times dominate the execution times. *The goal of DAS is to outperform both underlying schedulers in terms of execution time and EDP by dynamically switching between them as a function of system state and workload.*

7.1.2 Zero-Delay DAS Preselection Classifier

The first step of DAS is selecting the fast or slow scheduler. Since this decision is on the critical path of the fast scheduler, we must optimize it to approach our zero overhead goal. One of the novel contributions of DAS is recognizing this selection as a deterministic task that will eventually be executed with probability one. Hence, we prefetch the relevant features required for this decision to a pre-allocated local register. To minimize the overhead, we re-use a subset of counters shown in Table 7.1 to make this decision, discussed in Section 7.2.2.

The OS periodically refreshes the performance counters to reflect the current system state. Each time the features are refreshed, DAS preselection classifier updates its scheduler selection which will be used for the next ready task. This decision will always be up to date since it uses the features that reflect the most recent system state. This way, DAS determines which scheduler should be called even before a task is ready for scheduling. Hence, the preselection classifier introduces zero

latency and minimal energy overhead, as described next.

Offline Classifier Design: The first step to design the pre- selection classifier is generating the training data based on the domain applications known at design time. Each scenario in the training data consists of concurrent applications and their respective data rates (e.g., a combination of WiFi transmitter and receiver chains, at a specific upload and download speed). To this end, we run each scenario twice, as described in Fig. 7.1.

First Execution: The instrumentation enables us to run *both fast and slow schedulers* each time a task scheduling decision is made. If the decisions of the fast (D_F) and slow (D_S) schedulers for a task T_i are identical, then we label task T_i with F (i.e., the *fast scheduler*) and store the recent performance counters. If the schedulers return different decisions, then the label is left as *pending*, and the execution continues by

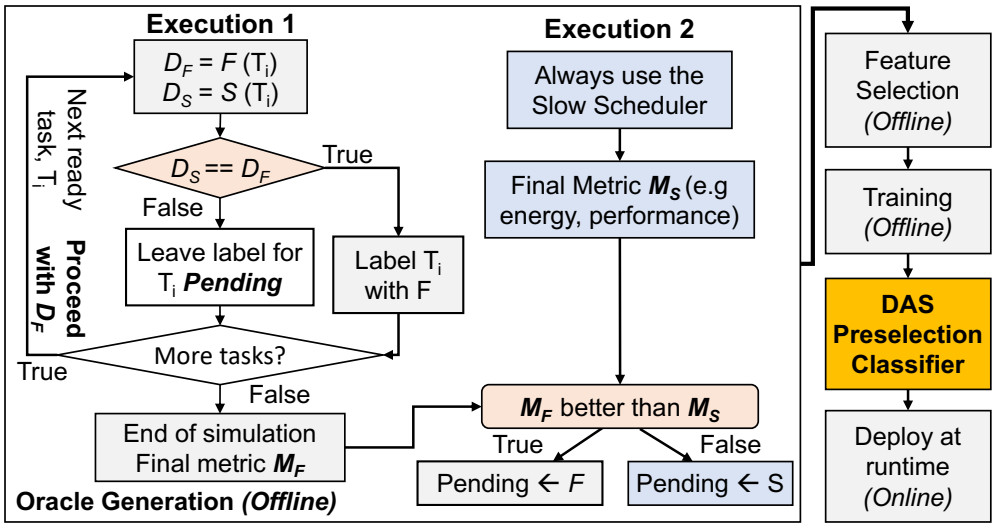

Figure 7.1: Flowchart describing the flow of the DAS framework: Oracle generation, feature selection, and training a model for the classifier.

following D_F, as shown in Fig. 7.1. At the end, the training data contains a mixture of both labeled (F) and pending decisions.

Second Execution: The same scenario is executed, *this time by always following* D_S. At the end, we analyze the target metric, such as the execution time. If the slow scheduler achieves a better result, the pending labels are replaced with S to indicate that the slow scheduler is preferred despite its larger overhead. Otherwise, we conclude that the fast scheduler's lower overhead pays off and replace the pending labels with F. An alternative approach is evaluating each pending label individually. However, this approach will not be scalable since the decision at time t_k affects the remaining execution.

We generate training data using these two runs as described in Section 7.2.1. Then, we design a low-overhead classifier using machine learning and feature selection techniques to select the fast or slow scheduler at runtime, as shown in Fig. 7.1.

Online Use of the Classifier: At runtime, a background pro- cess periodically updates a preallocated local memory with *a small subset of performance counters* required by the classifier. After each update, the classifier determines whether the fast *F* or slow *S* scheduler should be used for the next available task. When a new ready task becomes available, we know which scheduler is a better choice. Therefore, DAS does not incur any extra delay on the critical path. Moreover, it has a negligible energy overhead, as demonstrated in Section 7.2.

7.1.3 Fast & Slow (Sophisticated) (F&S) Schedulers

The DAS framework can work with any choice of fast and slow scheduling algorithms. This work uses a LUT implementation as the fast scheduler since the goal of the fast scheduler is to achieve almost zero overhead. The LUT stores the most energy-efficient processor in the system for each known task in the target domain. Unknown tasks are mapped to the next available CPU core. Hence, the only extra delay on the critical path and overhead is the LUT access. To profile the overhead, we developed an implementation using C with inline assembly code. Experiments show that *our fast scheduler takes ~7.2 cycles (6 ns on Arm Cortex-A53 at 1.2 GHz) on average and incurs negligible (2.3 nJ) energy overhead.*

This work uses a commonly used heuristic, earliest task first (ETF), as the slow scheduler [84]. ETF is chosen since it performs a comprehensive search to make a decision when the SoC is loaded with many tasks. It recursively iter- ates over the ready tasks and processors to find the schedule with the fastest finish time, as shown in Algorithm 4. Hence, its computational complexity is quadratic on the

Algorithm 4 ETF Scheduler

```
1: while ready queue 𝒯 is not empty do
2:     for task T_i ∈ 𝒯 do
3:         // 𝒫 = set of PEs
4:         for PE p_j ∈ 𝒫 do
5:             FT_{T_i,p_j} = Compute the finish time of T_i on p_j
6:         end for
7:     end for
8:     (T′, p′) = Find the task & PE pair that has the minimum FT
9:     Assign task T′ to PE p′
10: end while
```

number of ready tasks.

7.2 Experimental Evaluations

7.2.1 Experimental Setup

Domain Applications: The DAS framework is evalu- ated using five real-world streaming applications: 1) range detection; 2) temporal mitigation; 3) WiFi-transmitter; 4) WiFi-receiver applications; and 5) a proprietary industrial application (*App-1*) [84, 85]. We construct 40 different workloads by mixing applications in different ratios for our evaluations. More information is provided in our GitHub [84].

Emulation Environment: One of our key goals in this study is to conduct a realistic energy and runtime overhead analy- sis. For this purpose, we leverage an open-source Linux-based emulation framework [85]. For our analysis, we incorporate LUT and ETF schedulers into this emulation environment. We generate a wide range of workloads – ranging from all application instances belonging to a single application to a uniform distribution from all five applications. We measure the trend between the number of ready tasks and the scheduling overhead of ETF on the Xilinx Zynq ZCU102. Based on these measurements, we formulate the ETF scheduling overhead using a quadratic equation to evaluate the DAS scheduler.

Simulation Environment: We use DS3 [84], an open-source DSSoC simulation framework, for the detailed evaluation of DAS. DS3 includes built-in scheduling algorithms, models for PEs, interconnect, and memory systems. The framework has been validated with Xilinx Zynq ZCU102 and Odroid-XU3.

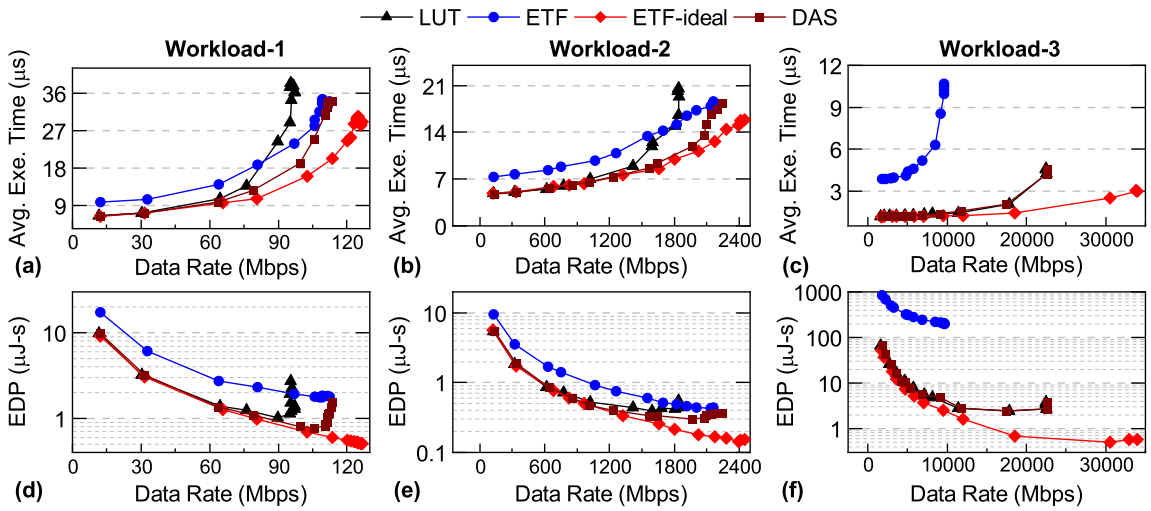

Figure 7.2: Comparison of (a)–(c) average execution time and (d)–(f) EDP between DAS, LUT, ETF, and ETF-ideal for three different workloads.

DSSoC Configuration: We construct a DSSoC configuration that comprises clusters of general-purpose cores and hardware accelerators. The application domains used in this study are wireless communications and radar systems. The DSSoC used in our experiments uses the Arm big.LITTLE architecture with 4 cores each. We also include dedicated accelerators for fast Fourier transform (FFT), forward error correction (FEC), finite impulse response (FIR), and a systolic array processor (SAP). We include 4 cores each for the FFT and FIR accelerators, one core for the FEC, and two cores of the SAP. In total, the DSSoC integrates 19 PEs with a mesh-based network-on-chip to enable efficient on-chip data movement.

7.2.2 Exploration of ML Techniques and Feature Space for DAS

Machine Learning Technique Exploration: We explore different classifiers to co-optimize the *classification accuracy* and *model size* towards our minimal overhead

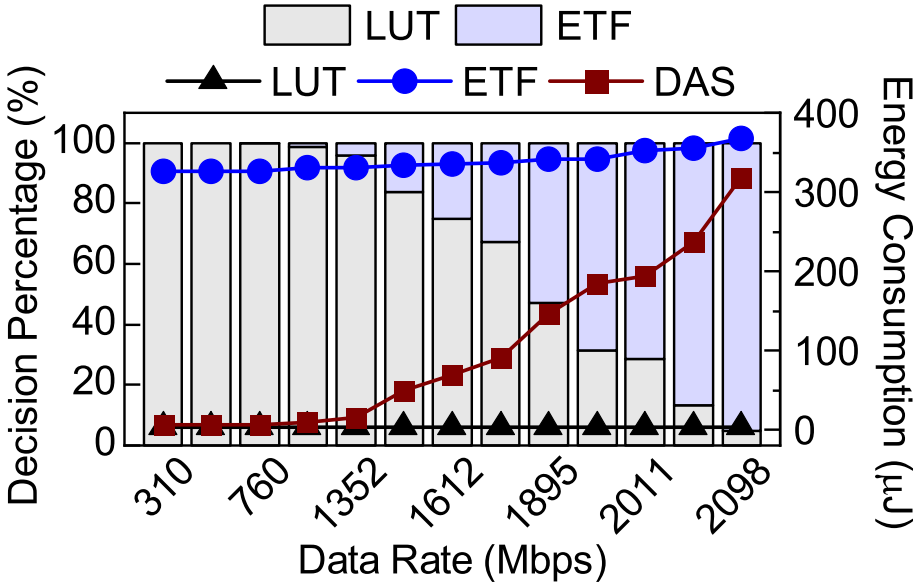

Figure 7.3: Decisions taken by the DAS framework as bar plots and total scheduling energy overheads of LUT, ETF, and DAS as line plots.

goal. Specifically, we investigated support vector classifiers (SVC), decision tree (DT), multi-layer perceptron (MLP), and logistic regression (LR). The training process with SVCs exceeded 24 h, rendering it infeasible. The latency and storage requirements of the MLP (one hidden layer and 16 neurons) did not fit the low-overhead requirements. Therefore, these two techniques are excluded from the rest of the analysis. Table 7.2 summarizes the classification accuracy and storage overheads for the LR and DTs as a function of the number of features. DTs achieve similar or higher accuracies compared to LR classifiers with lower storage overheads. While a DT with depth 16 achieves the best accuracy, there is a significant impact on the storage overhead, which influences the latency and energy consumption of the classifier. In comparison, DTs with depth 2 and 4 have negligible storage overheads with competitive accuracies (>85%). Hence, we adopt the DT with depth 2.

Feature Space Exploration: We collect 62 performance counters in our training. A systematic feature space exploration is performed using feature selection and importance methods. Growing the list from a single feature (*input data rate*) to two features with the addition of *the earliest availability time of the Arm big cluster* increases the accuracy from 63.66% to 85.48%. The data rate is tracked at runtime by an 8-entry×16-bit shift register. Therefore, we utilize only two most important features for a DT of depth 2 for the DAS classifier; this takes 13 ns on Arm Cortex-A53 cores at 1.2 GHz.

7.2.3 Performance and Scheduling Overhead Analysis

This section compares the DAS framework with LUT (*fast*), ETF (*slow*), and ETF-ideal schedulers. ETF-ideal is a version of the ETF scheduler which ignores the scheduling overhead. It helps us establish the theoretical limit of achievable execution time and EDP. Out of the 40 workloads described in Section 7.1.2, we choose three representative workloads for a detailed analysis of execution time and EDP. These workloads present different data rates. Workload-1 [Fig. 7.2(a) and (d)],

Table 7.2: Classification accuracies and storage overhead of DAS models with different machine learning classifiers and features

Classifier	Tree Depth	Number of Features	Classification Accuracy (%)	Storage (KB)
LR	-	2	79.23	0.01
LR	-	62	83.1	0.24
DT	2	**1**	**63.66**	0.01
DT	2	**2**	**85.48**	0.01
DT	4	6	85.51	0.03
DT	16	62	91.65	256

workload-2 [Fig. 7.2(b) and (e)] and workload-3 [Fig. 7.2(c) and (f)] represent low, moderate, and high data rate, respectively.

Fig. 7.2(a)–(c) [Fig. 7.2(d)–(f)] compare the execution times (EDP) of DAS, LUT, ETF, and ETF-ideal. For workloads 1 and 2, the SoC is not congested at low data rates. Hence, DAS performs similar to LUT. As data rates increase, DAS aptly chooses between LUT and ETF at runtime. Its execution time and EDP is 14% and 15% lower than LUT, and 15% and 42% lower than ETF. For workload-3, the execution time and EDP of ETF are significantly higher than LUT. DAS chooses LUT for >99% of the decisions and closely follows its trend.

This study is extended to all 40 workloads. At low data rates, DAS achieves 1.29× speedup and 45% lower EDP than ETF, and 1.28× speedup and 37% lower EDP than LUT, when the complexity increases. In summary, DAS consistently performs better than both of the underlying schedulers, successfully adapts to the workloads at runtime, and aptly chooses between LUT and ETF to achieve low execution time and EDP.

The left axis of Fig. 7.3 plots the decision distribution of DAS. It uses LUT for *all* decisions at the lowest data rate and ETF for 95% of decisions at the highest data rate. At a moderate workload of 1352 Mb/s, DAS still uses LUT for 96% of the decisions. The right axis of Fig. 7.3 shows the energy overhead of different schedulers. As DAS uses LUT and ETF based on the system load, its energy overhead varies from that of LUT to ETF. The average scheduling latency overhead of DAS under heavy workloads is 65 ns, and the energy overhead is 27.2 nJ.

We also compared DAS against a heuristic that chooses the fast scheduler when

the data rate is less than a predetermined threshold and uses the slow scheduler otherwise. The threshold is chosen judiciously by analyzing the training data used for DAS. Simulation results show that the heuristic closely follows LUT and ETF schedulers below and above the threshold, respectively. In contrast, DAS outperforms both schedulers and achieves 13% lower execution time than the heuristic.

8 CONCLUSIONS AND FUTURE DIRECTIONS

The slowdown of Moore's Law has limited the power and performance gains obtained with the evolution of technology process nodes over the years. Beyond the conventional approaches to address this challenge, IMC architectures promise to achieve superior energy efficiency by combining the computation with the communication inside the memory, thus alleviating the data communication and von Neumann bottleneck. This preliminary report addressed several critical challenges in IMC design and development to fully exploit their potential for large-scale computing to edge computing. First, we developed a heterogeneous big little chiplet architecture for IMC machine learning accelerators that optimizes the DNN model inference with a novel mapping and an NoP configuration. To the best of our knowledge, this is the first heterogeneous chiplet-based IMC architecture that leverages different IMC compute structures coupled with a heterogeneous NoP. We show that mapping the early layers to the little chiplet bank and the subsequent layers to the big chiplet bank achieves up to 2.8$\times$ higher IMC utilization and up to 329$\times$ improvement in the energy-delay-area product compared to homogeneous chiplet IMC architectures. Experimental evaluation of the proposed big-little chiplet-based RRAM IMC architecture for ResNet-50 on ImageNet shows 18.4$\times$ energy-efficiency improvement compared to SOTA chiplet-based architecture SIMBA. Then, we developed an energy-efficient on-chip learning framework for personalized home-based rehabilitation systems. To the best of our knowledge, this is the first on-chip learning framework that targets rehabilitation systems that demands a personalized solution

for patients. The proposed framework significantly improves the mmWave-based human pose estimation model through real-time on-chip learning without sacrificing user privacy. Experimental evaluations show that the proposed framework achieves up to a 45.82% decrease in MPJPE and up to 14.0× speedup for training compared to an edge device.

In summary, this report addressed the following critical gaps in the design and development of IMCs by making the following contributions:

- A detailed and comprehensive literature review on IMC architectures for large-scale computing and edge computing,
- Big Little Chiplets, a heterogeneous big little chiplet architecture for IMC machine learning accelerators [28],
- An energy-efficient on-chip training framework for home-based rehabilitation systems [29],
- DAS, dynamic adaptive scheduling framework for heterogeneous SoCs [86].

8.0.1 Future Directions

Neural Network Optimization: Machine learning (ML) algorithms exhibit high communication volume, and the communication can alone contribute to a significant portion of total inference latency in IMC-based architectures. Sparse inter-layer connections can enable energy-efficient DNNs.

This report proposes to optimize neural networks using communication-aware sparse training and latency-aware mapping to alleviate the increasing contribution of communication to the total execution.

Carbon Footprint Optimization: Computational loads continue to grow exponentially, as evidenced by the growth in AI applications and training demands. Energy consumption and resource utilization surge led to high carbon footprints. Carbon footprint is directly proportional to energy consumption given a location and a source mix of energy. Therefore, there is a need for optimization of carbon footprint.

This report proposes to optimize the carbon footprint of architecture and the DNN model by applying layer freezing and quantization. For in-memory computing-based accelerators, integrating a carbon footprint tracker into an IMC simulator can enable designers to consider the carbon footprint during the development phase.

BIBLIOGRAPHY

[1] Jason Shao, Yakun Sophia Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. SIMBA: Scaling Deep-Learning Inference with Multi-Chip-Module-based Architecture. In *IEEE/ACM MICRO*, 2019.

[2] Gokul Krishnan, Sumit K Mandal, Manvitha Pannala, Chaitali Chakrabarti, Jae-Sun Seo, Umit Y Ogras, and Yu Cao. SIAM: Chiplet-based Scalable In-Memory Acceleration with Mesh for Deep Neural Networks. *ACM TECS*, 2021.

[3] Anirban Shafiee, Ali Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. ISAAC: A Convolutional Neural Network Accelerator with in-situ Analog Arithmetic in Crossbars. *ACM/IEEE ISCA*, 2016.

[4] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring Randomly Wired Neural Networks for Image Recognition. In *IEEE/CVF ICCV*, 2019.

[5] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for Mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324, 2019.

[6] Xuehai Song, Linghao Qian, Hai Li, and Yiran Chen. Pipelayer: A Pipelined Reram-based Accelerator for Deep Learning. In *IEEE HPCA*, pages 541–552, 2017.

[7] Gokul Krishnan, Sumit K Mandal, Chaitali Chakrabarti, Jae sun Seo, Umit Y Ogras, and Yu Cao. Interconnect-aware Area and Energy Optimization for In-Memory Acceleration of DNNs. *IEEE Design & Test*, 37(6):79–87, 2020.

[8] Sumit K Mandal, Gokul Krishnan, Chaitali Chakrabarti, Jae-Sun Seo, Yu Cao, and Umit Y Ogras. A Latency-Optimized Reconfigurable NoC for In-Memory Acceleration of DNNs. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(3):362–375, 2020.

[9] Sumit K Mandal, Gokul Krishnan, A. Alper Goksoy, Gopikrishnan Ravindran Nair, Yu Cao, and Umit Y Ogras. COIN: Communication-Aware In-Memory Acceleration for Graph Convolutional Networks. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2022.

[10] Jingyang Pal, Saptadeep Liu, Irina Alam, Nicholas Cebry, Haris Suhail, Shi Bu, Subramanian S Iyer, Sudhakar Pamarti, Rakesh Kumar, and Puneet Gupta. Designing a 2048-Chiplet, 14336-Core Waferscale Processor. In *ACM/IEEE DAC*, 2021.

[11] Liangzhen Kwon, Hyoukjun Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. Heterogeneous Dataflow Accelerators for Multi-DNN Workloads. In *IEEE HPCA*, 2021.

[12] Hongyu Tan, Zhanhong Cai, Runpei Dong, and Kaisheng Ma. NN-Baton: DNN Workload Orchestration and Chiplet Granularity Exploration for Multi-chip Accelerators. In *ACM/IEEE ISCA*, 2021.

[13] Mengdi Wang, Ying Wang, Cheng Liu, and Lei Zhang. Network-on-Interposer Design for Agile Neural-Network Processor Chip Customization. In *ACM/IEEE DAC*, 2021.

[14] Gauthaman Kim, Jinwoo Murali, Heechun Park, Eric Qin, Hyoukjun Kwon, Venkata Chaitanya Krishna Chekuri, Nael Mizanur Rahman, Nihar Dasari, Arvind Singh, Minah Lee, et al. Architecture, Chip, and Package Codesign Flow for Interposer-Based 2.5D Chiplet Integration Enabling Heterogeneous IP Reuse. *IEEE TVLSI*, 2020.

[15] Eric Vivet, Pascal Guthmuller, Yvain Thonnart, Gael Pillonnet, César Fuguet, Ivan Miro-Panades, Guillaume Moritz, Jean Durupt, Christian Bernard, Didier

Varreau, et al. IntAct: A 96-core Processor with Six Chiplets 3D-stacked on an Active Interposer with Distributed Interconnects and Integrated Power Management. *IEEE JSSC*, 2020.

[16] Hao Zheng, Ke Wang, and Ahmed Louri. A Versatile and Flexible Chiplet-based System Design for Heterogeneous Manycore Architectures. In *ACM/IEEE DAC*, 2020.

[17] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. Centaur: A Chiplet-based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations. In *2020 ACM/IEEE 47th Annual ISCA*, pages 968–981. IEEE, 2020.

[18] Yuan Li, Ahmed Louri, and Avinash Karanth. Scaling deep-learning inference with chiplet-based architecture and photonic interconnects. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 931–936. IEEE, 2021.

[19] Yuan Li, Ahmed Louri, and Avinash Karanth. Spacx: Silicon photonics-based scalable chiplet accelerator for dnn inference. In *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, pages 1–13, 2022.

[20] Yuan Li, Ke Wang, Hao Zheng, Ahmed Louri, and Avinash Karanth. Ascend: A scalable and energy-efficient deep neural network accelerator with photonic interconnects. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2022.

[21] John W Turner, Walker J Poulton, John M Wilson, Xi Chen, Stephen G Tell, Matthew Fojtik, Thomas H Greer, Brian Zimmer, Sanquan Song, Nikola Nedovic, et al. Ground-Referenced Signaling for Intra-Chip and Short-Reach Chip-to-Chip Interconnects. In *IEEE CICC*, 2018.

[22] Robert Mahajan, Ravi Sankman, Neha Patel, Dae-Woo Kim, Kemal Aygun, Zhiguo Qian, Yidnekachew Mekonnen, Islam Salama, Sujit Sharan, Deepti Iyengar, et al. Embedded Multi-Die Interconnect Bridge (EMIB)–A High Density, High Bandwidth Packaging Interconnect. In *IEEE ECTC*, 2016.

[23] Min SH Aung et al. The automatic detection of chronic pain-related expression: requirements, challenges and the multimodal emopain dataset. *IEEE Trans. on Affective Computing*, 7(4):435–451, 2015.

[24] Aleksandar Vakanski et al. A data set of human body movements for physical rehabilitation exercises. *Data*, 3(1):2, 2018.

[25] Sizhe An and Umit Y Ogras. Mars: mmwave-based assistive rehabilitation system for smart healthcare. *ACM Trans. on Embedded Computing Syst.*, 20:1–22, 2021.

[26] Sizhe An, Yin Li, and Umit Ogras. mRI: Multi-modal 3d human pose estimation dataset using mmwave, RGB-d, and inertial sensors. In *Proc. of NeurIPS Datasets and Benchmarks Track*, 2022.

[27] Puck ME Schuivens et al. Impact of the covid-19 lockdown strategy on vascular surgery practice: more major amputations than usual. *Annals of Vascular Surgery*, 69:74–79, 2020.

[28] Gokul Krishnan, A. Alper Goksoy, Sumit K Mandal, Zhenyu Wang, Chaitali Chakrabarti, Jae-sun Seo, Umit Y Ogras, and Yu Cao. Big-little chiplets for in-memory acceleration of dnns: A scalable heterogeneous architecture. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[29] A. Alper Goksoy, Sizhe An, and Umit Y Ogras. Energy-efficient on-chip training for customized home-based rehabilitation systems. In *Proceedings of the 60th IEEE/ACM Design Automation Conference*, 2023.

[30] Nvidia. Jetson Xavier NX Developer Kit. `https://www.nvidia.com/en-us/autonomous-machines/embedded-Syst./jetson-xavier-nx/`, 2014. [Online; accessed 29 Sep. 2022].

[31] Leila Ma, Yenai Delshadtehrani, Cansu Demirkiran, José L Abellán, and Aiav Joshi. TAP-2.5 D: A Thermally-Aware Chiplet Placement Methodology for 2.5 D Systems. In *IEEE DATE*, 2021.

[32] Uneeb Rathore, Sumeet Singh Nagi, Subramanian Iyer, and Dejan Marković. A 16nm 785gmacs/j 784-core digital signal processor array with a multilayer switch box interconnect, assembled as a 2× 2 dielet with 10µm-pitch inter-dielet i/o for runtime multi-program reconfiguration. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 52–54. IEEE, 2022.

[33] Jieming Bharadwaj, Srikant Yin, Bradford Beckmann, and Tushar Krishna. Kite: A Family of Heterogeneous Interposer Topologies Enabled via Accurate Interconnect Modeling. In *ACM/IEEE DAC*, 2020.

[34] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 7291–7299, 2017.

[35] Ke Sun, Bin Xiao, Dong Liu, and Jingdong Wang. Deep high-resolution representation learning for human pose estimation. In *Proc. IEEE/CVF Conf. on Computer Vision and Pattern Recognition*, pages 5693–5703, 2019.

[36] Arindam Sengupta, Feng Jin, Renyuan Zhang, and Siyang Cao. mm-pose: Real-time human skeletal posture estimation using mmwave radars and cnns. *IEEE Sensors Journal*, 20(17):10032–10044, 2020.

[37] Hongfei Xue et al. mmmesh: Towards 3d real-time dynamic human mesh construction using millimeter-wave. In *Proc. 19th Int. Conf. on Mobile Syst., Applications, and Services*, pages 269–282, 2021.

[38] Shinhyun Choi et al. Sige epitaxial memory for neuromorphic computing with reproducible high performance based on engineered dislocations. *Nature Materials*, 17(4):335–340, 2018.

[39] Alessandro Grossi et al. Resistive ram endurance: Array-level characterization and correction techniques targeting deep learning applications. *IEEE Trans. on Electron Devices*, 66(3):1281–1288, 2019.

[40] Biresh Kumar Joardar et al. Learning to train cnns on faulty reram-based manycore accelerators. *ACM Trans. on Embedded Computing Syst.*, 20(5s):1–23, 2021.

[41] Xiaochen Peng et al. Dnn+ neurosim v2. 0: An end-to-end benchmarking framework for compute-in-memory accelerators for on-chip training. *IEEE TCAD*, 40(11):2306–2319, 2020.

[42] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. on Parallel and Distrib. Syst.*, 13(3):260–274, 2002.

[43] Luiz F Bittencourt, Rizos Sakellariou, and Edmundo RM Madeira. DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm. In *IEEE Euromicro Conf. on Parallel, Distrib. and Network-based Process.*, pages 27–34, 2010.

[44] Chandandeep Singh Pabla. Completely Fair Scheduler. *Linux Journal*, (184), 2009.

[45] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple Linux Utility for Resource Management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[46] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[47] Kallia Chronaki et al. Task Scheduling Techniques for Asymmetric Multi-core Systems. *IEEE Trans. on Parallel and Distrib. Systems*, 28(7):2074–2087, 2016.

[48] Junyan Zhou. Real-time Task Scheduling and Network Device Security for Complex Embedded Systems based on Deep Learning Networks. *Microprocessors and Microsystems*, 79:103282, 2020.

[49] Alireza Namazi, Saeed Safari, and Siamak Mohammadi. CMV: Clustered Majority Voting Reliability-aware Task Scheduling for Multicore Real-time Systems. *IEEE Trans. on Reliability*, 68(1):187–200, 2018.

[50] Anish Krishnakumar et al. Runtime Task Scheduling using Imitation Learning for Heterogeneous Many-core Systems. *IEEE Trans. on CAD of Integr. Circuits and Syst.*, 39(11):4064–4077, 2020.

[51] Ravindra Jejurikar and Rajesh Gupta. Energy-aware Task Scheduling with Task Synchronization for Embedded Real-time Systems. *IEEE Trans. on CAD of Integr. Circuits and Syst.*, 25(6):1024–1037, 2006.

[52] Poopak Azad and Nima Jafari Navimipour. An Energy-aware Task Scheduling in the Cloud Computing using a Hybrid Cultural and Ant Colony Optimization Algorithm. *Int. Journal of Cloud Applications and Computing*, 7(4):20–40, 2017.

[53] Achim Streit. A Self-tuning Job Scheduler Family with Dynamic Policy Switching. In *Workshop on Job Scheduling Strategies for Parallel Process.*, pages 1–23. Springer, 2002.

[54] Mohammad I Daoud and Nawwaf Kharma. A Hybrid Heuristic–genetic Algorithm for Task Scheduling in Heterogeneous Processor Networks. *Journal of Parallel and Distrib. Computing*, 71(11):1518–1531, 2011.

[55] Cristina Boeres, Alexandre Lima, and Vinod EF Rebello. Hybrid Task Scheduling: Integrating Static and Dynamic Heuristics. In *Proc. of 15th Symp. on Computer Arch. and High Perform. Computing*, pages 199–206, 2003.

[56] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural net-

works. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 45–54, New York, NY, USA, 2017. ACM.

[57] Trevor Mudge. Power: A First-class Architectural Design Constraint. *Computer*, 34(4):52–58, 2001.

[58] MICRON. Datasheet for DDR4 Model. `https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/4gb_ddr4_dram_2e0d.pdf` Accessed 29 Mar. 2021, 2014.

[59] Saransh Imani, Mohsen Gupta, Yeseong Kim, and Tajana Rosing. FloatPIM: In-memory Acceleration of Deep Neural Network Training with High Precision. In *ACM/IEEE ISCA*, 2019.

[60] David Greenhill et al. 3.3 A 14nm 1GHz FPGA with 2.5 D Transceiver Integration. In *2017 IEEE ISSCC*. IEEE, 2017.

[61] William J Poulton, John W Dally, Xi Chen, John G Eyles, Thomas H Greer, Stephen G Tell, and C Thomas Gray. A 0.54 pJ/b 20Gb/s Ground-Referenced Single-Ended Short-Haul Serial Link in 28nm CMOS for Advanced Packaging Applications. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, 2013.

[62] Michael Su, Bryan Black, Yu-Hsiang Hsiao, Chien-Lin Changchien, Chang-Chi Lee, and Hung-Jen Chang. 2.5 d ic micro-bump materials characterization and imcs evolution under reliability stress conditions. In *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*, pages 322–328. IEEE, 2016.

[63] Chester Liu, Jacob Botimer, and Zhengya Zhang. A 256gb/s/mm-shoreline aib-compatible 16nm finfet cmos chiplet for 2.5 d integration with stratix 10 fpga on emib and tiling on silicon interposer. In *2021 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–2. IEEE, 2021.

[64] Saurabh Sinha, Greg Yeric, Vikas Chandra, Brian Cline, and Yu Cao. Exploring Sub-20nm FinFET Design with Predictive Technology Models. In *DAC 2012*, pages 283–288. IEEE, 2012.

[65] CHIPS Alliance (INTEL). EMIB PHY RTL. `https://github.com/chipsalliance/aib-phy-hardware`, 2021. [Online; Accessed 20-April-2022].

[66] Texas Instruments. Datasheet. `https://www.ti.com/lit/ds/symlink/iwr1443.pdf`, 2014. [Online; accessed 8 Apr. 2022].

[67] Microsoft. Kinect sensor. `https://developer.microsoft.com/en-us/windows/kinect/`, 2014. [Online; accessed 29 Aug. 2022].

[68] Catalin Ionescu, Dragos Papava, Vlad Olaru, and Cristian Sminchisescu. Human3. 6m: Large scale datasets and predictive methods for 3d human sensing in natural environments. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 36(7):1325–1339, 2013.

[69] Chinthaka Gamanayake, Lahiru Jayasinghe, Benny Kai Kiat Ng, and Chau Yuen. Cluster Pruning: An Efficient Filter Pruning Method for Edge AI Vision Applications. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):802–816, 2020.

[70] Valentin Radu, Kuba Kaszyk, Yuan Wen, Jack Turner, José Cano, Elliot J Crowley, Björn Franke, Amos Storkey, and Michael O'Boyle. Performance Aware Convolutional Neural Network Channel Pruning for Embedded GPUs. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 24–34, 2019.

[71] Siling Yang, Weijian Chen, Xuechen Zhang, Shuibing He, Yanlong Yin, and Xian-He Sun. Auto-prune: Automated DNN Pruning and Mapping for ReRAM-based Accelerator. In *Proceedings of the ACM International Conference on Supercomputing*, pages 304–315, 2021.

[72] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable Training of Artificial Neural Networks with Adaptive Sparse Connectivity Inspired by Network Science. *Nature communications*, 9(1):1–12, 2018.

[73] Nan Wu, Lei Deng, Guoqi Li, and Yuan Xie. Core placement optimization for multi-chip many-core neural network systems with reinforcement learning. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(2):1–27, 2020.

[74] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[75] Kadan Lottick, Silvia Susai, Sorelle A Friedler, and Jonathan P Wilson. Energy usage reports: Environmental awareness as part of algorithmic accountability. *Workshop on Tackling Climate Change with Machine Learning at NeurIPS 2019*, 2019.

[76] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *Workshop on Tackling Climate Change with Machine Learning at NeurIPS 2019*, 2019.

[77] Lasse F. Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. ICML Workshop on Challenges in Deploying and monitoring Machine Learning Systems, July 2020. arXiv:2007.03051.

[78] Loïc Lannelongue, Jason Grealey, and Michael Inouye. Green algorithms: quantifying the carbon footprint of computation. *Advanced science*, 8(12):2100707, 2021.

[79] Victor Schmidt, Kamal Goyal, Aditya Joshi, Boris Feld, Liam Conell, Nikolas Laskaris, Doug Blank, Jonathan Wilson, Sorelle Friedler, and Sasha Luccioni.

Codecarbon: estimate and track carbon emissions from machine learning computing (2021). *DOI: https://doi. org/10.5281/zenodo*, 4658424, 2021.

[80] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. Towards the systematic reporting of the energy and carbon footprints of machine learning. *The Journal of Machine Learning Research*, 21(1):10039–10081, 2020.

[81] SA Budennyy, VD Lazarev, NN Zakharenko, AN Korovin, OA Plosskaya, DV Dimitrov, VS Akhripkin, IV Pavlov, IV Oseledets, IS Barsola, et al. Eco2ai: carbon emissions tracking of machine learning models as the first step towards sustainable ai. In *Doklady Mathematics*, pages 1–11. Springer, 2023.

[82] Ji Lin, Ligeng Zhu, Wei-ming Chen, Wei-chen Wang, Chuang Gan, and Song Han. On-device training under 256kb memory. In *Annual Conference on Neural Information Processing Systems*, 2022.

[83] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga, Jinshi Huang, Charles Bai, et al. Sustainable ai: Environmental implications, challenges and opportunities. *Proceedings of Machine Learning and Systems*, 4:795–813, 2022.

[84] Samet Egemen Arda et al. DS3: A System-Level Domain-Specific System-on-Chip Simulation Framework. *IEEE Trans. on Computers*, 69(8):1248–1262, 2020.

[85] Joshua Mack, Nirmal Kumbhare, Anish NK, Umit Y Ogras, and Ali Akoglu. User-Space Emulation Framework for Domain-Specific SoC Design. In *2020 IEEE Int. Parallel and Distrib. Process. Symp. Workshops*), pages 44–53, 2020.

[86] A Alper Goksoy, Anish Krishnakumar, Md Sahil Hassan, Allen J Farcas, Ali Akoglu, Radu Marculescu, and Umit Y Ogras. Das: Dynamic adaptive scheduling for energy-efficient heterogeneous socs. *IEEE Embedded Systems Letters*, 14(1):51–54, 2021.